Dask.distributed Documentation

Release 2021.05.0

Matthew Rocklin

May 14, 2021

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

ONE

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

TWO

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

THREE

CONTENTS

3.1 Install Dask.Distributed

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

3.1.1 Conda

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

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

3.1.2 Pip

Or install distributed with `pip`:

```
python -m pip install dask distributed --upgrade
```

3.1.3 Source

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

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

3.1.4 Notes

Note for Macports users: There is a known issue. with Python from macports that makes executables be placed in a location that is not available by default. A simple solution is to extend the `PATH` environment variable to the location where Python from macports install the binaries. For example, for Python 3.7:

```
$ export PATH=/opt/local/Library/Frameworks/Python.framework/Versions/3.7/bin:$PATH
```

3.2 Quickstart

3.2.1 Install

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

See *installation* document for more information.

3.2.2 Setup Dask.distributed the Easy Way

If you create a 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>
```

3.2.3 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 a 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 documentation 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
```

(continues on next page)

(continued from previous page)

```
>>> 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.

3.3 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.

3.3.1 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.

3.3.2 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*.

3.3.3 Pure Functions by Default

By default, `distributed` assumes that all functions are pure. Pure functions:

- always return the same output for a given set of inputs
- do not have side effects, like modifying global state or creating files

If this is not the case, you should use the `pure=False` keyword argument in methods like `Client.map()` and `Client.submit()`.

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'
```

3.3.4 Async/await Operation

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

```
async def f():
    client = await Client(asynchronous=True)
    future = client.submit(func, *args)
    result = await 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

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

See the *Asynchronous* documentation for more information.

3.3.5 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*

3.4 API

Client

Client([address, loop, timeout, ...])	Connect to and submit computation to a Dask cluster

Client.as_current()	Thread-local, Task-local context manager that causes the Client.current class method to return self.
Client.call_stack([futures, keys])	The actively running call stack of all relevant keys
Client.cancel(futures[, asynchronous, force])	Cancel running futures
Client.close([timeout])	Close this client
Client.collections_to_dsk(collections, ...)	Convert many collections into a single dask graph, after optimization
Client.compute(collections[, sync, ...])	Compute dask collections on cluster
Client.current([allow_global])	When running within the context of *as_client*, return the context-local current client.
Client.futures_of(futures)	
Client.gather(futures[, errors, direct, ...])	Gather futures from distributed memory
Client.get(dsk, keys[, workers, ...])	Compute dask graph
Client.get_dataset(name[, default])	Get named dataset from the scheduler if present.
Client.get_events([topic])	Retrieve structured topic logs
Client.get_executor(**kwargs)	Return a concurrent.futures Executor for submitting tasks on this Client
Client.get_metadata(keys[, default])	Get arbitrary metadata from scheduler
Client.get_scheduler_logs([n])	Get logs from scheduler
Client.get_task_stream([start, stop, count, ...])	Get task stream data from scheduler
Client.get_versions([check, packages])	Return version info for the scheduler, all workers and myself

continues on next page

Table 2 – continued from previous page

Client.get_worker_logs([n, workers, nanny])	Get logs from workers
Client.has_what([workers])	Which keys are held by which workers
Client.list_datasets(**kwargs)	List named datasets available on the scheduler
Client.log_event(topic, msg)	Log an event under a given topic
Client.map(func, *iterables[, key, workers, ...])	Map a function on a sequence of arguments
Client.nbytes([keys, summary])	The bytes taken up by each key on the cluster
Client.ncores([workers])	The number of threads/cores available on each worker node
Client.normalize_collection(collection)	Replace collection's tasks by already existing futures if they exist
Client.nthreads([workers])	The number of threads/cores available on each worker node
Client.persist(collections[, ...])	Persist dask collections on cluster
Client.processing([workers])	The tasks currently running on each worker
Client.profile([key, start, stop, workers, ...])	Collect statistical profiling information about recent work
Client.publish_dataset(*args, **kwargs)	Publish named datasets to scheduler
Client.rebalance([futures, workers])	Rebalance data within network
Client.register_worker_callbacks([setup])	Registers a setup callback function for all current and future workers.
Client.register_worker_plugin([plugin, name])	Registers a lifecycle worker plugin for all current and future workers.
Client.replicate(futures[, n, workers, ...])	Set replication of futures within network
Client.restart(**kwargs)	Restart the distributed network
Client.retire_workers([workers, close_workers])	Retire certain workers on the scheduler
Client.retry(futures[, asynchronous])	Retry failed futures
Client.run(function, *args, **kwargs)	Run a function on all workers outside of task scheduling system
Client.run_coroutine(function, *args, **kwargs)	Spawn a coroutine on all workers.
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.set_metadata(key, value)	Set arbitrary metadata in the scheduler
Client.shutdown()	Shut down the connected scheduler and workers
Client.start(**kwargs)	Start scheduler running in separate thread
Client.start_ipython(*args, **kwargs)	Deprecated - Method moved to start_ipython_workers
Client.start_ipython_scheduler([magic_name, ...])	Start IPython kernel on the scheduler
Client.start_ipython_workers([workers, ...])	Start IPython kernels on workers
Client.submit(func, *args[, key, workers, ...])	Submit a function application to the scheduler
Client.sync(func, *args[, asynchronous, ...])	
Client.unpublish_dataset(name, **kwargs)	Remove named datasets from scheduler
Client.unregister_worker_plugin(name)	Unregisters a lifecycle worker plugin
Client.upload_file(filename, **kwargs)	Upload local package to workers
Client.wait_for_workers([n_workers, timeout])	Blocking call to wait for n workers before continuing
Client.who_has([futures])	The workers storing each future's data

continues on next page

Table 2 – continued from previous page

Client.write_scheduler_file(scheduler_file)	Write the scheduler information to a json file.

worker_client([timeout, separate_thread])	Get client for this thread
get_worker()	Get the worker currently running this task
get_client([address, timeout, resolve_address])	Get a client while within a task.
secede()	Have this task secede from the worker's thread pool
rejoin()	Have this thread rejoin the ThreadPoolExecutor
Reschedule	Reschedule this task

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(**kwargs)	Cancel request to run this future
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.release([_in_destructor])	
Future.result([timeout])	Wait until computation completes, gather result to local process.
Future.retry(**kwargs)	Retry this future if it has failed
Future.traceback([timeout])	Return the traceback of a failed task

Client Coordination

Event([name, client])	Distributed Centralized Event equivalent to asyncio.Event
Lock([name, client])	Distributed Centralized Lock
MultiLock([names, client])	Distributed Centralized Lock
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	
wait(fs[, timeout, return_when])	Wait until all/any futures are finished
fire_and_forget(obj)	Run tasks at least once, even if we release the futures

continues on next page

Table 8 – continued from previous page

futures_of(o[, client])	Future objects in a collection
get_task_stream([client, plot, filename])	Collect task stream within a context block
get_task_metadata()	Collect task metadata within a context block

3.4.1 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.

3.4.2 Client

class distributed.**Client**(*address=None, loop=None, timeout='__no_default__', set_as_default=True, scheduler_file=None, security=None, asynchronous=False, name=None, heartbeat_interval=None, serializers=None, deserializers=None, extensions=[<class 'distributed.pubsub.PubSubClientExtension'>], direct_to_workers=None, connection_limit=512, **kwargs*)

Connect to and submit computation to a Dask cluster

The Client connects users to a Dask 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. When a Client is instantiated it takes over all `dask.compute` and `dask.persist` calls by default.

It is also common to create a Client without specifying the scheduler address , like `Client()`. In this case the Client creates a *LocalCluster* in the background and connects to that. Any extra keywords are passed from Client to LocalCluster in this case. See the LocalCluster documentation for more information.

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: Security or bool, optional Optional security information. If creating a local cluster can also pass in `True`, in which case temporary self-signed credentials will be created automatically.

asynchronous: bool (False by default) Set to True if using this client within async/await functions or within Tornado gen.coroutines. Otherwise this should remain False for normal use.

name: string (optional) Gives the client a name that will be included in logs generated on the scheduler for matters relating to this client

direct_to_workers: bool (optional) Whether or not to connect directly to the workers, or to ask the scheduler to serve as intermediary.

heartbeat_interval: int Time in milliseconds between heartbeats to scheduler

****kwargs:** If you do not pass a scheduler address, Client will create a `LocalCluster` object, passing any extra keyword arguments.

See also:

distributed.scheduler.Scheduler Internal scheduler

distributed.LocalCluster

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
```

You can also call Client with no arguments in order to create your own local cluster.

```
>>> client = Client()  # makes your own local "cluster"
```

Extra keywords will be passed directly to LocalCluster

```
>>> client = Client(processes=False, threads_per_worker=1)
```

as_current()
Thread-local, Task-local context manager that causes the Client.current class method to return self. Any Future objects deserialized inside this context manager will be automatically attached to this Client.

property 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.

call_stack(*futures=None*, *keys=None*)
The actively running call stack of all relevant keys

You can specify data of interest either by providing futures or collections in the `futures=` keyword or a list of explicit keys in the `keys=` keyword. If neither are provided then all call stacks will be returned.

Parameters

futures [list (optional)] List of futures, defaults to all data

keys [list (optional)] List of key names, defaults to all data

Examples

```
>>> df = dd.read_parquet(...).persist()
>>> client.call_stack(df)  # call on collections
```

```
>>> client.call_stack()  # Or call with no arguments for all activity
```

cancel(*futures*, *asynchronous=None*, *force=False*)
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]

force [boolean (False)] Cancel this future even if other clients desire it

close(*timeout='__no_default__'*)
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

static collections_to_dsk(*collections*, **args*, ***kwargs*)
Convert many collections into a single dask graph, after optimization

compute(*collections*, *sync=False*, *optimize_graph=True*, *workers=None*, *allow_other_workers=False*, *resources=None*, *retries=0*, *priority=0*, *fifo_timeout='60s'*, *actors=None*, *traverse=True*, ***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 [string or iterable of strings] 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*. Indicates whether or not the computations may be performed on workers that are not in the *workers* set(s).

retries [int (default to 0)] Number of allowed automatic retries if computing a result fails

priority [Number] Optional prioritization of task. Zero is default. Higher priorities take precedence

fifo_timeout [timedelta str (defaults to '60s')] Allowed amount of time between calls to consider the same priority

traverse [bool (defaults to True)] By default dask traverses builtin python collections looking for dask objects passed to `compute`. For large collections this can be expensive. If none of the arguments contain any dask objects, set `traverse=False` to avoid doing this traversal.

resources [dict (defaults to {})] Defines the *resources* each instance of this mapped task requires on the worker; e.g. `{'GPU': 2}`. See *worker resources* for details on defining resources.

actors [bool or dict (default None)] Whether these tasks should exist on the worker as stateful actors. Specified on a global (True/False) or per-task (`{'x': True, 'y': False}`) basis. See *Actors* for additional details.

****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()
```

(continues on next page)

(continued from previous page)

```
3
>>> yy.result()
6
```

Also support single arguments

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

classmethod current(*allow_global=True*)
When running within the context of *as_client*, return the context-local current client. Otherwise, return the latest initialised Client. If no Client instances exist, raise ValueError. If allow_global is set to False, raise ValueError if running outside of the *as_client* context manager.

property dashboard_link
Link to the scheduler's dashboard.

Returns

str Dashboard URL.

Examples

Opening the dashboard in your default web browser:

```
>>> import webbrowser
>>> from distributed import Client
>>> client = Client()
>>> webbrowser.open(client.dashboard_link)
```

gather(*futures*, *errors='raise'*, *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, 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

direct [boolean] Whether or not to connect directly to the workers, or to ask the scheduler to serve as intermediary. This can also be set when creating the Client.

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]
```

get(*dsk*, *keys*, *workers=None*, *allow_other_workers=None*, *resources=None*, *sync=True*, *asynchronous=None*, *direct=None*, *retries=None*, *priority=0*, *fifo_timeout='60s'*, *actors=None*, ***kwargs*)
Compute dask graph

Parameters

dsk [dict]

keys [object, or nested lists of objects]

workers [string or iterable of strings] A set of worker addresses or 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`. Indicates whether or not the computations may be performed on workers that are not in the *workers* set(s).

retries [int (default to 0)] Number of allowed automatic retries if computing a result fails

priority [Number] Optional prioritization of task. Zero is default. Higher priorities take precedence

resources [dict (defaults to {})] Defines the `resources` each instance of this mapped task requires on the worker; e.g. `{'GPU': 2}`. See *worker resources* for details on defining resources.

sync [bool (optional)] Returns Futures if False or concrete values if True (default).

direct [bool] Whether or not to connect directly to the workers, or to ask the scheduler to serve as intermediary. This can also be set when creating the Client.

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*, *default='_no_default_'*, ***kwargs*)
Get named dataset from the scheduler if present. Return the default or raise a KeyError if not present.

Parameters

name [name of the dataset to retrieve]

default [optional, not set by default] If set, do not raise a KeyError if the name is not present but return this default

kwargs [dict] additional arguments to _get_dataset

See also:

Client.publish_dataset

Client.list_datasets

get_events(*topic: Optional[str] = None*)
Retrieve structured topic logs

Parameters

topic [str, optional] Name of topic log to retrieve events for. If no `topic` is provided, then logs for all topics will be returned.

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.

get_metadata(*keys*, *default='__no_default__'*)
Get arbitrary metadata from scheduler

See set_metadata for the full docstring with examples

Parameters

keys [key or list] Key to access. If a list then gets within a nested collection

default [optional] If the key does not exist then return this value instead. If not provided then this raises a KeyError if the key is not present

See also:

Client.set_metadata

get_scheduler_logs(*n=None*)
Get logs from scheduler

Parameters

n [int] Number of logs to retrive. Maxes out at 10000 by default, configurable via the `distributed.admin.log-length` configuration value.

Returns

Logs in reversed order (newest first)

get_task_stream(*start=None*, *stop=None*, *count=None*, *plot=False*, *filename='task-stream.html'*, *bokeh_resources=None*)
Get task stream data from scheduler

This collects the data present in the diagnostic "Task Stream" plot on the dashboard. It includes the start, stop, transfer, and deserialization time of every task for a particular duration.

Note that the task stream diagnostic does not run by default. You may wish to call this function once before you start work to ensure that things start recording, and then again after you have completed.

Parameters

start [Number or string] When you want to start recording If a number it should be the result of calling time() If a string then it should be a time difference before now, like '60s' or '500 ms'

stop [Number or string] When you want to stop recording

count [int] The number of desired records, ignored if both start and stop are specified

plot [boolean, str] If true then also return a Bokeh figure If plot == 'save' then save the figure to a file

filename [str (optional)] The filename to save to if you set `plot='save'`

bokeh_resources [bokeh.resources.Resources (optional)] Specifies if the resource component is INLINE or CDN

Returns

L: List[Dict]

See also:

get_task_stream a context manager version of this method

Examples

```
>>> client.get_task_stream()  # prime plugin if not already connected
>>> x.compute()  # do some work
>>> client.get_task_stream()
[{'task': ...,
  'type': ...,
  'thread': ...,
  ...}]
```

Pass the `plot=True` or `plot='save'` keywords to get back a Bokeh figure

```
>>> data, figure = client.get_task_stream(plot='save', filename='myfile.html')
```

Alternatively consider the context manager

```
>>> from dask.distributed import get_task_stream
>>> with get_task_stream() as ts:
...     x.compute()
>>> ts.data
[...]
```

get_versions(*check=False*, *packages=[]*)

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

packages [List[str]] Extra package names to check

Examples

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

```
>>> c.get_versions(packages=['sklearn', 'geopandas'])
```

get_worker_logs(*n=None*, *workers=None*, *nanny=False*)
Get logs from workers

Parameters

n [int] Number of logs to retrive. Maxes out at 10000 by default, configurable via the `distributed.admin.log-length` configuration value.

workers [iterable] List of worker addresses to retrieve. Gets all workers by default.

nanny [bool, default False] Whether to get the logs from the workers (False) or the nannies (True). If specified, the addresses in *workers* should still be the worker addresses, not the nanny addresses.

Returns

Dictionary mapping worker address to logs.

Logs are returned in reversed order (newest first)

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

This returns the keys of the data that are held in each worker's memory.

Parameters

workers [list (optional)] A list of worker addresses, defaults to all

See also:

Client.who_has

Client.nthreads

Client.processing

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

log_event(*topic*, *msg*)
Log an event under a given topic

Parameters

topic [str, list] Name of the topic under which to log an event. To log the same event under multiple topics, pass a list of topic names.

msg Event message to log. Note this must be msgpack serializable.

Examples

```
>>> from time import time
>>> client.log_event("current-time", time())
```

map(*func*, **iterables*, *key=None*, *workers=None*, *retries=None*, *resources=None*, *priority=0*, *allow_other_workers=False*, *fifo_timeout='100 ms'*, *actor=False*, *actors=False*, *pure=None*, *batch_size=None*, ***kwargs*)
Map a function on a sequence of arguments

Arguments can be normal objects or Futures

Parameters

func [callable]

iterables [Iterables] List-like objects to map over. They should have the same length.

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`. See *Pure Functions by Default* for more details.

workers [string or iterable of strings] 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*. Indicates whether or not the computations may be performed on workers that are not in the *workers* set(s).

retries [int (default to 0)] Number of allowed automatic retries if a task fails

priority [Number] Optional prioritization of task. Zero is default. Higher priorities take precedence

fifo_timeout [str timedelta (default '100ms')] Allowed amount of time between calls to consider the same priority

resources [dict (defaults to {})] Defines the *resources* each instance of this mapped task requires on the worker; e.g. `{'GPU': 2}`. See *worker resources* for details on defining resources.

actor [bool (default False)] Whether these tasks should exist on the worker as stateful actors. See *Actors* for additional details.

actors [bool (default False)] Alias for *actor*

batch_size [int, optional] Submit tasks to the scheduler in batches of (at most) `batch_size`. Larger batch sizes can be useful for very large `iterables`, as the cluster can start processing tasks while later ones are submitted asynchronously.

****kwargs** [dict] Extra keywords to send to the function. Large values will be included explicitly in the task graph.

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.threads()
{'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_graph__())  # x is a dask collection with 100 tasks
100
>>> set(client.futures).intersection(x.__dask_graph__())  # some overlap␣
↪exists
10
```

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

nthreads(*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.threads()
{'192.168.1.141:46784': 8,
 '192.167.1.142:47548': 8,
 '192.167.1.143:47329': 8,
 '192.167.1.144:37297': 8}
```

persist(*collections*, *optimize_graph=True*, *workers=None*, *allow_other_workers=None*, *resources=None*, *retries=None*, *priority=0*, *fifo_timeout='60s'*, *actors=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 [string or iterable of strings] 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*. Indicates whether or not the computations may be performed on workers that are not in the *workers* set(s).

retries [int (default to 0)] Number of allowed automatic retries if computing a result fails

priority [Number] Optional prioritization of task. Zero is default. Higher priorities take precedence

fifo_timeout [timedelta str (defaults to '60s')] Allowed amount of time between calls to consider the same priority

resources [dict (defaults to {})] Defines the *resources* each instance of this mapped task requires on the worker; e.g. `{'GPU': 2}`. See *worker resources* for details on defining resources.

actors [bool or dict (default None)] Whether these tasks should exist on the worker as stateful actors. Specified on a global (True/False) or per-task (`{'x': True, 'y': False}`) basis. See *Actors* for additional details.

****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.who_has

Client.has_what

Client.nthreads

Examples

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

profile(*key=None*, *start=None*, *stop=None*, *workers=None*, *merge_workers=True*, *plot=False*, *filename=None*, *server=False*, *scheduler=False*)

Collect statistical profiling information about recent work

Parameters

key [str] Key prefix to select, this is typically a function name like 'inc' Leave as None to collect all data

start [time]

stop [time]

workers [list] List of workers to restrict profile information

server [bool] If true, return the profile of the worker's administrative thread rather than the worker threads. This is useful when profiling Dask itself, rather than user code.

scheduler [bool] If true, return the profile information from the scheduler's administrative thread rather than the workers. This is useful when profiling Dask's scheduling itself.

plot [boolean or string] Whether or not to return a plot object

filename [str] Filename to save the plot

Examples

```
>>> client.profile()  # call on collections
>>> client.profile(filename='dask-profile.html')  # save to html file
```

publish_dataset(**args*, ***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

args [list of objects to publish as name]

name [optional name of the dataset to publish]

override [bool (optional, default False)] if true, override any already present dataset with the same name

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)
```

Alternative invocation >>> c.publish_dataset(df, name='my_dataset')

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 is 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

register_worker_callbacks (*setup=None*)
Registers a setup callback function for all current and future workers.

This registers a new setup function for workers in this cluster. The function will run immediately on all currently connected workers. It will also be run upon connection by any workers that are added in the future. Multiple setup functions can be registered - these will be called in the order they were added.

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

Parameters

setup [callable(dask_worker: Worker) -> None] Function to register and run on all workers

register_worker_plugin (*plugin=None*, *name=None*, ***kwargs*)
Registers a lifecycle worker plugin for all current and future workers.

This registers a new object to handle setup, task state transitions and teardown for workers in this cluster. The plugin will instantiate itself on all currently connected workers. It will also be run on any worker that connects in the future.

The plugin may include methods `setup`, `teardown`, `transition`, and `release_key`. See the `dask.distributed.WorkerPlugin` class or the examples below for the interface and docstrings. It must be serializable with the pickle or cloudpickle modules.

If the plugin has a `name` attribute, or if the `name=` keyword is used then that will control idempotency. If a plugin with that name has already been registered then any future plugins will not run.

For alternatives to plugins, you may also wish to look into preload scripts.

Parameters

plugin [WorkerPlugin] The plugin object to pass to the workers

name [str, optional] A name for the plugin. Registering a plugin with the same name will have no effect. If plugin has no name attribute a random name is used.

****kwargs** [optional] If you pass a class as the plugin, instead of a class instance, then the class will be instantiated with any extra keyword arguments.

See also:

distributed.WorkerPlugin

unregister_worker_plugin

Examples

```
>>> class MyPlugin(WorkerPlugin):
...     def __init__(self, *args, **kwargs):
...         pass  # the constructor is up to you
...     def setup(self, worker: dask.distributed.Worker):
...         pass
...     def teardown(self, worker: dask.distributed.Worker):
...         pass
...     def transition(self, key: str, start: str, finish: str, **kwargs):
...         pass
...     def release_key(self, key: str, state: str, cause: Optional[str],␣
↪reason: None, report: bool):
...         pass
```

```
>>> plugin = MyPlugin(1, 2, 3)
>>> client.register_worker_plugin(plugin)
```

You can get access to the plugin with the `get_worker` function

```
>>> client.register_worker_plugin(other_plugin, name='my-plugin')
>>> def f():
...     worker = get_worker()
...     plugin = worker.plugins['my-plugin']
...     return plugin.my_state
```

```
>>> future = client.run(f)
```

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.

retire_workers (*workers=None*, *close_workers=True*, ***kwargs*)
Retire certain workers on the scheduler

See dask.distributed.Scheduler.retire_workers for the full docstring.

See also:

dask.distributed.Scheduler.retire_workers

Examples

You can get information about active workers using the following:

```
>>> workers = client.scheduler_info()['workers']
```

From that list you may want to select some workers to close

```
>>> client.retire_workers(workers=['tcp://address:port', ...])
```

retry (*futures*, *asynchronous=None*)
Retry failed futures

Parameters

futures [list of Futures]

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.

wait [boolean (optional)] If the function is asynchronous whether or not to wait until that function finishes.

nanny [bool, defualt False] Whether to run `function` on the nanny. By default, the function is run on the worker process. If specified, the addresses in `workers` should still be the worker addresses, not the nanny addresses.

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 asynchronous functions in the background:

```
>>> async def print_state(dask_worker):
...     while True:
...         print(dask_worker.status)
...         await asyncio.sleep(1)
```

```
>>> c.run(print_state, wait=False)
```

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

This spawns 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.tasks)
```

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

Run asynchronous functions in the background:

```
>>> async def print_state(dask_scheduler):
...     while True:
...         print(dask_scheduler.status)
...         await asyncio.sleep(1)
```

```
>>> c.run(print_state, wait=False)
```

scatter (*data*, *workers=None*, *broadcast=False*, *direct=None*, *hash=True*, *timeout='__no_default__'*, *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, dict, 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)] Whether or not to connect directly to the workers, or to ask the scheduler to serve as intermediary. This can also be set when creating the Client.

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)])
```

Broadcast data to all workers

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

Send scattered data to parallelized function using client futures interface

```
>>> data = c.scatter(data, broadcast=True)
>>> res = [c.submit(func, data, i) for i in range(100)]
```

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}}}
```

set_metadata (*key*, *value*)
Set arbitrary metadata in the scheduler

This allows you to store small amounts of data on the central scheduler process for administrative purposes. Data should be msgpack serializable (ints, strings, lists, dicts)

If the key corresponds to a task then that key will be cleaned up when the task is forgotten by the scheduler.

If the key is a list then it will be assumed that you want to index into a nested dictionary structure using those keys. For example if you call the following:

```
>>> client.set_metadata(['a', 'b', 'c'], 123)
```

Then this is the same as setting

```
>>> scheduler.task_metadata['a']['b']['c'] = 123
```

The lower level dictionaries will be created on demand.

See also:

get_metadata

Examples

```
>>> client.set_metadata('x', 123)
>>> client.get_metadata('x')
123
```

```
>>> client.set_metadata(['x', 'y'], 123)
>>> client.get_metadata('x')
{'y': 123}
```

```
>>> client.set_metadata(['x', 'w', 'z'], 456)
>>> client.get_metadata('x')
{'y': 123, 'w': {'z': 456}}
```

```
>>> client.get_metadata(['x', 'w'])
{'z': 456}
```

shutdown ()
Shut down the connected scheduler and workers

Note, this may disrupt other clients that may be using the same scheduler and workers.

See also:

Client.close close only this client

start(**kwargs)
Start scheduler running in separate thread

start_ipython(*args, **kwargs)
Deprecated - Method moved to start_ipython_workers

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*, *key=None*, *workers=None*, *resources=None*, *retries=None*, *priority=0*, *fifo_timeout='100 ms'*, *allow_other_workers=False*, *actor=False*, *actors=False*, *pure=None*, ***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`. See *Pure Functions by Default* for more details.

workers [string or iterable of strings] A set of worker addresses or hostnames on which computations may be performed. Leave empty to default to all workers (common case)

key [str] Unique identifier for the task. Defaults to function-name and hash

allow_other_workers [bool (defaults to False)] Used with `workers`. Indicates whether or not the computations may be performed on workers that are not in the *workers* set(s).

retries [int (default to 0)] Number of allowed automatic retries if the task fails

priority [Number] Optional prioritization of task. Zero is default. Higher priorities take precedence

fifo_timeout [str timedelta (default '100ms')] Allowed amount of time between calls to consider the same priority

resources [dict (defaults to {})] Defines the `resources` each instance of this mapped task requires on the worker; e.g. `{'GPU': 2}`. See *worker resources* for details on defining resources.

actor [bool (default False)] Whether this task should exist on the worker as a stateful actor. See *Actors* for additional details.

actors [bool (default False)] Alias for *actor*

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()
[]
```

unregister_worker_plugin(*name*)
Unregisters a lifecycle worker plugin

This unregisters an existing worker plugin. As part of the unregistration process the plugin's `teardown` method will be called.

Parameters

name [str] Name of the plugin to unregister. See the *Client.register_worker_plugin()* docstring for more information.

See also:

register_worker_plugin

Examples

```
>>> class MyPlugin(WorkerPlugin):
...     def __init__(self, *args, **kwargs):
...         pass  # the constructor is up to you
...     def setup(self, worker: dask.distributed.Worker):
...         pass
...     def teardown(self, worker: dask.distributed.Worker):
...         pass
...     def transition(self, key: str, start: str, finish: str, **kwargs):
...         pass
...     def release_key(self, key: str, state: str, cause: Optional[str],␣
↪reason: None, report: bool):
...         pass
```

```
>>> plugin = MyPlugin(1, 2, 3)
>>> client.register_worker_plugin(plugin, name='foo')
>>> client.unregister_worker_plugin(name='foo')
```

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, .egg or .zip files will be importable.

Parameters

filename [string] Filename of .py, .egg or .zip file to send to workers

Examples

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

wait_for_workers (*n_workers=0*, *timeout=None*)
Blocking call to wait for n workers before continuing

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.nthreads

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']}
```

write_scheduler_file(*scheduler_file*)

Write the scheduler information to a json file.

This facilitates easy sharing of scheduler information using a file system. The scheduler file can be used to instantiate a second Client using the same scheduler.

Parameters

scheduler_file [str] Path to a write the scheduler file.

Examples

```
>>> client = Client()
>>> client.write_scheduler_file('scheduler.json')
# connect to previous client's scheduler
>>> client2 = Client(scheduler_file='scheduler.json')
```

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>
```

3.4.3 Future

class distributed.**Future**(*key*, *client=None*, *inform=True*, *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.

Parameters

key: str, or tuple Key of remote data to which this future refers

client: Client Client that should own this future. Defaults to _get_global_client()

inform: bool Do we inform the scheduler that we need an update on this future

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(***kwargs*)
Cancel request to run this future

See also:

Client.cancel

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 `dask.distributed.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 `dask.distributed.TimeoutError` is raised.

retry(***kwargs*)
Retry this future if it has failed

See also:

Client.retry

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 `dask.distributed.TimeoutError` is raised.

See also:

Future.exception

Examples

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

3.4.4 Cluster

Classes relevant for cluster creation and management. Other libraries (like dask-jobqueue, dask-gateway, dask-kubernetes, dask-yarn etc.) provide additional cluster objects.

LocalCluster([name, n_workers, ...])	Create local Scheduler and Workers
SpecCluster([workers, scheduler, worker, ...])	Cluster that requires a full specification of workers

class distributed.**LocalCluster**(*name=None*, *n_workers=None*, *threads_per_worker=None*, *processes=True*, *loop=None*, *start=None*, *host=None*, *ip=None*, *scheduler_port=0*, *silence_logs=30*, *dashboard_address=':8787'*, *worker_dashboard_address=None*, *diagnostics_port=None*, *services=None*, *worker_services=None*, *service_kwargs=None*, *asynchronous=False*, *security=None*, *protocol=None*, *blocked_handlers=None*, *interface=None*, *worker_class=None*, *scheduler_kwargs=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.WARN` by default. Use a falsey value like False or None for no change.

host: string Host address on which the scheduler will listen, defaults to only localhost

ip: string Deprecated. See `host` above.

dashboard_address: str Address on which to listen for the Bokeh diagnostics server like 'localhost:8787' or '0.0.0.0:8787'. Defaults to ':8787'. Set to `None` to disable the dashboard. Use ':0' for a random port.

worker_dashboard_address: str Address on which to listen for the Bokeh worker diagnostics server like 'localhost:8787' or '0.0.0.0:8787'. Defaults to None which disables the dashboard. Use ':0' for a random port.

diagnostics_port: int Deprecated. See dashboard_address.

asynchronous: bool (False by default) Set to True if using this cluster within async/await functions or within Tornado gen.coroutines. This should remain False for normal use.

blocked_handlers: List[str] A list of strings specifying a blacklist of handlers to disallow on the Scheduler, like `['feed', 'run_function']`

service_kwargs: Dict[str, Dict] Extra keywords to hand to the running services

security [Security or bool, optional] Configures communication security in this cluster. Can be a security object, or True. If True, temporary self-signed credentials will be created automatically.

protocol: str (optional) Protocol to use like `tcp://`, `tls://`, `inproc://` This defaults to sensible choice given other keyword arguments like `processes` and `security`

interface: str (optional) Network interface to use. Defaults to lo/localhost

worker_class: Worker Worker class used to instantiate workers from.

****worker_kwargs:** Extra worker arguments. Any additional keyword arguments will be passed to the `Worker` class constructor.

Examples

```
>>> cluster = LocalCluster()  # Create a local cluster
>>> cluster
LocalCluster("127.0.0.1:8786", workers=8, threads=8)
```

```
>>> c = Client(cluster)  # connect to local cluster
```

Scale the cluster to three workers

```
>>> cluster.scale(3)
```

Pass extra keyword arguments to Bokeh

```
>>> LocalCluster(service_kwargs={'dashboard': {'prefix': '/foo'}})
```

class distributed.**SpecCluster**(*workers=None*, *scheduler=None*, *worker=None*, *asynchronous=False*, *loop=None*, *security=None*, *silence_logs=False*, *name=None*, *shutdown_on_close=True*)

Cluster that requires a full specification of workers

The SpecCluster class expects a full specification of the Scheduler and Workers to use. It removes any handling of user inputs (like threads vs processes, number of cores, and so on) and any handling of cluster resource managers (like pods, jobs, and so on). Instead, it expects this information to be passed in scheduler and worker specifications. This class does handle all of the logic around asynchronously cleanly setting up and tearing things down at the right times. Hopefully it can form a base for other more user-centric classes.

Parameters

workers: dict A dictionary mapping names to worker classes and their specifications See example below

scheduler: dict, optional A similar mapping for a scheduler

worker: dict A specification of a single worker. This is used for any new workers that are created.

asynchronous: bool If this is intended to be used directly within an event loop with async/await

silence_logs: bool Whether or not we should silence logging when setting up the cluster.

name: str, optional A name to use when printing out the cluster, defaults to type name

Examples

To create a SpecCluster you specify how to set up a Scheduler and Workers

```
>>> from dask.distributed import Scheduler, Worker, Nanny
>>> scheduler = {'cls': Scheduler, 'options': {"dashboard_address": ':8787'}}
>>> workers = {
...     'my-worker': {"cls": Worker, "options": {"nthreads": 1}},
...     'my-nanny': {"cls": Nanny, "options": {"nthreads": 2}},
... }
>>> cluster = SpecCluster(scheduler=scheduler, workers=workers)
```

The worker spec is stored as the `.worker_spec` attribute

```
>>> cluster.worker_spec
{
   'my-worker': {"cls": Worker, "options": {"nthreads": 1}},
   'my-nanny': {"cls": Nanny, "options": {"nthreads": 2}},
}
```

While the instantiation of this spec is stored in the `.workers` attribute

```
>>> cluster.workers
{
    'my-worker': <Worker ...>
    'my-nanny': <Nanny ...>
}
```

Should the spec change, we can await the cluster or call the `._correct_state` method to align the actual state to the specified state.

We can also `.scale(...)` the cluster, which adds new workers of a given form.

```
>>> worker = {'cls': Worker, 'options': {}}
>>> cluster = SpecCluster(scheduler=scheduler, worker=worker)
>>> cluster.worker_spec
{}
```

```
>>> cluster.scale(3)
>>> cluster.worker_spec
{
    0: {'cls': Worker, 'options': {}},
    1: {'cls': Worker, 'options': {}},
    2: {'cls': Worker, 'options': {}},
}
```

Note that above we are using the standard `Worker` and `Nanny` classes, however in practice other classes could be used that handle resource management like `KubernetesPod` or `SLURMJob`. The spec does not need to conform to the expectations of the standard Dask Worker class. It just needs to be called with the provided options, support `__await__` and `close` methods and the `worker_address` property..

Also note that uniformity of the specification is not required. Other API could be added externally (in subclasses) that adds workers of different specifications into the same dictionary.

If a single entry in the spec will generate multiple dask workers then please provide a *"group"* element to the spec, that includes the suffixes that will be added to each name (this should be handled by your worker class).

```
>>> cluster.worker_spec
{
    0: {"cls": MultiWorker, "options": {"processes": 3}, "group": ["-0", "-1", -2
↪"]}
    1: {"cls": MultiWorker, "options": {"processes": 2}, "group": ["-0", "-1"]}
}
```

These suffixes should correspond to the names used by the workers when they deploy.

```
>>> [ws.name for ws in cluster.scheduler.workers.values()]
["0-0", "0-1", "0-2", "1-0", "1-1"]
```

adapt(**args*, *minimum=0*, *maximum=inf*, *minimum_cores: Optional[int] = None*, *maximum_cores: Optional[int] = None*, *minimum_memory: Optional[str] = None*, *maximum_memory: Optional[str] = None*, ***kwargs*) → distributed.deploy.adaptive.Adaptive

Turn on adaptivity

This scales Dask clusters automatically based on scheduler activity.

Parameters

minimum [int] Minimum number of workers

maximum [int] Maximum number of workers

minimum_cores [int] Minimum number of cores/threads to keep around in the cluster

maximum_cores [int] Maximum number of cores/threads to keep around in the cluster

minimum_memory [str] Minimum amount of memory to keep around in the cluster Expressed as a string like "100 GiB"

maximum_memory [str] Maximum amount of memory to keep around in the cluster Expressed as a string like "100 GiB"

See also:

dask.distributed.Adaptive for more keyword arguments

Examples

```
>>> cluster.adapt(minimum=0, maximum_memory="100 GiB", interval='500ms')
```

classmethod from_name(*name: str*)
Create an instance of this class to represent an existing cluster by name.

new_worker_spec()
Return name and spec for the next worker

Returns

d: dict mapping names to worker specs

See also:

scale

scale(*n=0*, *memory=None*, *cores=None*)
Scale cluster to n workers

Parameters

n [int] Target number of workers

Examples

```
>>> cluster.scale(10)  # scale cluster to ten workers
```

scale_up(*n=0*, *memory=None*, *cores=None*)
Scale cluster to n workers

Parameters

n [int] Target number of workers

Examples

```
>>> cluster.scale(10)  # scale cluster to ten workers
```

3.4.5 Other

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

Parameters

futures: Collection of futures A list of Future objects to be iterated over in the order in which they complete

with_results: bool (False) Whether to wait and include results of futures as well; in this case *as_completed* yields a tuple of (future, result)

raise_errors: bool (True) Whether we should raise when the result of a future raises an exception; only affects behavior when *with_results=True*.

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], with_results=True)
>>> for future, result in ac:
...     print(result)
2
4
3
```

add(*future*)
Add a future to the collection

This future will emit from the iterator once it finishes

batches()
Yield all finished futures at once rather than one-by-one

This returns an iterator of lists of futures or lists of (future, result) tuples rather than individual futures or individual (future, result) tuples. It will yield these as soon as possible without waiting.

Examples

```
>>> for batch in as_completed(futures).batches():
...     results = client.gather(batch)
...     print(results)
[4, 2]
[1, 3, 7]
[5]
[6]
```

clear()
Clear out all submitted futures

count()
Return the number of futures yet to be returned

This includes both the number of futures still computing, as well as those that are finished, but have not yet been returned from this iterator.

has_ready()
Returns True if there are completed futures available.

is_empty()
Returns True if there no completed or computing futures

next_batch(*block=True*)
Get the next batch of completed futures.

Parameters

block [bool, optional] If True then wait until we have some result, otherwise return immediately, even with an empty list. Defaults to True.

Returns

List of futures or (future, result) tuples

Examples

```
>>> ac = as_completed(futures)
>>> client.gather(ac.next_batch())
[4, 1, 3]
```

```
>>> client.gather(ac.next_batch(block=False))
[]
```

update(*futures*)
Add multiple futures to the collection.

The added futures will emit from the iterator once they finish

distributed.diagnostics.**progress**()

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

Parameters

fs [list of futures]

timeout [number, optional] Time in seconds after which to raise a `dask.distributed.TimeoutError`

return_when [str, optional] One of *ALL_COMPLETED* or *FIRST_COMPLETED*

Returns

Named tuple of completed, not completed

`distributed.`**`fire_and_forget`**(*obj*)

Run tasks at least once, even if we release the futures

Under normal operation Dask will not run any tasks for which there is not an active future (this avoids unnecessary work in many situations). However sometimes you want to just fire off a task, not track its future, and expect it to finish eventually. You can use this function on a future or collection of futures to ask Dask to complete the task even if no active client is tracking it.

The results will not be kept in memory after the task completes (unless there is an active future) so this is only useful for tasks that depend on side effects.

Parameters

obj [Future, list, dict, dask collection] The futures that you want to run at least once

Examples

```
>>> fire_and_forget(client.submit(func, *args))
```

`distributed.`**`futures_of`**(*o*, *client=None*)

Future objects in a collection

Parameters

o [collection] A possibly nested collection of Dask objects

Returns

futures [List[Future]] A list of futures held by those collections

Examples

```
>>> futures_of(my_dask_dataframe)
[<Future: finished key: ...>,
 <Future: pending  key: ...>]
```

`distributed.`**`worker_client`**(*timeout=None*, *separate_thread=True*)

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 or String] Timeout after which to error out. Defaults to the `distributed.comm.timeouts.connect` configuration value.

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(timeout="10s") 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=None*, *resolve_address=True*)
Get a client while within a task.

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

Parameters

address [str, optional] The address of the scheduler to connect to. Defaults to the scheduler the worker is connected to.

timeout [int or str] Timeout (in seconds) for getting the Client. Defaults to the `distributed.comm.timeouts.connect` configuration value.

resolve_address [bool, default True] Whether to resolve *address* to its canonical form.

Returns

Client

See also:

get_worker

worker_client

secede

Examples

```
>>> def f():
...     client = get_client(timeout="10s")
...     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. This enables the client to schedule tasks on this node, which is especially useful while waiting for other jobs to finish (e.g., with `client.gather`).

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
```

distributed.**rejoin**()
Have this thread rejoin the ThreadPoolExecutor

This will block until a new slot opens up in the executor. The next thread to finish a task will leave the pool to allow this one to join.

See also:

secede leave the thread pool

class distributed.**Reschedule**
Reschedule this task

Raising this exception will stop the current execution of the task and ask the scheduler to reschedule this task, possibly on a different machine.

This does not guarantee that the task will move onto a different machine. The scheduler will proceed through its normal heuristics to determine the optimal machine to accept this task. The machine will likely change if the load across the cluster has significantly changed since first scheduling the task.

class distributed.**get_task_stream**(*client=None*, *plot=False*, *filename='task-stream.html'*)

Collect task stream within a context block

This provides diagnostic information about every task that was run during the time when this block was active.

This must be used as a context manager.

Parameters

plot: boolean, str If true then also return a Bokeh figure If plot == 'save' then save the figure to a file

filename: str (optional) The filename to save to if you set `plot='save'`

See also:

Client.get_task_stream Function version of this context manager

Examples

```
>>> with get_task_stream() as ts:
...     x.compute()
>>> ts.data
[...]
```

Get back a Bokeh figure and optionally save to a file

```
>>> with get_task_stream(plot='save', filename='task-stream.html') as ts:
...    x.compute()
>>> ts.figure
<Bokeh Figure>
```

To share this file with others you may wish to upload and serve it online. A common way to do this is to upload the file as a gist, and then serve it on https://raw.githack.com

```
$ python -m pip install gist
$ gist task-stream.html
https://gist.github.com/8a5b3c74b10b413f612bb5e250856ceb
```

You can then navigate to that site, click the "Raw" button to the right of the `task-stream.html` file, and then provide that URL to https://raw.githack.com . This process should provide a sharable link that others can use to see your task stream plot.

class distributed.**get_task_metadata**

Collect task metadata within a context block

This gathers `TaskState` metadata and final state from the scheduler for tasks which are submitted and finished within the scope of this context manager.

Examples

```
>>> with get_task_metadata() as tasks:
...     x.compute()
>>> tasks.metadata
{...}
>>> tasks.state
{...}
```

class distributed.**Event**(*name=None*, *client=None*)

Distributed Centralized Event equivalent to asyncio.Event

An event stores a single flag, which is set to false on start. The flag can be set to true (using the set() call) or back to false (with the clear() call). Every call to wait() blocks until the event flag is set to true.

Parameters

name: string (optional) Name of the event. Choosing the same name allows two disconnected processes to coordinate an event. If not given, a random name will be generated.

client: Client (optional) Client to use for communication with the scheduler. If not given, the default global client will be used.

Examples

```
>>> event_1 = Event('a')
>>> event_1.wait(timeout=1)
>>> # in another process
>>> event_2 = Event('a')
>>> event_2.set()
>>> # now event_1 will stop waiting
```

clear()

Clear the event (set its flag to false).

All waiters will now block.

is_set()

Check if the event is set

set()

Set the event (set its flag to false).

All waiters will now be released.

wait(*timeout=None*)

Wait until the event is set.

Parameters

timeout [number or string or timedelta, optional] Seconds to wait on the event in the scheduler. This does not include local coroutine time, network transfer time, etc.. Instead of number of seconds, it is also possible to specify a timedelta in string format, e.g. "200ms".

Returns

True if the event was set of false, if a timeout happend

Examples

```
>>> event = Event('a')
>>> event.wait(timeout="1s")
```

class distributed.**Lock**(*name=None*, *client=None*)
Distributed Centralized Lock

Parameters

name: string (optional) Name of the lock to acquire. Choosing the same name allows two disconnected processes to coordinate a lock. If not given, a random name will be generated.

client: Client (optional) Client to use for communication with the scheduler. If not given, the default global client will be used.

Examples

```
>>> lock = Lock('x')
>>> lock.acquire(timeout=1)
>>> # do things with protected resource
>>> lock.release()
```

acquire(*blocking=True*, *timeout=None*)
Acquire the lock

Parameters

blocking [bool, optional] If false, don't wait on the lock in the scheduler at all.

timeout [string or number or timedelta, optional] Seconds to wait on the lock in the scheduler. This does not include local coroutine time, network transfer time, etc.. It is forbidden to specify a timeout when blocking is false. Instead of number of seconds, it is also possible to specify a timedelta in string format, e.g. "200ms".

Returns

True or False whether or not it sucessfully acquired the lock

Examples

```
>>> lock = Lock('x')
>>> lock.acquire(timeout="1s")
```

release()
Release the lock if already acquired

class distributed.**MultiLock**(*names=[]*, *client=None*)
Distributed Centralized Lock

Parameters

names: List[str] Names of the locks to acquire. Choosing the same name allows two disconnected processes to coordinate a lock.

client: Client (optional) Client to use for communication with the scheduler. If not given, the default global client will be used.

Examples

```
>>> lock = MultiLock(['x', 'y'])
>>> lock.acquire(timeout=1)
>>> # do things with protected resource 'x' and 'y'
>>> lock.release()
```

acquire(*blocking=True*, *timeout=None*, *num_locks=None*)

Acquire the lock

Parameters

blocking [bool, optional] If false, don't wait on the lock in the scheduler at all.

timeout [string or number or timedelta, optional] Seconds to wait on the lock in the scheduler. This does not include local coroutine time, network transfer time, etc.. It is forbidden to specify a timeout when blocking is false. Instead of number of seconds, it is also possible to specify a timedelta in string format, e.g. "200ms".

num_locks [int, optional] Number of locks needed. If None, all locks are needed

Returns

True or False whether or not it successfully acquired the lock

Examples

```
>>> lock = MultiLock(['x', 'y'])
>>> lock.acquire(timeout="1s")
```

release()

Release the lock if already acquired

class distributed.**Semaphore**(*max_leases=1*, *name=None*, *register=True*, *scheduler_rpc=None*, *loop=None*)

This semaphore will track leases on the scheduler which can be acquired and released by an instance of this class. If the maximum amount of leases are already acquired, it is not possible to acquire more and the caller waits until another lease has been released.

The lifetime or leases are controlled using a timeout. This timeout is refreshed in regular intervals by the `Client` of this instance and provides protection from deadlocks or resource starvation in case of worker failure. The timeout can be controlled using the configuration option `distributed.scheduler.locks.lease-timeout` and the interval in which the scheduler verifies the timeout is set using the option `distributed.scheduler.locks.lease-validation-interval`.

A noticeable difference to the Semaphore of the python standard library is that this implementation does not allow to release more often than it was acquired. If this happens, a warning is emitted but the internal state is not modified.

Warning: This implementation is still in an experimental state and subtle changes in behavior may occur without any change in the major version of this library.

Warning: This implementation is susceptible to lease overbooking in case of lease timeouts. It is advised to monitor log information and adjust above configuration options to suitable values for the user application.

Parameters

max_leases: int (optional) The maximum amount of leases that may be granted at the same time. This effectively sets an upper limit to the amount of parallel access to a specific resource. Defaults to 1.

name: string (optional) Name of the semaphore to acquire. Choosing the same name allows two disconnected processes to coordinate. If not given, a random name will be generated.

register: bool If True, register the semaphore with the scheduler. This needs to be done before any leases can be acquired. If not done during initialization, this can also be done by calling the register method of this class. When registering, this needs to be awaited.

scheduler_rpc: ConnectionPool The ConnectionPool to connect to the scheduler. If None is provided, it uses the worker or client pool. This paramter is mostly used for testing.

loop: IOLoop The event loop this instance is using. If None is provided, reuse the loop of the active worker or client.

Notes

If a client attempts to release the semaphore but doesn't have a lease acquired, this will raise an exception.

When a semaphore is closed, if, for that closed semaphore, a client attempts to:

- Acquire a lease: an exception will be raised.
- Release: a warning will be logged.
- Close: nothing will happen.

dask executes functions by default assuming they are pure, when using semaphore acquire/releases inside such a function, it must be noted that there *are* in fact side-effects, thus, the function can no longer be considered pure. If this is not taken into account, this may lead to unexpected behavior.

Examples

```
>>> from distributed import Semaphore
... sem = Semaphore(max_leases=2, name='my_database')
...
... def access_resource(s, sem):
...     # This automatically acquires a lease from the semaphore (if available)␣
↪which will be
...     # released when leaving the context manager.
...     with sem:
...         pass
...
... futures = client.map(access_resource, range(10), sem=sem)
... client.gather(futures)
... # Once done, close the semaphore to clean up the state on scheduler side.
... sem.close()
```

acquire(*timeout=None*)

Acquire a semaphore.

If the internal counter is greater than zero, decrement it by one and return True immediately. If it is zero, wait until a release() is called and return True.

Parameters

timeout [number or string or timedelta, optional] Seconds to wait on acquiring the semaphore. This does not include local coroutine time, network transfer time, etc.. Instead of number of seconds, it is also possible to specify a timedelta in string format, e.g. "200ms".

get_value()
Return the number of currently registered leases.

release()
Release the semaphore.

Returns

bool This value indicates whether a lease was released immediately or not. Note that a user should *not* retry this operation. Under certain circumstances (e.g. scheduler overload) the lease may not be released immediately, but it will always be automatically released after a specific interval configured using "distributed.scheduler.locks.lease-validation-interval" and "distributed.scheduler.locks.lease-timeout".

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

Parameters

name: string (optional) Name used by other clients and the scheduler to identify the queue. If not given, a random name will be generated.

client: Client (optional) Client used for communication with the scheduler. If not given, the default global client will be used.

maxsize: int (optional) Number of items allowed in the queue. If 0 (the default), the queue size is unbounded.

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 or string or timedelta, optional] Time in seconds to wait before timing out. Instead of number of seconds, it is also possible to specify a timedelta in string format, e.g. "200ms".

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

Parameters

timeout [number or string or timedelta, optional] Time in seconds to wait before timing out. Instead of number of seconds, it is also possible to specify a timedelta in string format, e.g. "200ms".

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

Parameters

name: string (optional) Name used by other clients and the scheduler to identify the variable. If not given, a random name will be generated.

client: Client (optional) Client used for communication with the scheduler. If not given, the default global client will be used.

See also:

Queue shared multi-producer/multi-consumer queue between clients

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

Parameters

timeout [number or string or timedelta, optional] Time in seconds to wait before timing out. Instead of number of seconds, it is also possible to specify a timedelta in string format, e.g. "200ms".

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

Parameters

value [Future or object] Must be either a Future or a msgpack-encodable value

3.4.6 Adaptive

class distributed.deploy.**Adaptive**(*cluster=None*, *interval=None*, *minimum=None*, *maximum=None*, *wait_count=None*, *target_duration=None*, *worker_key=None*, ***kwargs*)
Adaptively allocate workers based on scheduler load. A superclass.

Contains logic to dynamically resize a Dask cluster based on current use. This class needs to be paired with a system that can create and destroy Dask workers using a cluster resource manager. Typically it is built into already existing solutions, rather than used directly by users. It is most commonly used from the `.adapt(...)` method of various Dask cluster classes.

Parameters

cluster: object Must have scale and scale_down methods/coroutines

interval [timedelta or str, default "1000 ms"] Milliseconds between checks

wait_count: int, default 3 Number of consecutive times that a worker should be suggested for removal before we remove it.

target_duration: timedelta or str, default "5s" Amount of time we want a computation to take. This affects how aggressively we scale up.

worker_key: Callable[WorkerState] Function to group workers together when scaling down See Scheduler.workers_to_close for more information

minimum: int Minimum number of workers to keep around

maximum: int Maximum number of workers to keep around

****kwargs:** Extra parameters to pass to Scheduler.workers_to_close

Notes

Subclasses can override *Adaptive.target()* and *Adaptive.workers_to_close()* to control when the cluster should be resized. The default implementation checks if there are too many tasks per worker or too little memory available (see `Scheduler.adaptive_target()`). The values for interval, min, max, wait_count and target_duration can be specified in the dask config under the distributed.adaptive key.

Examples

This is commonly used from existing Dask classes, like KubeCluster

```
>>> from dask_kubernetes import KubeCluster
>>> cluster = KubeCluster()
>>> cluster.adapt(minimum=10, maximum=100)
```

Alternatively you can use it from your own Cluster class by subclassing from Dask's Cluster superclass

```
>>> from distributed.deploy import Cluster
>>> class MyCluster(Cluster):
...     def scale_up(self, n):
...         """ Bring worker count up to n """
...     def scale_down(self, workers):
...         """ Remove worker addresses from cluster """
```

```
>>> cluster = MyCluster()
>>> cluster.adapt(minimum=10, maximum=100)
```

async recommendations(*target: int*) → dict
Make scale up/down recommendations based on current state and target

async target()
Determine target number of workers that should exist.

Returns

Target number of workers

See also:

Scheduler.adaptive_target

Notes

`Adaptive.target` dispatches to Scheduler.adaptive_target(), but may be overridden in subclasses.

async workers_to_close(*target: int*)
Determine which, if any, workers should potentially be removed from the cluster.

Returns

List of worker addresses to close, if any

See also:

Scheduler.workers_to_close

Notes

`Adaptive.workers_to_close` dispatches to Scheduler.workers_to_close(), but may be overridden in subclasses.

3.5 Examples

3.5.1 Word count in HDFS

Setup

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

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

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

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

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

Code example

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

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

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

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

Initialize a connection to the `distributed` client, replacing `SCHEDULER_IP` and `SCHEDULER_PORT` with the IP address and port of the `distributed` scheduler.

```
>>> client = Client('SCHEDULER_IP:SCHEDULER_PORT')
```

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

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

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

Print the first 1024 bytes of the first text file:

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

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

Create a function to count words in each file:

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

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

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

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

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

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

[(b'the', 144873),
 (b'of', 98122),
 (b'to', 97202),
 (b'and', 90575),
 (b'or', 60305),
```

(continues on next page)

(continued from previous page)

```
 (b'in', 53869),
 (b'a', 43300),
 (b'any', 31632),
 (b'by', 31515),
 (b'is', 30055)]
```

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

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

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

```
>>> len(futures)

161

>>> futures[:5]

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

>>> progress(futures)

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

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

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

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

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

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

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

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

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

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

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

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

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

8797842

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

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

The complete Python script for this example is shown below:

```
# word-count.py

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

hdfs = hdfs3.HDFileSystem('NAMENODE_HOSTNAME', port=NAMENODE_PORT)
client = Client('SCHEDULER_IP:SCHEDULER:PORT')

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


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

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

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

futures = client.map(count_words, filenames)
len(futures)
```

(continues on next page)

(continued from previous page)

```
futures[:5]
progress(futures)


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

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

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

print(len(all_counts))

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

3.6 Frequently Asked Questions

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

3.6.1 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.

3.6.2 Too many open file descriptors?

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

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

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

3.6.3 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

3.6.4 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.

3.6.5 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.

3.6.6 KilledWorker, CommsClosed, etc.

In the case that workers disappear unexpectedly from your cluster, you may see a range of error messages. After checking the logs of the workers affected, you should read the section *Why did my worker die?*.

3.7 Diagnosing Performance

Understanding the performance of a distributed computation can be difficult. This is due in part to the many components of a distributed computer that may impact performance:

1. Compute time
2. Memory bandwidth
3. Network bandwidth
4. Disk bandwidth
5. Scheduler overhead

6. Serialization costs

This difficulty is compounded because the information about these costs is spread among many machines and so there is no central place to collect data to identify performance issues.

Fortunately, Dask collects a variety of diagnostic information during execution. It does this both to provide performance feedback to users, but also for its own internal scheduling decisions. The primary place to observe this feedback is the diagnostic dashboard. This document describes the various pieces of performance information available and how to access them.

3.7.1 Task start and stop times

Workers capture durations associated to tasks. For each task that passes through a worker we record start and stop times for each of the following:

1. Serialization (gray)
2. Dependency gathering from peers (red)
3. Disk I/O to collect local data (orange)
4. Execution times (colored by task)

The main way to observe these times is with the task stream plot on the scheduler's `/status` page where the colors of the bars correspond to the colors listed above.

Alternatively if you want to do your own diagnostics on every task event you might want to create a *Scheduler plugin*. All of this information will be available when a task transitions from processing to memory or erred.

3.7.2 Statistical Profiling

For single-threaded profiling Python users typically depend on the CProfile module in the standard library (Dask developers recommend the snakeviz tool for single-threaded profiling). Unfortunately the standard CProfile module does not work with multi-threaded or distributed computations.

To address this Dask implements its own distributed statistical profiler. Every 10ms each worker process checks what each of its worker threads are doing. It captures the call stack and adds this stack to a counting data structure. This counting data structure is recorded and cleared every second in order to establish a record of performance over time.

Users typically observe this data through the `/profile` plot on either the worker or scheduler diagnostic dashboards. On the scheduler page they observe the total profile aggregated over all workers over all threads. Clicking on any of the bars in the profile will zoom the user into just that section, as is typical with most profiling tools. There is a timeline at the bottom of the page to allow users to select different periods in time.

Profiles are also grouped by the task that was being run at the time. You can select a task name from the selection menu at the top of the page. You can also click on the rectangle corresponding to the task in the main task stream plot on the `/status` page.

Users can also query this data directly using the *Client.profile* function. This will deliver the raw data structure used to produce these plots. They can also pass a filename to save the plot as an HTML file directly. Note that this file will have to be served from a webserver like `python -m http.server` to be visible.

The 10ms and 1s parameters can be controlled by the `profile-interval` and `profile-cycle-interval` entries in the config.yaml file.

3.7.3 Bandwidth

Dask workers track every incoming and outgoing transfer in the `Worker.outgoing_transfer_log` and `Worker.incoming_transfer_log` attributes including

1. Total bytes transferred
2. Compressed bytes transferred
3. Start/stop times
4. Keys moved
5. Peer

These are made available to users through the `/status` page of the Worker's diagnostic dashboard. You can capture their state explicitly by running a command on the workers:

```
client.run(lambda dask_worker: dask_worker.outgoing_transfer_log)
client.run(lambda dask_worker: dask_worker.incoming_transfer_log)
```

3.7.4 Performance Reports

Often when benchmarking and/or profiling, users may want to record a particular computation or even a full workflow. Dask can save the bokeh dashboards as static HTML plots including the task stream, worker profiles, bandwidths, etc. This is done wrapping a computation with the `performance_report` context manager:

```
from dask.distributed import performance_report

with performance_report(filename="dask-report.html"):
    ## some dask computation
```

The following video demonstrates the `performance_report` context manager in greater detail:

3.7.5 A note about times

Different computers maintain different clocks which may not match perfectly. To address this the Dask scheduler sends its current time in response to every worker heartbeat. Workers compare their local time against this time to obtain an estimate of differences. All times recorded in workers take this estimated delay into account. This helps, but still, imprecise measurements may exist.

All times are intended to be from the scheduler's perspective.

3.8 Efficiency

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

3.8.1 Leave data on the cluster

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

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

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

```
>>> x = 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)
```

3.8.2 Use larger tasks

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

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

Slow

```
>>> futures = 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 tlz 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
```

3.8.3 Adjust between Threads and Processes

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

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

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

3.8.4 Don't go distributed

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

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

3.9 Limitations

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

3.9.1 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.

3.9.2 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.

3.9.3 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.

3.10 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.

3.10.1 Current Policies

Task Submission

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

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

Data Scatter

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

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

Then Alice and Bob receive the following data

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

3.10.2 User Control

Complex algorithms may require more user control.

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

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

```
future = 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 all the 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')
```

3.10.3 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 `.__dask_graph__()` method):

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

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

3.11 Logging

There are several ways in which state and other activities are logged throughout a Dask cluster.

3.11.1 Logs

The scheduler, workers, and client all log various administrative events using Python's standard logging module. Both the logging level and logging handlers are customizable. See the Debugging docs for more information.

3.11.2 Task transition logs

The scheduler keeps track of all *state transitions* for each task. This gives insight into how tasks progressed through their computation and can be particularly valuable when debugging. To retrieve the transition logs for a given task, pass the task's key to the `Scheduler.story()` method.

```
>>> f = client.submit(inc, 123)
>>> f
<Future: finished, type: builtins.int, key: inc-aad7bbea25dc61c8e53d929c7ec50bed>
>>> s.story(f.key)
[('inc-aad7bbea25dc61c8e53d929c7ec50bed', 'released', 'waiting', {'inc-
↪aad7bbea25dc61c8e53d929c7ec50bed': 'processing'}, 1605143345.7283862),
 ('inc-aad7bbea25dc61c8e53d929c7ec50bed', 'waiting', 'processing', {}, 1605143345.
↪7284858),
 ('inc-aad7bbea25dc61c8e53d929c7ec50bed', 'processing', 'memory', {}, 1605143345.
↪731495)]
```

3.11.3 Structured logs

The scheduler, workers, and client all support logging structured events to a centralized ledger, which is indexed by topic. By default, Dask will log a few administrative events to this system (e.g. when workers enter and leave the cluster) but custom events can be logged using the `Scheduler.log_event()`, `Worker.log_event()`, or *`Client.log_event()`* methods.

For example, below we log start and stop times to the `"runtimes"` topic using the worker's `log_event` method:

```
>>> def myfunc(x):
...     start = time()
...     ...
...     stop = time()
...     dask.distributed.get_worker().log_event("runtimes", {"start": start, "stop":␣
↪stop})
>>> futures = client.map(myfunc, range(10))
>>> client.get_events("runtimes")
((1605207481.77175, {'start': 1605207481.769397, 'stop': 1605207481.769397}),
 (1605207481.772021, {'start': 1605207481.770036, 'stop': 1605207481.770037}),
 ...
)
```

Events for a given topic can be retrieved using the *`Client.get_events()`* method. In the above example, we retrieved the logged start and stop times with `client.get_events("runtimes")`. Note that `Client.get_events` returns a tuple for each logged event which contains the logged message along with a timestamp for when the event was logged.

When combined with scheduler and worker plugins, the structured events system can produce rich logging / diagnostic systems.

3.12 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.

3.12.1 Dask Collections to Concrete Values

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

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

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

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

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

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

3.12.2 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.

3.12.3 Concrete Values to Futures

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

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

3.12.4 Futures to Concrete Values

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

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

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

3.12.5 Futures to Dask Collections

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

```
>>> x = delayed(sum)(futures)
>>> y = delayed(product)(futures)
>>> future = 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

3.13 Managing Memory

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

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

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

3.13.1 Creating Futures

The following functions produce Futures

`Client.submit`(func, *args[, key, workers, ...])	Submit a function application to the scheduler
`Client.map`(func, *iterables[, key, workers, ...])	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.

3.13.2 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.

3.13.3 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
```

3.13.4 Clearing data

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

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

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

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

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

3.13.5 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()
```

3.13.6 Resilience

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

For more information see *Resilience*.

3.13.7 Advanced techniques

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

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

`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

3.14 Prioritizing Work

When there is more work than workers, Dask has to decide which tasks to prioritize over others. Dask can determine these priorities automatically to optimize performance, or a user can specify priorities manually according to their needs.

Dask uses the following priorities, in order:

1. **User priorities**: A user defined priority is provided by the `priority=` keyword argument to functions like `compute()`, `persist()`, `submit()`, or `map()`. Tasks with higher priorities run before tasks with lower priorities with the default priority being zero.

   ```
   future = client.submit(func, *args, priority=10)  # high priority task
   future = client.submit(func, *args, priority=-10)  # low priority task

   df = df.persist(priority=10)  # high priority computation
   ```

   Priorities can also be specified using the dask annotations machinery:

   ```
   with dask.annotate(priority=10):
       future = client.submit(func, *args)  # high priority task
   with dask.annotate(priority=-10):
       future = client.submit(func, *args)  # low priority task

   with dask.annotate(priority=10):
       df = df.persist()  # high priority computation
   ```

2. **First in first out chronologically**: Dask prefers computations that were submitted early. Because users can submit computations asynchronously it may be that several different computations are running on the workers at the same time. Generally Dask prefers those groups of tasks that were submitted first.

   As a nuance, tasks that are submitted within a close window are often considered to be submitted at the same time.

   ```
   x = x.persist()  # submitted first and so has higher priority
   # wait a while
   x = x.persist()  # submitted second and so has lower priority
   ```

In this case "a while" depends on the kind of computation. Operations that are often used in bulk processing, like `compute` and `persist` consider any two computations submitted in the same sixty seconds to have the same priority. Operations that are often used in real-time processing, like `submit` or `map` are considered the same priority if they are submitted within the 100 milliseconds of each other. This behavior can be controlled with the `fifo_timeout=` keyword:

```
x = x.persist()
# wait one minute
x = x.persist(fifo_timeout='10 minutes')  # has the same priority

a = client.submit(func, *args)
# wait no time at all
b = client.submit(func, *args, fifo_timeout='0ms')  # is lower priority
```

3. **Graph Structure**: Within any given computation (a compute or persist call) Dask orders tasks in such a way as to minimize the memory-footprint of the computation. This is discussed in more depth in the task ordering documentation.

If multiple tasks each have exactly the same priorities outlined above, then the order in which tasks arrive at a worker, in a last in first out manner, is used to determine the order in which tasks run.

3.15 Related Work

Writing the "related work" for a project called "distributed", is a Sisyphean task. We'll list a few notable projects that you've probably already heard of down below.

You may also find the dask comparison with spark of interest.

3.15.1 Big Data World

- The venerable Hadoop provides batch processing with the MapReduce programming paradigm. Python users typically use Hadoop Streaming or MRJob.
- Spark builds on top of HDFS systems with a nicer API and in-memory processing. Python users typically use PySpark.
- Storm provides streaming computation. Python users typically use streamparse.

This is a woefully inadequate representation of the excellent work blossoming in this space. A variety of projects have come into this space and rival or complement the projects above. Still, most "Big Data" processing hype probably centers around the three projects above, or their derivatives.

3.15.2 Python Projects

There are dozens of Python projects for distributed computing. Here we list a few of the more prominent projects that we see in active use today.

Task scheduling

- Celery: An asynchronous task scheduler, focusing on real-time processing.
- Luigi: A bulk big-data/batch task scheduler, with hooks to a variety of interesting data sources.

Ad hoc computation

- IPython Parallel: Allows for stateful remote control of several running ipython sessions.
- Scoop: Implements the concurrent.futures API on distributed workers. Notably allows tasks to spawn more tasks.

Direct Communication

- MPI4Py: Wraps the Message Passing Interface popular in high performance computing.
- PyZMQ: Wraps ZeroMQ, the high-performance asynchronous messaging library.

Venerable

There are a couple of older projects that often get mentioned

- Dispy: Embarrassingly parallel function evaluation
- Pyro: Remote objects / RPC

3.15.3 Relationship

In relation to these projects `distributed`...

- Supports data-local computation like Hadoop and Spark
- Uses a task graph with data dependencies abstraction like Luigi
- In support of ad-hoc applications, like IPython Parallel and Scoop

3.15.4 In depth comparison to particular projects

IPython Parallel

Short Description

IPython Parallel is a distributed computing framework from the IPython project. It uses a centralized hub to farm out jobs to several `ipengine` processes running on remote workers. It communicates over ZeroMQ sockets and centralizes communication through the central hub.

IPython parallel has been around for a while and, while not particularly fancy, is quite stable and robust.

IPython Parallel offers parallel `map` and remote `apply` functions that route computations to remote workers

```
>>> view = Client(...)[:]
>>> results = view.map(func, sequence)
>>> result = view.apply(func, *args, **kwargs)
>>> future = view.apply_async(func, *args, **kwargs)
```

It also provides direct execution of code in the remote process and collection of data from the remote namespace.

```
>>> view.execute('x = 1 + 2')
>>> view['x']
[3, 3, 3, 3, 3, 3]
```

Brief Comparison

Distributed and IPython Parallel are similar in that they provide `map` and `apply/submit` abstractions over distributed worker processes running Python. Both manage the remote namespaces of those worker processes.

They are dissimilar in terms of their maturity, how worker nodes communicate to each other, and in the complexity of algorithms that they enable.

Distributed Advantages

The primary advantages of `distributed` over IPython Parallel include

1. Peer-to-peer communication between workers
2. Dynamic task scheduling

`Distributed` workers share data in a peer-to-peer fashion, without having to send intermediate results through a central bottleneck. This allows `distributed` to be more effective for more complex algorithms and to manage larger datasets in a more natural manner. IPython parallel does not provide a mechanism for workers to communicate with each other, except by using the central node as an intermediary for data transfer or by relying on some other medium, like a shared file system. Data transfer through the central node can easily become a bottleneck and so IPython parallel has been mostly helpful in embarrassingly parallel work (the bulk of applications) but has not been used extensively for more sophisticated algorithms that require non-trivial communication patterns.

The distributed 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 3148**. 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.

3.16 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.

3.16.1 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.

3.16.2 Closed Network Connections

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

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

This has some fail cases.

1. If results depend on impure functions then you may get a different (although still entirely accurate) result

2. If the worker failed due to a bad function, for example a function that causes a segmentation fault, then that bad function will repeatedly be called on other workers. This function will be marked as "bad" after it kills a fixed number of workers (defaults to three).
3. Data sent out directly to the workers via a call to `scatter()` (instead of being created from a Dask task graph via other Dask functions) is not kept in the scheduler, as 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.

3.16.3 Hardware Failures

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

3.16.4 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.

3.16.5 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. This requires the nanny process (which is started by default).

3.17 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*.

3.17.1 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(ts, all_workers, ...)	Decide which worker should take task *ts*.

3.17.2 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.

3.18 Scheduling State

3.18.1 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 in turn.
3. The worker receives information about how to run the task, communicates with its peer workers to collect data dependencies, and then runs the relevant function on the appropriate data. It reports back to the scheduler that it has finished, keeping the result stored in the worker where it was computed.
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*.

The scheduler keeps internal state about several kinds of entities:

- Individual tasks known to the scheduler
- Workers connected to the scheduler
- Clients connected to the scheduler

Note: Everything listed in this page is an internal detail of how Dask operates. It may change between versions and you should probably avoid relying on it in user code (including on any APIs explained here).

3.18.2 Task State

Internally, the scheduler moves tasks between a fixed set of states, notably `released`, `waiting`, `no-worker`, `processing`, `memory`, `error`.

Tasks flow along 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 to be computed, but no appropriate worker exists (for example because of resource restrictions, or because no worker is connected at all).
- *Processing*: Actively being computed by one or more workers
- *Memory*: In memory on one or more workers
- *Erred*: Task computation, or one of its dependencies, has encountered an error
- *Forgotten* (not actually a state): Task is no longer needed by any client or dependent task

In addition to the literal state, though, other information needs to be kept and updated about each task. Individual task state is stored in an object named *TaskState* and consists of the following information:

class distributed.scheduler.**TaskState**(*key: str*, *run_spec: object*)
A simple object holding information about a task.

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

prefix: TaskPrefix
The broad class of tasks to which this task belongs like "inc" or "read_csv"

run_spec: object
A specification of how to run the task. The type and meaning of this value is opaque to the scheduler, as it is only interpreted by the worker to which the task is sent for executing.

As a special case, this attribute may also be `None`, in which case the task is "pure data" (such as, for example, a piece of data loaded in the scheduler using `Client.scatter()`). A "pure data" task cannot be computed again if its value is lost.

priority: tuple
The priority provides each task with a relative ranking which is used to break ties when many tasks are being considered for execution.

This ranking is generally a 2-item tuple. The first (and dominant) item corresponds to when it was submitted. Generally, earlier tasks take precedence. The second item 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*.

state: str
This task's current state. Valid states include `released`, `waiting`, `no-worker`, `processing`, `memory`, `erred` and `forgotten`. If it is `forgotten`, the task isn't stored in the `tasks` dictionary anymore and will probably disappear soon from memory.

dependencies: {TaskState}
The set of tasks this task depends on for proper execution. Only tasks still alive are listed in this set. If, for whatever reason, this task also depends on a forgotten task, the `has_lost_dependencies` flag is set.

A task can only be executed once all its dependencies have already been successfully executed and have their result stored on at least one worker. This is tracked by progressively draining the `waiting_on` set.

dependents: {TaskState}
The set of tasks which depend on this task. Only tasks still alive are listed in this set.

This is the reverse mapping of `dependencies`.

has_lost_dependencies: bool
Whether any of the dependencies of this task has been forgotten. For memory consumption reasons, forgotten tasks are not kept in memory even though they may have dependent tasks. When a task is forgotten, therefore, each of its dependents has their `has_lost_dependencies` attribute set to `True`.

If `has_lost_dependencies` is true, this task cannot go into the "processing" state anymore.

waiting_on: {TaskState}
The set of tasks this task is waiting on *before* it can be executed. This is always a subset of `dependencies`. Each time one of the dependencies has finished processing, it is removed from the `waiting_on` set.

Once `waiting_on` becomes empty, this task can move from the "waiting" state to the "processing" state (unless one of the dependencies errored out, in which case this task is instead marked "erred").

waiters: {TaskState}
The set of tasks which need this task to remain alive. This is always a subset of `dependents`. Each time one of the dependents has finished processing, it is removed from the `waiters` set.

Once both `waiters` and `who_wants` become empty, this task can be released (if it has a non-empty `run_spec`) or forgotten (otherwise) by the scheduler, and by any workers in `who_has`.

Note: Counter-intuitively, `waiting_on` and `waiters` are not reverse mappings of each other.

---

who_wants: {ClientState}
The set of clients who want this task's result to remain alive. This is the reverse mapping of `ClientState.wants_what`.

When a client submits a graph to the scheduler it also specifies which output tasks it desires, such that their results are not released from memory.

Once a task has finished executing (i.e. moves into the "memory" or "erred" state), the clients in `who_wants` are notified.

Once both `waiters` and `who_wants` become empty, this task can be released (if it has a non-empty `run_spec`) or forgotten (otherwise) by the scheduler, and by any workers in `who_has`.

who_has: {WorkerState}
The set of workers who have this task's result in memory. It is non-empty iff the task is in the "memory" state. There can be more than one worker in this set if, for example, `Client.scatter()` or `Client.replicate()` was used.

This is the reverse mapping of `WorkerState.has_what`.

processing_on: WorkerState (or None)
If this task is in the "processing" state, which worker is currently processing it. Otherwise this is `None`.

This attribute is kept in sync with `WorkerState.processing`.

retries: int
The number of times this task can automatically be retried in case of failure. If a task fails executing (the worker returns with an error), its `retries` attribute is checked. If it is equal to 0, the task is marked "erred". If it is greater than 0, the `retries` attribute is decremented and execution is attempted again.

nbytes: int (or None)
The number of bytes, as determined by `sizeof`, of the result of a finished task. This number is used for diagnostics and to help prioritize work.

type: str
The type of the object as a string. Only present for tasks that have been computed.

exception: object
If this task failed executing, the exception object is stored here. Otherwise this is `None`.

traceback: object
If this task failed executing, the traceback object is stored here. Otherwise this is `None`.

exception_blame: TaskState (or None)
If this task or one of its dependencies failed executing, the failed task is stored here (possibly itself). Otherwise this is `None`.

suspicious: int
The number of times this task has been involved in a worker death.

Some tasks may cause workers to die (such as calling `os._exit(0)`). When a worker dies, 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 dies, we mark each task currently processing on that worker (as recorded by `WorkerState.processing`) as suspicious.

If a task is involved in three deaths (or some other fixed constant) then we mark the task as `erred`.

host_restrictions: {hostnames}
A set of hostnames where this task can be run (or `None` if empty). Usually this is empty unless the task has been specifically restricted to only run on certain hosts. A hostname may correspond to one or several connected workers.

worker_restrictions: {worker addresses}
A set of complete worker addresses where this can be run (or `None` if empty). Usually this is empty unless the task has been specifically restricted to only run on certain workers.

Note this is tracking worker addresses, not worker states, since the specific workers may not be connected at this time.

resource_restrictions: {resource: quantity}
Resources required by this task, such as `{'gpu': 1}` or `{'memory': 1e9}` (or `None` if empty).

These are user-defined names and are matched against the contents of each `WorkerState.resources` dictionary.

loose_restrictions: bool
If `False`, each of `host_restrictions`, `worker_restrictions` and `resource_restrictions` is a hard constraint: if no worker is available satisfying those restrictions, the task cannot go into the "processing" state and will instead go into the "no-worker" state.

If `True`, the above restrictions are mere preferences: if no worker is available satisfying those restrictions, the task can still go into the "processing" state and be sent for execution to another connected worker.

metadata: dict
Metadata related to task.

actor: bool
Whether or not this task is an Actor.

group: TaskGroup
The group of tasks to which this one belongs.

annotations: dict
Task annotations

The scheduler keeps track of all the *TaskState* objects (those not in the "forgotten" state) using several containers:

tasks: {str: TaskState}
A dictionary mapping task keys (usually strings) to *TaskState* objects. Task keys are how information about tasks is communicated between the scheduler and clients, or the scheduler and workers; this dictionary is then used to find the corresponding *TaskState* object.

unrunnable: {TaskState}
A set of *TaskState* objects in the "no-worker" state. These tasks already have all their `dependencies` satisfied (their `waiting_on` set is empty), and are waiting for an appropriate worker to join the network before computing.

3.18.3 Worker State

Each worker's current state is stored in a *WorkerState* object. This information is involved in deciding *which worker to run a task on*.

class distributed.scheduler.**WorkerState**(*address: Optional[str] = None*, *pid: ctypes.c_long = 0*, *name: Optional[object] = None*, *nthreads: ctypes.c_long = 0*, *memory_limit: ctypes.c_long = 0*, *local_directory: Optional[str] = None*, *services: Optional[dict] = None*, *versions: Optional[dict] = None*, *nanny: Optional[str] = None*, *extra: Optional[dict] = None*)
A simple object holding information about a worker.

address: str
This worker's unique key. This can be its connected address (such as `'tcp://127.0.0.1:8891'`) or an alias (such as `'alice'`).

processing: {TaskState: cost}
A dictionary of tasks that have been submitted to this worker. Each task state is asssociated with the expected cost in seconds of running that task, summing both the task's expected computation time and the expected communication time of its result.

Multiple tasks may be submitted to a worker in advance and the worker will run them eventually, depending on its execution resources (but see *Work Stealing*).

All the tasks here are in the "processing" state.

This attribute is kept in sync with `TaskState.processing_on`.

executing: {TaskState: duration}
A dictionary of tasks that are currently being run on this worker. Each task state is asssociated with the duration in seconds which the task has been running.

has_what: {TaskState}
The set of tasks which currently reside on this worker. All the tasks here are in the "memory" state.

This is the reverse mapping of `TaskState.who_has`.

nbytes: int
The total memory size, in bytes, used by the tasks this worker holds in memory (i.e. the tasks in this worker's `has_what`).

nthreads: int
The number of CPU threads made available on this worker.

resources: {str: Number}
The available resources on this worker like `{'gpu': 2}`. These are abstract quantities that constrain certain tasks from running at the same time on this worker.

used_resources: {str: Number}
The sum of each resource used by all tasks allocated to this worker. The numbers in this dictionary can only be less or equal than those in this worker's `resources`.

occupancy: double
The total expected runtime, in seconds, of all tasks currently processing on this worker. This is the sum of all the costs in this worker's `processing` dictionary.

status: str
The current status of the worker, either `'running'` or `'closed'`

nanny: str
Address of the associated Nanny, if present

last_seen: Py_ssize_t
The last time we received a heartbeat from this worker, in local scheduler time.

actors: {TaskState}
A set of all TaskStates on this worker that are actors. This only includes those actors whose state actually lives on this worker, not actors to which this worker has a reference.

In addition to individual worker state, the scheduler maintains two containers to help with scheduling tasks:

Scheduler.saturated: {WorkerState}
A set of workers whose computing power (as measured by `WorkerState.nthreads`) is fully exploited by processing tasks, and whose current `occupancy` is a lot greater than the average.

Scheduler.idle: {WorkerState}
A set of workers whose computing power is not fully exploited. These workers are assumed to be able to start computing new tasks immediately.

These two sets are disjoint. Also, some workers may be *neither* "idle" nor "saturated". "Idle" workers will be preferred when *deciding a suitable worker* to run a new task on. Conversely, "saturated" workers may see their workload lightened through *Work Stealing*.

3.18.4 Client State

Information about each individual client of the scheduler is kept in a `ClientState` object:

class distributed.scheduler.**ClientState**(*client: str*, *versions: Optional[dict] = None*)
A simple object holding information about a client.

client_key: str
A unique identifier for this client. This is generally an opaque string generated by the client itself.

wants_what: {TaskState}
A set of tasks this client wants kept in memory, so that it can download its result when desired. This is the reverse mapping of `TaskState.who_wants`.

Tasks are typically removed from this set when the corresponding object in the client's space (for example a `Future` or a Dask collection) gets garbage-collected.

3.18.5 Understanding a Task's Flow

As seen above, there are numerous pieces of information pertaining to task and worker state, and some of them can be computed, updated or removed during a task's transitions.

The table below shows which state variable a task is in, depending on the task's state. Cells with a check mark (✓) indicate the task key *must* be present in the given state variable; cells with an question mark (*?*) indicate the task key *may* be present in the given state variable.

State variable	Released	Waiting	No-worker	Processing	Memory	Erred
`TaskState.dependencies`	✓	✓	✓	✓	✓	✓
`TaskState.dependents`	✓	✓	✓	✓	✓	✓
`TaskState.host_restrictions`	?	?	?	?	?	?
`TaskState.worker_restrictions`	?	?	?	?	?	?
`TaskState.resource_restrictions`	?	?	?	?	?	?
`TaskState.loose_restrictions`	?	?	?	?	?	?
`TaskState.waiting_on`		✓		✓		
`TaskState.waiters`		✓		✓		
`TaskState.processing_on`				✓		
`WorkerState.processing`				✓		
`TaskState.who_has`					✓	
`WorkerState.has_what`					✓	
`TaskState.nbytes` *(1)*	?	?	?	?	✓	?
`TaskState.exception` *(2)*						?
`TaskState.traceback` *(2)*						?
`TaskState.exception_blame`						✓
`TaskState.retries`	?	?	?	?	?	?
`TaskState.suspicious_tasks`	?	?	?	?	?	?

Notes:

1. `TaskState.nbytes`: this attribute can be known as long as a task has already been computed, even if it has been later released.
2. `TaskState.exception` and `TaskState.traceback` should be looked up on the `TaskState.exception_blame` task.

The table below shows which worker state variables are updated on each task state transition.

Transition	Affected worker state
released → waiting	occupancy, idle, saturated
waiting → processing	occupancy, idle, saturated, used_resources
waiting → memory	idle, saturated, nbytes
processing → memory	occupancy, idle, saturated, used_resources, nbytes
processing → erred	occupancy, idle, saturated, used_resources
processing → released	occupancy, idle, saturated, used_resources
memory → released	nbytes
memory → forgotten	nbytes

Note: Another way of understanding this table is to observe that entering or exiting a specific task state updates a well-defined set of worker state variables. For example, entering and exiting the "memory" state updates `WorkerState.nbytes`.

3.18.6 Implementation

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()
```

(continues on next page)

(continued from previous page)

```
        new = self.transition(key, finish)
        recommendations.update(new)
```

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

3.18.7 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.

3.18.8 API

class distributed.scheduler.**Scheduler**(*loop=None*, *delete_interval='500ms'*, *synchronize_worker_interval='60s'*, *services=None*, *service_kwargs=None*, *allowed_failures=None*, *extensions=None*, *validate=None*, *scheduler_file=None*, *security=None*, *worker_ttl=None*, *idle_timeout=None*, *interface=None*, *host=None*, *port=0*, *protocol=None*, *dashboard_address=None*, *dashboard=None*, *http_prefix='/'*, *preload=None*, *preload_argv=()*, *plugins=()*, ***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: `{task key: TaskState}`** Tasks currently known to the scheduler
- **unrunnable: `{TaskState}`** Tasks in the "no-worker" state
- **workers: `{worker key: WorkerState}`** Workers currently connected to the scheduler
- **idle: `{WorkerState}`:** Set of workers that are not fully utilized
- **saturated: `{WorkerState}`:** Set of workers that are not over-utilized
- **host_info: `{hostname: dict}`:** Information about each worker host
- **clients: `{client key: ClientState}`** Clients currently connected to the scheduler
- **services: `{str: port}`:** Other services running on this scheduler, like Bokeh
- **loop: `IOLoop`:** The running Tornado IOLoop
- **client_comms: `{client key: Comm}`** For each client, a Comm object used to receive task requests and report task status updates.
- **stream_comms: `{worker key: Comm}`** For each worker, a Comm object 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}`

`adaptive_target`(*comm=None*, *target_duration=None*)

Desired number of workers based on the current workload

This looks at the current running tasks and memory use, and returns a number of desired workers. This is often used by adaptive scheduling.

Parameters

target_duration [str] A desired duration of time for computations to take. This affects how rapidly the scheduler will ask to scale.

See also:

distributed.deploy.Adaptive

async add_client(*comm*, *client=None*, *versions=None*)
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. However, it is sent by workers from time to time.

add_plugin(*plugin=None*, *idempotent=False*, ***kwargs*)
Add external plugin to scheduler

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

async add_worker(*comm=None*, *address=None*, *keys=()*, *nthreads=None*, *name=None*, *resolve_address=True*, *nbytes=None*, *types=None*, *now=None*, *resources=None*, *host_info=None*, *memory_limit=None*, *metrics=None*, *pid=0*, *services=None*, *local_directory=None*, *versions=None*, *nanny=None*, *extra=None*)
Add a new worker to the cluster

async broadcast(*comm=None*, *msg=None*, *workers=None*, *hosts=None*, *nanny=False*, *serializers=None*)
Broadcast message to workers, return all results

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

client_heartbeat(*client=None*)
Handle heartbeats from Client

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

client_send(*client*, *msg*)
Send message to client

async close(*comm=None*, *fast=False*, *close_workers=False*)
Send cleanup signal to all coroutines then wait until finished

See also:

Scheduler.cleanup

async close_worker(*comm=None*, *worker=None*, *safe=None*)
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.

async feed(*comm*, *function=None*, *setup=None*, *teardown=None*, *interval='1s'*, ***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.

async gather(*comm=None*, *keys=None*, *serializers=None*)
Collect data in from workers

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

Parameters

worker [address]

service_name [str] Common services include 'bokeh' and 'nanny'

protocol [boolean] Whether or not to include a full address with protocol (True) or just a (host, port) pair

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.

async handle_worker(*comm=None*, *worker=None*)
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

async proxy(*comm=None*, *msg=None*, *worker=None*, *serializers=None*)
Proxy a communication through the scheduler to some other worker

async rebalance(*comm=None*, *keys=None*, *workers=None*)
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(*worker_index: ctypes.c_long = 0*)
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.

async register_worker_plugin(*comm*, *plugin*, *name=None*)
Registers a setup function, and call it on every worker

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

remove_plugin(*plugin*)
Remove external plugin from scheduler

async 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.

async replicate(*comm=None*, *keys=None*, *n=None*, *workers=None*, *branching_factor=2*, *delete=True*, *lock=True*)
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. The larger the branching factor, the more data we copy in a single step, but the more a given worker risks being swamped by data requests.

See also:

Scheduler.rebalance

report(*msg: dict*, *ts: Optional[*distributed.scheduler.TaskState*] = None*, *client: Optional[str] = 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.

reschedule(*key=None*, *worker=None*)
Reschedule a task

Things may have shifted and this task may now be better suited to run elsewhere

async restart(*client=None*, *timeout=3*)
Restart all workers. Reset local state.

async retire_workers(*comm=None*, *workers=None*, *remove=True*, *close_workers=False*, *names=None*, *lock=True*, ***kwargs*) → dict
Gracefully retire workers from cluster

Parameters

workers: list (optional) List of worker addresses to retire. If not provided we call `workers_to_close` which finds a good set

names: list (optional) List of worker names to retire.

remove: bool (defaults to True) Whether or not to remove the worker metadata immediately or else wait for the worker to contact us

close_workers: bool (defaults to False) Whether or not to actually close the worker explicitly from here. Otherwise we expect some external job scheduler to finish off the worker.

****kwargs: dict** Extra options to pass to workers_to_close to determine which workers we should drop

Returns

Dictionary mapping worker ID/address to dictionary of information about

that worker for each retired worker.

See also:

Scheduler.workers_to_close

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

See also:

Client.run_on_scheduler

async scatter(*comm=None*, *data=None*, *workers=None*, *client=None*, *broadcast=False*, *timeout=2*)
Send data out to workers

See also:

Scheduler.broadcast

send_all(*client_msgs: dict*, *worker_msgs: dict*)
Send messages to client and workers

send_task_to_worker(*worker*, *ts:* distributed.scheduler.TaskState, *duration: ctypes.c_double = -1*)
Send a single computational task to a worker

async start()
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*, *force=False*)
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

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

transition(*key*, *finish: str*, **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: dict*)
Process transitions until none are left

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

async unregister_worker_plugin(*comm*, *name*)
Unregisters a worker plugin

update_data(*comm=None*, *who_has=None*, *nbytes: Optional[dict] = None*, *client=None*, *serializers=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*, *retries=None*, *user_priority=0*, *actors=None*, *fifo_timeout=0*, *annotations=None*)
Add new computations to the internal dask graph

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

worker_send(*worker*, *msg*)
Send message to worker

This also handles connection failures by adding a callback to remove the worker on the next cycle.

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(*comm=None*, *memory_ratio=None*, *n=None*, *key=None*, *minimum=None*, *target=None*, *attribute='address'*)
Find workers that we can close with low cost

This returns a list of workers that are good candidates to retire. These workers are 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.

n [int] Number of workers to close

minimum [int] Minimum number of workers to keep around

key [Callable(WorkerState)] An optional callable mapping a WorkerState object to a group affiliation. Groups will be closed together. This is useful when closing workers must be done collectively, such as by hostname.

target [int] Target number of workers to have after we close

attribute [str] The attribute of the WorkerState object to return, like "address" or "name". Defaults to "address".

Returns

to_close: list of worker addresses that are OK to close

See also:

Scheduler.retire_workers

Examples

```
>>> scheduler.workers_to_close()
['tcp://192.168.0.1:1234', 'tcp://192.168.0.2:1234']
```

Group workers by hostname prior to closing

```
>>> scheduler.workers_to_close(key=lambda ws: ws.host)
['tcp://192.168.0.1:1234', 'tcp://192.168.0.1:4567']
```

Remove two workers

```
>>> scheduler.workers_to_close(n=2)
```

Keep enough workers to have twice as much memory as we we need.

```
>>> scheduler.workers_to_close(memory_ratio=2)
```

distributed.scheduler.**decide_worker**(*ts:* distributed.scheduler.TaskState, *all_workers*, *valid_workers:* *set*, *objective*) → *distributed.scheduler.WorkerState*

Decide which worker should take task *ts*.

We choose the worker that has the data on which *ts* depends.

If several workers have dependencies then we choose the less-busy worker.

Optionally provide *valid_workers* of where jobs are allowed to occur (if all workers are allowed to take the task, pass None instead).

If the task requires data communication because no eligible worker has all the dependencies already, then we choose to minimize the number of bytes sent between workers. This is determined by calling the *objective* function.

3.19 Worker

3.19.1 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!
```

3.19.2 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.

3.19.3 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 process:

```
$ dask-worker scheduler:8786 --nprocs 8 --nthreads 1
```

This will launch 8 worker processes each of which has its own ThreadPoolExecutor of size 1.

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.

3.19.4 Command Line tool

Use the `dask-worker` command line tool to start an individual worker. For more details on the command line options, please have a look at the command line tools documentation.

3.19.5 Internal Scheduling

Internally tasks that come to the scheduler proceed through the following pipeline as *distributed.worker.TaskState* objects. Tasks which follow this path have a `distributed.worker.TaskState.runspec` defined which instructs the worker how to execute them.

Data dependencies are also represented as *distributed.worker.TaskState* objects and follow a simpler path through the execution pipeline. These tasks do not have a `distributed.worker.TaskState.runspec` defined and instead contain a listing of workers to collect their result from.

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. In the event that the network comms between two workers are saturated, a dependency task may cycle between `fetch` and `flight` until it is successfully collected.

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.

Tasks slated for execution and tasks marked for collection from other workers must follow their respective transition paths as defined above. The only exceptions to this are when:

- A task is stolen, in which case a task which might have been collected will instead be executed on the thieving worker
- Scheduler intercession, in which the scheduler reassigns a task that was previously assigned to a separate worker to a new worker. This most commonly occurs when a worker dies during computation.

3.19.6 Memory Management

Workers are given a target memory limit to stay under with the command line `--memory-limit` keyword or the `memory_limit=` Python keyword argument, which sets the memory limit per worker processes launched by dask-worker

```
$ dask-worker tcp://scheduler:port --memory-limit=auto  # TOTAL_MEMORY * min(1,␣
↪nthreads / total_nthreads)
$ dask-worker tcp://scheduler:port --memory-limit=4e9  # four gigabytes per worker␣
↪process.
```

Workers use a few different heuristics to keep memory use beneath this limit:

1. At 60% of memory load (as estimated by `sizeof`), spill least recently used data to disk
2. At 70% of memory load, spill least recently used data to disk regardless of what is reported by `sizeof`
3. At 80% of memory load, stop accepting new work on local thread pool
4. At 95% of memory load, terminate and restart the worker

These values can be configured by modifying the `~/.config/dask/distributed.yaml` file

```
distributed:
  worker:
    # Fractions of worker memory at which we take action to avoid memory blowup
    # Set any of the lower three values to False to turn off the behavior entirely
    memory:
      target: 0.60  # target fraction to stay below
      spill: 0.70  # fraction at which we spill to disk
      pause: 0.80  # fraction at which we pause worker threads
      terminate: 0.95  # fraction at which we terminate the worker
```

Spill data to Disk

Every time the worker finishes a task it estimates the size in bytes that the result costs to keep in memory using the `sizeof` function. This function defaults to `sys.getsizeof` for arbitrary objects which uses the standard Python __sizeof__ protocol, but also has special-cased implementations for common data types like NumPy arrays and Pandas dataframes.

When the sum of the number of bytes of the data in memory exceeds 60% of the available threshold the worker will begin to dump the least recently used data to disk. You can control this location with the `--local-directory` keyword.:

```
$ dask-worker tcp://scheduler:port --memory-limit 4e9 --local-directory /scratch
```

That data is still available and will be read back from disk when necessary. On the diagnostic dashboard status page disk I/O will show up in the task stream plot as orange blocks. Additionally the memory plot in the upper left will become orange and then red.

Monitor process memory load

The approach above can fail for a few reasons

1. Custom objects may not report their memory size accurately
2. User functions may take up more RAM than expected
3. Significant amounts of data may accumulate in network I/O buffers

To address this we periodically monitor the memory of the worker process every 200 ms. If the system reported memory use is above 70% of the target memory usage then the worker will start dumping unused data to disk, even if internal `sizeof` recording hasn't yet reached the normal 60% limit.

Halt worker threads

At 80% load the worker's thread pool will stop accepting new tasks. This gives time for the write-to-disk functionality to take effect even in the face of rapidly accumulating data.

Kill Worker

At 95% memory load a worker's nanny process will terminate it. This is to avoid having our worker job being terminated by an external job scheduler (like YARN, Mesos, SGE, etc..). After termination the nanny will restart the worker in a fresh state.

Nanny

Dask workers are by default launched, monitored, and managed by a small Nanny process.

class distributed.nanny.**Nanny**(*scheduler_ip=None*, *scheduler_port=None*, *scheduler_file=None*, *worker_port=0*, *nthreads=None*, *ncores=None*, *loop=None*, *local_dir=None*, *local_directory=None*, *services=None*, *name=None*, *memory_limit='auto'*, *reconnect=True*, *validate=False*, *quiet=False*, *resources=None*, *silence_logs=None*, *death_timeout=None*, *preload=None*, *preload_argv=None*, *preload_nanny=None*, *preload_nanny_argv=None*, *security=None*, *contact_address=None*, *listen_address=None*, *worker_class=None*, *env=None*, *interface=None*, *host=None*, *port=None*, *protocol=None*, *config=None*, ***worker_kwargs*)

A process to manage worker processes

The nanny spins up Worker processes, watches then, and kills or restarts them as necessary. It is necessary if you want to use the `Client.restart` method, or to restart the worker automatically if it gets to the terminate fractiom of its memory limit.

The parameters for the Nanny are mostly the same as those for the Worker.

See also:

Worker

3.19.7 API Documentation

class distributed.worker.**TaskState**(*key*, *runspec=None*)
Holds volatile state relating to an individual Dask task

- **dependencies: set(TaskState instances)** The data needed by this key to run
- **dependents: set(TaskState instances)** The keys that use this dependency. Only keys which are not available already are tracked in this structure and dependents made available are actively removed. Only after all dependents have been removed, this task is allowed to be forgotten
- **duration: float** Expected duration the a task
- **priority: tuple** The priority this task given by the scheduler. Determines run order.
- **state: str** The current state of the task. One of ["waiting", "ready", "executing", "fetch", "memory", "flight", "long-running", "rescheduled", "error"]
- **who_has: set(worker)** Workers that we believe have this data
- **coming_from: str** The worker that current task data is coming from if task is in flight
- **waiting_for_data: set(keys of dependencies)** A dynamic version of dependencies. All dependencies that we still don't have for a particular key.
- **resource_restrictions: {str: number}** Abstract resources required to run a task
- **exception: str** The exception caused by running a task if it erred
- **traceback: str** The exception caused by running a task if it erred
- **type: type** The type of a particular piece of data
- **suspicious_count: int** The number of times a dependency has not been where we expected it
- **startstops: [{startstop}]** Log of transfer, load, and compute times for a task
- **start_time: float** Time at which task begins running
- **stop_time: float** Time at which task finishes running
- **metadata: dict** Metadata related to task. Stored metadata should be msgpack serializable (e.g. int, string, list, dict).
- **nbytes: int** The size of a particular piece of data

Parameters

key: str

runspec: SerializedTask A named tuple containing the `function`, `args`, `kwargs` and `task` associated with this *TaskState* instance. This defaults to `None` and can remain empty if it is a dependency that this worker will receive from another worker.

class distributed.worker.**Worker**(*scheduler_ip=None*, *scheduler_port=None*, *scheduler_file=None*, *ncores=None*, *nthreads=None*, *loop=None*, *local_dir=None*, *local_directory=None*, *services=None*, *service_ports=None*, *service_kwargs=None*, *name=None*, *reconnect=True*, *memory_limit='auto'*, *executor=None*, *resources=None*, *silence_logs=None*, *death_timeout=None*, *preload=None*, *preload_argv=None*, *security=None*, *contact_address=None*, *memory_monitor_interval='200ms'*, *extensions=None*, *metrics={}*, *startup_information={}*, *data=None*, *interface=None*, *host=None*, *port=None*, *protocol=None*, *dashboard_address=None*, *dashboard=False*, *http_prefix='/'*, *nanny=None*, *plugins=()*, *low_level_profiler=False*, *validate=None*, *profile_cycle_interval=None*, *lifetime=None*, *lifetime_stagger=None*, *lifetime_restart=None*, ***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.

- **nthreads: `int`:** Number of nthreads used by this worker process
- **executor: `concurrent.futures.ThreadPoolExecutor`:** Executor used to perform computation This can also be the string "offload" in which case this uses the same thread pool used for offloading communications. This results in the same thread being used for deserialization and computation.
- **local_directory: `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_out_connections: `int`** The maximum number of concurrent outgoing requests for data
- **total_in_connections: `int`** The maximum number of concurrent incoming requests for data
- **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

These 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.

- **tasks: `{key: TaskState}`** The tasks currently executing on this worker (and any dependencies of those tasks)
- **data: `{key: object}`:** Prefer using the **host** attribute instead of this, unless memory_limit and at least one of memory_target_fraction or memory_spill_fraction values are defined, in that case, this attribute is a zict.Buffer, from which information on LRU cache can be queried.
- **data.memory: `{key: object}`:** Dictionary mapping keys to actual values stored in memory. Only available if condition for **data** being a zict.Buffer is met.
- **data.disk: `{key: object}`:** Dictionary mapping keys to actual values stored on disk. Only available if condition for **data** being a zict.Buffer is met.
- **data_needed: deque(keys)** The keys which still require data in order to execute, arranged in a deque
- **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_count: `int`** A count of tasks currently executing on this worker
- **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.
- **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: `int`** A count of the number of tasks that are coming to us in current peer-to-peer connections
- **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
- **threads: `{key: int}`** The ID of the thread on which the task ran
- **active_threads: `{int: key}`** The keys currently running on active threads
- **waiting_for_data_count: `int`** A count of how many tasks are currently waiting for data

Parameters

scheduler_ip: str

scheduler_port: int

ip: str, optional

data: MutableMapping, type, None The object to use for storage, builds a disk-backed LRU dict by default

nthreads: int, optional

loop: tornado.ioloop.IOLoop

local_directory: str, optional Directory where we place local resources

name: str, optional

memory_limit: int, float, string Number of bytes of memory that this worker should use. Set to zero for no limit. Set to 'auto' to calculate as system.MEMORY_LIMIT * min(1, nthreads / total_cores) Use strings or numbers like 5GB or 5e9

memory_target_fraction: float Fraction of memory to try to stay beneath

memory_spill_fraction: float Fraction of memory at which we start spilling to disk

memory_pause_fraction: float Fraction of memory at which we stop running new tasks

executor: concurrent.futures.Executor

resources: dict Resources that this worker has like `{'GPU': 2}`

nanny: str Address on which to contact nanny, if it exists

lifetime: str Amount of time like "1 hour" after which we gracefully shut down the worker. This defaults to None, meaning no explicit shutdown time.

lifetime_stagger: str Amount of time like "5 minutes" to stagger the lifetime value The actual lifetime will be selected uniformly at random between lifetime +/- lifetime_stagger

lifetime_restart: bool Whether or not to restart a worker after it has reached its lifetime Default False

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
```

3.20 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.

3.20.1 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.

3.20.2 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.

3.20.3 Transactional Work Stealing

To avoid running the same task twice, Dask implements transactional work stealing. When the scheduler identifies a task that should be moved it first sends a request to the busy worker. The worker inspects its current state of the task and sends a response to the scheduler:

1. If the task is not yet running, then the worker cancels the task and informs the scheduler that it can reroute the task elsewhere.
2. If the task is already running or complete then the worker tells the scheduler that it should not replicate the task elsewhere.

This avoids redundant work, and also the duplication of side effects for more exotic tasks. However, concurrent or repeated execution of the same task *is still possible* in the event of worker death or a disrupted network connection.

3.20.4 Disabling Work Stealing

Work stealing is a toggleable setting on the Dask Scheduler; to disable work stealing, you can toggle the scheduler `work-stealing` configuration option to `"False"` either by setting `DASK_DISTRIBUTED__SCHEDULER__WORK_STEALING="False"` or through your Dask configuration file

3.21 Why did my worker die?

A Dask worker can cease functioning for a number of reasons. These fall into the following categories:

- the worker chooses to exit
- an unrecoverable exception happens within the worker
- the worker process is shut down by some external action

Each of these cases will be described in more detail below. The *symptoms* you will experience when these things happen range from simply work not getting done anymore, to various exceptions appearing when you interact with your local client, such as `KilledWorker`, `TimeoutError` and `CommClosedError`.

Note the special case of `KilledWorker`: this means that a particular task was tried on a worker, and it died, and then the same task was sent to another worker, which also died. After a configurable number of deaths (config key " `distributed.scheduler.allowed-failures`), Dask decides to blame the task itself, and returns this exception. Note, that it is possible for a task to be unfairly blamed - the worker happened to die while the task was active, perhaps due to another thread - complicating diagnosis.

In every case, the first place to look for further information is the logs of the given worker, which may well give a complete description of what happened. These logs are printed by the worker to its "standard error", which may appear in the text console from which you launched the worker, or some logging system maintained by the cluster infrastructure. It is also helpful to watch the diagnostic dashboard to look for memory spikes, but of course this is only possible while the worker is still alive.

In all cases, the scheduler will notice that the worker has gone, either because of an explicit de-registration, or because the worker no longer produces heartbeats, and it should be possible to reroute tasks to other workers and have the system keep running.

3.21.1 Scenarios

Worker chose to exit

Workers may exit in normal functioning because they have been asked to, e.g., they received a keyboard interrupt (^C), or the scheduler scaled down the cluster. In such cases, the work that was being done by the worker will be redirected to other workers, if there are any left.

You should expect to see the following message at the end of the worker's log:

```
distributed.dask_worker - INFO - End worker
```

In these cases, there is not normally anything which you need to do, since the behaviour is expected.

Unrecoverable Exception

The worker is a python process, and like any other code, an exception may occur which causes the process to exit. One typical example of this might be a version mismatch between the packages of the client and worker, so that a message sent to the worker errors while being unpacked. There are a number of packages that need to match, not only `dask` and `distributed`.

In this case, you should expect to see the full python traceback in the worker's log. In the event of a version mismatch, this might be complaining about a bad import or missing attribute. However, other fatal exceptions are also possible, such as trying to allocate more memory than the system has available, or writing temporary files without appropriate permissions.

To assure that you have matching versions, you should run (more recent versions of distributed may do this automatically)

```
client.get_versions(check=True)
```

For other errors, you might want to run the computation in your local client, if possible, or try grabbing just the task that errored and using *`recreate_error_locally`*, as you would for ordinary exceptions happening during task execution.

Specifically for connectivity problems (e.g., timeout exceptions in the worker logs), you will need to diagnose your networking infrastructure, which is more complicated than can be described here. Commonly, it may involve logging into the machine running the affected worker (although you can *Launch IPython within Dask Workers*).

Killed by Nanny

The Dask "nanny" is a process which watches the worker, and restarts it if necessary. It also tracks the worker's memory usage, and if it should cross a given fraction of total memory, then also the worker will be restarted, interrupting any work in progress. The log will show a message like

```
Worker exceeded X memory budget. Restarting
```

Where X is the memory fraction. You can set this critical fraction using the configuration, see *Memory Management*. If you have an external system for watching memory usage provided by your cluster infrastructure (HPC, kubernetes, etc.), then it may be reasonable to turn off this memory limit. Indeed, in these cases, restarts might be handled for you too, so you could do without the nanny at all (`--no-nanny` CLI option or configuration equivalent).

Sudden Exit

The worker process may stop working without notice. This can happen due to something internal to the worker, e.g., a memory violation (common if interfacing with compiled code), or due to something external, e.g., the `kill` command, or stopping of the container or machine on which the worker is running.

In the best case, you may have a line in the logs from the OS saying that the worker was shut down, such as the single word "killed" or something more descriptive. In these cases, the fault may well be in your code, and you might be able to use the same debugging tools as in the previous section.

However, if the action was initiated by some outside framework, then the worker will have no time to leave a logging message, and the death *may* have nothing to do with what the worker was doing at the time. For example, if kubernetes decides to evict a pod, or your ec2 instance goes down for maintenance, the worker is not at fault. Hopefully, the system provides a reasonable message of what happened in the process output. However, if the memory allocation (or other resource) exceeds toleration, then it *is* the code's fault - although you may be able to fix with better configuration of Dask's own limits, or simply with a bigger cluster. In any case, your deployment framework has its own logging system, and you should look there for the reason that the dask worker was taken down.

Specifically for memory issues, refer to the memory section of best practices.

3.22 Actors

Note: This is an experimental feature and is subject to change without notice

Note: This is an advanced feature and may not be suitable for beginning users. It is rarely necessary for common workloads.

Actors enable stateful computations within a Dask workflow. They are useful for some rare algorithms that require additional performance and are willing to sacrifice resilience.

An actor is a pointer to a user-defined-object living on a remote worker. Anyone with that actor can call methods on that remote object.

3.22.1 Example

Here we create a simple `Counter` class, instantiate that class on one worker, and then call methods on that class remotely.

```
class Counter:
    """ A simple class to manage an incrementing counter """
    n = 0

    def __init__(self):
        self.n = 0

    def increment(self):
        self.n += 1
        return self.n

    def add(self, x):
        self.n += x
        return self.n

from dask.distributed import Client           # Start a Dask Client
client = Client()

future = client.submit(Counter, actor=True)   # Create a Counter on a worker
counter = future.result()                     # Get back a pointer to that object

counter
# <Actor: Counter, key=Counter-1234abcd>

future = counter.increment()                  # Call remote method
future.result()                               # Get back result
# 1

future = counter.add(10)                      # Call remote method
future.result()                               # Get back result
# 11
```

3.22.2 Motivation

Actors are motivated by some of the challenges of using pure task graphs.

Normal Dask computations are composed of a graph of functions. This approach has a few limitations that are good for resilience, but can negatively affect performance:

1. **State**: The functions should not mutate their inputs in-place or rely on global state. They should instead operate in a pure-functional manner, consuming inputs and producing separate outputs.
2. **Central Overhead**: The execution location and order is determined by the centralized scheduler. Because the scheduler is involved in every decision it can sometimes create a central bottleneck.

Some workloads may need to update state directly, or may involve more tiny tasks than the scheduler can handle (the scheduler can coordinate about 4000 tasks per second).

Actors side-step both of these limitations:

1. **State**: Actors can hold on to and mutate state. They are allowed to update their state in-place.
2. **Overhead**: Operations on actors do not inform the central scheduler, and so do not contribute to the 4000 task/second overhead. They also avoid an extra network hop and so have lower latencies.

3.22.3 Create an Actor

You create an actor by submitting a Class to run on a worker using normal Dask computation functions like `submit`, `map`, `compute`, or `persist`, and using the `actors=` keyword (or `actor=` on `submit`).

```
future = client.submit(Counter, actors=True)
```

You can use all other keywords to these functions like `workers=`, `resources=`, and so on to control where this actor ends up.

This creates a normal Dask future on which you can call `.result()` to get the Actor once it has successfully run on a worker.

```
>>> counter = future.result()
>>> counter
<Actor: Counter, key=...>
```

A `Counter` object has been instantiated on one of the workers, and this `Actor` object serves as our proxy to that remote object. It has the same methods and attributes.

```
>>> dir(counter)
['add', 'increment', 'n']
```

3.22.4 Call Remote Methods

However accessing an attribute or calling a method will trigger a communication to the remote worker, run the method on the remote worker in a separate thread pool, and then communicate the result back to the calling side. For attribute access these operations block and return when finished, for method calls they return an `ActorFuture` immediately.

```
>>> future = counter.increment()  # Immediately returns an ActorFuture
>>> future.result()               # Block until finished and result arrives
1
```

`ActorFuture` are similar to normal Dask `Future` objects, but not as fully featured. They curently *only* support the `result` method and nothing else. They don't currently work with any other Dask functions that expect futures, like `as_completed`, `wait`, or `client.gather`. They can't be placed into additional submit or map calls to form dependencies. They communicate their results immediately (rather than waiting for result to be called) and cache the result on the future itself.

3.22.5 Access Attributes

If you define an attribute at the class level then that attribute will be accessible to the actor.

```
class Counter:
    n = 0   # Recall that we defined our class with `n` as a class variable

    ...

>>> counter.n                      # Blocks until finished
1
```

Attribute access blocks automatically. It's as though you called `.result()`.

3.22.6 Execution on the Worker

When you call a method on an actor, your arguments get serialized and sent to the worker that owns the actor's object. If you do this from a worker this communication is direct. If you do this from a Client then this will be direct if the Client has direct access to the workers (create a client with `Client(..., direct_to_workers=True)` if direct connections are possible) or by proxying through the scheduler if direct connections from the client to the workers are not possible.

The appropriate method of the Actor's object is then called in a separate thread, the result captured, and then sent back to the calling side. Currently workers have only a single thread for actors, but this may change in the future.

The result is sent back immediately to the calling side, and is not stored on the worker with the actor. It is cached on the `ActorFuture` object.

3.22.7 Calling from coroutines and async/await

If you use actors within a coroutine or async/await function then actor methods and attrbute access will return Tornado futures

```
async def f():
    counter = await client.submit(Counter, actor=True)

    await counter.increment()
    n = await counter.n
```

3.22.8 Coroutines and async/await on the Actor

If you define an `async def` function on the actor class then that method will run on the Worker's event loop thread rather than a separate thread.

```
def Waiter:
    def __init__(self):
        self.event = asyncio.Event()

    async def set(self):
        self.event.set()

    async def wait(self):
        await self.event.wait()

waiter = client.submit(Waiter, actor=True).result()
waiter.wait().result()  # waits until set, without consuming a worker thread
```

3.22.9 Performance

Worker operations currently have about 1ms of latency, on top of any network latency that may exist. However other activity in a worker may easily increase these latencies if enough other activities are present.

3.22.10 Limitations

Actors offer advanced capabilities, but with some cost:

1. **No Resilience:** No effort is made to make actor workloads resilient to worker failure. If the worker dies while holding an actor that actor is lost forever.
2. **No Diagnostics:** Because the scheduler is not informed about actor computations no diagnostics are available about these computations.
3. **No Load balancing:** Actors are allocated onto workers evenly, without serious consideration given to avoiding communication.
4. **Experimental:** Actors are a new feature and subject to change without warning

3.23 Asynchronous Operation

Dask can run fully asynchronously 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).

3.23.1 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, such as a function defined with `async/await` syntax.

```
async def f():
    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 use an asynchronous function with a synchronous `Client` (one made without the `asynchronous=True` keyword) then you can apply the `asynchronous=True` keyword at each method call and use the `Client.sync` function to run the asynchronous function:

```
from dask.distributed import Client

client = Client()  # normal blocking client

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

client.sync(f)
```

```
async with Client(asynchronous=True) as client:
    arr = da.random.random((1000, 1000), chunks=(1000, 100))
    await client.compute(arr.mean())
```

3.23.2 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 Asyncio and Tornado.

Python 3 with Tornado or Asyncio

```
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

# Either use Tornado
from tornado.ioloop import IOLoop
IOLoop().run_sync(f)

# Or use asyncio
import asyncio
asyncio.get_event_loop().run_until_complete(f())
```

3.23.3 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.

3.24 IPython Integration

Dask.distributed integrates with IPython in three ways:

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

3.24.1 Launch Dask from IPyParallel

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

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

This is possible with the Client.become_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>
```

3.24.2 Launch IPython within Dask Workers

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

`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)
```

3.25 HTTP endpoints

A subset of the following pages will be available from the scheduler or workers of a running cluster. The list of currently available endpoints can be found by examining `/sitemap.json`.

3.25.1 Main dashboard links

Dynamic bokeh pages. The root redirects to /status, and each page links to the others via a header navbar.

- `/status`: entry point to the dashboards, shows cluster-wide memory and tasks
- `/workers`: currently connected workers and their CPU/memory usage
- `/tasks`: task block view with longer look-back than on /status
- `/system`: global stats for the cluster, CPU, memory, bandwidth, file descriptors
- `/profile`: flamegraph statistical profiling across the cluster
- `/graph`: currently processing graphs in a dependnecy tree view
- `/info`: redirect to `/info/main/workers.html`

3.25.2 Scheduler HTTP

Pages and JSON endpoints served by the scheduler

- `/health`: check server is alive
- `/info/main/workers.html` basic info about workers and links to their dashboards and logs
- `info/worker/(worker).html`: more detail about given worker, keyed by TCP address; links to tasks
- `info/task/(task).html`: details about a task on the cluster, by dask key; links to worker, related tasks, and client
- `/info/call-stacks/(worker).html`: tasks currently handled by given worker
- `/info/call-stack/(task).html`: state of task (where it is running)
- `/info/main/logs.html`: scheduler logs
- `/info/logs/(worker).html`: logs of given worker
- `/individual-plots.json`: map of path to description for available individual (i.e,. one-pane, non-dashboard) plots
- `/eventstream`: scheduler events on a websocket
- `/proxy/(port)/(address)/(path)`: proxy to worker HTTP locations (if you have jupyter-server-proxy)
- `/metrics`: prometheus endpoint
- `/json/counts.json`: cluster count stats
- `/json/identity.json`: scheduler information
- `/json/index.html`: link to the above two endpoints
- `/sitemap.json`: list of available endpoints
- `/statics/()`: static file content (CSS, etc)
- `/stealing`: worker occupancy metrics, to evaluate task stealing

3.25.3 Individual bokeh plots

- `/individual-task-stream`
- `/individual-progress`
- `/individual-graph`
- `/individual-profile`
- `/individual-profile-server`
- `/individual-nbytes`
- `/individual-nbytes-cluster`
- `/individual-cpu`
- `/individual-nprocessing`
- `/individual-workers`
- `/individual-bandwidth-types`
- `/individual-bandwidth-workers`
- `/individual-memory-by-key`
- `/individual-compute-time-per-key`
- `/individual-aggregate-time-per-action`
- `/individual-scheduler-system`
- `/individual-gpu-memory` (GPU only)
- `/individual-gpu-utilization` (GPU only)

3.25.4 Worker HTTP

- `/status`:
- `/counters`:
- `/crossfilter`:
- `/sitemap.json`: list of available endpoints
- `/system`:
- `/health`: check server is alive
- `/metrics`: prometheus endpoint
- `/statics/()`: static file content (CSS, etc)

3.26 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

3.26.1 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.

3.26.2 Dictionary interface

Alternatively you can use the `.datasets` mapping on the client to publish, list, get, and delete global datasets.

```
>>> client.datasets['negative_accounts'] = df

>>> list(client.datasets)
['negative_accounts']
>>> df = client.datasets['negative_accounts']
```

This mapping is globally shared among all clients connected to the same scheduler.

3.26.3 Notes

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

Any client can publish or unpublish a dataset.

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

3.26.4 API

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

3.27 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.

3.27.1 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})
```

Equivalently, we can specify resource constraints using the dask annotations machinery:

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

3.27.2 Specifying Resources

Resources can be specifed in several ways. The easiest option will depend on exactly how your cluster is being created.

From the command line

Resources can be provided when starting the worker process, as shown above:

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

The keys are used as the resource name and the values are parsed into a numeric value.

From Dask's configuration system

Alternatively, resources can be specified using Dask's configuration system.

```
from distributed import LocalCluster

with dask.config.set({"distributed.worker.resources.GPU": 2}):
    cluster = LocalCluster()
```

The configuration will need to be set in the process that's spawning the actual worker. This might be easiest to achieve by specifying resources as an environment variable (shown in the next section).

From environment variables

Like any other Dask config value, resources can be specified as environment variables before starting the process. Using Bash syntax

```
$ DASK_DISTRIBUTED__WORKER__RESOURCES__GPU=2 dask-worker
...
```

This might be the easiest solution if you aren't able to pass options to the `distributed.Worker` class.

3.27.3 Resources are applied separately to each worker process

If you are using `dask-worker --nprocs <nprocs>` the resource will be applied separately to each of the `nprocs` worker processes. Suppose you have 2 GPUs on your machine, if you want to use two worker processes, you have 1 GPU per worker process so you need to do something like this:

```
dask-worker scheduler:8786 --nprocs 2 --resources "GPU=1"
```

Here is an example that illustrates how to use resources to ensure each task is run inside a separate process, which is useful to execute non thread-safe tasks or tasks that uses multithreading internally:

```
dask-worker scheduler:8786 --nprocs 3 --nthreads 2 --resources "process=1"
```

With the code below, there will be at most 3 tasks running concurrently and each task will run in a separate process:

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

futures = [client.submit(non_thread_safe_function, arg,
                         resources={'process': 1}) for arg in args]
```

3.27.4 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.

3.27.5 Resources with collections

You can also use resources with Dask collections, like arrays, dataframes, and delayed objects. You can annotate operations on collections with specific resources that should be required perform the computation using the dask annotations machinery.

```
x = dd.read_csv(...)
with dask.annotate(resources={'GPU': 1}):
    y = x.map_partitions(func1)
z = y.map_partitions(func2)

z.compute(optimize_graph=False)
```

In most cases (such as the case above) the annotations for `y` may be lost during graph optimization before execution. You can avoid that by passing the `optimize_graph=False` keyword.

3.28 Launch Tasks from Tasks

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

3.28.1 Motivating example

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

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

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

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

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

3.28.2 Gather the data locally

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

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

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

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

3.28.3 Submit tasks from client

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

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

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

>>> futures = client.map(process_element, elements)
>>> analysis = client.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.

Extended Example

Computing the Fibonacci numbers creates involves a recursive function. When the function is run, it calls itself using values it computed. We will use this as an example throughout this documentation to illustrate different techniques of submitting tasks from tasks.

```
def fib(n):
    if n < 2:
        return n
    a = fib(n - 1)
    b = fib(n - 2)
    return a + b

print(fib(10))  # prints "55"
```

We will use this example to show the different interfaces.

3.28.4 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 are three options for this:

1. `dask.delayed` and `dask.compute`
2. `get_client` with `secede` and `rejoin`
3. `worker_client`

dask.delayed

The Dask delayed behaves as normal: it submits the functions to the graph, optimizes for less bandwidth/computation and gathers the results. For more detail, see dask.delayed.

```
from distributed import Client
from dask import delayed, compute


@delayed
def fib(n):
    if n < 2:
        return n
    # We can use dask.delayed and dask.compute to launch
    # computation from within tasks
    a = fib(n - 1)  # these calls are delayed
    b = fib(n - 2)
    a, b = compute(a, b)  # execute both in parallel
    return a + b

if __name__ == "__main__":
    # these features require the dask.distributed scheduler
    client = Client()
```

(continues on next page)

(continued from previous page)

```
    result = fib(10).compute()
    print(result)  # prints "55"
```

Getting the client on a worker

The *get_client* function provides a normal Client object that gives full access to the dask cluster, including the ability to submit, scatter, and gather results.

```
from distributed import Client, get_client, secede, rejoin

def fib(n):
    if n < 2:
        return n
    client = get_client()
    a_future = client.submit(fib, n - 1)
    b_future = client.submit(fib, n - 2)
    a, b = client.gather([a_future, b_future])
    return a + b

if __name__ == "__main__":
    client = Client()
    future = client.submit(fib, 10)
    result = future.result()
    print(result)  # prints "55"
```

However, this can deadlock the scheduler if too many tasks request jobs at once. Each task does not communicate to the scheduler that they are waiting on results and are free to compute other tasks. This can deadlock the cluster if every scheduling slot is running a task and they all request more tasks.

To avoid this deadlocking issue we can use *secede* and *rejoin*. These functions will remove and rejoin the current task from the cluster respectively.

```
def fib(n):
    if n < 2:
        return n
    client = get_client()
    a_future = client.submit(fib, n - 1)
    b_future = client.submit(fib, n - 2)
    secede()
    a, b = client.gather([a_future, b_future])
    rejoin()
    return a + b
```

Connection with context manager

The *worker_client* function performs the same task as *get_client*, but is implemented as a context manager. Using *worker_client* as a context manager ensures proper cleanup on the worker.

```
from dask.distributed import worker_client


def fib(n):
    if n < 2:
```

(continues on next page)

(continued from previous page)

```
        return n
    with worker_client() as client:
        a_future = client.submit(fib, n - 1)
        b_future = client.submit(fib, n - 2)
        a, b = client.gather([a_future, b_future])
    return a + b

if __name__ == "__main__":
    client = Client()
    future = client.submit(fib, 10)
    result = future.result()
    print(result)  # prints "55"
```

Tasks that invoke *worker_client* are conservatively assumed to be *long running*. They can take a long time, 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.

Establishing a connection to the scheduler takes a few milliseconds and so it is wise for computations that use this feature to be at least a few times longer in duration than this.

3.29 TLS/SSL

Dask distributed has 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.

3.29.1 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)
```

3.29.2 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.

3.29.3 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.

3.29.4 API

class distributed.security.**Security**(*require_encryption=None*, ***kwargs*)
Security configuration for a Dask cluster.

Default values are loaded from Dask's configuration files, and can be overridden in the constructor.

Parameters

require_encryption [bool, optional] Whether TLS encryption is required for all connections.

tls_ca_file [str, optional] Path to a CA certificate file encoded in PEM format.

tls_ciphers [str, optional] An OpenSSL cipher string of allowed ciphers. If not provided, the system defaults will be used.

tls_client_cert [str, optional] Path to a certificate file for the client, encoded in PEM format.

tls_client_key [str, optional] Path to a key file for the client, encoded in PEM format. Alternatively, the key may be appended to the cert file, and this parameter be omitted.

tls_scheduler_cert [str, optional] Path to a certificate file for the scheduler, encoded in PEM format.

tls_scheduler_key [str, optional] Path to a key file for the scheduler, encoded in PEM format. Alternatively, the key may be appended to the cert file, and this parameter be omitted.

tls_worker_cert [str, optional] Path to a certificate file for a worker, encoded in PEM format.

tls_worker_key [str, optional] Path to a key file for a worker, encoded in PEM format. Alternatively, the key may be appended to the cert file, and this parameter be omitted.

extra_conn_args [mapping, optional] Mapping with keyword arguments to pass down to connections.

get_connection_args(*role*)
Get the *connection_args* argument for a connect() call with the given *role*.

get_listen_args(*role*)
Get the *connection_args* argument for a listen() call with the given *role*.

get_tls_config_for_role(*role*)
Return the TLS configuration for the given role, as a flat dict.

classmethod temporary(***kwargs*)
Create a new temporary Security object.

This creates a new self-signed key/cert pair suitable for securing communication for all roles in a Dask cluster. These keys/certs exist only in memory, and are stored in this object.

This method requires the library `cryptography` be installed.

3.30 Changelog

3.30.1 2021.05.0

Released on May 14, 2021

- Merge global annotations on the client (GH#4691) Mads R. B. Kristensen
- Add support for `click` 8 (GH#4810) James Bourbeau

- Add HTML reprs to some scheduler classes (GH#4795) James Bourbeau
- Use JupyterLab theme variables (GH#4796) Ian Rose
- Allow the dashboard to run on multiple ports (GH#4786) Jacob Tomlinson
- Remove `release_dep` from `WorkerPlugin` API (GH#4791) James Bourbeau
- Support for UCX 1.10+ (GH#4787) Peter Andreas Entschev
- Reduce complexity of `test_gather_allow_worker_reconnect` (GH#4739) Florian Jetter
- Fix doctests in `utils.py` (GH#4785) Jacob Tomlinson
- Ensure deps are actually logged in worker (GH#4753) Florian Jetter
- Add `stacklevel` keyword into `performance_report()` to allow for selecting calling code to be displayed (GH#4777) Nathan Danielsen
- Unregister worker plugin (GH#4748) Naty Clementi
- Fixes some pickling issues in the Cythonized `Scheduler` (GH#4768) jakirkham
- Improve graceful shutdown if nanny is involved (GH#4725) Florian Jetter
- Update cythonization in CI (GH#4764) James Bourbeau
- Use `contextlib.nullcontext` (GH#4763) James Bourbeau
- Cython fixes for `MemoryState` (GH#4761) jakirkham
- Fix errors in `check_thread_leak` (GH#4747) James Bourbeau
- Handle missing `key` case in `report_on_key` (GH#4755) jakirkham
- Drop temporary `set` variables `s` (GH#4758) jakirkham

3.30.2 2021.04.1

Released on April 23, 2021

- Avoid `active_threads` changing size during iteration (GH#4729) James Bourbeau
- Fix `UnboundLocalError` in `AdaptiveCore.adapt()` (GH#4731) Anderson Banihirwe
- Minor formatting updates for HTTP endpoints doc (GH#4736) James Bourbeau
- Unit test for `metrics["memory"]=None` (GH#4727) crusaderky
- Enable configuration of prometheus metrics namespace (GH#4722) Jacob Tomlinson
- Reintroduce `weight` function (GH#4723) James Bourbeau
- Add `ready->memory` to transitions in worker (GH#4728) Gil Forsyth
- Fix regressions in GH#4651 (GH#4719) crusaderky
- Add descriptions for UCX config options (GH#4683) Charles Blackmon-Luca
- Split RAM measure into dask keys/other old/other new (GH#4651) crusaderky
- Fix `DeprecationWarning` on Python 3.9 (GH#4717) George Sakkis
- ipython causes `test_profile_nested_sizeof` crash on windows (GH#4713) crusaderky
- Add `iterate_collection` argument to `serialize` (GH#4641) Richard J Zamora
- When closing `Server`, close all listeners (GH#4704) Florian Jetter

- Fix timeout in `client.restart` (GH#4690) Matteo De Wint
- Avoid repeatedly using the same worker on first task with quiet cluster (GH#4638) Doug Davis
- Grab `func` for `finish` case only if used (GH#4702) jakirkham
- Remove hostname check in `test_dashboard` (GH#4706) James Bourbeau
- Faster `tests_semaphore::test_worker_dies` (GH#4703) Florian Jetter
- Clean up `test_dashboard` (GH#4700) crusaderky
- Add timing information to `TaskGroup` (GH#4671) Matthew Rocklin
- Remove `WSSConnector` TLS presence check (GH#4695) Marcos Moyano
- Fix typo and remove unused `time.time` import (GH#4689) Hristo Georgiev
- Don't initialize CUDA context in monitor (GH#4688) Charles Blackmon-Luca
- Add support for extra conn args for HTTP protocols (GH#4682) Marcos Moyano
- Adjust timings in `test_threadpoolworkers` (GH#4681) Florian Jetter
- Add GPU metrics to `SystemMonitor` (GH#4661) Charles Blackmon-Luca
- Removing `dumps_msgpack()` and `loads_msgpack()` (GH#4677) Mads R. B. Kristensen
- Expose worker `SystemMonitor` s to scheduler via RPC (GH#4657) Charles Blackmon-Luca

3.30.3 2021.04.0

Released on April 2, 2021

- Fix un-merged frames (GH#4666) Matthew Rocklin
- Add informative error message to install uvloop (GH#4664) Matthew Rocklin
- Remove incorrect comment regarding default `LocalCluster` creation (GH#4660) cameron16
- Treat empty/missing `writeable` as a no-op (GH#4659) jakirkham
- Avoid list mutation in `pickle_loads` (GH#4653) Matthew Rocklin
- Ignore `OSError` exception when scaling down (GH#4633) Gerald
- Add `isort` to pre-commit hooks, package resorting (GH#4647) Charles Blackmon-Luca
- Use powers-of-two when displaying RAM (GH#4649) crusaderky
- Support Websocket communication protocols (GH#4396) Marcos Moyano
- `scheduler.py` / `worker.py` code cleanup (GH#4626) crusaderky
- Update out-of-date references to `config.yaml` (GH#4643) Hristo Georgiev
- Suppress `OSError` on `SpecCluster` shutdown (GH#4567) Jacob Tomlinson
- Replace conda with mamba (GH#4585) crusaderky
- Expand documentation on pure functions (GH#4644) James Lamb

3.30.4 2021.03.1

Released on March 26, 2021

- Add standalone dashboard page for GPU usage (GH#4556) Jacob Tomlinson
- Handle `stream is None` case in TCP comm finalizer (GH#4631) James Bourbeau
- Include `LIST_PICKLE` in NumPy array serialization (GH#4632) James Bourbeau
- Rename annotation plugin in `test_highlevelgraph.py` (GH#4618) James Bourbeau
- UCX use `nbytes` instead of `len` (GH#4621) Mads R. B. Kristensen
- Skip NumPy and pandas tests if not importable (GH#4563) Ben Greiner
- Remove `utils.shutting_down` in favor of `sys.is_finalizing` (GH#4624) James Bourbeau
- Handle `async` clients when closing (GH#4623) Matthew Rocklin
- Drop `log` from `remove_key_from_stealable` (GH#4609) jakirkham
- Introduce events log length config option (GH#4615) Fabian Gebhart
- Upstream config serialization and inheritance (GH#4372) Jacob Tomlinson
- Add check to scheduler creation in `SpecCluster` (GH#4605) Jacob Tomlinson
- Make length of events `deque` configurable (GH#4604) Fabian Gebhart
- Add explicit `fetch` state to worker `TaskState` (GH#4470) Gil Forsyth
- Update `develop.rst` (GH#4603) Florian Jetter
- `pickle_loads()`: Handle empty `memoryview` (GH#4595) Mads R. B. Kristensen
- Switch documentation builds for PRs to readthedocs (GH#4599) James Bourbeau
- Track frame sizes along with frames (GH#4593) jakirkham
- Add support for a list of keys when using `batch_size` in `client.map` (GH#4592) Sultan Orazbayev
- If `SpecCluster` fails to start attempt to gracefully close out again (GH#4590) Jacob Tomlinson
- Multi-lock extension (GH#4503) Mads R. B. Kristensen
- Update `PipInstall` plugin command (GH#4584) James Bourbeau
- IPython magics: remove deprecated `ioloop` workarounds (GH#4530) Min RK
- Add GitHub actions workflow to cancel duplicate builds (GH#4581) James Bourbeau
- Remove outdated macOS build badge from `README` (GH#4576) James Bourbeau
- Dask master -> main (GH#4569) Julia Signell
- Drop support for Python 3.6 (GH#4390) James Bourbeau
- Add docstring for `dashboard_link` property (GH#4572) Doug Davis
- Change default branch from master to main (GH#4495) Julia Signell
- Msgpack handles extract serialize (GH#4531) Mads R. B. Kristensen

3.30.5 2021.03.0

Released on March 5, 2021

Note: This is the first release with support for Python 3.9 and the last release with support for Python 3.6

- `tcp.write()`: cast `memoryview` to byte itemsize (GH#4555) Mads R. B. Kristensen
- Refcount the `thread_state.asynchronous` flag (GH#4557) Mads R. B. Kristensen
- Python 3.9 (GH#4460) crusaderky
- Better bokeh defaults for dashboard (GH#4554) Benjamin Zaitlen
- Expose system monitor dashboard as individual plot for lab extension (GH#4540) Jacob Tomlinson
- Pass on original temp dir from nanny to worker (GH#4549) Martin Durant
- Serialize and split (GH#4541) Mads R. B. Kristensen
- Use the new HLG pack/unpack API in Dask (GH#4489) Mads R. B. Kristensen
- Handle annotations for culled tasks (GH#4544) Tom Augspurger
- Make sphinx autosummary and autoclass consistent (GH#4367) Casey Clements
- Move `_transition*` to `SchedulerState` (GH#4545) jakirkham
- Migrate from travis to GitHub actions (GH#4504) crusaderky
- Move `new_task` to `SchedulerState` (GH#4527) jakirkham
- Batch more Scheduler sends (GH#4526) jakirkham
- `transition_memory_released` and `get_nbytes()` optimizations (GH#4516) jakirkham
- Pin `black` pre-commit (GH#4533) James Bourbeau
- Read & write all frames in one pass (GH#4506) jakirkham
- Skip `stream.write` call for empty frames (GH#4507) jakirkham
- Prepend frame metadata header (GH#4505) jakirkham
- `transition_processing_memory` optimizations, etc. (GH#4487) jakirkham
- Attempt to get client from worker in `Queue` and `Variable` (GH#4490) James Bourbeau
- Use `main` branch for `zict` (GH#4499) jakirkham
- Use a callback to close TCP Comms, rather than check every time (GH#4453) Matthew Rocklin

3.30.6 2021.02.0

Released on February 5, 2021

- Bump minimum Dask to 2021.02.0 (GH#4486) James Bourbeau
- Update `TaskState` documentation about dependents attribute (GH#4440) Florian Jetter
- DOC: Autoreformat all functions docstrings (GH#4475) Matthias Bussonnier
- Use cached version of `is_coroutine_function` in stream handling to (GH#4481) Ian Rose
- Optimize `transitions` (GH#4451) jakirkham

- Create `PULL_REQUEST_TEMPLATE.md` (GH#4476) Ray Bell
- DOC: typo, directives ends with 2 colons `::` (GH#4472) Matthias Bussonnier
- DOC: Proper numpydoc syntax for `distributed/protocol/*.py` (GH#4473) Matthias Bussonnier
- Update `pytest.skip` usage in `test_server_listen` (GH#4467) James Bourbeau
- Unify annotations (GH#4406) Ian Rose
- Added worker resources from config (GH#4456) Tom Augspurger
- Fix var name in worker validation func (GH#4457) Gil Forsyth
- Refactor `task_groups` & `task_prefixes` (GH#4452) jakirkham
- Use `parent._tasks` in `heartbeat` (GH#4450) jakirkham
- Refactor `SchedulerState` from `Scheduler` (GH#4365) jakirkham

3.30.7 2021.01.1

Released on January 22, 2021

- Make system monitor interval configurable (GH#4447) Matthew Rocklin
- Add `uvloop` config value (GH#4448) Matthew Rocklin
- Additional optimizations to stealing (GH#4445) jakirkham
- Give clusters names (GH#4426) Jacob Tomlinson
- Use worker comm pool in `Semaphore` (GH#4195) Florian Jetter
- Set `runspec` on all new tasks to avoid deadlocks (GH#4432) Florian Jetter
- Support `TaskState` objects in story methods (GH#4434) Matthew Rocklin
- Support missing event loop in `Client.asynchronous` (GH#4436) Matthew Rocklin
- Don't require network to inspect tests (GH#4433) Matthew Rocklin

3.30.8 2021.01.0

Released on January 15, 2021

- Add time started to scheduler info (GH#4425) Jacob Tomlinson
- Log adaptive error (GH#4422) Jacob Tomlinson
- Xfail normalization tests (GH#4411) Jacob Tomlinson
- Use `dumps_msgpack` and `loads_msgpack` when packing high level graphs (GH#4409) Mads R. B. Kristensen
- Add `nprocs` auto option to `dask-worker` CLI (GH#4377) Jacob Tomlinson
- Type annotation of `_reevaluate_occupancy_worker` (GH#4398) jakirkham
- Type `TaskGroup` in `active_states` (GH#4408) jakirkham
- Fix `test_as_current_is_thread_local` (GH#4402) jakirkham
- Use `list` comprehensions to bind `TaskGroup` type (GH#4401) jakirkham
- Make tests pass after 2028 (GH#4403) Bernhard M. Wiedemann

- Fix compilation warnings, `decide_worker` now a C func, stealing improvements (GH#4375) jakirkham
- Drop custom `__eq__` from `Status` (GH#4270) jakirkham
- `test_performance_report`: skip without bokeh (GH#4388) Bruno Pagani
- `Nanny` now respects dask settings from ctx mgr (GH#4378) Florian Jetter
- Better task duration estimates for outliers (GH#4213) selshowk
- Dask internal inherit config (GH#4364) Jacob Tomlinson
- Provide `setup.py` option to profile Cython code (GH#4362) jakirkham
- Optimizations of `*State` and `Task*` objects and stealing (GH#4358) jakirkham
- Cast `SortedDict` s to `dict` s in a few key places & other minor changes (GH#4355) jakirkham
- Use task annotation priorities for user-level priorities (GH#4354) James Bourbeau
- Added docs to highlevelgraph pack/unpack (GH#4352) Mads R. B. Kristensen
- Optimizations in notable functions used by transitions (GH#4351) jakirkham
- Silence exception when releasing futures on process shutdown (GH#4309) Benjamin Zaitlen

3.30.9 2020.12.0

Released on December 10, 2020

Highlights

- Switched to CalVer for versioning scheme.
- The scheduler can now receives Dask `HighLevelGraph` s instead of raw dictionary task graphs. This allows for a much more efficient communication of task graphs from the client to the scheduler.
- Added support for using custom `Layer`-level annotations like `priority`, `retries`, etc. with the `dask.annotations` context manager.
- Updated minimum supported version of Dask to 2020.12.0.
- Added many type annotations and updates to allow for gradually Cythonizing the scheduler.

All changes

- Some common optimizations across transitions (GH#4348) jakirkham
- Drop serialize extension (GH#4344) jakirkham
- Log duplciate workers in scheduler (GH#4338) Matthew Rocklin
- Annotation of some comm related methods in the `Scheduler` (GH#4341) jakirkham
- Optimize `assert` in `validate_waiting` (GH#4342) jakirkham
- Optimize `decide_worker` (GH#4332) jakirkham
- Store occupancy in `_reevaluate_occupancy_worker` (GH#4337) jakirkham
- Handle `WorkerState` `memory_limit` of `None` (GH#4335) jakirkham
- Use `bint` to annotate boolean attributes (GH#4334) jakirkham

- Optionally use offload executor in worker (GH#4307) Matthew Rocklin
- Optimize `send_task_to_worker` (GH#4331) jakirkham
- Optimize `valid_workers` (GH#4329) jakirkham
- Store occupancy in `transition_waiting_processing` (GH#4330) jakirkham
- Optimize `get_comm_cost` (GH#4328) jakirkham
- Use `.pop(...)` to remove `key` (GH#4327) jakirkham
- Use `operator.attrgetter` on `WorkerState.address` (GH#4324) jakirkham
- Annotate `Task*` objects for Cythonization (GH#4302) jakirkham
- Ensure `retire_workers` always `return` a `dict` (GH#4323) jakirkham
- Some Cython fixes for `WorkerState` (GH#4321) jakirkham
- Optimize `WorkerState.__eq__` (GH#4320) jakirkham
- Swap order of `TaskGroup` and `TaskPrefix` (GH#4319) jakirkham
- Check traceback object can be unpickled (GH#4299) jakirkham
- Move `TaskGroup` & `TaskPrefix` before *TaskState* (GH#4318) jakirkham
- Remove empty `test_highgraph.py` file (GH#4313) James Bourbeau
- Ensure that `retire_workers` returns a `dict` (GH#4315) Matthew Rocklin
- Annotate `WorkerState` for Cythonization (GH#4294) jakirkham
- Close `comm` on low-level errors (GH#4239) jochen-ott-by
- Coerce new `TaskState.nbytes` value to `int` (GH#4311) jakirkham
- Remove offload `try`/`except` for `thread_name_prefix` keyword (GH#4308) James Bourbeau
- Fix `pip` install issue on CI (GH#4310) jakirkham
- Transmit `Layer` annotations to scheduler (GH#4279) Simon Perkins
- Ignores any compiled files generated by Cython (GH#4301) jakirkham
- Protect against missing key in `get_metrics` (GH#4300) Matthew Rocklin
- Provide option to build Distributed with Cython (GH#4292) jakirkham
- Set `WorkerState.processing` w/`dict` in `clean` (GH#4295) jakirkham
- Annotate `ClientState` for Cythonization (GH#4290) jakirkham
- Annotate `check_idle_saturated` for Cythonization (GH#4289) jakirkham
- Avoid flicker in `TaskStream` with "Scheduler is empty" message (GH#4284) Matthew Rocklin
- Make `gather_dep` robust to missing tasks (GH#4285) Matthew Rocklin
- Annotate `extract_serialize` (for Cythonization) (GH#4283) jakirkham
- Move `nbytes` from Worker's state to `TaskState` (GH#4274) Gil Forsyth
- Drop extra type check in `_extract_serialize` (GH#4281) jakirkham
- Move Status to top-level import (GH#4280) Matthew Rocklin
- Add `__hash__` and `__eq__` for `TaskState` (GH#4278) jakirkham
- Add `__hash__` and `__eq__` for `ClientState` (GH#4276) jakirkham

- Collect `report`'s `client_key``s in a ``list` (GH#4275) jakirkham
- Precompute `hash` for `WorkerState` (GH#4271) jakirkham
- Use `Status Enum` in `remove_worker` (GH#4269) jakirkham
- Add aggregated topic logs and `log_event` method (GH#4230) James Bourbeau
- Find the set of workers instead of their frequency (GH#4267) jakirkham
- Use `set.update` to include other `comms` (GH#4268) jakirkham
- Support string timeouts in `sync` (GH#4266) James Bourbeau
- Use `dask.utils.stringify()` instead of `distributed.utils.tokey()` (GH#4255) Mads R. B. Kristensen
- Use `.items()` to walk through keys and values (GH#4261) jakirkham
- Simplify frame length packing in TCP write (GH#4257) jakirkham
- Comm/tcp listener: do not pass comm with failed handshake to `comm_handler` (GH#4240) jochen-ott-by
- Fuse steps in `extract_serialize` (GH#4254) jakirkham
- Drop `test_sklearn` (GH#4253) jakirkham
- Document task priority tie breaking (GH#4252) James Bourbeau
- `__dask_distributed_pack__()`: client argument (GH#4248) Mads R. B. Kristensen
- Configurable timeouts for `worker_client` and `get_client` (GH#4146) GeethanjaliEswaran
- Add dask/distributed versions to `performance_report` (GH#4249) Matthew Rocklin
- Update miniconda GitHub action (GH#4250) James Bourbeau
- UCX closing ignore error (GH#4236) Mads R. B. Kristensen
- Redirect to `dask-worker` cli documentation (GH#4247) Timost
- Upload file worker plugin (GH#4238) Ian Rose
- Create dependency `TaskState` as needed in `gather_dep` (GH#4241) Gil Forsyth
- Instantiate plugin if needed in `register_worker_plugin` (GH#4198) Julia Signell
- Allow actors to call actors on the same worker (GH#4225) Martin Durant
- Special case profile thread in leaked thread check (GH#4229) James Bourbeau
- Use `intersection()` on a set instead of `dict_keys` in `update_graph` (GH#4227) Mads R. B. Kristensen
- Communicate `HighLevelGraphs` directly to the `Scheduler` (GH#4140) Mads R. B. Kristensen
- Add `get_task_metadata` context manager (GH#4216) James Bourbeau
- Task state logs and data fix (GH#4206) Gil Forsyth
- Send active task durations from worker to scheduler (GH#4192) James Bourbeau
- Fix state check in `test_close_gracefully` (GH#4203) Gil Forsyth
- Avoid materializing layers in `Client.compute()` (GH#4196) Mads R. B. Kristensen
- Add `TaskState` metadata (GH#4191) James Bourbeau
- Fix regression in task stealing for already released keys (GH#4182) Florian Jetter
- Fix `_graph_to_futures` bug for futures-based dependencies (GH#4178) Richard J Zamora

- High level graph `dumps/loads` support (GH#4174) Mads R. B. Kristensen
- Implement pass HighLevelGraphs through `_graph_to_futures` (GH#4139) Mads R. B. Kristensen
- Support `async` preload click commands (GH#4170) James Bourbeau
- `dask-worker` cli memory limit option doc fix (GH#4172) marwan116
- Add `TaskState` to `worker.py` (GH#4107) Gil Forsyth
- Increase robustness of `Semaphore.release` (GH#4151) Lucas Rademaker
- Skip batched comm test win / tornado5 (GH#4166) Tom Augspurger
- Set Zict buffer target to maxsize when `memory_target_fraction` is `False` (GH#4156) Krishan Bhasin
- Add `PipInstall WorkerPlugin` (GH#3216) Matthew Rocklin
- Log `KilledWorker` events in the scheduler (GH#4157) Matthew Rocklin
- Fix `test_gpu_metrics` failure (GH#4154) jakirkham

3.30.10 2.30.1 - 2020-11-03

- Pin `pytest-asyncio` version (GH#4212) James Bourbeau
- Replace `AsyncProcess` exit handler by `weakref.finalize` (GH#4184) Peter Andreas Entschev
- Remove hard coded connect handshake timeouts (GH#4176) Florian Jetter

3.30.11 2.30.0 - 2020-10-06

- Support `SubgraphCallable` in `str_graph()` (GH#4148) Mads R. B. Kristensen
- Handle exceptions in `BatchedSend` (GH#4135) Tom Augspurger
- Fix for missing `:` in autosummary docs (GH#4143) Gil Forsyth
- Limit GPU metrics to visible devices only (GH#3810) Jacob Tomlinson

3.30.12 2.29.0 - 2020-10-02

- Use `pandas.testing` (GH#4138) jakirkham
- Fix a few typos (GH#4131) Pav A
- Return right away in `Cluster.close` if cluster is already closed (GH#4116) Tom Rochette
- Update async doc with example on `.compute()` vs `client.compute()` (GH#4137) Benjamin Zaitlen
- Correctly tear down `LoopRunner` in `Client` (GH#4112) Sergey Kozlov
- Simplify `Client._graph_to_futures()` (GH#4127) Mads R. B. Kristensen
- Cleanup new exception traceback (GH#4125) Krishan Bhasin
- Stop writing config files by default (GH#4123) Matthew Rocklin

3.30.13 2.28.0 - 2020-09-25

- Fix SSL `connection_args` for `progressbar` connect (GH#4122) jennalc

3.30.14 2.27.0 - 2020-09-18

- Fix registering a worker plugin with `name` arg (GH#4105) Nick Evans
- Support different `remote_python` paths on cluster nodes (GH#4085) Abdulelah Bin Mahfoodh
- Allow `RuntimeError` s when closing global clients (GH#4115) Matthew Rocklin
- Match `pre-commit` in dask (GH#4049) Julia Signell
- Update `super` usage (GH#4110) Poruri Sai Rahul

3.30.15 2.26.0 - 2020-09-11

- Add logging for adaptive start and stop (GH#4101) Matthew Rocklin
- Don't close a nannied worker if it hasn't yet started (GH#4093) Matthew Rocklin
- Respect timeouts when closing clients synchronously (GH#4096) Matthew Rocklin
- Log when downloading a preload script (GH#4094) Matthew Rocklin
- `dask-worker --nprocs` accepts negative values (GH#4089) Dror Speiser
- Support zero-worker clients (GH#4090) Matthew Rocklin
- Exclude `fire-and-forget` client from metrics (GH#4078) Tom Augspurger
- Drop `Serialized.deserialize()` method (GH#4073) jakirkham
- Add `timeout=` keyword to `Client.wait_for_workers` method (GH#4087) Matthew Rocklin

3.30.16 2.25.0 - 2020-08-28

- Update for black (GH#4081) Tom Augspurger
- Provide informative error when connecting an older version of Dask (GH#4076) Matthew Rocklin
- Simplify `pack_frames` (GH#4068) jakirkham
- Simplify `frame_split_size` (GH#4067) jakirkham
- Use `list.insert` to add prelude up front (GH#4066) jakirkham
- Graph helper text (GH#4064) Julia Signell
- Graph dashboard: Reset container data if task number is too large (GH#4056) Florian Jetter
- Ensure semaphore picks correct `IOLoop` for threadpool workers (GH#4060) Florian Jetter
- Add cluster log method (GH#4051) Jacob Tomlinson
- Cleanup more exception tracebacks (GH#4054) Krishan Bhasin
- Improve documentation of `scheduler.locks` options (GH#4062) Florian Jetter

3.30.17 2.24.0 - 2020-08-22

- Move toolbar to above and fix y axis (#4043) Julia Signell
- Make behavior clearer for how to get worker dashboard (#4047) Julia Signell
- Worker dashboard clean up (#4046) Julia Signell
- Add a default argument to the datasets and a possibility to override datasets (#4052) Nils Braun
- Discover HTTP endpoints (#3744) Martin Durant

3.30.18 2.23.0 - 2020-08-14

- Tidy up exception traceback in TCP Comms (GH#4042) Krishan Bhasin
- Angle on the x-axis labels (GH#4030) Mathieu Dugré
- Always set RMM's strides in the `header` (GH#4039) jakirkham
- Fix documentation `upload_file` (GH#4038) Roberto Panai
- Update UCX tests for new handshake step (GH#4036) jakirkham
- Add test for informative errors in serialization cases (GH#4029) Matthew Rocklin
- Add compression, pickle protocol to comm contexts (GH#4019) Matthew Rocklin
- Make GPU plots robust to not having GPUs (GH#4008) Matthew Rocklin
- Update `PendingDeprecationWarning` with correct version number (GH#4025) Matthias Bussonnier
- Install PyTorch on CI (GH#4017) jakirkham
- Try getting cluster `dashboard_link` before asking scheduler (GH#4018) Matthew Rocklin
- Ignore writeable frames with builtin `array` (GH#4016) jakirkham
- Just extend `frames2` by `frames` (GH#4015) jakirkham
- Serialize builtin array (GH#4013) jakirkham
- Use cuDF's `assert_eq` (GH#4014) jakirkham
- Clear function cache whenever we upload a new file (GH#3993) Jack Xiaosong Xu
- Emmit warning when assign/comparing string with `Status Enum` (GH#3875) Matthias Bussonnier
- Track mutable frames (GH#4004) jakirkham
- Improve `bytes` and `bytearray` serialization (GH#4009) jakirkham
- Fix memory histogram values in dashboard (GH#4006) Willi Rath

3.30.19 2.22.0 - 2020-07-31

- Only call `frame_split_size` when there are frames (GH#3996) jakirkham
- Fix failing `test_bandwidth` (GH#3999) jakirkham
- Handle sum of memory percentage when `memory_limit` is 0 (GH#3984) Julia Signell
- Drop msgpack pre-0.5.2 compat code (GH#3977) jakirkham
- Revert to localhost for local IP if no network available (GH#3991) Matthew Rocklin
- Add missing backtick in inline directive. (GH#3988) Matthias Bussonnier
- Warn when `threads_per_worker` is set to zero (GH#3986) Julia Signell
- Use `memoryview` in `unpack_frames` (GH#3980) jakirkham
- Iterate over list of comms (GH#3959) Matthew Rocklin
- Streamline `pack_frames`/`unpack_frames` frames (GH#3973) jakirkham
- Always attempt to create `dask-worker-space` folder and continue if it exists (GH#3972) Jendrik Jördening
- Use `merge_frames` with host memory only (GH#3971) jakirkham
- Simplify `pack_frames_prelude` (GH#3961) jakirkham
- Use continuation prompt for proper example parsing (GH#3966) Matthias Bussonnier
- Ensure writable frames (GH#3967) jakirkham

3.30.20 2.21.0 - 2020-07-17

- Fix data replication error (GH#3963) Andrew Fulton
- Treat falsey local directory as `None` (GH#3964) Tom Augspurger
- Unpin `numpydoc` now that 1.1 is released (GH#3957) Gil Forsyth
- Error hard when Dask has mismatched versions or lz4 installed (GH#3936) Matthew Rocklin
- Skip coercing to `bytes` in `merge_frames` (GH#3960) jakirkham
- UCX: reuse endpoints in order to fix NVLINK issue (GH#3953) Mads R. B. Kristensen
- Optionally use `pickle5` (GH#3849) jakirkham
- Update time per task chart with filtering and pie (GH#3933) Benjamin Zaitlen
- UCX: explicit shutdown message (GH#3950) Mads R. B. Kristensen
- Avoid too aggressive retry of connections (GH#3944) Matthias Bussonnier
- Parse timeouts in `Client.sync` (GH#3952) Matthew Rocklin
- Synchronize on non-trivial CUDA frame transmission (GH#3949) jakirkham
- Serialize `memoryview` with `shape` and `format` (GH#3947) jakirkham
- Move `scheduler_comm` into `Cluster.__init__` (GH#3945) Matthew Rocklin

3.30.21 2.20.0 - 2020-07-02

- Link issue on using `async` with `executor_submit` (GH#3939) jakirkham
- Make dashboard server listens on all IPs by default even when interface is set explicitly (GH#3941) Loïc Estève
- Update logic for worker removal in check ttl (GH#3927) Benjamin Zaitlen
- Close a created cluster quietly (GH#3935) Matthew Rocklin
- Ensure `Worker.run*` handles `kwargs` correctly (GH#3937) jakirkham
- Restore `Scheduler.time_started` for Dask Gateway (GH#3934) Tom Augspurger
- Fix exception handling in `_wait_until_connected` (GH#3912) Alexander Clausen
- Make local directory if it does not exist (GH#3928) Matthew Rocklin
- Install vanilla status route if bokeh dependency is not satisfied (GH#3844) joshreback
- Make `Worker.delete_data` sync (GH#3922) Peter Andreas Entschev
- Fix `ensure_bytes` import location (GH#3919) jakirkham
- Fix race condition in repeated calls to `cluster.adapt()` (GH#3915) Jacob Tomlinson

3.30.22 2.19.0 - 2020-06-19

- Notify worker plugins when a task is released (GH#3817) Nick Evans
- Update heartbeat checks in scheduler (GH#3896) Benjamin Zaitlen
- Make encryption default if `Security` is given arguments (GH#3887) Matthew Rocklin
- Show `cpu_fraction` on hover for dashboard workers circle plot. (GH#3906) Loïc Estève
- Prune virtual client on variable deletion (GH#3910) Marco Neumann
- Fix total aggregated metrics in dashboard (GH#3897) Loïc Estève
- Support Bokeh 2.1 (GH#3904) Matthew Rocklin
- Update `related-work.rst` (GH#3889) DomHudson
- Skip `test_pid_file` in older versions of Python (GH#3888) Matthew Rocklin
- Replace `stream=` with `comm=` in handlers (GH#3860) Julien Jerphanion
- Check hosts for `None` value in SSH cluster. (GH#3883) Matthias Bussonnier
- Allow dictionaries in `security=` keywords (GH#3874) Matthew Rocklin
- Use pickle protocol 5 with NumPy object arrays (GH#3871) jakirkham
- Cast any `frame` to `uint8` (same type as `bytes`) (GH#3870) jakirkham
- Use `Enum` for worker, scheduler and nanny status. (GH#3853) Matthias Bussonnier
- Drop legacy `buffer_interface` assignment (GH#3869) jakirkham
- Drop old frame splitting in NumPy serialization (GH#3868) jakirkham
- Drop no longer needed local `import pickle` (GH#3865) jakirkham
- Fix typo in `feed`'s log message (GH#3867) jakirkham
- Tidy pickle (GH#3866) jakirkham

- Handle empty times in task stream (GH#3862) Benjamin Zaitlen
- Change `asyncssh` objects to sphinx references (GH#3861) Jacob Tomlinson
- Improve `SSHCluster` docstring for `connect_options` (GH#3859) Jacob Tomlinson
- Validate address parameter in client constructor (GH#3842) joshreback
- Use `SpecCluster` name in worker names (GH#3855) Loïc Estève
- Allow async `add_worker` and `remove_worker` plugin methods (GH#3847) James Bourbeau

3.30.23 2.18.0 - 2020-06-05

- Merge frames in `deserialize_bytes` (GH#3639) John Kirkham
- Allow `SSHCluster` to take a list of `connect_options` (GH#3854) Jacob Tomlinson
- Add favicon to performance report (GH#3852) Jacob Tomlinson
- Add dashboard plots for the amount of time spent per key and for transfer/serialization (GH#3792) Benjamin Zaitlen
- Fix variable name in journey of a task documentation (GH#3840) Matthias Bussonnier
- Fix typo in journey of a task doc (GH#3838) James Bourbeau
- Register `dask_cudf` serializers (GH#3832) John Kirkham
- Fix key check in `rebalance` missing keys (GH#3834) Jacob Tomlinson
- Allow collection of partial profile information in case of exceptions (GH#3773) Florian Jetter

3.30.24 2.17.0 - 2020-05-26

- Record the time since the last run task on the scheduler (GH#3830) Matthew Rocklin
- Set colour of `nbytes` pane based on thresholds (GH#3805) Krishan Bhasin
- Include total number of tasks in the performance report (GH#3822) Abdulelah Bin Mahfoodh
- Allow to pass in task key strings in the worker restrictions (GH#3826) Nils Braun
- Control de/ser offload (GH#3793) Martin Durant
- Parse timeout parameters in `Variable`/`Event`/`Lock` to support text timeouts (GH#3825) Nils Braun
- Don't send empty dependencies (GH#3423) Jakub Beránek
- Add distributed Dask `Event` that mimics `threading.Event` (GH#3821) Nils Braun
- Enhance `VersionMismatchWarning` messages (GH#3786) Abdulelah Bin Mahfoodh
- Support Pickle's protocol 5 (GH#3784) jakirkham
- Replace `utils.ignoring` with `contextlib.suppress` (GH#3819) Nils Braun
- Make re-creating conda environments from the CI output easier (GH#3816) Lucas Rademaker
- Add prometheus metrics for semaphore (GH#3757) Lucas Rademaker
- Fix worker plugin called with superseded transition (GH#3812) Nick Evans
- Add retries to server listen (GH#3801) Jacob Tomlinson
- Remove commented out lines from `scheduler.py` (GH#3803) James Bourbeau

- Fix `RuntimeWarning` for never awaited coroutine when using `distributed.Semaphore` (GH#3713) Florian Jetter
- Fix profile thread leakage during test teardown on some platforms (GH#3795) Florian Jetter
- Await self before handling comms (GH#3788) Matthew Rocklin
- Fix typo in `Cluster` docstring (GH#3787) Scott Sanderson

3.30.25 2.16.0 - 2020-05-08

- `Client.get_dataset` to always create `Futures` attached to itself (GH#3729) crusaderky
- Remove dev-requirements since it is unused (GH#3782) Julia Signell
- Use bokeh column for `/system` instead of custom css (GH#3781) Julia Signell
- Attempt to fix `test_preload_remote_module` on windows (GH#3775) James Bourbeau
- Fix broadcast for TLS comms (GH#3766) Florian Jetter
- Don't validate http preloads locally (GH#3768) Rami Chowdhury
- Allow range of ports to be specified for `Workers` (GH#3704) James Bourbeau
- Add UCX support for RDMACM (GH#3759) Peter Andreas Entschev
- Support web addresses in preload (GH#3755) Matthew Rocklin

3.30.26 2.15.2 - 2020-05-01

- Connect to dashboard when address provided (GH#3758) Tom Augspurger
- Move `test_gpu_metrics test` (GH#3721) Tom Augspurger
- Nanny closing worker on `KeyboardInterrupt` (GH#3747) Mads R. B. Kristensen
- Replace `OrderedDict` with `dict` in scheduler (GH#3740) Matthew Rocklin
- Fix exception handling typo (GH#3751) Jonas Haag

3.30.27 2.15.1 - 2020-04-28

- Ensure `BokehTornado` uses prefix (GH#3746) James Bourbeau
- Warn if cluster closes before starting (GH#3735) Matthew Rocklin
- Memoryview serialisation (GH#3743) Martin Durant
- Allows logging config under distributed key (GH#2952) Dillon Niederhut

3.30.28 2.15.0 - 2020-04-24

- Reinstate support for legacy `@gen_cluster` functions (GH#3738) crusaderky
- Relax NumPy requirement in UCX (GH#3731) jakirkham
- Add Configuration Schema (GH#3696) Matthew Rocklin
- Reuse CI scripts for local installation process (GH#3698) crusaderky
- Use `PeriodicCallback` class from tornado (GH#3725) James Bourbeau
- Add `remote_python` option in ssh cmd (GH#3709) Abdulelah Bin Mahfoodh
- Configurable polling interval for cluster widget (GH#3723) Julia Signell
- Fix copy-paste in docs (GH#3728) Julia Signell
- Replace `gen.coroutine` with async-await in tests (GH#3706) crusaderky
- Fix flaky `test_oversubscribing_leases` (GH#3726) Florian Jetter
- Add `batch_size` to `Client.map` (GH#3650) Tom Augspurger
- Adjust semaphore test timeouts (GH#3720) Florian Jetter
- Dask-serialize dicts longer than five elements (GH#3689) Richard J Zamora
- Force `threads_per_worker` (GH#3715) crusaderky
- Idempotent semaphore acquire with retries (GH#3690) Florian Jetter
- Always use `readinto` in TCP (GH#3711) jakirkham
- Avoid `DeprecationWarning` from pandas (GH#3712) Tom Augspurger
- Allow modification of `distributed.comm.retry` at runtime (GH#3705) Florian Jetter
- Do not log an error on unset variable delete (GH#3652) Jonathan J. Helmus
- Add `remote_python` keyword to the new `SSHCluster` (GH#3701) Abdulelah Bin Mahfoodh
- Replace Example with Examples in docstrings (GH#3697) Matthew Rocklin
- Add `Cluster` `__enter__` and `__exit__` methods (GH#3699) Matthew Rocklin
- Fix propagating inherit config in `SSHCluster` for non-bash shells (GH#3688) Abdulelah Bin Mahfoodh
- Add `Client.wait_to_workers` to `Client` autosummary table (GH#3692) James Bourbeau
- Replace Bokeh Server with Tornado HTTPServer (GH#3658) Matthew Rocklin
- Fix `dask-ssh` after removing `local-directory` from `dask_scheduler` cli (GH#3684) Abdulelah Bin Mahfoodh
- Support preload modules in `Nanny` (GH#3678) Matthew Rocklin
- Refactor semaphore internals: make `_get_lease` synchronous (GH#3679) Lucas Rademaker
- Don't make task graphs too big (GH#3671) Martin Durant
- Pass through `connection`/`listen_args` as splatted keywords (GH#3674) Matthew Rocklin
- Run preload at import, start, and teardown (GH#3673) Matthew Rocklin
- Use relative URL in scheduler dashboard (GH#3676) Nicholas Smith
- Expose `Security` object as public API (GH#3675) Matthew Rocklin
- Add zoom tools to profile plots (GH#3672) James Bourbeau

- Update `Scheduler.rebalance` return value when data is missing (GH#3670) James Bourbeau

3.30.29 2.14.0 - 2020-04-03

- Enable more UCX tests (GH#3667) jakirkham
- Remove openssl 1.1.1d pin for Travis (GH#3668) Jonathan J. Helmus
- More documentation for `Semaphore` (GH#3664) Florian Jetter
- Get CUDA context to finalize Numba `DeviceNDArray` (GH#3666) jakirkham
- Add Resouces option to `get_task_stream` and call `output_file` (GH#3653) Prasun Anand
- Add `Semaphore` extension (GH#3573) Lucas Rademaker
- Replace `ncores` with `nthreads` in work stealing tests (GH#3615) James Bourbeau
- Clean up some test warnings (GH#3662) Matthew Rocklin
- Write "why killed" docs (GH#3596) Martin Durant
- Update Python version checking (GH#3660) James Bourbeau
- Add newlines to ensure code formatting for `retire_workers` (GH#3661) Rami Chowdhury
- Clean up performance report test (GH#3655) Matthew Rocklin
- Avoid diagnostics time in performance report (GH#3654) Matthew Rocklin
- Introduce config for default task duration (GH#3642) Gabriel Sailer
- UCX simplify receiving frames in `comm` (GH#3651) jakirkham
- Bump checkout GitHub action to v2 (GH#3649) James Bourbeau
- Handle exception in `faulthandler` (GH#3646) Jacob Tomlinson
- Add prometheus metric for suspicious tasks (GH#3550) Gabriel Sailer
- Remove `local-directory` keyword (GH#3620) Prasun Anand
- Don't create output Futures in Client when there are mixed Client Futures (GH#3643) James Bourbeau
- Add link to `contributing.md` (GH#3621) Prasun Anand
- Update bokeh dependency in CI builds (GH#3637) James Bourbeau

3.30.30 2.13.0 - 2020-03-25

- UCX synchronize default stream only on CUDA frames (GH#3638) Peter Andreas Entschev
- Add `as_completed.clear` method (GH#3617) Matthew Rocklin
- Drop unused line from `pack_frames_prelude` (GH#3634) John Kirkham
- Add logging message when closing idle dask scheduler (GH#3632) Matthew Rocklin
- Include frame lengths of CUDA objects in `header["lengths"]` (GH#3631) John Kirkham
- Ensure `Client` connection pool semaphore attaches to the `Client` event loop (GH#3546) James Bourbeau
- Remove dead stealing code (GH#3619) Florian Jetter
- Check `nbytes` and `types` before reading `data` (GH#3628) John Kirkham
- Ensure that we don't steal blacklisted fast tasks (GH#3591) Florian Jetter

- Support async `Listener.stop` functions (GH#3613) Matthew Rocklin
- Add str/repr methods to `as_completed` (GH#3618) Matthew Rocklin
- Add backoff to comm connect attempts. (GH#3496) Matthias Urlichs
- Make `Listeners` awaitable (GH#3611) Matthew Rocklin
- Increase number of visible mantissas in dashboard plots (GH#3585) Scott Sievert
- Pin openssl to 1.1.1d for Travis (GH#3602) Jacob Tomlinson
- Replace `tornado.queues` with `asyncio.queues` (GH#3607) James Bourbeau
- Remove `dill` from CI environments (GH#3608) Loïc Estève
- Fix linting errors (GH#3604) James Bourbeau
- Synchronize default CUDA stream before UCX send/recv (GH#3598) Peter Andreas Entschev
- Add configuration for `Adaptive` arguments (GH#3509) Gabriel Sailer
- Change `Adaptive` docs to reference `adaptive_target` (GH#3597) Julia Signell
- Optionally compress on a frame-by-frame basis (GH#3586) Matthew Rocklin
- Add Python version to version check (GH#3567) James Bourbeau
- Import `tlz` (GH#3579) John Kirkham
- Pin `numpydoc` to avoid double escaped `*` (GH#3530) Gil Forsyth
- Avoid `performance_report` crashing when a worker dies mid-compute (GH#3575) Krishan Bhasin
- Pin `bokeh` in CI builds (GH#3570) James Bourbeau
- Disable fast fail on GitHub Actions Windows CI (GH#3569) James Bourbeau
- Fix typo in `Client.shutdown` docstring (GH#3562) John Kirkham
- Add `local_directory` option to `dask-ssh` (GH#3554) Abdulelah Bin Mahfoodh

3.30.31 2.12.0 - 2020-03-06

- Update `TaskGroup` remove logic (GH#3557) James Bourbeau
- Fix-up CuPy sparse serialization (GH#3556) John Kirkham
- API docs for `LocalCluster` and `SpecCluster` (GH#3548) Tom Augspurger
- Serialize sparse arrays (GH#3545) John Kirkham
- Allow tasks with restrictions to be stolen (GH#3069) Stan Seibert
- Use UCX default configuration instead of raising (GH#3544) Peter Andreas Entschev
- Support using other serializers with `register_generic` (GH#3536) John Kirkham
- DOC: update to async await (GH#3543) Tom Augspurger
- Use `pytest.raises` in `test_ucx_config.py` (GH#3541) John Kirkham
- Fix/more ucx config options (GH#3539) Benjamin Zaitlen
- Update heartbeat `CommClosedError` error handling (GH#3529) James Bourbeau
- Use `makedirs` when constructing `local_directory` (GH#3538) John Kirkham
- Mark `None` as MessagePack serializable (GH#3537) John Kirkham

- Mark `bool` as MessagePack serializable (GH#3535) John Kirkham
- Use 'temporary-directory' from `dask.config` for Nanny's directory (GH#3531) John Kirkham
- Add try-except around getting source code in performance report (GH#3505) Matthew Rocklin
- Fix typo in docstring (GH#3528) Davis Bennett
- Make work stealing callback time configurable (GH#3523) Lucas Rademaker
- RMM/UCX Config Flags (GH#3515) Benjamin Zaitlen
- Revise develop-docs: conda env example (GH#3406) Darren Weber
- Remove `import ucp` from the top of `ucx.py` (GH#3510) Peter Andreas Entschev
- Rename `logs` to `get_logs` (GH#3473) Jacob Tomlinson
- Stop keep alives when worker reconnecting to the scheduler (GH#3493) Jacob Tomlinson

3.30.32 2.11.0 - 2020-02-19

- Add dask serialization of CUDA objects (GH#3482) John Kirkham
- Suppress cuML `ImportError` (GH#3499) John Kirkham
- Msgpack 1.0 compatibility (GH#3494) James Bourbeau
- Register cuML serializers (GH#3485) John Kirkham
- Check exact equality for worker state (GH#3483) Brett Naul
- Serialize 1-D, contiguous, `uint8` CUDA frames (GH#3475) John Kirkham
- Update NumPy array serialization to handle non-contiguous slices (GH#3474) James Bourbeau
- Propose fix for collection based resources docs (GH#3480) Chris Roat
- Remove `--verbose` flag from CI runs (GH#3484) Matthew Rocklin
- Do not duplicate messages in scheduler report (GH#3477) Jakub Beránek
- Register Dask cuDF serializers (GH#3478) John Kirkham
- Add support for Python 3.8 (GH#3249) James Bourbeau
- Add last seen column to worker table and highlight errant workers (GH#3468) kaelgreco
- Change default value of `local_directory` from empty string to `None` (GH#3441) condoratberlin
- Clear old docs (GH#3458) Matthew Rocklin
- Change default multiprocessing behavior to spawn (GH#3461) Matthew Rocklin
- Split dashboard host on additional slashes to handle inproc (GH#3466) Jacob Tomlinson
- Update `locality.rst` (GH#3470) Dustin Tindall
- Minor `gen.Return` cleanup (GH#3469) James Bourbeau
- Update comparison logic for worker state (GH#3321) rockwellw
- Update minimum `tblib` version to 1.6.0 (GH#3451) James Bourbeau
- Add total row to workers plot in dashboard (GH#3464) Julia Signell
- Workaround `RecursionError` on profile data (GH#3455) Tom Augspurger
- Include code and summary in performance report (GH#3462) Matthew Rocklin

- Skip `test_open_close_many_workers` on Python 3.6 (GH#3459) Matthew Rocklin
- Support serializing/deserializing `rmm.DeviceBuffer` s (GH#3442) John Kirkham
- Always add new `TaskGroup` to `TaskPrefix` (GH#3322) James Bourbeau
- Rerun `black` on the code base (GH#3444) John Kirkham
- Ensure `__causes__` s of exceptions raised on workers are serialized (GH#3430) Alex Adamson
- Adjust `numba.cuda` import and add check (GH#3446) John Kirkham
- Fix name of Numba serialization test (GH#3447) John Kirkham
- Checks for command parameters in `ssh2` (GH#3078) Peter Andreas Entschev
- Update `worker_kwargs` description in `LocalCluster` constructor (GH#3438) James Bourbeau
- Ensure scheduler updates task and worker states after successful worker data deletion (GH#3401) James Bourbeau
- Avoid `loop=` keyword in asyncio coordination primitives (GH#3437) Matthew Rocklin
- Call pip as a module to avoid warnings (GH#3436) Cyril Shcherbin
- Add documentation of parameters in coordination primitives (GH#3434) Søren Fuglede Jørgensen
- Replace `tornado.locks` with asyncio for Events/Locks/Conditions/Semaphore (GH#3397) Matthew Rocklin
- Remove object from class hierarchy (GH#3432) Anderson Banihirwe
- Add `dashboard_link` property to `Client` (GH#3429) Jacob Tomlinson
- Allow memory monitor to evict data more aggressively (GH#3424) fjetter
- Make `_get_ip` return an IP address when defaulting (GH#3418) Pierre Glaser
- Support version checking with older versions of Dask (GH#3390) Igor Gotlibovych
- Add Mac OS build to CI (GH#3358) James Bourbeau

3.30.33 2.10.0 - 2020-01-28

- Fixed `ZeroDivisionError` in dashboard when no workers were present (GH#3407) James Bourbeau
- Respect the `dashboard-prefix` when redirecting from the root (GH#3387) Chrysostomos Nanakos
- Allow enabling / disabling work-stealing after the cluster has started (GH#3410) John Kirkham
- Support `*args` and `**kwargs` in offload (GH#3392) Matthew Rocklin
- Add lifecycle hooks to SchedulerPlugin (GH#3391) Matthew Rocklin

3.30.34 2.9.3 - 2020-01-17

- Raise `RuntimeError` if no running loop (GH#3385) James Bourbeau
- Fix `get_running_loop` import (GH#3383) James Bourbeau
- Get JavaScript document location instead of window and handle proxied url (GH#3382) Jacob Tomlinson

3.30.35 2.9.2 - 2020-01-16

- Move Windows CI to GitHub Actions (GH#3373) Jacob Tomlinson
- Add client join and leave hooks (GH#3371) Jacob Tomlinson
- Add cluster map dashboard (GH#3361) Jacob Tomlinson
- Close connection comm on retry (GH#3365) James Bourbeau
- Fix scheduler state in case of worker name collision (GH#3366) byjott
- Add `--worker-class` option to `dask-worker` CLI (GH#3364) James Bourbeau
- Remove `locale` check that fails on OS X (GH#3360) Jacob Tomlinson
- Rework version checking (GH#2627) Matthew Rocklin
- Add websocket scheduler plugin (GH#3335) Jacob Tomlinson
- Return task in `dask-worker` `on_signal` function (GH#3354) James Bourbeau
- Fix failures on mixed integer/string worker names (GH#3352) Benedikt Reinartz
- Avoid calling `nbytes` multiple times when sending data (GH#3349) Markus Mohrhard
- Avoid setting event loop policy if within IPython kernel and no running event loop (GH#3336) Mana Borwornpadungkitti
- Relax intermittent failing `test_profile_server` (GH#3346) Matthew Rocklin

3.30.36 2.9.1 - 2019-12-27

- Add lock around dumps_function cache (GH#3337) Matthew Rocklin
- Add setuptools to dependencies (GH#3320) James Bourbeau
- Use TaskPrefix.name in Graph layout (GH#3328) Matthew Rocklin
- Add missing " in performance report example (GH#3329) John Kirkham
- Add performance report docs and color definitions to docs (GH#3325) Benjamin Zaitlen
- Switch startstops to dicts and add worker name to transfer (GH#3319) Jacob Tomlinson
- Add plugin entry point for out-of-tree comms library (GH#3305) Patrick Sodré
- All scheduler task states in prometheus (GH#3307) fjetter
- Use worker name in logs (GH#3309) Stephan Erb
- Add TaskGroup and TaskPrefix scheduler state (GH#3262) Matthew Rocklin
- Update latencies with heartbeats (GH#3310) fjetter
- Update inlining Futures in task graph in Client._graph_to_futures (GH#3303) James Bourbeau
- Use hostname as default IP address rather than localhost (GH#3308) Matthew Rocklin

- Clean up flaky test_nanny_throttle (GH#3295) Tom Augspurger
- Add lock to scheduler for sensitive operations (GH#3259) Matthew Rocklin
- Log address for each of the Scheduler listerners (GH#3306) Matthew Rocklin
- Make ConnectionPool.close asynchronous (GH#3304) Matthew Rocklin

3.30.37 2.9.0 - 2019-12-06

- Add `dask-spec` CLI tool (GH#3090) Matthew Rocklin
- Connectionpool: don't hand out closed connections (GH#3301) byjott
- Retry operations on network issues (GH#3294) byjott
- Skip `Security.temporary()` tests if cryptography not installed (GH#3302) James Bourbeau
- Support multiple listeners in the scheduler (GH#3288) Matthew Rocklin
- Updates RMM comment to the correct release (GH#3299) John Kirkham
- Add title to `performance_report` (GH#3298) Matthew Rocklin
- Forgot to fix slow test (GH#3297) Benjamin Zaitlen
- Update `SSHCluster` docstring parameters (GH#3296) James Bourbeau
- `worker.close()` awaits `batched_stream.close()` (GH#3291) Mads R. B. Kristensen
- Fix asynchronous listener in UCX (GH#3292) Benjamin Zaitlen
- Avoid repeatedly adding deps to already in memory stack (GH#3293) James Bourbeau
- xfail ucx empty object typed dataframe (GH#3279) Benjamin Zaitlen
- Fix `distributed.wait` documentation (GH#3289) Tom Rochette
- Move Python 3 syntax tests into main tests (GH#3281) Matthew Rocklin
- xfail `test_workspace_concurrency` for Python 3.6 (GH#3283) Matthew Rocklin
- Add `performance_report` context manager for static report generation (GH#3282) Matthew Rocklin
- Update function serialization caches with custom LRU class (GH#3260) James Bourbeau
- Make `Listener.start` asynchronous (GH#3278) Matthew Rocklin
- Remove `dask-submit` and `dask-remote` (GH#3280) Matthew Rocklin
- Worker profile server (GH#3274) Matthew Rocklin
- Improve bandwidth workers plot (GH#3273) Matthew Rocklin
- Make profile coroutines consistent between `Scheduler` and `Worker` (GH#3277) Matthew Rocklin
- Enable saving profile information from server threads (GH#3271) Matthew Rocklin
- Remove memory use plot (GH#3269) Matthew Rocklin
- Add offload size to configuration (GH#3270) Matthew Rocklin
- Fix layout scaling on profile plots (GH#3268) Jacob Tomlinson
- Set `x_range` in CPU plot based on the number of threads (GH#3266) Matthew Rocklin
- Use base-2 values for byte-valued axes in dashboard (GH#3267) Matthew Rocklin
- Robust gather in case of connection failures (GH#3246) fjetter

- Use `DeviceBuffer` from newer RMM releases (GH#3261) John Kirkham
- Fix dev requirements for pytest (GH#3264) Elliott Sales de Andrade
- Add validate options to configuration (GH#3258) Matthew Rocklin

3.30.38 2.8.1 - 2019-11-22

- Fix hanging worker when the scheduler leaves (GH#3250) Tom Augspurger
- Fix NumPy writeable serialization bug (GH#3253) James Bourbeau
- Skip `numba.cuda` tests if CUDA is not available (GH#3255) Peter Andreas Entschev
- Add new dashboard plot for memory use by key (GH#3243) Matthew Rocklin
- Fix `array.shape() -> array.shape` (GH#3247) Jed Brown
- Fixed typos in `pubsub.py` (GH#3244) He Jia
- Fixed cupy array going out of scope (GH#3240) Mads R. B. Kristensen
- Remove `gen.coroutine` usage in scheduler (GH#3242) Jim Crist-Harif
- Use `inspect.isawaitable` where relevant (GH#3241) Jim Crist-Harif

3.30.39 2.8.0 - 2019-11-14

- Add UCX config values (GH#3135) Matthew Rocklin
- Relax test_MultiWorker (GH#3210) Matthew Rocklin
- Avoid ucp.init at import time (GH#3211) Matthew Rocklin
- Clean up rpc to avoid intermittent test failure (GH#3215) Matthew Rocklin
- Respect protocol if given to Scheduler (GH#3212) Matthew Rocklin
- Use legend_field= keyword in bokeh plots (GH#3218) Matthew Rocklin
- Cache psutil.Process object in Nanny (GH#3207) Matthew Rocklin
- Replace gen.sleep with asyncio.sleep (GH#3208) Matthew Rocklin
- Avoid offloading serialization for small messages (GH#3224) Matthew Rocklin
- Add desired_workers metric (GH#3221) Gabriel Sailer
- Fail fast when importing distributed.comm.ucx (GH#3228) Matthew Rocklin
- Add module name to Future repr (GH#3231) Matthew Rocklin
- Add name to Pub/Sub repr (GH#3235) Matthew Rocklin
- Import CPU_COUNT from dask.system (GH#3199) James Bourbeau
- Efficiently serialize zero strided NumPy arrays (GH#3180) James Bourbeau
- Cache function deserialization in workers (GH#3234) Matthew Rocklin
- Respect ordering of futures in futures_of (GH#3236) Matthew Rocklin
- Bump dask dependency to 2.7.0 (GH#3237) James Bourbeau
- Avoid setting inf x_range (GH#3229) rockwellw
- Clear task stream based on recent behavior (GH#3200) Matthew Rocklin

- Use the percentage field for profile plots (GH#3238) Matthew Rocklin

3.30.40 2.7.0 - 2019-11-08

This release drops support for Python 3.5

- Adds badges to README.rst [skip ci] (GH#3152) James Bourbeau
- Don't overwrite *self.address* if it is present (GH#3153) Gil Forsyth
- Remove outdated references to debug scheduler and worker bokeh pages. (GH#3160) darindf
- Update CONTRIBUTING.md (GH#3159) Jacob Tomlinson
- Add Prometheus metric for a worker's executing tasks count (GH#3163) darindf
- Update Prometheus documentation (GH#3165) darindf
- Fix Numba serialization when strides is None (GH#3166) Peter Andreas Entschev
- Await cluster in Adaptive.recommendations (GH#3168) Simon Boothroyd
- Support automatic TLS (GH#3164) Jim Crist
- Avoid swamping high-memory workers with data requests (GH#3071) Tom Augspurger
- Update UCX variables to use sockcm by default (GH#3177) Peter Andreas Entschev
- Get protocol in Nanny/Worker from scheduler address (GH#3175) Peter Andreas Entschev
- Add worker and tasks state for Prometheus data collection (GH#3174) darindf
- Use async def functions for offload to/from_frames (GH#3171) Mads R. B. Kristensen
- Subprocesses inherit the global dask config (GH#3192) Mads R. B. Kristensen
- XFail test_open_close_many_workers (GH#3194) Matthew Rocklin
- Drop Python 3.5 (GH#3179) James Bourbeau
- UCX: avoid double init after fork (GH#3178) Mads R. B. Kristensen
- Silence warning when importing while offline (GH#3203) James A. Bednar
- Adds docs to Client methods for resources, actors, and traverse (GH#2851) IPetrik
- Add test for concurrent scatter operations (GH#2244) Matthew Rocklin
- Expand async docs (GH#2293) Dave Hirschfeld
- Add PatchedDeviceArray to drop stride attribute for cupy<7.0 (GH#3198) Richard J Zamora

3.30.41 2.6.0 - 2019-10-15

- Refactor dashboard module (GH#3138) Jacob Tomlinson
- Use `setuptools.find_packages` in `setup.py` (GH#3150) Matthew Rocklin
- Move death timeout logic up to `Node.start` (GH#3115) Matthew Rocklin
- Only include metric in `WorkerTable` if it is a scalar (GH#3140) Matthew Rocklin
- Add `Nanny(config={...})` keyword (GH#3134) Matthew Rocklin
- Xfail `test_worksapce_concurrency` on Python 3.6 (GH#3132) Matthew Rocklin
- Extend Worker plugin API with transition method (GH#2994) matthieubulte

- Raise exception if the user passes in unused keywords to `Client` (GH#3117) Jonathan De Troye
- Move new `SSHCluster` to top level (GH#3128) Matthew Rocklin
- Bump dask dependency (GH#3124) Jim Crist

3.30.42 2.5.2 - 2019-10-04

- Make dask-worker close quietly when given sigint signal (GH#3116) Matthew Rocklin
- Replace use of tornado.gen with asyncio in dask-worker (GH#3114) Matthew Rocklin
- UCX: allocate CUDA arrays using RMM and Numba (GH#3109) Mads R. B. Kristensen
- Support calling *cluster.scale* as async method (GH#3110) Jim Crist
- Identify lost workers in SpecCluster based on address not name (GH#3088) James Bourbeau
- Add Client.shutdown method (GH#3106) Matthew Rocklin
- Collect worker-worker and type bandwidth information (GH#3094) Matthew Rocklin
- Send noise over the wire to keep dask-ssh connection alive (GH#3105) Gil Forsyth
- Retry scheduler connect multiple times (GH#3104) Jacob Tomlinson
- Add favicon of logo to the dashboard (GH#3095) James Bourbeau
- Remove utils.py functions for their dask/utils.py equivalents (GH#3042) Matthew Rocklin
- Lower default bokeh log level (GH#3087) Philipp Rudiger
- Check if self.cluster.scheduler is a local scheduler (GH#3099) Jacob Tomlinson

3.30.43 2.5.1 - 2019-09-27

- Support clusters that don't have .security or ._close methods (GH#3100) Matthew Rocklin

3.30.44 2.5.0 - 2019-09-27

- Use the new UCX Python bindings (GH#3059) Mads R. B. Kristensen
- Fix worker preload config (GH#3027) byjott
- Fix widget with spec that generates multiple workers (GH#3067) Loïc Estève
- Make Client.get_versions async friendly (GH#3064) Jacob Tomlinson
- Add configuation option for longer error tracebacks (GH#3086) Daniel Farrell
- Have Client get Security from passed Cluster (GH#3079) Matthew Rocklin
- Respect Cluster.dashboard_link in Client._repr_html_ if it exists (GH#3077) Matthew Rocklin
- Add monitoring with dask cluster docs (GH#3072) Arpit Solanki
- Protocol of cupy and numba handles serialization exclusively (GH#3047) Mads R. B. Kristensen
- Allow specification of worker type in SSHCLuster (GH#3061) Jacob Tomlinson
- Use Cluster.scheduler_info for workers= value in repr (GH#3058) Matthew Rocklin
- Allow SpecCluster to scale by memory and cores (GH#3057) Matthew Rocklin

- Allow full script in preload inputs (GH#3052) Matthew Rocklin
- Check multiple cgroups dirs, ceil fractional cpus (GH#3056) Jim Crist
- Add blurb about disabling work stealing (GH#3055) Chris White

3.30.45 2.4.0 - 2019-09-13

- Remove six (GH#3045) Matthew Rocklin
- Add missing test data to sdist tarball (GH#3050) Elliott Sales de Andrade
- Use mock from unittest standard library (GH#3049) Elliott Sales de Andrade
- Use cgroups resource limits to determine default threads and memory (GH#3039) Jim Crist
- Move task deserialization to immediately before task execution (GH#3015) James Bourbeau
- Drop joblib shim module in distributed (GH#3040) John Kirkham
- Redirect configuration doc page (GH#3038) Matthew Rocklin
- Support `--name 0` and `--nprocs` keywords in dask-worker cli (GH#3037) Matthew Rocklin
- Remove lost workers from `SpecCluster.workers` (GH#2990) Guillaume Eynard-Bontemps
- Clean up `test_local.py::test_defaults` (GH#3017) Matthew Rocklin
- Replace print statement in `Queue.__init__` with debug message (GH#3035) Mikhail Akimov
- Set the `x_range` limit of the Meory utilization plot to memory-limit (GH#3034) Matthew Rocklin
- Rely on cudf codebase for cudf serialization (GH#2998) Benjamin Zaitlen
- Add fallback html repr for Cluster (GH#3023) Jim Crist
- Add support for zstandard compression to comms (GH#2970) Abael He
- Avoid collision when using `os.environ` in `dashboard_link` (GH#3021) Matthew Rocklin
- Fix `ConnectionPool` limit handling (GH#3005) byjott
- Support Spec jobs that generate multiple workers (GH#3013) Matthew Rocklin
- Tweak `Logs` styling (GH#3012) Jim Crist
- Better name for cudf deserialization function name (GH#3008) Benjamin Zaitlen
- Make `spec.ProcessInterface` a valid no-op worker (GH#3004) Matthew Rocklin
- Return dictionaries from `new_worker_spec` rather than name/worker pairs (GH#3000) Matthew Rocklin
- Fix minor typo in documentation (GH#3002) Mohammad Noor
- Permit more keyword options when scaling with cores and memory (GH#2997) Matthew Rocklin
- Add `cuda_ipc` to UCX environment for NVLink (GH#2996) Benjamin Zaitlen
- Add `threads=` and `memory=` to Cluster and Client reprs (GH#2995) Matthew Rocklin
- Fix PyNVML initialization (GH#2993) Richard J Zamora

3.30.46 2.3.2 - 2019-08-23

- Skip exceptions in startup information (GH#2991) Jacob Tomlinson

3.30.47 2.3.1 - 2019-08-22

- Add support for separate external address for SpecCluster scheduler (GH#2963) Jacob Tomlinson
- Defer cudf serialization/deserialization to that library (GH#2881) Benjamin Zaitlen
- Workaround for hanging test now calls ucp.fin() (GH#2967) Mads R. B. Kristensen
- Remove unnecessary bullet point (GH#2972) Pav A
- Directly import progress from diagnostics.progressbar (GH#2975) Matthew Rocklin
- Handle buffer protocol objects in ensure_bytes (GH#2969) Tom Augspurger
- Fix documentatation syntax and tree (GH#2981) Pav A
- Improve get_ip_interface error message when interface does not exist (GH#2964) Loïc Estève
- Add cores= and memory= keywords to scale (GH#2974) Matthew Rocklin
- Make workers robust to bad custom metrics (GH#2984) Matthew Rocklin

3.30.48 2.3.0 - 2019-08-16

- Except all exceptions when checking `pynvml` (GH#2961) Matthew Rocklin
- Pass serialization down through small base collections (GH#2948) Peter Andreas Entschev
- Use `pytest.warning(Warning)` rather than `Exception` (GH#2958) Matthew Rocklin
- Allow `server_kwargs` to override defaults in dashboard (GH#2955) Bruce Merry
- Update `utils_perf.py` (GH#2954) Shayan Amani
- Normalize names with `str` in `retire_workers` (GH#2949) Matthew Rocklin
- Update `client.py` (GH#2951) Shayan Amani
- Add `GPUCurrentLoad` dashboard plots (GH#2944) Matthew Rocklin
- Pass GPU diagnostics from worker to scheduler (GH#2932) Matthew Rocklin
- Import from `collections.abc` (GH#2938) Jim Crist
- Fixes Worker docstring formatting (GH#2939) James Bourbeau
- Redirect setup docs to docs.dask.org (GH#2936) Matthew Rocklin
- Wrap offload in `gen.coroutine` (GH#2934) Matthew Rocklin
- Change `TCP.close` to a coroutine to avoid task pending warning (GH#2930) Matthew Rocklin
- Fixup black string normalization (GH#2929) Jim Crist
- Move core functionality from `SpecCluster` to `Cluster` (GH#2913) Matthew Rocklin
- Add aenter/aexit protocols to `ProcessInterface` (GH#2927) Matthew Rocklin
- Add real-time CPU utilization plot to dashboard (GH#2922) Matthew Rocklin
- Always kill processes in clean tests, even if we don't check (GH#2924) Matthew Rocklin

- Add timeouts to processes in SSH tests (GH#2925) Matthew Rocklin
- Add documentation around `spec.ProcessInterface` (GH#2923) Matthew Rocklin
- Cleanup async warnings in tests (GH#2920) Matthew Rocklin
- Give 404 when requesting nonexistent tasks or workers (GH#2921) Martin Durant
- Raise informative warning when rescheduling an unknown task (GH#2916) James Bourbeau
- Fix docstring (GH#2917) Martin Durant
- Add keep-alive message between worker and scheduler (GH#2907) Matthew Rocklin
- Rewrite `Adaptive`/`SpecCluster` to support slowly arriving workers (GH#2904) Matthew Rocklin
- Call heartbeat rather than reconnect on disconnection (GH#2906) Matthew Rocklin

3.30.49 2.2.0 - 2019-07-31

- Respect security configuration in LocalCluster (GH#2822) Russ Bubley
- Add Nanny to worker docs (GH#2826) Christian Hudon
- Don't make False add-keys report to scheduler (GH#2421) tjb900
- Include type name in SpecCluster repr (GH#2834) Jacob Tomlinson
- Extend prometheus metrics endpoint (GH#2833) Gabriel Sailer
- Add alternative SSHCluster implementation (GH#2827) Matthew Rocklin
- Dont reuse closed worker in get_worker (GH#2841) Pierre Glaser
- SpecCluster: move init logic into start (GH#2850) Jacob Tomlinson
- Document distributed.Reschedule in API docs (GH#2860) James Bourbeau
- Add fsspec to installation of test builds (GH#2859) Martin Durant
- Make await/start more consistent across Scheduler/Worker/Nanny (GH#2831) Matthew Rocklin
- Add cleanup fixture for asyncio tests (GH#2866) Matthew Rocklin
- Use only remote connection to scheduler in Adaptive (GH#2865) Matthew Rocklin
- Add Server.finished async function (GH#2864) Matthew Rocklin
- Align text and remove bullets in Client HTML repr (GH#2867) Matthew Rocklin
- Test dask-scheduler –idle-timeout flag (GH#2862) Matthew Rocklin
- Remove `Client.upload_environment` (GH#2877) Jim Crist
- Replace gen.coroutine with async/await in core (GH#2871) Matthew Rocklin
- Forcefully kill all processes before each test (GH#2882) Matthew Rocklin
- Cleanup Security class and configuration (GH#2873) Jim Crist
- Remove unused variable in SpecCluster scale down (GH#2870) Jacob Tomlinson
- Add SpecCluster ProcessInterface (GH#2874) Jacob Tomlinson
- Add Log(str) and Logs(dict) classes for nice HTML reprs (GH#2875) Jacob Tomlinson
- Pass Client._asynchronous to Cluster._asynchronous (GH#2890) Matthew Rocklin
- Add default logs method to Spec Cluster (GH#2889) Matthew Rocklin

- Add processes keyword back into clean (GH#2891) Matthew Rocklin
- Update black (GH#2901) Matthew Rocklin
- Move Worker.local_dir attribute to Worker.local_directory (GH#2900) Matthew Rocklin
- Link from TapTools to worker info pages in dashboard (GH#2894) Matthew Rocklin
- Avoid exception in Client._ensure_connected if closed (GH#2893) Matthew Rocklin
- Convert Pythonic kwargs to CLI Keywords for SSHCluster (GH#2898) Matthew Rocklin
- Use kwargs in CLI (GH#2899) Matthew Rocklin
- Name SSHClusters by providing name= keyword to SpecCluster (GH#2903) Matthew Rocklin
- Request feed of worker information from Scheduler to SpecCluster (GH#2902) Matthew Rocklin
- Clear out compatibillity file (GH#2896) Matthew Rocklin
- Remove future imports (GH#2897) Matthew Rocklin
- Use click's show_default=True in relevant places (GH#2838) Christian Hudon
- Close workers more gracefully (GH#2905) Matthew Rocklin
- Close workers gracefully with –lifetime keywords (GH#2892) Matthew Rocklin
- Add closing <li> tags to Client._repr_html_ (GH#2911) Matthew Rocklin
- Add endline spacing in Logs._repr_html_ (GH#2912) Matthew Rocklin

3.30.50 2.1.0 - 2019-07-08

- Fix typo that prevented error message (GH#2825) Russ Bubley
- Remove `dask-mpi` (GH#2824) Matthew Rocklin
- Updates to use `update_graph` in task journey docs (GH#2821) James Bourbeau
- Fix Client repr with `memory_info=None` (GH#2816) Matthew Rocklin
- Fix case where key, rather than `TaskState`, could end up in `ts.waiting_on` (GH#2819) tjb900
- Use Keyword-only arguments (GH#2814) Matthew Rocklin
- Relax check for worker references in cluster context manager (GH#2813) Matthew Rocklin
- Add HTTPS support for the dashboard (GH#2812) Jim Crist
- Use `dask.utils.format_bytes` (GH#2810) Tom Augspurger

3.30.51 2.0.1 - 2019-06-26

We neglected to include `python_requires=` in our setup.py file, resulting in confusion for Python 2 users who erroneously get packages for 2.0.0. This is fixed in 2.0.1 and we have removed the 2.0.0 files from PyPI.

- Add python_requires entry to setup.py (GH#2807) Matthew Rocklin
- Correctly manage tasks beyond deque limit in TaskStream plot (GH#2797) Matthew Rocklin
- Fix diagnostics page for memory_limit=None (GH#2770) Brett Naul

3.30.52 2.0.0 - 2019-06-25

- **Drop support for Python 2**
- Relax warnings before release (GH#2796) Matthew Rocklin
- Deprecate –bokeh/–no-bokeh CLI (GH#2800) Tom Augspurger
- Typo in bokeh service_kwargs for dask-worker (GH#2783) Tom Augspurger
- Update command line cli options docs (GH#2794) James Bourbeau
- Remove “experimental” from TLS docs (GH#2793) James Bourbeau
- Add warnings around ncores= keywords (GH#2791) Matthew Rocklin
- Add –version option to scheduler and worker CLI (GH#2782) Tom Augspurger
- Raise when workers initialization times out (GH#2784) Tom Augspurger
- Replace ncores with nthreads throughout codebase (GH#2758) Matthew Rocklin
- Add unknown pytest markers (GH#2764) Tom Augspurger
- Delay lookup of allowed failures. (GH#2761) Tom Augspurger
- Change address -> worker in ColumnDataSource for nbytes plot (GH#2755) Matthew Rocklin
- Remove module state in Prometheus Handlers (GH#2760) Matthew Rocklin
- Add stress test for UCX (GH#2759) Matthew Rocklin
- Add nanny logs (GH#2744) Tom Augspurger
- Move some of the adaptive logic into the scheduler (GH#2735) Matthew Rocklin
- Add SpecCluster.new_worker_spec method (GH#2751) Matthew Rocklin
- Worker dashboard fixes (GH#2747) Matthew Rocklin
- Add async context managers to scheduler/worker classes (GH#2745) Matthew Rocklin
- Fix the resource key representation before sending graphs (GH#2733) Michael Spiegel
- Allow user to configure whether workers are daemon. (GH#2739) Caleb
- Pin pytest >=4 with pip in appveyor and python 3.5 (GH#2737) Matthew Rocklin
- Add Experimental UCX Comm (GH#2591) Ben Zaitlen Tom Augspurger Matthew Rocklin
- Close nannies gracefully (GH#2731) Matthew Rocklin
- add kwargs to progressbars (GH#2638) Manuel Garrido
- Add back LocalCluster.__repr__. (GH#2732) Loïc Estève
- Move bokeh module to dashboard (GH#2724) Matthew Rocklin
- Close clusters at exit (GH#2730) Matthew Rocklin
- Add SchedulerPlugin TaskState example (GH#2622) Matt Nicolls
- Add SpecificationCluster (GH#2675) Matthew Rocklin
- Replace register_worker_callbacks with worker plugins (GH#2453) Matthew Rocklin
- Proxy worker dashboards from scheduler dashboard (GH#2715) Ben Zaitlen
- Add docstring to Scheduler.check_idle_saturated (GH#2721) Matthew Rocklin
- Refer to LocalCluster in Client docstring (GH#2719) Matthew Rocklin

- Remove special casing of Scikit-Learn BaseEstimator serialization (GH#2713) Matthew Rocklin
- Fix two typos in Pub class docstring (GH#2714) Magnus Nord
- Support uploading files with multiple modules (GH#2587) Sam Grayson
- Change the main workers bokeh page to /status (GH#2689) Ben Zaitlen
- Cleanly stop periodic callbacks in Client (GH#2705) Matthew Rocklin
- Disable pan tool for the Progress, Byte Stored and Tasks Processing plot (GH#2703) Mathieu Dugré
- Except errors in Nanny's memory monitor if process no longer exists (GH#2701) Matthew Rocklin
- Handle heartbeat when worker has just left (GH#2702) Matthew Rocklin
- Modify styling of histograms for many-worker dashboard plots (GH#2695) Mathieu Dugré
- Add method to wait for n workers before continuing (GH#2688) Daniel Farrell
- Support computation on delayed(None) (GH#2697) Matthew Rocklin
- Cleanup localcluster (GH#2693) Matthew Rocklin
- Use 'temporary-directory' from dask.config for Worker's directory (GH#2654) Matthew Rocklin
- Remove support for Iterators and Queues (GH#2671) Matthew Rocklin

3.30.53 1.28.1 - 2019-05-13

This is a small bugfix release due to a config change upstream.

- Use config accessor method for "scheduler-address" (GH#2676) James Bourbeau

3.30.54 1.28.0 - 2019-05-08

- Add Type Attribute to TaskState (GH#2657) Matthew Rocklin
- Add waiting task count to progress title bar (GH#2663) James Bourbeau
- DOC: Clean up reference to cluster object (GH#2664) K.-Michael Aye
- Allow scheduler to politely close workers as part of shutdown (GH#2651) Matthew Rocklin
- Check direct_to_workers before using get_worker in Client (GH#2656) Matthew Rocklin
- Fixed comment regarding keeping existing level if less verbose (GH#2655) Brett Randall
- Add idle timeout to scheduler (GH#2652) Matthew Rocklin
- Avoid deprecation warnings (GH#2653) Matthew Rocklin
- Use an LRU cache for deserialized functions (GH#2623) Matthew Rocklin
- Rename Worker._close to Worker.close (GH#2650) Matthew Rocklin
- Add Comm closed bookkeeping (GH#2648) Matthew Rocklin
- Explain LocalCluster behavior in Client docstring (GH#2647) Matthew Rocklin
- Add last worker into KilledWorker exception to help debug (GH#2610) @plbertrand
- Set working worker class for dask-ssh (GH#2646) Martin Durant
- Add as_completed methods to docs (GH#2642) Jim Crist
- Add timeout to Client._reconnect (GH#2639) Jim Crist

- Limit test_spill_by_default memory, reenable it (GH#2633) Peter Andreas Entschev
- Use proper address in worker -> nanny comms (GH#2640) Jim Crist
- Fix deserialization of bytes chunks larger than 64MB (GH#2637) Peter Andreas Entschev

3.30.55 1.27.1 - 2019-04-29

- Adaptive: recommend close workers when any are idle (GH#2330) Michael Delgado
- Increase GC thresholds (GH#2624) Matthew Rocklin
- Add interface= keyword to LocalCluster (GH#2629) Matthew Rocklin
- Add worker_class argument to LocalCluster (GH#2625) Matthew Rocklin
- Remove Python 2.7 from testing matrix (GH#2631) Matthew Rocklin
- Add number of trials to diskutils test (GH#2630) Matthew Rocklin
- Fix parameter name in LocalCluster docstring (GH#2626) Loïc Estève
- Integrate stacktrace for low-level profiling (GH#2575) Peter Andreas Entschev
- Apply Black to standardize code styling (GH#2614) Matthew Rocklin
- added missing whitespace to start_worker cmd (GH#2613) condoratberlin
- Updated logging module doc links from docs.python.org/2 to docs.python.org/3. (GH#2635) Brett Randall

3.30.56 1.27.0 - 2019-04-12

- Add basic health endpoints to scheduler and worker bokeh. (GH#2607) amerkel2
- Improved description accuracy of –memory-limit option. (GH#2601) Brett Randall
- Check self.dependencies when looking at dependent tasks in memory (GH#2606) deepthirajagopalan7
- Add RabbitMQ SchedulerPlugin example (GH#2604) Matt Nicolls
- add resources to scheduler update_graph plugin (GH#2603) Matt Nicolls
- Use ensure_bytes in serialize_error (GH#2588) Matthew Rocklin
- Specify data storage explicitly from Worker constructor (GH#2600) Matthew Rocklin
- Change bokeh port keywords to dashboard_address (GH#2589) Matthew Rocklin
- .detach_() pytorch tensor to serialize data as numpy array. (GH#2586) Muammar El Khatib
- Add warning if creating scratch directories takes a long time (GH#2561) Matthew Rocklin
- Fix typo in pub-sub doc. (GH#2599) Loïc Estève
- Allow return_when='FIRST_COMPLETED' in wait (GH#2598) Nikos Tsaousis
- Forward kwargs through Nanny to Worker (GH#2596) Brian Chu
- Use ensure_dict instead of dict (GH#2594) James Bourbeau
- Specify protocol in LocalCluster (GH#2489) Matthew Rocklin

3.30.57 1.26.1 - 2019-03-29

- Fix LocalCluster to not overallocate memory when overcommitting threads per worker (GH#2541) George Sakkis
- Make closing resilient to lacking an address (GH#2542) Matthew Rocklin
- fix typo in comment (GH#2546) Brett Jurman
- Fix double init of prometheus metrics (GH#2544) Marco Neumann
- Skip test_duplicate_clients without bokeh. (GH#2553) Elliott Sales de Andrade
- Add blocked_handlers to servers (GH#2556) Chris White
- Always yield Server.handle_comm coroutine (GH#2559) Tom Augspurger
- Use yaml.safe_load (GH#2566) Matthew Rocklin
- Fetch executables from build root. (GH#2551) Elliott Sales de Andrade
- Fix Torando 6 test failures (GH#2570) Matthew Rocklin
- Fix test_sync_closed_loop (GH#2572) Matthew Rocklin

3.30.58 1.26.0 - 2019-02-25

- Update style to fix recent flake8 update (GH#2500) (GH#2509) Matthew Rocklin
- Fix typo in gen_cluster log message (GH#2503) Loïc Estève
- Allow KeyError when closing event loop (GH#2498) Matthew Rocklin
- Avoid thread testing for TCP ThreadPoolExecutor (GH#2510) Matthew Rocklin
- Find Futures inside SubgraphCallable (GH#2505) Jim Crist
- Avoid AttributeError when closing and sending a message (GH#2514) Matthew Rocklin
- Add deprecation warning to dask_mpi.py (GH#2522) Julia Kent
- Relax statistical profiling test (GH#2527) Matthew Rocklin
- Support alternative –remote-dask-worker SSHCluster() and dask-ssh CLI (GH#2526) Adam Beberg
- Iterate over full list of plugins in transition (GH#2518) Matthew Rocklin
- Create Prometheus Endpoint (GH#2499) Adam Beberg
- Use pytest.importorskip for prometheus test (GH#2533) Matthew Rocklin
- MAINT skip prometheus test when no installed (GH#2534) Olivier Grisel
- Fix intermittent testing failures (GH#2535) Matthew Rocklin
- Avoid using nprocs keyword in dask-ssh if set to one (GH#2531) Matthew Rocklin
- Bump minimum Tornado version to 5.0

3.30.59 1.25.3 - 2019-01-31

- Fix excess threading on missing connections (GH#2403) Daniel Farrell
- Fix typo in doc (GH#2457) Loïc Estève
- Start fewer but larger workers with LocalCluster (GH#2452) Matthew Rocklin
- Check for non-zero `length` first in `read` loop (GH#2465) John Kirkham
- DOC: Use of local cluster in script (GH#2462) Peter Killick
- DOC/API: Signature for base class write / read (GH#2472) Tom Augspurger
- Support Pytest 4 in Tests (GH#2478) Adam Beberg
- Ensure async behavior in event loop with LocalCluster (GH#2484) Matthew Rocklin
- Fix spurious CancelledError (GH#2485) Loïc Estève
- Properly reset dask.config scheduler and shuffle when closing the client (GH#2475) George Sakkis
- Make it more explict that resources are per worker. (GH#2470) Loïc Estève
- Remove references to center (GH#2488) Matthew Rocklin
- Expand client clearing timeout to 10s in testing (GH#2493) Matthew Rocklin
- Propagate key keyword in progressbar (GH#2492) Matthew Rocklin
- Use provided cluster's IOLoop if present in Client (GH#2494) Matthew Rocklin

3.30.60 1.25.2 - 2019-01-04

- Clean up LocalCluster logging better in async mode (GH#2448) Matthew Rocklin
- Add short error message if bokeh cannot be imported (GH#2444) Dirk Petersen
- Add optional environment variables to Nanny (GH#2431) Matthew Rocklin
- Make the direct keyword docstring entries uniform (GH#2441) Matthew Rocklin
- Make LocalCluster.close async friendly (GH#2437) Matthew Rocklin
- gather_dep: don't request dependencies we already found out we don't want (GH#2428) tjb900
- Add parameters to Client.run docstring (GH#2429) Matthew Rocklin
- Support coroutines and async-def functions in run/run_scheduler (GH#2427) Matthew Rocklin
- Name threads in ThreadPoolExecutors (GH#2408) Matthew Rocklin

3.30.61 1.25.1 - 2018-12-15

- Serialize numpy.ma.masked objects properly (GH#2384) Jim Crist
- Turn off bokeh property validation in dashboard (GH#2387) Jim Crist
- Fully initialize WorkerState objects (GH#2388) Jim Crist
- Fix typo in scheduler docstring (GH#2393) Russ Bubley
- DOC: fix typo in distributed.worker.Worker docstring (GH#2395) Loïc Estève
- Remove clients and workers from event log after removal (GH#2394) tjb900

- Support msgpack 0.6.0 by providing length keywords (GH#2399) tjb900
- Use async-await on large messages test (GH#2404) Matthew Rocklin
- Fix race condition in normalize_collection (GH#2386) Jim Crist
- Fix redict collection after HighLevelGraph fix upstream (GH#2413) Matthew Rocklin
- Add a blocking argument to Lock.acquire() (GH#2412) Stephan Hoyer
- Fix long traceback test (GH#2417) Matthew Rocklin
- Update x509 certificates to current OpenSSL standards. (GH#2418) Diane Trout

3.30.62 1.25.0 - 2018-11-28

- Fixed the 404 error on the Scheduler Dashboard homepage (GH#2361) Michael Wheeler
- Consolidate two Worker classes into one (GH#2363) Matthew Rocklin
- Avoid warnings in pyarrow and msgpack (GH#2364) Matthew Rocklin
- Avoid race condition in Actor's Future (GH#2374) Matthew Rocklin
- Support missing packages keyword in Client.get_versions (GH#2379) Matthew Rocklin
- Fixup serializing masked arrays (GH#2373) Jim Crist

3.30.63 1.24.2 - 2018-11-15

- Add support for Bokeh 1.0 (GH#2348) (GH#2356) Matthew Rocklin
- Fix regression that dropped support for Tornado 4 (GH#2353) Roy Wedge
- Avoid deprecation warnings (GH#2355) (GH#2357) Matthew Rocklin
- Fix typo in worker documentation (GH#2349) Tom Rochette

3.30.64 1.24.1 - 2018-11-09

- Use tornado's builtin AnyThreadLoopEventPolicy (GH#2326) Matthew Rocklin
- Adjust TLS tests for openssl 1.1 (GH#2331) Marius van Niekerk
- Avoid setting event loop policy if within Jupyter notebook server (GH#2343) Matthew Rocklin
- Add preload script to conf (GH#2325) Guillaume Eynard-Bontemps
- Add serializer for Numpy masked arrays (GH#2335) Peter Killick
- Use psutil.Process.oneshot (GH#2339) NotSqrt
- Use worker SSL context when getting client from worker. (GH#2301) Anonymous

3.30.65 1.24.0 - 2018-10-26

- Remove Joblib Dask Backend from codebase (GH#2298) Matthew Rocklin
- Include worker tls protocol in Scheduler.restart (GH#2295) Matthew Rocklin
- Adapt to new Bokeh selection for 1.0 (GH#2292) Matthew Rocklin
- Add explicit retry method to Future and Client (GH#2299) Matthew Rocklin
- Point to main worker page in bokeh links (GH#2300) Matthew Rocklin
- Limit concurrency when gathering many times (GH#2303) Matthew Rocklin
- Add tls_cluster pytest fixture (GH#2302) Matthew Rocklin
- Convert ConnectionPool.open and active to properties (GH#2304) Matthew Rocklin
- change export_tb to format_tb (GH#2306) Eric Ma
- Redirect joblib page to dask-ml (GH#2307) Matthew Rocklin
- Include unserializable object in error message (GH#2310) Matthew Rocklin
- Import Mapping, Iterator, Set from collections.abc in Python 3 (GH#2315) Gaurav Sheni
- Extend Client.scatter docstring (GH#2320) Eric Ma
- Update for new flake8 (GH#2321) Matthew Rocklin

3.30.66 1.23.3 - 2018-10-05

- Err in dask serialization if not a NotImplementedError (GH#2251) Matthew Rocklin
- Protect against key missing from priority in GraphLayout (GH#2259) Matthew Rocklin
- Do not pull data twice in Client.gather (GH#2263) Adam Klein
- Add pytest fixture for cluster tests (GH#2262) Matthew Rocklin
- Cleanup bokeh callbacks (GH#2261) (GH#2278) Matthew Rocklin
- Fix bokeh error for *memory_limit=None* (GH#2255) Brett Naul
- Place large keywords into task graph in Client.map (GH#2281) Matthew Rocklin
- Remove redundant blosc threading code from protocol.numpy (GH#2284) Mike Gevaert
- Add ncores to workertable (GH#2289) Matthew Rocklin
- Support upload_file on files with no extension (GH#2290) Matthew Rocklin

3.30.67 1.23.2 - 2018-09-17

- Discard dependent rather than remove (GH#2250) Matthew Rocklin
- Use dask_sphinx_theme Matthew Rocklin
- Drop the Bokeh index page (GH#2241) John Kirkham
- Revert change to keep link relative (GH#2242) Matthew Rocklin
- docs: Fix broken AWS link in setup.rst file (GH#2240) Vladyslav Moisieienkov
- Return cancelled futures in as_completed (GH#2233) Chris White

3.30.68 1.23.1 - 2018-09-06

- Raise informative error when mixing futures between clients (GH#2227) Matthew Rocklin
- add byte_keys to unpack_remotedata call (GH#2232) Matthew Rocklin
- Add documentation for gist/rawgit for get_task_stream (GH#2236) Matthew Rocklin
- Quiet Client.close by waiting for scheduler stop signal (GH#2237) Matthew Rocklin
- Display system graphs nicely on different screen sizes (GH#2239) Derek Ludwig
- Mutate passed in workers dict in TaskStreamPlugin.rectangles (GH#2238) Matthew Rocklin

3.30.69 1.23.0 - 2018-08-30

- Add direct_to_workers to Client Matthew Rocklin
- Add Scheduler.proxy to workers Matthew Rocklin
- Implement Actors Matthew Rocklin
- Fix tooltip (GH#2168) Loïc Estève
- Fix scale / avoid returning coroutines (GH#2171) Joe Hamman
- Clarify dask-worker –nprocs (GH#2173) Yu Feng
- Concatenate all bytes of small messages in TCP comms (GH#2172) Matthew Rocklin
- Add dashboard_link property (GH#2176) Jacob Tomlinson
- Always offload to_frames (GH#2170) Matthew Rocklin
- Warn if desired port is already in use (GH#2191) (GH#2199) Matthew Rocklin
- Add profile page for event loop thread (GH#2144) Matthew Rocklin
- Use dispatch for dask serialization, also add sklearn, pytorch (GH#2175) Matthew Rocklin
- Handle corner cases with busy signal (GH#2182) Matthew Rocklin
- Check self.dependencies when looking at tasks in memory (GH#2196) Matthew Rocklin
- Add ability to log additional custom metrics from each worker (GH#2169) Loïc Estève
- Fix formatting when port is a tuple (GH#2204) Loïc Estève
- Describe what ZeroMQ is (GH#2211) Mike DePalatis
- Tiny typo fix (GH#2214) Anderson Banihirwe
- Add Python 3.7 to travis.yml (GH#2203) Matthew Rocklin
- Add plot= keyword to get_task_stream (GH#2198) Matthew Rocklin
- Add support for optional versions in Client.get_versions (GH#2216) Matthew Rocklin
- Add routes for solo bokeh figures in dashboard (GH#2185) Matthew Rocklin
- Be resilient to missing dep after busy signal (GH#2217) Matthew Rocklin
- Use CSS Grid to layout status page on the dashboard (GH#2213) Derek Ludwig and Luke Canavan
- Fix deserialization of queues on main ioloop thread (GH#2221) Matthew Rocklin
- Add a worker initialization function (GH#2201) Guillaume Eynard-Bontemps

- Collapse navbar in dashboard (GH#2223) Luke Canavan

3.30.70 1.22.1 - 2018-08-03

- Add worker_class= keyword to Nanny to support different worker types (GH#2147) Martin Durant
- Cleanup intermittent worker failures (GH#2152) (GH#2146) Matthew Rocklin
- Fix msgpack PendingDeprecationWarning for encoding='utf-8' (GH#2153) Olivier Grisel
- Make bokeh coloring deterministic using hash function (GH#2143) Matthew Rocklin
- Allow client to query the task stream plot (GH#2122) Matthew Rocklin
- Use PID and counter in thread names (GH#2084) (GH#2128) Dror Birkman
- Test that worker restrictions are cleared after cancellation (GH#2107) Matthew Rocklin
- Expand resources in graph_to_futures (GH#2131) Matthew Rocklin
- Add custom serialization support for pyarrow (GH#2115) Dave Hirschfeld
- Update dask-scheduler cli help text for preload (GH#2120) Matt Nicolls
- Added another nested parallelism test (GH#1710) Tom Augspurger
- insert newline by default after TextProgressBar (GH#1976) Phil Tooley
- Retire workers from scale (GH#2104) Matthew Rocklin
- Allow worker to refuse data requests with busy signal (GH#2092) Matthew Rocklin
- Don't forget released keys (GH#2098) Matthew Rocklin
- Update example for stopping a worker (GH#2088) John Kirkham
- removed hardcoded value of memory terminate fraction from a log message (GH#2096) Bartosz Marcinkowski
- Adjust worker doc after change in config file location and treatment (GH#2094) Aurélien Ponte
- Prefer gathering data from same host (GH#2090) Matthew Rocklin
- Handle exceptions on deserialized comm with text error (GH#2093) Matthew Rocklin
- Fix typo in docstring (GH#2087) Loïc Estève
- Provide communication context to serialization functions (GH#2054) Matthew Rocklin
- Allow *name* to be explicitly passed in publish_dataset (GH#1995) Marius van Niekerk
- Avoid accessing Worker.scheduler_delay around yield point (GH#2074) Matthew Rocklin
- Support TB and PB in format bytes (GH#2072) Matthew Rocklin
- Add test for as_completed for loops in Python 2 (GH#2071) Matthew Rocklin
- Allow adaptive to exist without a cluster (GH#2064) Matthew Rocklin
- Have worker data transfer wait until recipient acknowledges (GH#2052) Matthew Rocklin
- Support async def functions in Client.sync (GH#2070) Matthew Rocklin
- Add asynchronous parameter to docstring of LocalCluster Matthew Rocklin
- Normalize address before comparison (GH#2066) Tom Augspurger
- Use ConnectionPool for Worker.scheduler Matthew Rocklin
- Avoid reference cycle in str_graph Matthew Rocklin

- Pull data outside of while loop in gather (GH#2059) Matthew Rocklin

3.30.71 1.22.0 - 2018-06-14

- Overhaul configuration (GH#1948) Matthew Rocklin
- Replace get= keyword with scheduler= (GH#1959) Matthew Rocklin
- Use tuples in msgpack (GH#2000) Matthew Rocklin and Marius van Niekerk
- Unify handling of high-volume connections (GH#1970) Matthew Rocklin
- Automatically scatter large arguments in joblib connector (GH#2020) (GH#2030) Olivier Grisel
- Turn click Python 3 locales failure into a warning (GH#2001) Matthew Rocklin
- Rely on dask implementation of sizeof (GH#2042) Matthew Rocklin
- Replace deprecated workers.iloc with workers.values() (GH#2013) Grant Jenks
- Introduce serialization families (GH#1912) Matthew Rocklin
- Add PubSub (GH#1999) Matthew Rocklin
- Add Dask stylesheet to documentation Matthew Rocklin
- Avoid recomputation on partially-complete results (GH#1840) Matthew Rocklin
- Use sys.prefix in popen for testing (GH#1954) Matthew Rocklin
- Include yaml files in manifest Matthew Rocklin
- Use self.sync so Client.processing works in asynchronous context (GH#1962) Henry Doupe
- Fix bug with bad repr on closed client (GH#1965) Matthew Rocklin
- Parse –death-timeout keyword in dask-worker (GH#1967) Matthew Rocklin
- Support serializers in BatchedSend (GH#1964) Matthew Rocklin
- Use normal serialization mechanisms to serialize published datasets (GH#1972) Matthew Rocklin
- Add security support to LocalCluster. (GH#1855) Marius van Niekerk
- add ConnectionPool.remove method (GH#1977) Tony Lorenzo
- Cleanly close workers when scheduler closes (GH#1981) Matthew Rocklin
- Add .pyz support in upload_file (GH#1781) @bmaisson
- add comm to packages (GH#1980) Matthew Rocklin
- Replace dask.set_options with dask.config.set Matthew Rocklin
- Exclude versions of sortedcontainers which do not have .iloc. (GH#1993) Russ Bubley
- Exclude gc statistics under PyPy (GH#1997) Marius van Niekerk
- Manage recent config and dataframe changes in dask (GH#2009) Matthew Rocklin
- Cleanup lingering clients in tests (GH#2012) Matthew Rocklin
- Use timeouts during *Client._ensure_connected* (GH#2011) Martin Durant
- Avoid reference cycle in joblib backend (GH#2014) Matthew Rocklin, also Olivier Grisel
- DOC: fixed test example (GH#2017) Tom Augspurger
- Add worker_key parameter to Adaptive (GH#1992) Matthew Rocklin

- Prioritize tasks with their true keys, before stringifying (GH#2006) Matthew Rocklin
- Serialize worker exceptions through normal channels (GH#2016) Matthew Rocklin
- Include exception in progress bar (GH#2028) Matthew Rocklin
- Avoid logging orphaned futures in All (GH#2008) Matthew Rocklin
- Don't use spill-to-disk dictionary if we're not spilling to disk Matthew Rocklin
- Only avoid recomputation if key exists (GH#2036) Matthew Rocklin
- Use client connection and serialization arguments in progress (GH#2035) Matthew Rocklin
- Rejoin worker client on closing context manager (GH#2041) Matthew Rocklin
- Avoid forgetting erred tasks when losing dependencies (GH#2047) Matthew Rocklin
- Avoid collisions in graph_layout (GH#2050) Matthew Rocklin
- Avoid recursively calling bokeh callback in profile plot (GH#2048) Matthew Rocklin

3.30.72 1.21.8 - 2018-05-03

- Remove errant print statement (GH#1957) Matthew Rocklin
- Only add reevaluate_occupancy callback once (GH#1953) Tony Lorenzo

3.30.73 1.21.7 - 2018-05-02

- Newline needed for doctest rendering (GH#1917) Loïc Estève
- Support Client._repr_html_ when in async mode (GH#1909) Matthew Rocklin
- Add parameters to dask-ssh command (GH#1910) Irene Rodriguez
- Santize get_dataset trace (GH#1888) John Kirkham
- Fix bug where queues would not clean up cleanly (GH#1922) Matthew Rocklin
- Delete cached file safely in upload file (GH#1921) Matthew Rocklin
- Accept KeyError when closing tornado IOLoop in tests (GH#1937) Matthew Rocklin
- Quiet the client and scheduler when gather(. . . , errors='skip') (GH#1936) Matthew Rocklin
- Clarify couldn't gather keys warning (GH#1942) Kenneth Koski
- Support submit keywords in joblib (GH#1947) Matthew Rocklin
- Avoid use of external resources in bokeh server (GH#1934) Matthew Rocklin
- Drop *__contains__* from *Datasets* (GH#1889) John Kirkham
- Fix bug with queue timeouts (GH#1950) Matthew Rocklin
- Replace msgpack-python by msgpack (GH#1927) Loïc Estève

3.30.74 1.21.6 - 2018-04-06

- Fix numeric environment variable configuration (GH#1885) Joseph Atkins-Kurkish
- support bytearrays in older lz4 library (GH#1886) Matthew Rocklin
- Remove started timeout in nanny (GH#1852) Matthew Rocklin
- Don't log errors in sync (GH#1894) Matthew Rocklin
- downgrade stale lock warning to info logging level (GH#1890) Matthew Rocklin
- Fix `UnboundLocalError` for `key` (GH#1900) John Kirkham
- Resolve deployment issues in Python 2 (GH#1905) Matthew Rocklin
- Support retries and priority in Client.get method (GH#1902) Matthew Rocklin
- Add additional attributes to task page if applicable (GH#1901) Matthew Rocklin
- Add count method to as_completed (GH#1897) Matthew Rocklin
- Extend default timeout to 10s (GH#1904) Matthew Rocklin

3.30.75 1.21.5 - 2018-03-31

- Increase default allowable tick time to 3s (GH#1854) Matthew Rocklin
- Handle errant workers when another worker has data (GH#1853) Matthew Rocklin
- Close multiprocessing queue in Nanny to reduce open file descriptors (GH#1862) Matthew Rocklin
- Extend nanny started timeout to 30s, make configurable (GH#1865) Matthew Rocklin
- Comment out the default config file (GH#1871) Matthew Rocklin
- Update to fix bokeh 0.12.15 update errors (GH#1872) Matthew Rocklin
- Downgrade Event Loop unresponsive warning to INFO level (GH#1870) Matthew Rocklin
- Add fifo timeout to control priority generation (GH#1828) Matthew Rocklin
- Add retire_workers API to Client (GH#1876) Matthew Rocklin
- Catch NoSuchProcess error in Nanny.memory_monitor (GH#1877) Matthew Rocklin
- Add uid to nanny queue communitcations (GH#1880) Matthew Rocklin

3.30.76 1.21.4 - 2018-03-21

- Avoid passing bytearrays to snappy decompression (GH#1831) Matthew Rocklin
- Specify IOLoop in Adaptive (GH#1841) Matthew Rocklin
- Use connect-timeout config value throughout client (GH#1839) Matthew Rocklin
- Support direct= keyword argument in Client.get (GH#1845) Matthew Rocklin

3.30.77 1.21.3 - 2018-03-08

- Add cluster superclass and improve adaptivity (GH#1813) Matthew Rocklin
- Fixup tests and support Python 2 for Tornado 5.0 (GH#1818) Matthew Rocklin
- Fix bug in recreate_error when dependencies are dropped (GH#1815) Matthew Rocklin
- Add worker time to live in Scheduler (GH#1811) Matthew Rocklin
- Scale adaptive based on total_occupancy (GH#1807) Matthew Rocklin
- Support calling compute within worker_client (GH#1814) Matthew Rocklin
- Add percentage to profile plot (GH#1817) Brett Naul
- Overwrite option for remote python in dask-ssh (GH#1812) Sven Kreiss

3.30.78 1.21.2 - 2018-03-05

- Fix bug where we didn't check idle/saturated when stealing (GH#1801) Matthew Rocklin
- Fix bug where client was noisy when scheduler closed unexpectedly (GH#1806) Matthew Rocklin
- Use string-based timedeltas (like `'500 ms'`) everywhere (GH#1804) Matthew Rocklin
- Keep logs in scheduler and worker even if silenced (GH#1803) Matthew Rocklin
- Support minimum, maximum, wait_count keywords in Adaptive (GH#1797) Jacob Tomlinson and Matthew Rocklin
- Support async protocols for LocalCluster, replace start= with asynchronous= (GH#1798) Matthew Rocklin
- Avoid restarting workers when nanny waits on scheduler (GH#1793) Matthew Rocklin
- Use `IOStream.read_into()` when available (GH#1477) Antoine Pitrou
- Reduce LocalCluster logging threshold from CRITICAL to WARN (GH#1785) Andy Jones
- Add *futures_of* to API docs (GH#1783) John Kirkham
- Make diagnostics link in client configurable (GH#1810) Matthew Rocklin

3.30.79 1.21.1 - 2018-02-22

- Fixed an uncaught exception in `distributed.joblib` with a `LocalCluster` using only threads (GH#1775) Tom Augspurger
- Format bytes in info worker page (GH#1752) Matthew Rocklin
- Add pass-through arguments for scheduler/worker *–preload* modules. (GH#1634) Alexander Ford
- Use new LZ4 API (GH#1757) Thrasibule
- Replace dask.optimize with dask.optimization (GH#1754) Matthew Rocklin
- Add graph layout engine and bokeh plot (GH#1756) Matthew Rocklin
- Only expand name with –nprocs if name exists (GH#1776) Matthew Rocklin
- specify IOLoop for stealing PeriodicCallback (GH#1777) Matthew Rocklin
- Fixed distributed.joblib with no processes Tom Augspurger
- Use set.discard to avoid KeyErrors in stealing (GH#1766) Matthew Rocklin

- Avoid KeyError when task has been released during steal (GH#1765) Matthew Rocklin
- Add versions routes to avoid the use of run in Client.get_versions (GH#1773) Matthew Rocklin
- Add write_scheduler_file to Client (GH#1778) Joe Hamman
- Default host to tls:// if tls information provided (GH#1780) Matthew Rocklin

3.30.80 1.21.0 - 2018-02-09

- Refactor scheduler to use TaskState objects rather than dictionaries (GH#1594) Antoine Pitrou
- Plot CPU fraction of total in workers page (GH#1624) Matthew Rocklin
- Use thread CPU time in Throttled GC (GH#1625) Antoine Pitrou
- Fix bug with `memory_limit=None` (GH#1639) Matthew Rocklin
- Add futures_of to top level api (GH#1646) Matthew Rocklin
- Warn on serializing large data in Client (GH#1636) Matthew Rocklin
- Fix intermittent windows failure when removing lock file (GH#1652) Antoine Pitrou
- Add diagnosis and logging of poor GC Behavior (GH#1635) Antoine Pitrou
- Add client-scheduler heartbeats (GH#1657) Matthew Rocklin
- Return dictionary of worker info in `retire_workers` (GH#1659) Matthew Rocklin
- Ensure dumps_function works with unhashable functions (GH#1662) Matthew Rocklin
- Collect client name ids rom client-name config variable (GH#1664) Matthew Rocklin
- Allow simultaneous use of –name and –nprocs in dask-worker (GH#1665) Matthew Rocklin
- Add support for grouped adaptive scaling and adaptive behavior overrides (GH#1632) Alexander Ford
- Share scheduler RPC between worker and client (GH#1673) Matthew Rocklin
- Allow `retries=` in ClientExecutor (GH#1672) @rqx
- Improve documentation for get_client and dask.compute examples (GH#1638) Scott Sievert
- Support DASK_SCHEDULER_ADDRESS environment variable in worker (GH#1680) Matthew Rocklin
- Support tuple-keys in retries (GH#1681) Matthew Rocklin
- Use relative links in bokeh dashboard (GH#1682) Matthew Rocklin
- Make message log length configurable, default to zero (GH#1691) Matthew Rocklin
- Deprecate `Client.shutdown` (GH#1699) Matthew Rocklin
- Add warning in configuration docs to install pyyaml (GH#1701) Cornelius Riemenschneider
- Handle nested parallelism in distributed.joblib (GH#1705) Tom Augspurger
- Don't wait for Worker.executor to shutdown cleanly when restarting process (GH#1708) Matthew Rocklin
- Add support for user defined priorities (GH#1651) Matthew Rocklin
- Catch and log OSErrors around worker lock files (GH#1714) Matthew Rocklin
- Remove worker prioritization. Coincides with changes to dask.order (GH#1730) Matthew Rocklin
- Use process-measured memory rather than nbytes in Bokeh dashboard (GH#1737) Matthew Rocklin
- Enable serialization of Locks (GH#1738) Matthew Rocklin

- Support Tornado 5 beta (GH#1735) Matthew Rocklin
- Cleanup remote_magic client cache after tests (GH#1743) Min RK
- Allow service ports to be specified as (host, port) (GH#1744) Bruce Merry

3.30.81 1.20.2 - 2017-12-07

- Clear deque handlers after each test (GH#1586) Antoine Pitrou
- Handle deserialization in FutureState.set_error (GH#1592) Matthew Rocklin
- Add process leak checker to tests (GH#1596) Antoine Pitrou
- Customize process title for subprocess (GH#1590) Antoine Pitrou
- Make linting a separate CI job (GH#1599) Antoine Pitrou
- Fix error from get_client() with no global client (GH#1595) Daniel Li
- Remove Worker.host_health, correct WorkerTable metrics (GH#1600) Matthew Rocklin
- Don't mark tasks as suspicious when retire_workers called. Addresses (GH#1607) Russ Bubley
- Do not include processing workers in workers_to_close (GH#1609) Russ Bubley
- Disallow simultaneous scale up and down in Adaptive (GH#1608) Russ Bubley
- Parse bytestrings in –memory-limit (GH#1615) Matthew Rocklin
- Use environment variable for scheduler address if present (GH#1610) Matthew Rocklin
- Fix deprecation warning from logger.warn (GH#1616) Brett Naul

3.30.82 1.20.1 - 2017-11-26

- Wrap `import ssl` statements with try-except block for ssl-crippled environments, (GH#1570) Xander Johnson
- Support zero memory-limit in Nanny (GH#1571) Matthew Rocklin
- Avoid PeriodicCallback double starts (GH#1573) Matthew Rocklin
- Add disposable workspace facility (GH#1543) Antoine Pitrou
- Use format_time in task_stream plots (GH#1575) Matthew Rocklin
- Avoid delayed finalize calls in compute (GH#1577) Matthew Rocklin
- Doc fix about secede (GH#1583) Scott Sievert
- Add tracemalloc option when tracking test leaks (GH#1585) Antoine Pitrou
- Add JSON routes to Bokeh server (GH#1584) Matthew Rocklin
- Handle exceptions cleanly in Variables and Queues (GH#1580) Matthew Rocklin

3.30.83 1.20.0 - 2017-11-17

- Drop use of pandas.msgpack (GH#1473) Matthew Rocklin
- Add methods to get/set scheduler metadata Matthew Rocklin
- Add distributed lock Matthew Rocklin
- Add reschedule exception for worker tasks Matthew Rocklin
- Fix `nbytes()` for `bytearrays` Matthew Rocklin
- Capture scheduler and worker logs Matthew Rocklin
- Garbage collect after data eviction on high worker memory usage (GH#1488) Olivier Grisel
- Add scheduler HTML routes to bokeh server (GH#1478) (GH#1514) Matthew Rocklin
- Add pytest plugin to test for resource leaks (GH#1499) Antoine Pitrou
- Improve documentation for scheduler states (GH#1498) Antoine Pitrou
- Correct warn_if_longer timeout in ThrottledGC (GH#1496) Fabian Keller
- Catch race condition in as_completed on cancelled futures (GH#1507) Matthew Rocklin
- Transactional work stealing (GH#1489) (GH#1528) Matthew Rocklin
- Avoid forkserver in PyPy (GH#1509) Matthew Rocklin
- Add dict access to get/set datasets (GH#1508) Mike DePalatis
- Support Tornado 5 (GH#1509) (GH#1512) (GH#1518) (GH#1534) Antoine Pitrou
- Move thread_state in Dask (GH#1523) Jim Crist
- Use new Dask collections interface (GH#1513) Matthew Rocklin
- Add nanny flag to dask-mpi Matthew Rocklin
- Remove JSON-based HTTP servers Matthew Rocklin
- Avoid doing I/O in repr/str (GH#1536) Matthew Rocklin
- Fix URL for MPI4Py project (GH#1546) Ian Hopkinson
- Allow automatic retries of a failed task (GH#1524) Antoine Pitrou
- Clean and accelerate tests (GH#1548) (GH#1549) (GH#1552) (GH#1553) (GH#1560) (GH#1564) Antoine Pitrou
- Move HDFS functionality to the hdfs3 library (GH#1561) Jim Crist
- Fix bug when using events page with no events (GH#1562) @rbubley
- Improve diagnostic naming of tasks within tuples (GH#1566) Kelvyn Yang

3.30.84 1.19.3 - 2017-10-16

- Handle None case in profile.identity (GH#1456)
- Asyncio rewrite (GH#1458)
- Add rejoin function partner to secede (GH#1462)
- Nested compute (GH#1465)
- Use LooseVersion when comparing Bokeh versions (GH#1470)

3.30.85 1.19.2 - 2017-10-06

- as_completed doesn't block on cancelled futures (GH#1436)
- Notify waiting threads/coroutines on cancellation (GH#1438)
- Set Future(inform=True) as default (GH#1437)
- Rename Scheduler.transition_story to story (GH#1445)
- Future uses default client by default (GH#1449)
- Add keys= keyword to Client.call_stack (GH#1446)
- Add get_current_task to worker (GH#1444)
- Ensure that Client remains asynchornous before ioloop starts (GH#1452)
- Remove "click for worker page" in bokeh plot (GH#1453)
- Add Client.current() (GH#1450)
- Clean handling of restart timeouts (GH#1442)

3.30.86 1.19.1 - September 25th, 2017

- Fix tool issues with TaskStream plot (GH#1425)
- Move profile module to top level (GH#1423)

3.30.87 1.19.0 - September 24th, 2017

- Avoid storing messages in message log (GH#1361)
- fileConfig does not disable existing loggers (GH#1380)
- Offload upload_file disk I/O to separate thread (GH#1383)
- Add missing SSLContext (GH#1385)
- Collect worker thread information from sys._curent_frames (GH#1387)
- Add nanny timeout (GH#1395)
- Restart worker if memory use goes above 95% (GH#1397)
- Track workers memory use with psutil (GH#1398)
- Track scheduler delay times in workers (GH#1400)
- Add time slider to profile plot (GH#1403)

- Change memory-limit keyword to refer to maximum number of bytes (GH#1405)
- Add `cancel(force=)` keyword (GH#1408)

3.30.88 1.18.2 - September 2nd, 2017

- Silently pass on cancelled futures in as_completed (GH#1366)
- Fix unicode keys error in Python 2 (GH#1370)
- Support numeric worker names
- Add dask-mpi executable (GH#1367)

3.30.89 1.18.1 - August 25th, 2017

- Clean up forgotten keys in fire-and-forget workloads (GH#1250)
- Handle missing extensions (GH#1263)
- Allow recreate_exception on persisted collections (GH#1253)
- Add asynchronous= keyword to blocking client methods (GH#1272)
- Restrict to horizontal panning in bokeh plots (GH#1274)
- Rename client.shutdown to client.close (GH#1275)
- Avoid blocking on event loop (GH#1270)
- Avoid cloudpickle errors for Client.get_versions (GH#1279)
- Yield on Tornado IOStream.write futures (GH#1289)
- Assume async behavior if inside a sync statement (GH#1284)
- Avoid error messages on closing (GH#1297), (GH#1296) (GH#1318) (GH#1319)
- Add timeout= keyword to get_client (GH#1290)
- Respect timeouts when restarting (GH#1304)
- Clean file descriptor and memory leaks in tests (GH#1317)
- Deprecate Executor (GH#1302)
- Add timeout to ThreadPoolExecutor.shutdown (GH#1330)
- Clean up AsyncProcess handling (GH#1324)
- Allow unicode keys in Python 2 scheduler (GH#1328)
- Avoid leaking stolen data (GH#1326)
- Improve error handling on failed nanny starts (GH#1337), (GH#1331)
- Make Adaptive more flexible
- Support `--contact-address` and `--listen-address` in worker (GH#1278)
- Remove old dworker, dscheduler executables (GH#1355)
- Exit workers if nanny process fails (GH#1345)
- Auto pep8 and flake (GH#1353)

3.30.90 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)

3.30.91 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)

3.30.92 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)

3.30.93 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)

3.30.94 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)

3.30.95 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)

3.30.96 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)

3.30.97 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)

3.30.98 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)

3.30.99 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)

3.30.100 1.14.3 - November 13th, 2016

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

3.30.101 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)

3.30.102 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

3.30.103 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

3.30.104 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

3.30.105 1.13.1

- Fix critical Windows error with dask_worker executable

3.30.106 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

3.30.107 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.

3.30.108 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

3.30.109 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

3.31 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.

3.31.1 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*, *strict=False*)
Split address into its scheme and scheme-dependent location string.

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

If strict is set to true the address must have a scheme.

`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'
```

3.31.2 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.

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

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

This method is a coroutine.

abstract closed()
Return whether the stream is closed.

property 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.

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

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

abstract read(*deserializers=None*)
Read and return a message (a Python object).

This method is a coroutine.

Parameters

deserializers [Optional[Dict[str, Tuple[Callable, Callable, bool]]]] An optional dict appropriate for distributed.protocol.deserialize. See *Serialization* for more.

abstract write(*msg*, *serializers=None*, *on_error=None*)
Write a message (a Python object).

This method is a coroutine.

Parameters

msg

on_error [Optional[str]] The behavior when serialization fails. See `distributed.protocol.core.dumps` for valid values.

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

async distributed.comm.**connect**(*addr*, *timeout=None*, *deserialize=True*, *handshake_overrides=None*, ***connection_args*)
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*, ***kwargs*)
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**

abstract property 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'.

abstract property listen_address
The listening address as a URI string.

abstract async start()
Start listening for incoming connections.

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

3.31.3 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.

Out-of-tree backends can be registered under the group `distributed.comm.backends` in setuptools entry_points. For example, a hypothetical `dask_udp` package would register its UDP backend class by including the following in its `setup.py` file:

```
setup(name="dask_udp",
      entry_points={
        "distributed.comm.backends": [
            "udp=dask_udp.backend:UDPBackend",
        ]
      },
      ...
)
```

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

abstract 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.

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

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

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

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

Simple implementations may return *loc* unchanged.

3.32 Development Guidelines

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

3.32.1 Install

Clone this repository with git:

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

Install all dependencies:

All OS:

1. Install anaconda or miniconda
2. ```
   conda env create --file continuous_integration/environment-3.8.yaml
   conda activate dask-distributed
   python -m pip install -e .
   ```

To keep a fork in sync with the upstream source:

```
cd distributed
git remote add upstream git@github.com:dask/distributed.git
git remote -v
git fetch -a upstream
git checkout main
git pull upstream main
git push origin main
```

3.32.2 Test

Test using `py.test`:

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

3.32.3 Tornado

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

3.32.4 Writing Tests

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

The test suite contains three kinds of tests

1. `@gen_cluster`: Fully asynchronous tests where all components live in the same event loop in the main thread. These are good for testing complex logic and inspecting the state of the system directly. They are also easier to debug and cause the fewest problems with shutdowns.
2. `def test_foo(client)`: Tests with multiple processes forked from the main 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, e.g.

```
# tests/test_submit.py

from distributed.utils_test import gen_cluster, inc
from distributed import Client, Future, Scheduler, Worker

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

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

    # result = future.result()  # This synchronous API call would block
    result = await 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 (_) and awaited with `await`. 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 the `client` pytest fixture style test, which sets up a scheduler and workers for you in different forked processes:

```
from distributed.utils_test import client

def test_submit(client):
    future = client.submit(inc, 10)
    assert future.result() == 11
```

Additionally, if you want access to the scheduler and worker processes you can also add the `s, a, b` fixtures as well.

```
from distributed.utils_test import client

def test_submit(client, s, a, b):
    future = client.submit(inc, 10)
    assert future.result() == 11  # use the synchronous/blocking API here

    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 `await` 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 `client` pytest fixture tests test basic interface and resilience.

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

3.33 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.

3.33.1 Concurrency with Tornado Coroutines

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

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

3.33.2 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.

3.33.3 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*, *blocked_handlers=None*, *stream_handlers=None*, *connection_limit=512*, *deserialize=True*, *serializers=None*, *deserializers=None*, *connection_args=None*, *timeout=None*, *io_loop=None*, ***kwargs*)

Dask Distributed 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*, *serializers=None*, *deserializers=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()
```

3.33.4 Examples

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

Server Side

```
import asyncio
from distributed.core import Server

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

async def stream_data(comm, interval=1):  # complex handler, multiple responses
    data = 0
    while True:
        await asyncio.sleep(interval)
        data += 1
        await comm.write(data)

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

asyncio.get_event_loop().run_forever()
```

Client Side

```
import asyncio
from distributed.core import connect

async def f():
    comm = await connect('tcp://127.0.0.1:8888')
    await comm.write({'op': 'add', 'x': 1, 'y': 2})
    result = await comm.read()
    await comm.close()
```

(continues on next page)

(continued from previous page)

```
    print(result)

>>> asyncio.get_event_loop().run_until_complete(g())
3

async def g():
    comm = await connect('tcp://127.0.0.1:8888')
    await comm.write({'op': 'stream_data', 'interval': 1})
    while True:
        result = await comm.read()
        print(result)

>>> asyncio.get_event_loop().run_until_complete(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.

```
import asyncio
from distributed.core import rpc

async def f():
    # comm = await connect('tcp://127.0.0.1', 8888)
    # await comm.write({'op': 'add', 'x': 1, 'y': 2})
    # result = await comm.read()
    with rpc('tcp://127.0.0.1:8888') as r:
        result = await r.add(x=1, y=2)

    print(result)

>>> asyncio.get_event_loop().run_until_complete(f())
3
```

3.34 Journey of a Task

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

3.34.1 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 = client.submit(...)
y = client.submit(...)

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

print(z.result())
```

3.34.2 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'`.

3.34.3 Step 2: Arrive in the Scheduler

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

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

```
scheduler.update_graph(tasks=msg['tasks'], keys=msg['keys'])
```

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_graph()`. While this is fairly complex and tedious to describe rest assured that it all happens in constant time and in about a millisecond.

3.34.4 Step 3: Select a Worker

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

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

3. If there are multiple workers that require the minimum number of communication bytes then we select the worker that is the least busy

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

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

3.34.5 Step 4: Transmit to the Worker

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

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

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

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

3.34.6 Step 5: Execute on the Worker

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

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

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

Eventually the computation completes. The Worker stores this result in its local memory:

```
data['z'] = ...
```

And transmits back a success, and the number of bytes of the result:

```
Worker: Hey Scheduler, 'z' worked great.
        I'm holding onto it.
        It takes up 64 bytes.
```

The worker does not transmit back the actual value for `z`.

3.34.7 Step 6: Scheduler Aftermath

The scheduler receives this message and does a few things:

1. It notes that the worker has a free core, and sends up another task if available
2. If `x` or `y` are no longer needed then it sends a message out to relevant workers to delete them from local memory.
3. It sends a message to all of the clients that `z` is ready and so all client `Future` objects that are currently waiting should, wake up. In particular, this wakes up the `z.result()` command executed by the user originally.

3.34.8 Step 7: Gather

When the user calls `z.result()` they wait both on the completion of the computation and for the computation to be sent back over the wire to the local process. Usually this isn't necessary, usually you don't want to move data back to the local process but instead want to keep in on the cluster.

But perhaps the user really wanted to actually know this value, so they called `z.result()`.

The scheduler checks who has `z` and sends them a message asking for the result. This message doesn't wait in a queue or for other jobs to complete, it starts instantly. The value gets serialized, sent over TCP, and then deserialized and returned to the user (passing through a queue or two on the way.)

3.34.9 Step 8: Garbage Collection

The user leaves this part of their code and the local variable `z` goes out of scope. The Python garbage collector cleans it up. This triggers a decremented reference on the 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.

3.34.10 Overhead

The user experiences this in about 10 milliseconds, depending on network latency.

3.35 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',
 'nthreads': 4}

{'x': b'...',
 'y': b'...'}
```

When we communicate these messages between nodes we need to serialize these messages down to a string of bytes that can then be deserialized on the other end to their in-memory dictionary form. For simple cases several options exist like JSON, MsgPack, Protobuffers, and Thrift. The situation is made more complex by concerns like serializing Python functions and Python objects, optional compression, cross-language support, large messages, and efficiency.

This document describes the protocol used by `dask.distributed` today. Be advised that this protocol changes rapidly as we continue to optimize for performance.

3.35.1 Overview

We may split a single message into multiple message-part to suit different protocols. Generally small bits of data are encoded with MsgPack while large bytestrings 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.

3.35.2 MsgPack for Messages

Most messages are encoded with MsgPack, a self describing semi-structured serialization format that is very similar to JSON, but smaller, faster, not human-readable, and supporting of bytestrings and (soon) timestamps. We chose MsgPack as a base serialization format for the following reasons:

- It does not require separate headers, and so is easy and flexible to use which is particularly important in an early stage project like `dask.distributed`
- It is very fast, much faster than JSON, and there are nicely optimized implementations. With few exceptions (described later) MsgPack does not come anywhere near being a bottleneck, even under heavy use.
- Unlike JSON it supports bytestrings
- It covers the standard set of types necessary to encode most information
- It is widely implemented in a number of languages (see cross language section below)

However, MsgPack fails (correctly) in the following ways:

- It does not provide any way for us to encode Python functions or user defined data types
- It does not support bytestrings greater than 4GB and is generally inefficient for very large messages.

Because of these failings we supplement it with a language-specific protocol and a special case for large bytestrings.

3.35.3 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.

3.35.4 Cross Language Specialization

The Client and Workers must agree on a language-specific serialization format. In the standard `dask.distributed` client and worker objects this ends up being the following:

```
bytes = cloudpickle.dumps(obj, protocol=pickle.HIGHEST_PROTOCOL)
obj = cloudpickle.loads(bytes)
```

This varies between Python 2 and 3 and so your client and workers must match their Python versions and software environments.

However, the Scheduler never uses the language-specific serialization and instead only deals with MsgPack. If the client sends a pickled function up to the scheduler the scheduler will not unpack function but will instead keep it as bytes. Eventually those bytes will be sent to a worker, which will then unpack the bytes into a proper Python function. Because the Scheduler never unpacks language-specific serialized bytes it may be in a different language.

The client and workers must share the same language and software environment, the scheduler may differ.

This has a few advantages:

1. The Scheduler is protected from unpickling unsafe code
2. The Scheduler can be run under `pypy` for improved performance. This is only useful for larger clusters.
3. We could conceivably implement workers and clients for other languages (like R or Julia) and reuse the Python scheduler. The worker and client code is fairly simple and much easier to reimplement than the scheduler, which is complex.
4. The scheduler might some day be rewritten in more heavily optimized C or Go

3.35.5 Compression

Fast compression libraries like LZ4 or Snappy may increase effective bandwidth by compressing data before sending and decompressing it after reception. This is especially valuable on lower-bandwidth networks.

If either of these libraries is available (we prefer LZ4 to Snappy) then for every message greater than 1kB we try to compress the message and, if the compression is at least a 10% improvement, we send the compressed bytes rather than the original payload. We record the compression used within the header as a string like `'lz4'` or `'snappy'`.

To avoid compressing large amounts of uncompressable data we first try to compress a sample. We take 10kB chunks from five locations in the dataset, arrange them together, and try compressing the result. If this doesn't result in significant compression then we don't try to compress the full result.

3.35.6 Header

The header is a small dictionary encoded in msgpack that includes some metadata about the message, such as compression.

3.35.7 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'...'
```

3.35.8 Frames

At the end of the pipeline we have a sequence of bytestrings or frames. We need to tell the receiving end how many frames there are and how long each these frames are. We order the frames and lengths of frames as follows:

1. The number of frames, stored as an 8 byte unsigned integer
2. The length of each frame, each stored as an 8 byte unsigned integer
3. Each of the frames

In the following sections we describe how we create these frames.

3.35.9 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\xf0?'`

3.36 Serialization

When we communicate data between computers we first convert that data into a sequence of bytes that can be communicated across the network. Choices made in serialization can affect performance and security.

The standard Python solution to this, Pickle, is often but not always the right solution. Dask uses a number of different serialization schemes in different situations. These are extensible to allow users to control in sensitive situations and also to enable library developers to plug in more performant serialization solutions.

This document first describes Dask's default solution for serialization and then discusses ways to control and extend that serialiation.

3.36.1 Defaults

There are three kinds of messages passed through the Dask network:

1. Small administrative messages like "Worker A has finished task X" or "I'm running out of memory". These are always serialized with msgpack.
2. Movement of program data, such as Numpy arrays and Pandas dataframes. This uses a combination of pickle and custom serializers and is the topic of the next section
3. Computational tasks like `f(x)` that are defined and serialized on client processes and deserialized and run on worker processes. These are serialized using a fixed scheme decided on by those libraries. Today this is a combination of pickle and cloudpickle.

3.36.2 Serialization families

Use

For the movement of program data (item 2 above) we can use a few different families of serializers. By default the following families are built in:

1. Pickle and cloudpickle
2. Msgpack
3. Custom per-type serializers that come with Dask for the special serialization of important classes of data like Numpy arrays

You can choose which families you want to use to serialize data and to deserialize data when you create a Client

```
from dask.distributed import Client
client = Client('tcp://scheduler-address:8786',
                serializers=['dask', 'pickle'],
                deserializers=['dask', 'msgpack'])
```

This can be useful if, for example, you are sensitive about receiving Pickle-serialized data for security reasons.

Dask uses the serializers `['dask', 'pickle']` by default, trying to use dask custom serializers (described below) if they work and then falling back to pickle/cloudpickle.

Extend

These families can be extended by creating two functions, dumps and loads, which return and consume a msgpack-encodable header, and a list of byte-like objects. These must then be included in the `distributed.protocol.serialize` dictionary with an appropriate name. Here is the definition of `pickle_dumps` and `pickle_loads` to serve as an example.

```
import pickle

def pickle_dumps(x):
    header = {'serializer': 'pickle'}
    frames = [pickle.dumps(x)]
    return header, frames

def pickle_loads(header, frames):
    if len(frames) > 1:  # this may be cut up for network reasons
        frame = ''.join(frames)
    else:
        frame = frames[0]
    return pickle.loads(frame)

from distributed.protocol.serialize import register_serialization_family
register_serialization_family('pickle', pickle_dumps, pickle_loads)
```

After this the name `'pickle'` can be used in the `serializers=` and `deserializers=` keywords in `Client` and other parts of Dask.

Communication Context

Note: This is an experimental feature and may change without notice

Dask *Comms* can provide additional context to serialization family functions if they provide a `context=` keyword. This allows serialization to behave differently according to how it is being used.

```
def my_dumps(x, context=None):
    if context and 'recipient' in context:
        # check if we're sending to the same host or not
```

The context depends on the kind of communication. For example when sending over TCP, the address of the sender (us) and the recipient are available in a dictionary.

```
>>> context
{'sender': 'tcp://127.0.0.1:1234', 'recipient': 'tcp://127.0.0.1:5678'}
```

Other comms may provide other information.

3.36.3 Dask Serialization Family

Use

Dask maintains its own custom serialization family that special cases a few important types, like Numpy arrays. These serializers either operate more efficiently than Pickle, or serialize types that Pickle can not handle.

You don't need to do anything special to use this family of serializers. It is on by default (along with pickle). Note that Dask custom serializers may use pickle internally in some cases. It should not be considered more secure.

Extend

dask_serialize	Single Dispatch for dask_serialize
dask_deserialize	Single Dispatch for dask_deserialize

As with serialization families in general, the Dask family in particular is *also* extensible. This is a good way to support custom serialization of a single type of object. The method is similar, you create serialize and deserialize function that create and consume a header and frames, and then register them with Dask.

```
class Human:
    def __init__(self, name):
        self.name = name

from distributed.protocol import dask_serialize, dask_deserialize

@dask_serialize.register(Human)
def serialize(human: Human) -> Tuple[Dict, List[bytes]]:
    header = {}
    frames = [human.name.encode()]
    return header, frames

@dask_deserialize.register(Human)
```

(continues on next page)

(continued from previous page)

```
def deserialize(header: Dict, frames: List[bytes]) -> Human:
    return Human(frames[0].decode())
```

Traverse attributes

register_generic(cls[, serializer_name, ...])	Register (de)serialize to traverse through __dict__

A common case is that your object just wraps Numpy arrays or other objects that Dask already serializes well. For example, Scikit-Learn estimators mostly surround Numpy arrays with a bit of extra metadata. In these cases you can register your class for custom Dask serialization with the `register_generic` function.

3.36.4 API

serialize(x[, serializers, on_error, ...])	Convert object to a header and list of bytestrings
deserialize(header, frames[, deserializers])	Convert serialized header and list of bytestrings back to a Python object
dask_serialize	Single Dispatch for dask_serialize
dask_deserialize	Single Dispatch for dask_deserialize
register_generic(cls[, serializer_name, ...])	Register (de)serialize to traverse through __dict__

distributed.protocol.serialize.**serialize**(*x*, *serializers=None*, *on_error='message'*, *context=None*, *iterate_collection=None*)

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.

The serialization protocols to use are configurable: a list of names define the set of serializers to use, in order. These names are keys in the `serializer_registry` dict (e.g., 'pickle', 'msgpack'), which maps to the de/serialize functions. The name 'dask' is special, and will use the per-class serialization methods. `None` gives the default list `['dask', 'pickle']`.

Notes on the `iterate_collection` argument (only relevant when `x` is a collection): - `iterate_collection=True`: Serialize collection elements separately. - `iterate_collection=False`: Serialize collection elements together. - `iterate_collection=None` (default): Infer the best setting.

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*, *deserializers=None*)

Convert serialized header and list of bytestrings back to a Python object

Parameters

header [dict]

frames [list of bytes]

deserializers [Optional[Dict[str, Tuple[Callable, Callable, bool]]]] An optional dict mapping a name to a (de)serializer. See *dask_serialize* and *dask_deserialize* for more.

See also:

serialize

distributed.protocol.serialize.**dask_serialize**(*arg*, **args*, ***kwargs*)

Single Dispatch for dask_serialize

distributed.protocol.serialize.**dask_deserialize**(*arg*, **args*, ***kwargs*)

Single Dispatch for dask_deserialize

distributed.protocol.serialize.**register_generic**(*cls*, *serializer_name='dask'*, *serialize_func=<dask.utils.Dispatch object>*, *deserialize_func=<dask.utils.Dispatch object>*)

Register (de)serialize to traverse through __dict__

Normally when registering new classes for Dask's custom serialization you need to manage headers and frames, which can be tedious. If all you want to do is traverse through your object and apply serialize to all of your object's attributes then this function may provide an easier path.

This registers a class for the custom Dask serialization family. It serializes it by traversing through its __dict__ of attributes and applying `serialize` and `deserialize` recursively. It collects a set of frames and keeps small attributes in the header. Deserialization reverses this process.

This is a good idea if the following hold:

1. Most of the bytes of your object are composed of data types that Dask's custom serializtion already handles well, like Numpy arrays.
2. Your object doesn't require any special constructor logic, other than object.__new__(cls)

See also:

dask_serialize

dask_deserialize

Examples

```
>>> import sklearn.base
>>> from distributed.protocol import register_generic
>>> register_generic(sklearn.base.BaseEstimator)
```

3.37 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
```

```
>>> plugin = Counter()
>>> scheduler.add_plugin(plugin)
```

add_client(*scheduler=None*, *client=None*, ***kwargs*)

Run when a new client connects

add_worker(*scheduler=None*, *worker=None*, ***kwargs*)

Run when a new worker enters the cluster

async close()

Run when the scheduler closes down

This runs at the beginning of the Scheduler shutdown process, but after workers have been asked to shut down gracefully

remove_client(*scheduler=None*, *client=None*, ***kwargs*)

Run when a client disconnects

remove_worker(*scheduler=None*, *worker=None*, ***kwargs*)
Run when a worker leaves the cluster

restart(*scheduler*, ***kwargs*)
Run when the scheduler restarts itself

async start(*scheduler*)
Run when the scheduler starts up

This runs at the end of the Scheduler startup process

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

3.37.1 RabbitMQ Example

RabbitMQ is a distributed messaging queue that we can use to post updates about task transitions. By posting transitions to RabbitMQ, we allow other machines to do the processing of transitions and keep scheduler processing to a minimum. See the RabbitMQ tutorial for more information on RabbitMQ and how to consume the messages.

```
import json
from distributed.diagnostics.plugin import SchedulerPlugin
import pika

class RabbitMQPlugin(SchedulerPlugin):
    def __init__(self):
        # Update host to be your RabbitMQ host
        self.connection = pika.BlockingConnection(
            pika.ConnectionParameters(host='localhost'))
        self.channel = self.connection.channel()
        self.channel.queue_declare(queue='dask_task_status', durable=True)

    def transition(self, key, start, finish, *args, **kwargs):
        message = dict(
            key=key,
            start=start,
            finish=finish,
        )
        self.channel.basic_publish(
            exchange='',
            routing_key='dask_task_status',
            body=json.dumps(message),
            properties=pika.BasicProperties(
                delivery_mode=2,  # make message persistent
```

(continues on next page)

(continued from previous page)

```
            ))

@click.command()
def dask_setup(scheduler):
    plugin = RabbitMQPlugin()
    scheduler.add_plugin(plugin)
```

Run with: `dask-scheduler --preload <filename.py>`

3.37.2 Accessing Full Task State

If you would like to access the full *distributed.scheduler.TaskState* stored in the scheduler you can do this by passing and storing a reference to the scheduler as so:

```
from distributed.diagnostics.plugin import SchedulerPlugin

class MyPlugin(SchedulerPlugin):
    def __init__(self, scheduler):
        self.scheduler = scheduler

    def transition(self, key, start, finish, *args, **kwargs):
        # Get full TaskState
        ts = self.scheduler.tasks[key]

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

3.38 Worker Plugins

distributed.diagnostics.plugin.WorkerPlugin provides a base class for creating your own worker plugins. In addition, Dask provides some *built-in plugins*.

class distributed.diagnostics.plugin.**WorkerPlugin**

Interface to extend the Worker

A worker plugin enables custom code to run at different stages of the Workers' lifecycle: at setup, during task state transitions, when a task or dependency is released, and at teardown.

A plugin enables custom code to run at each of step of a Workers's life. Whenever such an event happens, the corresponding method on this class will be called. Note that the user code always runs within the Worker's main thread.

To implement a plugin implement some of the methods of this class and register the plugin to your client in order to have it attached to every existing and future workers with `Client.register_worker_plugin`.

Examples

```
>>> class ErrorLogger(WorkerPlugin):
...     def __init__(self, logger):
...         self.logger = logger
...
...     def setup(self, worker):
...         self.worker = worker
...
...     def transition(self, key, start, finish, *args, **kwargs):
...         if finish == 'error':
...             exc = self.worker.exceptions[key]
...             self.logger.error("Task '%s' has failed with exception: %s" %␣
↪(key, str(exc)))
```

```
>>> plugin = ErrorLogger()
>>> client.register_worker_plugin(plugin)
```

release_key(*key*, *state*, *cause*, *reason*, *report*)
Called when the worker releases a task.

Parameters

key [string]

state [string] State of the released task. One of waiting, ready, executing, long-running, memory, error.

cause [string or None] Additional information on what triggered the release of the task.

reason [None] Not used.

report [bool] Whether the worker should report the released task to the scheduler.

setup(*worker*)
Run when the plugin is attached to a worker. This happens when the plugin is registered and attached to existing workers, or when a worker is created after the plugin has been registered.

teardown(*worker*)
Run when the worker to which the plugin is attached to is closed

transition(*key*, *start*, *finish*, ***kwargs*)
Throughout the lifecycle of a task (see *Worker*), Workers are instructed by the scheduler to compute certain tasks, resulting in transitions in the state of each task. The Worker owning the task is then notified of this state transition.

Whenever a task changes its state, this method will be called.

Parameters

key [string]

start [string] Start state of the transition. One of waiting, ready, executing, long-running, memory, error.

finish [string] Final state of the transition.

kwargs [More options passed when transitioning]

3.38.1 Built-In Worker Plugins

class distributed.diagnostics.plugin.**PipInstall**(*packages*, *pip_options=None*, *restart=False*)

A Worker Plugin to pip install a set of packages

This accepts a set of packages to install on all workers. You can also optionally ask for the worker to restart itself after performing this installation.

Note: This will increase the time it takes to start up each worker. If possible, we recommend including the libraries in the worker environment or image. This is primarily intended for experimentation and debugging.

Additional issues may arise if multiple workers share the same file system. Each worker might try to install the packages simultaneously.

Parameters

packages [List[str]] A list of strings to place after "pip install" command

pip_options [List[str]] Additional options to pass to pip.

restart [bool, default False] Whether or not to restart the worker after pip installing Only functions if the worker has an attached nanny process

Examples

```
>>> from dask.distributed import PipInstall
>>> plugin = PipInstall(packages=["scikit-learn"], pip_options=["--upgrade"])
```

```
>>> client.register_worker_plugin(plugin)
```

class distributed.diagnostics.plugin.**UploadFile**(*filepath*)

A WorkerPlugin to upload a local file to workers.

Parameters

filepath: str A path to the file (.py, egg, or zip) to upload

Examples

```
>>> from distributed.diagnostics.plugin import UploadFile
```

```
>>> client.register_worker_plugin(UploadFile("/path/to/file.py"))
```

INDEX

E

F

G

H

I

L

M

N

P

Q

R

S

T

U

V

W