IPython Parallel Documentation

Release 6.0.2

The IPython Development Team

Feb 23, 2017

Contents

Release 6.0.2

Date Feb 23, 2017

CHAPTER 1

Installing IPython Parallel

As of 4.0, IPython parallel is now a standalone package called *ipyparallel*. You can install it with:

```
pip install ipyparallel
```

or:

```
conda install ipyparallel
```

And if you want the IPython clusters tab extension in your Jupyter Notebook dashboard:

```
ipcluster nbextension enable
```

CHAPTER 2

Contents

Changes in IPython Parallel

6.0.2

Upload fixed sdist for 6.0.1.

6.0.1

Small encoding fix for Python 2.

6.0

Due to a compatibility change and semver, this is a major release. However, it is not a big release. The main compatibility change is that all timestamps are now timezone-aware UTC timestamps. This means you may see comparison errors if you have code that uses datetime objects without timezone info (so-called naïve datetime objects).

Other fixes:

- Rename `Client.become_distributed()` to `Client.become_dask()`. `become_distributed()` remains as an alias.
- import joblib from a public API instead of a private one when using IPython Parallel as a joblib backend.
- Compatibility fix in extensions for security changes in notebook 4.3

5.2

- Fix compatibility with changes in ipykernel 4.3, 4.4
- Improve inspection of `@remote` decorated functions

- `Client.wait()` accepts any Future.
- Add `--user` flag to **`ipcluster nbextension`**
- Default to one core per worker in `Client.become_distributed()`. Override by specifying *ncores* keyword-argument.
- Subprocess logs are no longer sent to files by default in **`ipcluster`**.

5.1

dask, joblib

IPython Parallel 5.1 adds integration with other parallel computing tools, such as dask.distributed and joblib.

To turn an IPython cluster into a dask.distributed cluster, call `become_distributed()`:

```
executor = client.become_distributed(ncores=1)
```

which returns a distributed `Executor` instance.

To register IPython Parallel as the backend for joblib:

```
import ipyparallel as ipp
ipp.register_joblib_backend()
```

nbextensions

IPython parallel now supports the notebook-4.2 API for enabling server extensions, to provide the IPython clusters tab:

```
jupyter serverextension enable --py ipyparallel
jupyter nbextension install --py ipyparallel
jupyter nbextension enable --py ipyparallel
```

though you can still use the more convenient single-call:

```
ipcluster nbextension enable
```

which does all three steps above.

Slurm support

Slurm support is added to ipcluster.

5.1.0

5.1.0 on GitHub

5.0

5.0.1

5.0.1 on GitHub

- Fix imports in `use_cloudpickle()`, `use_dill()`.
- Various typos and documentation updates to catch up with 5.0.

5.0.0

5.0 on GitHub

The highlight of ipyparallel 5.0 is that the Client has been reorganized a bit to use Futures. AsyncResults are now a Future subclass, so they can be *yield* ed in coroutines, etc. Views have also received an Executor interface. This rewrite better connects results to their handles, so the Client.results cache should no longer grow unbounded.

See also:

- The Executor API `ipyparallel.ViewExecutor`
- Creating an Executor from a Client: `ipyparallel.Client.executor()`
- Each View has an `executor` attribute

Part of the Future refactor is that Client IO is now handled in a background thread, which means that `Client.spin_thread()` is obsolete and deprecated.

Other changes:

- Add **`ipcluster nbextension enable|disable`** to toggle the clusters tab in Jupyter notebook

Less interesting development changes for users:

Some IPython-parallel extensions to the IPython kernel have been moved to the ipyparallel package:

- `ipykernel.datapub` is now `ipyparallel.datapub`
- ipykernel Python serialization is now in `ipyparallel.serialize`
- apply_request message handling is implememented in a Kernel subclass, rather than the base ipykernel Kernel.

4.1

4.1 on GitHub

- Add `Client.wait_interactive()`
- Improvements for specifying engines with SSH launcher.

4.0

4.0 on GitHub

First release of `ipyparallel` as a standalone package.

Overview and getting started

Examples

We have various example scripts and notebooks for using ipyparallel in our `examples` directory, or they can be viewed using nbviewer. Some of these are covered in more detail in the *examples* section.

Introduction

This section gives an overview of IPython's sophisticated and powerful architecture for parallel and distributed computing. This architecture abstracts out parallelism in a very general way, which enables IPython to support many different styles of parallelism including:

- Single program, multiple data (SPMD) parallelism.
- Multiple program, multiple data (MPMD) parallelism.
- Message passing using MPI.
- Task farming.
- Data parallel.
- Combinations of these approaches.
- Custom user defined approaches.

Most importantly, IPython enables all types of parallel applications to be developed, executed, debugged and monitored *interactively*. Hence, the `I` in IPython. The following are some example usage cases for IPython:

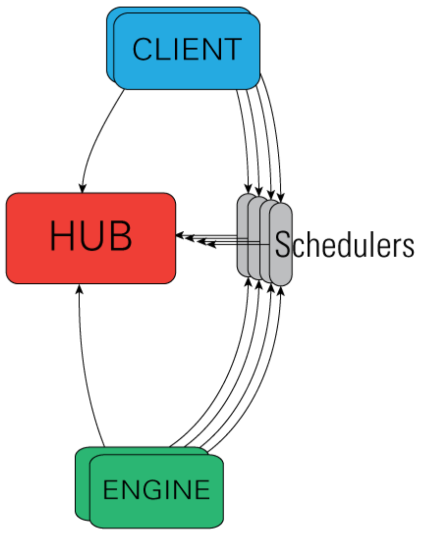
- Quickly parallelize algorithms that are embarrassingly parallel using a number of simple approaches. Many simple things can be parallelized interactively in one or two lines of code.
- Steer traditional MPI applications on a supercomputer from an IPython session on your laptop.
- Analyze and visualize large datasets (that could be remote and/or distributed) interactively using IPython and tools like matplotlib/TVTK.
- Develop, test and debug new parallel algorithms (that may use MPI) interactively.
- Tie together multiple MPI jobs running on different systems into one giant distributed and parallel system.
- Start a parallel job on your cluster and then have a remote collaborator connect to it and pull back data into their local IPython session for plotting and analysis.
- Run a set of tasks on a set of CPUs using dynamic load balancing.

Tip: At the SciPy 2011 conference in Austin, Min Ragan-Kelley presented a complete 4-hour tutorial on the use of these features, and all the materials for the tutorial are now available online. That tutorial provides an excellent, hands-on oriented complement to the reference documentation presented here.

Architecture overview

The IPython architecture consists of four components:

- The IPython engine.
- The IPython hub.



- The IPython schedulers.
- The controller client.

These components live in the *ipyparallel* package and are installed with IPython. They do, however, have additional dependencies that must be installed. For more information, see our installation documentation.

IPython engine

The IPython engine is an extension of the IPython kernel for Jupyter. The engine listens for requests over the network, runs code, and returns results. IPython parallel extends the Jupyter messaging protocol to support native Python object serialization. When multiple engines are started, parallel and distributed computing becomes possible.

IPython controller

The IPython controller processes provide an interface for working with a set of engines. At a general level, the controller is a collection of processes to which IPython engines and clients can connect. The controller is composed of a `Hub` and a collection of `Schedulers`. These Schedulers are typically run in separate processes but on the same machine as the Hub, but can be run anywhere from local threads or on remote machines.

The controller also provides a single point of contact for users who wish to access the engines connected to the controller. There are different ways of working with a controller. In IPython, all of these models are implemented via the `View.apply()` method, after constructing `View` objects to represent subsets of engines. The two primary models for interacting with engines are:

- A **Direct** interface, where engines are addressed explicitly.
- A **LoadBalanced** interface, where the Scheduler is trusted with assigning work to appropriate engines.

Advanced users can readily extend the View models to enable other styles of parallelism.

Note: A single controller and set of engines can be used with multiple models simultaneously. This opens the door for lots of interesting things.

The Hub

The center of an IPython cluster is the Hub. This is the process that keeps track of engine connections, schedulers, clients, as well as all task requests and results. The primary role of the Hub is to facilitate queries of the cluster state, and minimize the necessary information required to establish the many connections involved in connecting new clients and engines.

Schedulers

All actions that can be performed on the engine go through a Scheduler. While the engines themselves block when user code is run, the schedulers hide that from the user to provide a fully asynchronous interface to a set of engines.

IPython client and views

There is one primary object, the `Client`, for connecting to a cluster. For each execution model, there is a corresponding `View`. These views allow users to interact with a set of engines through the interface. Here are the two default views:

- The `DirectView` class for explicit addressing.
- The `LoadBalancedView` class for destination-agnostic scheduling.

Security

IPython uses ZeroMQ for networking, which has provided many advantages, but one of the setbacks is its utter lack of security *[ZeroMQ]*. By default, no IPython connections are encrypted, but open ports only listen on localhost. The only source of encryption for IPython is via ssh-tunnel. IPython supports both shell (*openssh*) and *paramiko* based tunnels for connections. There is a key used to authenticate requests, but due to the lack of encryption, it does not provide significant security if loopback traffic is compromised.

In our architecture, the controller is the only process that listens on network ports, and is thus the main point of vulnerability. The standard model for secure connections is to designate that the controller listen on localhost, and use ssh-tunnels to connect clients and/or engines.

To connect and authenticate to the controller an engine or client needs some information that the controller has stored in a JSON file. Thus, the JSON files need to be copied to a location where the clients and engines can find them. Typically, this is the `~/.ipython/profile_default/security` directory on the host where the client/engine is running (which could be a different host than the controller). Once the JSON files are copied over, everything should work fine.

Currently, there are two JSON files that the controller creates:

ipcontroller-engine.json This JSON file has the information necessary for an engine to connect to a controller.

ipcontroller-client.json The client's connection information. This may not differ from the engine's, but since the controller may listen on different ports for clients and engines, it is stored separately.

ipcontroller-client.json will look something like this, under default localhost circumstances:

```
{
  "url":"tcp:\/\/127.0.0.1:54424",
  "exec_key":"a361fe89-92fc-4762-9767-e2f0a05e3130",
  "ssh":"",
  "location":"10.19.1.135"
}
```

If, however, you are running the controller on a work node on a cluster, you will likely need to use ssh tunnels to connect clients from your laptop to it. You will also probably need to instruct the controller to listen for engines coming from other work nodes on the cluster. An example of ipcontroller-client.json, as created by:

```
$> ipcontroller --ip=* --ssh=login.mycluster.com
```

```
{
  "url":"tcp:\/\/*:54424",
  "exec_key":"a361fe89-92fc-4762-9767-e2f0a05e3130",
  "ssh":"login.mycluster.com",
  "location":"10.0.0.2"
}
```

More details of how these JSON files are used are given below.

A detailed description of the security model and its implementation in IPython can be found *here*.

Warning: Even at its most secure, the Controller listens on ports on localhost, and every time you make a tunnel, you open a localhost port on the connecting machine that points to the Controller. If localhost on the Controller's machine, or the machine of any client or engine, is untrusted, then your Controller is insecure. There is no way around this with ZeroMQ.

Getting Started

To use IPython for parallel computing, you need to start one instance of the controller and one or more instances of the engine. Initially, it is best to simply start a controller and engines on a single host using the **`ipcluster`** command. To start a controller and 4 engines on your localhost, just do:

```
$ ipcluster start -n 4
```

More details about starting the IPython controller and engines can be found *here*

Once you have started the IPython controller and one or more engines, you are ready to use the engines to do something useful. To make sure everything is working correctly, try the following commands:

```
In [1]: import ipyparallel as ipp

In [2]: c = ipp.Client()

In [4]: c.ids
Out[4]: [0, 1, 2, 3]

In [5]: c[:].apply_sync(lambda : "Hello, World")
Out[5]: [ 'Hello, World', 'Hello, World', 'Hello, World', 'Hello, World' ]
```

When a client is created with no arguments, the client tries to find the corresponding JSON file in the local *~/.ipython/profile_default/security* directory. Or if you specified a profile, you can use that with the Client. This should cover most cases:

```
In [2]: c = Client(profile='myprofile')
```

If you have put the JSON file in a different location or it has a different name, create the client like this:

```
In [2]: c = Client('/path/to/my/ipcontroller-client.json')
```

Remember, a client needs to be able to see the Hub's ports to connect. So if they are on a different machine, you may need to use an ssh server to tunnel access to that machine, then you would connect to it with:

```
In [2]: c = Client('/path/to/my/ipcontroller-client.json', sshserver='me@myhub.
↪example.com')
```

Where 'myhub.example.com' is the url or IP address of the machine on which the Hub process is running (or another machine that has direct access to the Hub's ports).

The SSH server may already be specified in ipcontroller-client.json, if the controller was instructed at its launch time.

You are now ready to learn more about the *Direct* and *LoadBalanced* interfaces to the controller.

Starting the IPython controller and engines

To use IPython for parallel computing, you need to start one instance of the controller and one or more instances of the engine. The controller and each engine can run on different machines or on the same machine. Because of this, there are many different possibilities.

Broadly speaking, there are two ways of going about starting a controller and engines:

- In an automated manner using the **`ipcluster`** command.
- In a more manual way using the **`ipcontroller`** and **`ipengine`** commands.

This document describes both of these methods. We recommend that new users start with the **`ipcluster`** command as it simplifies many common usage cases.

General considerations

Before delving into the details about how you can start a controller and engines using the various methods, we outline some of the general issues that come up when starting the controller and engines. These things come up no matter which method you use to start your IPython cluster.

If you are running engines on multiple machines, you will likely need to instruct the controller to listen for connections on an external interface. This can be done by specifying the `ip` argument on the command-line, or the `HubFactory.ip` configurable in `ipcontroller_config.py`.

If your machines are on a trusted network, you can safely instruct the controller to listen on all interfaces with:

```
$> ipcontroller --ip="*"
```

Or you can set the same behavior as the default by adding the following line to your `ipcontroller_config.py`:

```
c.HubFactory.ip = '*'
# c.HubFactory.location = '10.0.1.1'
```

Note: `--ip=*` instructs ZeroMQ to listen on all interfaces, but it does not contain the IP needed for engines / clients to know where the controller actually is. This can be specified with `--location=10.0.0.1`, the specific

IP address of the controller, as seen from engines and/or clients. IPython tries to guess this value by default, but it will not always guess correctly. Check the `location` field in your connection files if you are having connection trouble.

Note: Due to the lack of security in ZeroMQ, the controller will only listen for connections on localhost by default. If you see Timeout errors on engines or clients, then the first thing you should check is the ip address the controller is listening on, and make sure that it is visible from the timing out machine.

See also:

Our notes on security in the new parallel computing code.

Let's say that you want to start the controller on `host0` and engines on hosts `host1-hostn`. The following steps are then required:

1. Start the controller on `host0` by running **`ipcontroller`** on `host0`. The controller must be instructed to listen on an interface visible to the engine machines, via the `ip` command-line argument or `HubFactory.ip` in `ipcontroller_config.py`.
2. Move the JSON file (`ipcontroller-engine.json`) created by the controller from `host0` to hosts `host1-hostn`.
3. Start the engines on hosts `host1-hostn` by running **`ipengine`**. This command has to be told where the JSON file (`ipcontroller-engine.json`) is located.

At this point, the controller and engines will be connected. By default, the JSON files created by the controller are put into the `IPYTHONDIR/profile_default/security` directory. If the engines share a filesystem with the controller, step 2 can be skipped as the engines will automatically look at that location.

The final step required to actually use the running controller from a client is to move the JSON file `ipcontroller-client.json` from `host0` to any host where clients will be run. If these file are put into the `IPYTHONDIR/profile_default/security` directory of the client's host, they will be found automatically. Otherwise, the full path to them has to be passed to the client's constructor.

Using `ipcluster`

The **`ipcluster`** command provides a simple way of starting a controller and engines in the following situations:

1. When the controller and engines are all run on localhost. This is useful for testing or running on a multicore computer.
2. When engines are started using the **`mpiexec`** command that comes with most MPI *[MPI]* implementations
3. When engines are started using the PBS *[PBS]* batch system (or other *qsub* systems, such as SGE).
4. When the controller is started on localhost and the engines are started on remote nodes using **`ssh`**.
5. When engines are started using the Windows HPC Server batch system.

Note: Currently **`ipcluster`** requires that the `IPYTHONDIR/profile_<name>/security` directory live on a shared filesystem that is seen by both the controller and engines. If you don't have a shared file system you will need to use **`ipcontroller`** and **`ipengine`** directly.

Under the hood, **`ipcluster`** just uses **`ipcontroller`** and **`ipengine`** to perform the steps described above.

The simplest way to use ipcluster requires no configuration, and will launch a controller and a number of engines on the local machine. For instance, to start one controller and 4 engines on localhost, just do:

```
$ ipcluster start -n 4
```

To see other command line options, do:

```
$ ipcluster -h
```

Configuring an IPython cluster

Cluster configurations are stored as *profiles*. You can create a new profile with:

```
$ ipython profile create --parallel --profile=myprofile
```

This will create the directory `IPYTHONDIR/profile_myprofile`, and populate it with the default configuration files for the three IPython cluster commands. Once you edit those files, you can continue to call ipcluster/ipcontroller/ipengine with no arguments beyond `profile=myprofile`, and any configuration will be maintained.

There is no limit to the number of profiles you can have, so you can maintain a profile for each of your common use cases. The default profile will be used whenever the profile argument is not specified, so edit `IPYTHONDIR/profile_default/*_config.py` to represent your most common use case.

The configuration files are loaded with commented-out settings and explanations, which should cover most of the available possibilities.

Using various batch systems with `ipcluster`

`ipcluster` has a notion of Launchers that can start controllers and engines with various remote execution schemes. Currently supported models include **`ssh`**, **`mpiexec`**, PBS-style (Torque, SGE, LSF), and Windows HPC Server.

In general, these are configured by the `IPClusterEngines.engine_set_launcher_class`, and `IPClusterStart.controller_launcher_class` configurables, which can be the fully specified object name (e.g. `'ipyparallel.apps.launcher.LocalControllerLauncher'`), but if you are using IPython's builtin launchers, you can specify just the class name, or even just the prefix e.g:

```
c.IPClusterEngines.engine_launcher_class = 'SSH'
# equivalent to
c.IPClusterEngines.engine_launcher_class = 'SSHEngineSetLauncher'
# both of which expand to
c.IPClusterEngines.engine_launcher_class = 'ipyparallel.apps.launcher.
↪SSHEngineSetLauncher'
```

The shortest form being of particular use on the command line, where all you need to do to get an IPython cluster running with engines started with MPI is:

```
$> ipcluster start --engines=MPI
```

Assuming that the default MPI config is sufficient.

Note: shortcuts for builtin launcher names were added in 0.12, as was the `_class` suffix on the configurable names. If you use the old 0.11 names (e.g. `engine_set_launcher`), they will still work, but you will get a deprecation warning that the name has changed.

Note: The Launchers and configuration are designed in such a way that advanced users can subclass and configure them to fit their own system that we have not yet supported (such as Condor)

Using `ipcluster` in mpiexec/mpirun mode

The mpiexec/mpirun mode is useful if you:

1. Have MPI installed.
2. Your systems are configured to use the **`mpiexec`** or **`mpirun`** commands to start MPI processes.

If these are satisfied, you can create a new profile:

```
$ ipython profile create --parallel --profile=mpi
```

and edit the file `IPYTHONDIR/profile_mpi/ipcluster_config.py`.

There, instruct ipcluster to use the MPI launchers by adding the lines:

```
c.IPClusterEngines.engine_launcher_class = 'MPIEngineSetLauncher'
```

If the default MPI configuration is correct, then you can now start your cluster, with:

```
$ ipcluster start -n 4 --profile=mpi
```

This does the following:

1. Starts the IPython controller on current host.
2. Uses **`mpiexec`** to start 4 engines.

If you have a reason to also start the Controller with mpi, you can specify:

```
c.IPClusterStart.controller_launcher_class = 'MPIControllerLauncher'
```

Note: The Controller *will not* be in the same MPI universe as the engines, so there is not much reason to do this unless sysadmins demand it.

On newer MPI implementations (such as OpenMPI), this will work even if you don't make any calls to MPI or call `MPI_Init()`. However, older MPI implementations actually require each process to call `MPI_Init()` upon starting. The easiest way of having this done is to install the mpi4py *[mpi4py]* package and then specify the `c.MPI.use` option in `ipengine_config.py`:

```
c.MPI.use = 'mpi4py'
```

Unfortunately, even this won't work for some MPI implementations. If you are having problems with this, you will likely have to use a custom Python executable that itself calls `MPI_Init()` at the appropriate time. Fortunately, mpi4py comes with such a custom Python executable that is easy to install and use. However, this custom Python executable approach will not work with **`ipcluster`** currently.

More details on using MPI with IPython can be found *here*.

Using `ipcluster` in PBS mode

The PBS mode uses the Portable Batch System (PBS) to start the engines.

As usual, we will start by creating a fresh profile:

```
$ ipython profile create --parallel --profile=pbs
```

And in `ipcluster_config.py`, we will select the PBS launchers for the controller and engines:

```
c.IPClusterStart.controller_launcher_class = 'PBSControllerLauncher'
c.IPClusterEngines.engine_launcher_class = 'PBSEngineSetLauncher'
```

Note: Note that the configurable is IPClusterEngines for the engine launcher, and IPClusterStart for the controller launcher. This is because the start command is a subclass of the engine command, adding a controller launcher. Since it is a subclass, any configuration made in IPClusterEngines is inherited by IPClusterStart unless it is overridden.

IPython does provide simple default batch templates for PBS and SGE, but you may need to specify your own. Here is a sample PBS script template:

```
#PBS -N ipython
#PBS -j oe
#PBS -l walltime=00:10:00
#PBS -l nodes={n//4}:ppn=4
#PBS -q {queue}

cd $PBS_O_WORKDIR
export PATH=$HOME/usr/local/bin
export PYTHONPATH=$HOME/usr/local/lib/python2.7/site-packages
/usr/local/bin/mpiexec -n {n} ipengine --profile-dir={profile_dir}
```

There are a few important points about this template:

1. This template will be rendered at runtime using IPython's `EvalFormatter`. This is simply a subclass of `string.Formatter` that allows simple expressions on keys.
2. Instead of putting in the actual number of engines, use the notation `{n}` to indicate the number of engines to be started. You can also use expressions like `{n//4}` in the template to indicate the number of nodes. There will always be `{n}` and `{profile_dir}` variables passed to the formatter. These allow the batch system to know how many engines, and where the configuration files reside. The same is true for the batch queue, with the template variable `{queue}`.
3. Any options to **`ipengine`** can be given in the batch script template, or in `ipengine_config.py`.
4. Depending on the configuration of you system, you may have to set environment variables in the script template.

The controller template should be similar, but simpler:

```
#PBS -N ipython
#PBS -j oe
#PBS -l walltime=00:10:00
#PBS -l nodes=1:ppn=4
#PBS -q {queue}

cd $PBS_O_WORKDIR
export PATH=$HOME/usr/local/bin
export PYTHONPATH=$HOME/usr/local/lib/python2.7/site-packages
ipcontroller --profile-dir={profile_dir}
```

Once you have created these scripts, save them with names like `pbs.engine.template`. Now you can load them into the `ipcluster_config` with:

```
c.PBSEngineSetLauncher.batch_template_file = "pbs.engine.template"

c.PBSControllerLauncher.batch_template_file = "pbs.controller.template"
```

Alternately, you can just define the templates as strings inside `ipcluster_config`.

Whether you are using your own templates or our defaults, the extra configurables available are the number of engines to launch (`{n}`, and the batch system queue to which the jobs are to be submitted (`{queue}`)). These are configurables, and can be specified in `ipcluster_config`:

```
c.PBSLauncher.queue = 'veryshort.q'
c.IPClusterEngines.n = 64
```

Note that assuming you are running PBS on a multi-node cluster, the Controller's default behavior of listening only on localhost is likely too restrictive. In this case, also assuming the nodes are safely behind a firewall, you can simply instruct the Controller to listen for connections on all its interfaces, by adding in `ipcontroller_config`:

```
c.HubFactory.ip = '*'
```

You can now run the cluster with:

```
$ ipcluster start --profile=pbs -n 128
```

Additional configuration options can be found in the PBS section of `ipcluster_config`.

Note: Due to the flexibility of configuration, the PBS launchers work with simple changes to the template for other **`qsub`**-using systems, such as Sun Grid Engine, and with further configuration in similar batch systems like Condor.

Using `ipcluster` in SSH mode

The SSH mode uses **`ssh`** to execute **`ipengine`** on remote nodes and **`ipcontroller`** can be run remotely as well, or on localhost.

Note: When using this mode it highly recommended that you have set up SSH keys and are using ssh-agent *[SSH]* for password-less logins.

As usual, we start by creating a clean profile:

```
$ ipython profile create --parallel --profile=ssh
```

To use this mode, select the SSH launchers in `ipcluster_config.py`:

```
c.IPClusterEngines.engine_launcher_class = 'SSHEngineSetLauncher'
# and if the Controller is also to be remote:
c.IPClusterStart.controller_launcher_class = 'SSHControllerLauncher'
```

The controller's remote location and configuration can be specified:

```
# Set the user and hostname for the controller
# c.SSHControllerLauncher.hostname = 'controller.example.com'
# c.SSHControllerLauncher.user = os.environ.get('USER','username')
# Set the arguments to be passed to ipcontroller
# note that remotely launched ipcontroller will not get the contents of
# the local ipcontroller_config.py unless it resides on the *remote host*
# in the location specified by the `profile-dir` argument.
# c.SSHControllerLauncher.controller_args = ['--reuse', '--ip=*', '--profile-dir=/
↪path/to/cd']
```

Engines are specified in a dictionary, by hostname and the number of engines to be run on that host.

```
c.SSHEngineSetLauncher.engines = { 'host1.example.com' : 2,
            'host2.example.com' : 5,
            'host3.example.com' : (1, ['--profile-dir=/home/different/location']),
            'host4.example.com' : {'n': 3, 'engine_args': ['--profile-dir=/away/
↪location'], 'engine_cmd': ['/home/venv/bin/python', '-m', 'ipyparallel.engine']},
            'host5.example.com' : 8 }
```

- The *engines* dict, where the keys are the host we want to run engines on and the value is the number of engines to run on that host.
- on host3, the value is a tuple, where the number of engines is first, and the arguments to be passed to **ipengine** are the second element.
- on host4, a dictionary configures the engine. The dictionary can be used to specify the number of engines to be run on that host *n*, the engine arguments *engine_args*, as well as the engine command itself *engine_cmd*. This is particularly useful for virtual environments on heterogeneous clusters where the location of the python executable might vary from host to host.

For engines without explicitly specified arguments, the default arguments are set in a single location:

```
c.SSHEngineSetLauncher.engine_args = ['--profile-dir=/path/to/profile_ssh']
```

Current limitations of the SSH mode of **ipcluster** are:

- Untested and unsupported on Windows. Would require a working **ssh** on Windows. Also, we are using shell scripts to setup and execute commands on remote hosts.

Moving files with SSH

SSH launchers will try to move connection files, controlled by the `to_send` and `to_fetch` configurables. If your machines are on a shared filesystem, this step is unnecessary, and can be skipped by setting these to empty lists:

```
c.SSHLauncher.to_send = []
c.SSHLauncher.to_fetch = []
```

If our default guesses about paths don't work for you, or other files should be moved, you can manually specify these lists as tuples of (local_path, remote_path) for to_send, and (remote_path, local_path) for to_fetch. If you do specify these lists explicitly, IPython *will not* automatically send connection files, so you must include this yourself if they should still be sent/retrieved.

IPython on EC2 with StarCluster

The excellent StarCluster toolkit for managing Amazon EC2 clusters has a plugin which makes deploying IPython on EC2 quite simple. The starcluster plugin uses **ipcluster** with the SGE launchers to distribute engines across the EC2 cluster. See their ipcluster plugin documentation for more information.

Using the `ipcontroller` and `ipengine` commands

It is also possible to use the **`ipcontroller`** and **`ipengine`** commands to start your controller and engines. This approach gives you full control over all aspects of the startup process.

Starting the controller and engine on your local machine

To use **`ipcontroller`** and **`ipengine`** to start things on your local machine, do the following.

First start the controller:

```
$ ipcontroller
```

Next, start however many instances of the engine you want using (repeatedly) the command:

```
$ ipengine
```

The engines should start and automatically connect to the controller using the JSON files in `IPYTHONDIR/profile_default/security`. You are now ready to use the controller and engines from IPython.

Warning: The order of the above operations may be important. You *must* start the controller before the engines, unless you are reusing connection information (via `--reuse`), in which case ordering is not important.

Note: On some platforms (OS X), to put the controller and engine into the background you may need to give these commands in the form `(ipcontroller &)` and `(ipengine &)` (with the parentheses) for them to work properly.

Starting the controller and engines on different hosts

When the controller and engines are running on different hosts, things are slightly more complicated, but the underlying ideas are the same:

1. Start the controller on a host using **`ipcontroller`**. The controller must be instructed to listen on an interface visible to the engine machines, via the `ip` command-line argument or `HubFactory.ip` in `ipcontroller_config.py`:

   ```
   $ ipcontroller --ip=192.168.1.16
   ```

   ```
   # in ipcontroller_config.py
   HubFactory.ip = '192.168.1.16'
   ```

2. Copy `ipcontroller-engine.json` from `IPYTHONDIR/profile_<name>/security` on the controller's host to the host where the engines will run.

3. Use **`ipengine`** on the engine's hosts to start the engines.

The only thing you have to be careful of is to tell **`ipengine`** where the `ipcontroller-engine.json` file is located. There are two ways you can do this:

- Put `ipcontroller-engine.json` in the `IPYTHONDIR/profile_<name>/security` directory on the engine's host, where it will be found automatically.
- Call **`ipengine`** with the `--file=full_path_to_the_file` flag.

The `file` flag works like this:

```
$ ipengine --file=/path/to/my/ipcontroller-engine.json
```

Note: If the controller's and engine's hosts all have a shared file system (`IPYTHONDIR/profile_<name>/security` is the same on all of them), then things will just work!

SSH Tunnels

If your engines are not on the same LAN as the controller, or you are on a highly restricted network where your nodes cannot see each others ports, then you can use SSH tunnels to connect engines to the controller.

Note: This does not work in all cases. Manual tunnels may be an option, but are highly inconvenient. Support for manual tunnels will be improved.

You can instruct all engines to use ssh, by specifying the ssh server in `ipcontroller-engine.json`:

```
{
  "url":"tcp://192.168.1.123:56951",
  "exec_key":"26f4c040-587d-4a4e-b58b-030b96399584",
  "ssh":"user@example.com",
  "location":"192.168.1.123"
}
```

This will be specified if you give the `--enginessh=use@example.com` argument when starting **`ipcontroller`**.

Or you can specify an ssh server on the command-line when starting an engine:

```
$> ipengine --profile=foo --ssh=my.login.node
```

For example, if your system is totally restricted, then all connections will actually be loopback, and ssh tunnels will be used to connect engines to the controller:

```
[node1] $> ipcontroller --enginessh=node1
[node2] $> ipengine
[node3] $> ipcluster engines --n=4
```

Or if you want to start many engines on each node, the command *ipcluster engines –n=4* without any configuration is equivalent to running ipengine 4 times.

An example using ipcontroller/engine with ssh

No configuration files are necessary to use ipcontroller/engine in an SSH environment without a shared filesystem. You simply need to make sure that the controller is listening on an interface visible to the engines, and move the connection file from the controller to the engines.

1. start the controller, listening on an ip-address visible to the engine machines:

   ```
   [controller.host] $ ipcontroller --ip=192.168.1.16

   [IPControllerApp] Using existing profile dir: u'/Users/me/.ipython/profile_default
   ↪'
   ```

```
[IPControllerApp] Hub listening on tcp://192.168.1.16:63320 for registration.
[IPControllerApp] Hub using DB backend: 'ipyparallel.controller.dictdb.DictDB'
[IPControllerApp] hub::created hub
[IPControllerApp] writing connection info to /Users/me/.ipython/profile_default/
↪security/ipcontroller-client.json
[IPControllerApp] writing connection info to /Users/me/.ipython/profile_default/
↪security/ipcontroller-engine.json
[IPControllerApp] task::using Python leastload Task scheduler
[IPControllerApp] Heartmonitor started
[IPControllerApp] Creating pid file: /Users/me/.ipython/profile_default/pid/
↪ipcontroller.pid
Scheduler started [leastload]
```

2. on each engine, fetch the connection file with scp:

```
[engine.host.n] $ scp controller.host:.ipython/profile_default/security/
↪ipcontroller-engine.json ./
```

Note: The log output of ipcontroller above shows you where the json files were written. They will be in `~/.ipython` under `profile_default/security/ipcontroller-engine.json`

3. start the engines, using the connection file:

```
[engine.host.n] $ ipengine --file=./ipcontroller-engine.json
```

A couple of notes:

- You can avoid having to fetch the connection file every time by adding `--reuse` flag to ipcontroller, which instructs the controller to read the previous connection file for connection info, rather than generate a new one with randomized ports.
- In step 2, if you fetch the connection file directly into the security dir of a profile, then you need not specify its path directly, only the profile (assumes the path exists, otherwise you must create it first):

```
[engine.host.n] $ scp controller.host:.ipython/profile_default/security/
↪ipcontroller-engine.json ~/.ipython/profile_ssh/security/
[engine.host.n] $ ipengine --profile=ssh
```

  Of course, if you fetch the file into the default profile, no arguments must be passed to ipengine at all.
- Note that ipengine *did not* specify the ip argument. In general, it is unlikely for any connection information to be specified at the command-line to ipengine, as all of this information should be contained in the connection file written by ipcontroller.

Make JSON files persistent

At fist glance it may seem that that managing the JSON files is a bit annoying. Going back to the house and key analogy, copying the JSON around each time you start the controller is like having to make a new key every time you want to unlock the door and enter your house. As with your house, you want to be able to create the key (or JSON file) once, and then simply use it at any point in the future.

To do this, the only thing you have to do is specify the *–reuse* flag, so that the connection information in the JSON files remains accurate:

```
$ ipcontroller --reuse
```

Then, just copy the JSON files over the first time and you are set. You can start and stop the controller and engines any many times as you want in the future, just make sure to tell the controller to reuse the file.

Note: You may ask the question: what ports does the controller listen on if you don't tell is to use specific ones? The default is to use high random port numbers. We do this for two reasons: i) to increase security through obscurity and ii) to multiple controllers on a given host to start and automatically use different ports.

Log files

All of the components of IPython have log files associated with them. These log files can be extremely useful in debugging problems with IPython and can be found in the directory `IPYTHONDIR/profile_<name>/log`. Sending the log files to us will often help us to debug any problems.

Configuring *ipcontroller*

The IPython Controller takes its configuration from the file `ipcontroller_config.py` in the active profile directory.

Ports and addresses

In many cases, you will want to configure the Controller's network identity. By default, the Controller listens only on loopback, which is the most secure but often impractical. To instruct the controller to listen on a specific interface, you can set the `HubFactory.ip` trait. To listen on all interfaces, simply specify:

```
c.HubFactory.ip = '*'
```

When connecting to a Controller that is listening on loopback or behind a firewall, it may be necessary to specify an SSH server to use for tunnels, and the external IP of the Controller. If you specified that the HubFactory listen on loopback, or all interfaces, then IPython will try to guess the external IP. If you are on a system with VM network devices, or many interfaces, this guess may be incorrect. In these cases, you will want to specify the 'location' of the Controller. This is the IP of the machine the Controller is on, as seen by the clients, engines, or the SSH server used to tunnel connections.

For example, to set up a cluster with a Controller on a work node, using ssh tunnels through the login node, an example `ipcontroller_config.py` might contain:

```
# allow connections on all interfaces from engines
# engines on the same node will use loopback, while engines
# from other nodes will use an external IP
c.HubFactory.ip = '*'

# you typically only need to specify the location when there are extra
# interfaces that may not be visible to peer nodes (e.g. VM interfaces)
c.HubFactory.location = '10.0.1.5'
# or to get an automatic value, try this:
import socket
hostname = socket.gethostname()
# alternate choices for hostname include `socket.getfqdn()`
# or `socket.gethostname() + '.local'`
```

```
ex_ip = socket.gethostbyname_ex(hostname)[-1][-1]
c.HubFactory.location = ex_ip

# now instruct clients to use the login node for SSH tunnels:
c.HubFactory.ssh_server = 'login.mycluster.net'
```

After doing this, your `ipcontroller-client.json` file will look something like this:

```
{
  "url":"tcp:\/\/*:43447",
  "exec_key":"9c7779e4-d08a-4c3b-ba8e-db1f80b562c1",
  "ssh":"login.mycluster.net",
  "location":"10.0.1.5"
}
```

Then this file will be all you need for a client to connect to the controller, tunneling SSH connections through login.mycluster.net.

Database Backend

The Hub stores all messages and results passed between Clients and Engines. For large and/or long-running clusters, it would be unreasonable to keep all of this information in memory. For this reason, we have two database backends: *[MongoDB]* via PyMongo, and SQLite with the stdlib `sqlite`.

MongoDB is our design target, and the dict-like model it uses has driven our design. As far as we are concerned, BSON can be considered essentially the same as JSON, adding support for binary data and datetime objects, and any new database backend must support the same data types.

See also:

MongoDB BSON doc

To use one of these backends, you must set the `HubFactory.db_class` trait:

```
# for a simple dict-based in-memory implementation, use dictdb
# This is the default and the fastest, since it doesn't involve the filesystem
c.HubFactory.db_class = 'ipyparallel.controller.dictdb.DictDB'

# To use MongoDB:
c.HubFactory.db_class = 'ipyparallel.controller.mongodb.MongoDB'

# and SQLite:
c.HubFactory.db_class = 'ipyparallel.controller.sqlitedb.SQLiteDB'

# You can use NoDB to disable the database altogether, in case you don't need
# to reuse tasks or results, and want to keep memory consumption under control.
c.HubFactory.db_class = 'ipyparallel.controller.dictdb.NoDB'
```

When using the proper databases, you can actually allow for tasks to persist from one session to the next by specifying the MongoDB database or SQLite table in which tasks are to be stored. The default is to use a table named for the Hub's Session, which is a UUID, and thus different every time.

```
# To keep persistent task history in MongoDB:
c.MongoDB.database = 'tasks'
```

```
# and in SQLite:
c.SQLiteDB.table = 'tasks'
```

Since MongoDB servers can be running remotely or configured to listen on a particular port, you can specify any arguments you may need to the PyMongo Connection:

```
# positional args to pymongo.Connection
c.MongoDB.connection_args = []

# keyword args to pymongo.Connection
c.MongoDB.connection_kwargs = {}
```

But sometimes you are moving lots of data around quickly, and you don't need that information to be stored for later access, even by other Clients to this same session. For this case, we have a dummy database, which doesn't actually store anything. This lets the Hub stay small in memory, at the obvious expense of being able to access the information that would have been stored in the database (used for task resubmission, requesting results of tasks you didn't submit, etc.). To use this backend, simply pass `--nodb` to **`ipcontroller`** on the command-line, or specify the `NoDB` class in your `ipcontroller_config.py` as described above.

See also:

For more information on the database backends, see the *db backend reference*.

Configuring *ipengine*

The IPython Engine takes its configuration from the file `ipengine_config.py`

The Engine itself also has some amount of configuration. Most of this has to do with initializing MPI or connecting to the controller.

To instruct the Engine to initialize with an MPI environment set up by mpi4py, add:

```
c.MPI.use = 'mpi4py'
```

In this case, the Engine will use our default mpi4py init script to set up the MPI environment prior to execution. We have default init scripts for mpi4py and pytrilinos. If you want to specify your own code to be run at the beginning, specify *c.MPI.init_script*.

You can also specify a file or python command to be run at startup of the Engine:

```
c.IPEngineApp.startup_script = u'/path/to/my/startup.py'

c.IPEngineApp.startup_command = 'import numpy, scipy, mpi4py'
```

These commands/files will be run again, after each

It's also useful on systems with shared filesystems to run the engines in some scratch directory. This can be set with:

```
c.IPEngineApp.work_dir = u'/path/to/scratch/'
```

IPython's Direct interface

The direct, or multiengine, interface represents one possible way of working with a set of IPython engines. The basic idea behind the multiengine interface is that the capabilities of each engine are directly and explicitly exposed to the user. Thus, in the multiengine interface, each engine is given an id that is used to identify the engine and give it work to do. This interface is very intuitive and is designed with interactive usage in mind, and is the best place for new users of IPython to begin.

Starting the IPython controller and engines

To follow along with this tutorial, you will need to start the IPython controller and four IPython engines. The simplest way of doing this is to use the **ipcluster** command:

```
$ ipcluster start -n 4
```

For more detailed information about starting the controller and engines, see our *introduction* to using IPython for parallel computing.

Creating a `DirectView` instance

The first step is to import the IPython `ipyparallel` module and then create a `Client` instance:

```
In [1]: from ipyparallel import Client

In [2]: rc = Client()
```

This form assumes that the default connection information (stored in `ipcontroller-client.json` found in `IPYTHONDIR/profile_default/security`) is accurate. If the controller was started on a remote machine, you must copy that connection file to the client machine, or enter its contents as arguments to the Client constructor:

```
# If you have copied the json connector file from the controller:
In [2]: rc = Client('/path/to/ipcontroller-client.json')
# or to connect with a specific profile you have set up:
In [3]: rc = Client(profile='mpi')
```

To make sure there are engines connected to the controller, users can get a list of engine ids:

```
In [3]: rc.ids
Out[3]: [0, 1, 2, 3]
```

Here we see that there are four engines ready to do work for us.

For direct execution, we will make use of a `DirectView` object, which can be constructed via list-access to the client:

```
In [4]: dview = rc[:] # use all engines
```

See also:

For more information, see the in-depth explanation of *Views*.

Quick and easy parallelism

In many cases, you simply want to apply a Python function to a sequence of objects, but *in parallel*. The client interface provides a simple way of accomplishing this: using the DirectView's `map()` method.

Parallel map

Python's builtin `map()` functions allows a function to be applied to a sequence element-by-element. This type of code is typically trivial to parallelize. In fact, since IPython's interface is all about functions anyway, you can just use the builtin `map()` with a `RemoteFunction`, or a DirectView's `map()` method:

```
In [62]: serial_result = map(lambda x:x**10, range(32))

In [63]: parallel_result = dview.map_sync(lambda x: x**10, range(32))

In [67]: serial_result==parallel_result
Out[67]: True
```

Note: The `DirectView`'s version of `map()` does not do dynamic load balancing. For a load balanced version, use a `LoadBalancedView`.

See also:

`map()` is implemented via `ParallelFunction`.

Remote function decorators

Remote functions are just like normal functions, but when they are called, they execute on one or more engines, rather than locally. IPython provides two decorators:

```
In [10]: @dview.remote(block=True)
   ....: def getpid():
   ....:     import os
   ....:     return os.getpid()
   ....:

In [11]: getpid()
Out[11]: [12345, 12346, 12347, 12348]
```

The `@parallel` decorator creates parallel functions, that break up an element-wise operations and distribute them, reconstructing the result.

```
In [12]: import numpy as np

In [13]: A = np.random.random((64,48))

In [14]: @dview.parallel(block=True)
   ....: def pmul(A,B):
   ....:     return A*B

In [15]: C_local = A*A

In [16]: C_remote = pmul(A,A)

In [17]: (C_local == C_remote).all()
Out[17]: True
```

Calling a `@parallel` function *does not* correspond to map. It is used for splitting element-wise operations that operate on a sequence or array. For `map` behavior, parallel functions do have a map method.

call	pfunc(seq)	pfunc.map(seq)
# of tasks	# of engines (1 per engine)	# of engines (1 per engine)
# of remote calls	# of engines (1 per engine)	`len(seq)`
argument to remote	`seq[i:j]` (sub-sequence)	`seq[i]` (single element)

A quick example to illustrate the difference in arguments for the two modes:

```
In [16]: @dview.parallel(block=True)
   ....: def echo(x):
   ....:     return str(x)
   ....:

In [17]: echo(range(5))
Out[17]: ['[0, 1]', '[2]', '[3]', '[4]']

In [18]: echo.map(range(5))
Out[18]: ['0', '1', '2', '3', '4']
```

See also:

See the `parallel()` and `remote()` decorators for options.

Calling Python functions

The most basic type of operation that can be performed on the engines is to execute Python code or call Python functions. Executing Python code can be done in blocking or non-blocking mode (non-blocking is default) using the `View.execute()` method, and calling functions can be done via the `View.apply()` method.

apply

The main method for doing remote execution (in fact, all methods that communicate with the engines are built on top of it), is `View.apply()`.

We strive to provide the cleanest interface we can, so *apply* has the following signature:

```
view.apply(f, *args, **kwargs)
```

There are various ways to call functions with IPython, and these flags are set as attributes of the View. The `DirectView` has just two of these flags:

dv.block [bool] whether to wait for the result, or return an *`AsyncResult`* object immediately

dv.track [bool] whether to instruct pyzmq to track when zeromq is done sending the message. This is primarily useful for non-copying sends of numpy arrays that you plan to edit in-place. You need to know when it becomes safe to edit the buffer without corrupting the message.

dv.targets [int, list of ints] which targets this view is associated with.

Creating a view is simple: index-access on a client creates a `DirectView`.

```
In [4]: view = rc[1:3]
Out[4]: <DirectView [1, 2]>

In [5]: view.apply<tab>
view.apply  view.apply_async  view.apply_sync
```

For convenience, you can set block temporarily for a single call with the extra sync/async methods.

Blocking execution

In blocking mode, the `DirectView` object (called `dview` in these examples) submits the command to the controller, which places the command in the engines' queues for execution. The `apply()` call then blocks until the engines are done executing the command:

```
In [2]: dview = rc[:] # A DirectView of all engines
In [3]: dview.block=True
In [4]: dview['a'] = 5

In [5]: dview['b'] = 10

In [6]: dview.apply(lambda x: a+b+x, 27)
Out[6]: [42, 42, 42, 42]
```

You can also select blocking execution on a call-by-call basis with the `apply_sync()` method:

```
In [7]: dview.block=False

In [8]: dview.apply_sync(lambda x: a+b+x, 27)
Out[8]: [42, 42, 42, 42]
```

Python commands can be executed as strings on specific engines by using a View's `execute` method:

```
In [6]: rc[::2].execute('c=a+b')

In [7]: rc[1::2].execute('c=a-b')

In [8]: dview['c'] # shorthand for dview.pull('c', block=True)
Out[8]: [15, -5, 15, -5]
```

Non-blocking execution

In non-blocking mode, `apply()` submits the command to be executed and then returns a *`AsyncResult`* object immediately. The *`AsyncResult`* object gives you a way of getting a result at a later time through its `get()` method.

See also:

Docs on the *AsyncResult* object.

This allows you to quickly submit long running commands without blocking your local Python/IPython session:

```
# define our function
In [6]: def wait(t):
   ....:     import time
   ....:     tic = time.time()
   ....:     time.sleep(t)
   ....:     return time.time()-tic

# In non-blocking mode
In [7]: ar = dview.apply_async(wait, 2)

# Now block for the result
In [8]: ar.get()
Out[8]: [2.0006198883056641, 1.9997570514678955, 1.9996809959411621, 2.
↪0003249645233154]
```

```
# Again in non-blocking mode
In [9]: ar = dview.apply_async(wait, 10)

# Poll to see if the result is ready
In [10]: ar.ready()
Out[10]: False

# ask for the result, but wait a maximum of 1 second:
In [45]: ar.get(1)
---------------------------------------------------------------------------
TimeoutError                              Traceback (most recent call last)
/home/you/<ipython-input-45-7cd858bbb8e0> in <module>()
----> 1 ar.get(1)

/path/to/site-packages/IPython/parallel/asyncresult.pyc in get(self, timeout)
     62                 raise self._exception
     63         else:
---> 64             raise error.TimeoutError("Result not ready.")
     65
     66     def ready(self):

TimeoutError: Result not ready.
```

Note: Note the import inside the function. This is a common model, to ensure that the appropriate modules are imported where the task is run. You can also manually import modules into the engine(s) namespace(s) via *view.execute('import numpy')*.

Often, it is desirable to wait until a set of `AsyncResult` objects are done. For this, there is a the method `wait()`. This method takes a tuple of `AsyncResult` objects (or *msg_ids* or indices to the client's History), and blocks until all of the associated results are ready:

```
In [72]: dview.block=False

# A trivial list of AsyncResults objects
In [73]: pr_list = [dview.apply_async(wait, 3) for i in range(10)]

# Wait until all of them are done
In [74]: dview.wait(pr_list)

# Then, their results are ready using get() or the `.r` attribute
In [75]: pr_list[0].get()
Out[75]: [2.9982571601867676, 2.9982588291168213, 2.9987530708312988, 2.
↪9990990161895752]
```

The `block` and `targets` keyword arguments and attributes

Most DirectView methods (excluding `apply()`) accept `block` and `targets` as keyword arguments. As we have seen above, these keyword arguments control the blocking mode and which engines the command is applied to. The `View` class also has `block` and `targets` attributes that control the default behavior when the keyword arguments are not provided. Thus the following logic is used for `block` and `targets`:

- If no keyword argument is provided, the instance attributes are used.
- The Keyword arguments, if provided overrides the instance attributes for the duration of a single call.

The following examples demonstrate how to use the instance attributes:

```
In [16]: dview.targets = [0,2]

In [17]: dview.block = False

In [18]: ar = dview.apply(lambda : 10)

In [19]: ar.get()
Out[19]: [10, 10]

In [20]: dview.targets = rc.ids # all engines (4)

In [21]: dview.block = True

In [22]: dview.apply(lambda : 42)
Out[22]: [42, 42, 42, 42]
```

The `block` and `targets` instance attributes of the `DirectView` also determine the behavior of the parallel magic commands.

See also:

See the documentation of the *Parallel Magics*.

Moving Python objects around

In addition to calling functions and executing code on engines, you can transfer Python objects to and from your IPython session and the engines. In IPython, these operations are called `push()` (sending an object to the engines) and `pull()` (getting an object from the engines).

Basic push and pull

Here are some examples of how you use `push()` and `pull()`:

```
In [38]: dview.push(dict(a=1.03234,b=3453))
Out[38]: [None,None,None,None]

In [39]: dview.pull('a')
Out[39]: [ 1.03234, 1.03234, 1.03234, 1.03234]

In [40]: dview.pull('b', targets=0)
Out[40]: 3453

In [41]: dview.pull(('a','b'))
Out[41]: [ [1.03234, 3453], [1.03234, 3453], [1.03234, 3453], [1.03234, 3453] ]

In [42]: dview.push(dict(c='speed'))
Out[42]: [None,None,None,None]
```

In non-blocking mode `push()` and `pull()` also return *`AsyncResult`* objects:

```
In [48]: ar = dview.pull('a', block=False)

In [49]: ar.get()
Out[49]: [1.03234, 1.03234, 1.03234, 1.03234]
```

Dictionary interface

Since a Python namespace is just a `dict`, `DirectView` objects provide dictionary-style access by key and methods such as `get()` and `update()` for convenience. This make the remote namespaces of the engines appear as a local dictionary. Underneath, these methods call `apply()`:

```
In [51]: dview['a']=['foo','bar']

In [52]: dview['a']
Out[52]: [ ['foo', 'bar'], ['foo', 'bar'], ['foo', 'bar'], ['foo', 'bar'] ]
```

Scatter and gather

Sometimes it is useful to partition a sequence and push the partitions to different engines. In MPI language, this is know as scatter/gather and we follow that terminology. However, it is important to remember that in IPython's `Client` class, `scatter()` is from the interactive IPython session to the engines and `gather()` is from the engines back to the interactive IPython session. For scatter/gather operations between engines, MPI, pyzmq, or some other direct interconnect should be used.

```
In [58]: dview.scatter('a',range(16))
Out[58]: [None,None,None,None]

In [59]: dview['a']
Out[59]: [ [0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15] ]

In [60]: dview.gather('a')
Out[60]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Other things to look at

How to do parallel list comprehensions

In many cases list comprehensions are nicer than using the map function. While we don't have fully parallel list comprehensions, it is simple to get the basic effect using `scatter()` and `gather()`:

```
In [66]: dview.scatter('x',range(64))

In [67]: %px y = [i**10 for i in x]
Parallel execution on engines: [0, 1, 2, 3]

In [68]: y = dview.gather('y')

In [69]: print y
[0, 1, 1024, 59049, 1048576, 9765625, 60466176, 282475249, 1073741824,...]
```

Remote imports

Sometimes you will want to import packages both in your interactive session and on your remote engines. This can be done with the `ContextManager` created by a DirectView's `sync_imports()` method:

```
In [69]: with dview.sync_imports():
   ....:     import numpy
importing numpy on engine(s)
```

Any imports made inside the block will also be performed on the view's engines. sync_imports also takes a *local* boolean flag that defaults to True, which specifies whether the local imports should also be performed. However, support for *local=False* has not been implemented, so only packages that can be imported locally will work this way. Note that the usual renaming of the import handle in the same line like in *import matplotlib.pyplot as plt' does not work on the remote engine, the 'as plt* is ignored remotely, while it executes locally. One could rename the remote handle with *%px plt = pyplot* though after the import.

You can also specify imports via the `@require` decorator. This is a decorator designed for use in Dependencies, but can be used to handle remote imports as well. Modules or module names passed to `@require` will be imported before the decorated function is called. If they cannot be imported, the decorated function will never execute and will fail with an UnmetDependencyError. Failures of single Engines will be collected and raise a CompositeError, as demonstrated in the next section.

```
In [69]: from ipyparallel import require

In [70]: @require('re')
   ....: def findall(pat, x):
   ....:     # re is guaranteed to be available
   ....:     return re.findall(pat, x)

# you can also pass modules themselves, that you already have locally:
In [71]: @require(time)
   ....: def wait(t):
   ....:     time.sleep(t)
   ....:     return t
```

Note: `sync_imports()` does not allow `import foo as bar` syntax, because the assignment represented by the `as bar` part is not available to the import hook.

Parallel exceptions

In the multiengine interface, parallel commands can raise Python exceptions, just like serial commands. But it is a little subtle, because a single parallel command can actually raise multiple exceptions (one for each engine the command was run on). To express this idea, we have a `CompositeError` exception class that will be raised in most cases. The `CompositeError` class is a special type of exception that wraps one or more other types of exceptions. Here is how it works:

```
In [78]: dview.block = True

In [79]: dview.execute("1/0")
[0:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero

[1:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero

[2:execute]:
---------------------------------------------------------------------------
```

```
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero

[3:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero
```

Notice how the error message printed when `CompositeError` is raised has information about the individual exceptions that were raised on each engine. If you want, you can even raise one of these original exceptions:

```
In [79]: from ipyparallel import CompositeError

In [80]: try:
   ....:     dview.execute('1/0', block=True)
   ....: except CompositeError, e:
   ....:     e.raise_exception()
   ....:
   ....:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero
```

If you are working in IPython, you can simple type `%debug` after one of these `CompositeError` exceptions is raised, and inspect the exception instance:

```
In [81]: dview.execute('1/0')
[0:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero

[1:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero

[2:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero

[3:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero

In [82]: %debug
> /.../site-packages/IPython/parallel/client/asyncresult.py(125)get()
    124                 else:
--> 125                     raise self._exception
```

```
    126         else:

# Here, self._exception is the CompositeError instance:

ipdb> e = self._exception
ipdb> e
CompositeError(4)

# we can tab-complete on e to see available methods:
ipdb> e.<TAB>
e.args                  e.message               e.traceback
e.elist                 e.msg
e.ename                 e.print_traceback
e.engine_info           e.raise_exception
e.evalue                e.render_traceback

# We can then display the individual tracebacks, if we want:
ipdb> e.print_traceback(1)
[1:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero
```

Since you might have 100 engines, you probably don't want to see 100 tracebacks for a simple NameError because of a typo. For this reason, CompositeError truncates the list of exceptions it will print to `CompositeError.tb_limit` (default is five). You can change this limit to suit your needs with:

```
In [20]: from ipyparallel import CompositeError
In [21]: CompositeError.tb_limit = 1
In [22]: %px x=z
[0:execute]:
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
----> 1 x=z
NameError: name 'z' is not defined

... 3 more exceptions ...
```

All of this same error handling magic even works in non-blocking mode:

```
In [83]: dview.block=False

In [84]: ar = dview.execute('1/0')

In [85]: ar.get()
[0:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero

... 3 more exceptions ...
```

Parallel Magic Commands

We provide a few IPython magic commands that make it a bit more pleasant to execute Python commands on the engines interactively. These are mainly shortcuts to `DirectView.execute()` and `AsyncResult.display_outputs()` methods respectively.

These magics will automatically become available when you create a Client:

```
In [1]: from ipyparallel import Client
In [2]: rc = Client()
```

The initially active View will have attributes `targets='all', block=True`, which is a blocking view of all engines, evaluated at request time (adding/removing engines will change where this view's tasks will run).

The Magics

%px

The %px magic executes a single Python command on the engines specified by the `targets` attribute of the `DirectView` instance:

```
# import numpy here and everywhere
In [25]: with rc[:].sync_imports():
   ....:     import numpy
importing numpy on engine(s)

In [27]: %px a = numpy.random.rand(2,2)
Parallel execution on engines: [0, 1, 2, 3]

In [28]: %px numpy.linalg.eigvals(a)
Parallel execution on engines: [0, 1, 2, 3]
Out [0:68]: array([ 0.77120707, -0.19448286])
Out [1:68]: array([ 1.10815921,  0.05110369])
Out [2:68]: array([ 0.74625527, -0.37475081])
Out [3:68]: array([ 0.72931905,  0.07159743])

In [29]: %px print 'hi'
Parallel execution on engine(s): all
[stdout:0] hi
[stdout:1] hi
[stdout:2] hi
[stdout:3] hi
```

Since engines are IPython as well, you can even run magics remotely:

```
In [28]: %px %pylab inline
Parallel execution on engine(s): all
[stdout:0]
Populating the interactive namespace from numpy and matplotlib
[stdout:1]
Populating the interactive namespace from numpy and matplotlib
[stdout:2]
Populating the interactive namespace from numpy and matplotlib
[stdout:3]
Populating the interactive namespace from numpy and matplotlib
```

And once in pylab mode with the inline backend, you can make plots and they will be displayed in your frontend if it supports the inline figures (e.g. notebook or qtconsole):

```
In [40]: %px plot(rand(100))
Parallel execution on engine(s): all
<plot0>
<plot1>
<plot2>
<plot3>
Out[0:79]: [<matplotlib.lines.Line2D at 0x10a6286d0>]
Out[1:79]: [<matplotlib.lines.Line2D at 0x10b9476d0>]
Out[2:79]: [<matplotlib.lines.Line2D at 0x110652750>]
Out[3:79]: [<matplotlib.lines.Line2D at 0x10c6566d0>]
```

%%px Cell Magic

%%px can be used as a Cell Magic, which accepts some arguments for controlling the execution.

Targets and Blocking

%%px accepts `--targets` for controlling which engines on which to run, and `--[no]block` for specifying the blocking behavior of this cell, independent of the defaults for the View.

```
In [6]: %%px --targets ::2
   ...: print "I am even"
   ...:
Parallel execution on engine(s): [0, 2]
[stdout:0] I am even
[stdout:2] I am even

In [7]: %%px --targets 1
   ...: print "I am number 1"
   ...:
Parallel execution on engine(s): 1
I am number 1

In [8]: %%px
   ...: print "still 'all' by default"
   ...:
Parallel execution on engine(s): all
[stdout:0] still 'all' by default
[stdout:1] still 'all' by default
[stdout:2] still 'all' by default
[stdout:3] still 'all' by default

In [9]: %%px --noblock
   ...: import time
   ...: time.sleep(1)
   ...: time.time()
   ...:
Async parallel execution on engine(s): all
Out[9]: <AsyncResult: execute>

In [10]: %pxresult
Out[0:12]: 1339454561.069116
Out[1:10]: 1339454561.076752
```

```
Out[2:12]: 1339454561.072837
Out[3:10]: 1339454561.066665
```

See also:

%pxconfig accepts these same arguments for changing the *default* values of targets/blocking for the active View.

Output Display

%%px also accepts a `--group-outputs` argument, which adjusts how the outputs of multiple engines are presented.

See also:

`AsyncResult.display_outputs()` for the grouping options.

```
In [50]: %%px --block --group-outputs=engine
   ....: import numpy as np
   ....: A = np.random.random((2,2))
   ....: ev = numpy.linalg.eigvals(A)
   ....: print ev
   ....: ev.max()
   ....:
Parallel execution on engine(s): all
[stdout:0] [ 0.60640442  0.95919621]
Out [0:73]: 0.9591962130899806
[stdout:1] [ 0.38501813  1.29430871]
Out [1:73]: 1.2943087091452372
[stdout:2] [-0.85925141  0.9387692 ]
Out [2:73]: 0.93876920456230284
[stdout:3] [ 0.37998269  1.24218246]
Out [3:73]: 1.2421824618493817
```

%pxresult

If you are using %px in non-blocking mode, you won't get output. You can use %pxresult to display the outputs of the latest command, just as is done when %px is blocking:

```
In [39]: dv.block = False

In [40]: %px print 'hi'
Async parallel execution on engine(s): all

In [41]: %pxresult
[stdout:0] hi
[stdout:1] hi
[stdout:2] hi
[stdout:3] hi
```

%pxresult simply calls `AsyncResult.display_outputs()` on the most recent request. It accepts the same output-grouping arguments as %%px, so you can use it to view a result in different ways.

%autopx

The %autopx magic switches to a mode where everything you type is executed on the engines until you do %autopx again.

```
In [30]: dv.block=True

In [31]: %autopx
%autopx enabled

In [32]: max_evals = []

In [33]: for i in range(100):
   ....:     a = numpy.random.rand(10,10)
   ....:     a = a+a.transpose()
   ....:     evals = numpy.linalg.eigvals(a)
   ....:     max_evals.append(evals[0].real)
   ....:

In [34]: print "Average max eigenvalue is: %f" % (sum(max_evals)/len(max_evals))
[stdout:0] Average max eigenvalue is: 10.193101
[stdout:1] Average max eigenvalue is: 10.064508
[stdout:2] Average max eigenvalue is: 10.055724
[stdout:3] Average max eigenvalue is: 10.086876

In [35]: %autopx
Auto Parallel Disabled
```

%pxconfig

The default targets and blocking behavior for the magics are governed by the `block` and `targets` attribute of the active View. If you have a handle for the view, you can set these attributes directly, but if you don't, you can change them with the %pxconfig magic:

```
In [3]: %pxconfig --block

In [5]: %px print 'hi'
Parallel execution on engine(s): all
[stdout:0] hi
[stdout:1] hi
[stdout:2] hi
[stdout:3] hi

In [6]: %pxconfig --targets ::2

In [7]: %px print 'hi'
Parallel execution on engine(s): [0, 2]
[stdout:0] hi
[stdout:2] hi

In [8]: %pxconfig --noblock

In [9]: %px print 'are you there?'
Async parallel execution on engine(s): [0, 2]
Out[9]: <AsyncResult: execute>

In [10]: %pxresult
```

```
[stdout:0] are you there?
[stdout:2] are you there?
```

Multiple Active Views

The parallel magics are associated with a particular `DirectView` object. You can change the active view by calling the `activate()` method on any view.

```
In [11]: even = rc[::2]

In [12]: even.activate()

In [13]: %px print 'hi'
Async parallel execution on engine(s): [0, 2]
Out[13]: <AsyncResult: execute>

In [14]: even.block = True

In [15]: %px print 'hi'
Parallel execution on engine(s): [0, 2]
[stdout:0] hi
[stdout:2] hi
```

When activating a View, you can also specify a *suffix*, so that a whole different set of magics are associated with that view, without replacing the existing ones.

```
# restore the original DirecView to the base %px magics
In [16]: rc.activate()
Out[16]: <DirectView all>

In [17]: even.activate('_even')

In [18]: %px print 'hi all'
Parallel execution on engine(s): all
[stdout:0] hi all
[stdout:1] hi all
[stdout:2] hi all
[stdout:3] hi all

In [19]: %px_even print "We aren't odd!"
Parallel execution on engine(s): [0, 2]
[stdout:0] We aren't odd!
[stdout:2] We aren't odd!
```

This suffix is applied to the end of all magics, e.g. %autopx_even, %pxresult_even, etc.

For convenience, the `Client` has a `activate()` method as well, which creates a DirectView with block=True, activates it, and returns the new View.

The initial magics registered when you create a client are the result of a call to `rc.activate()` with default args.

Engines as Kernels

Engines are really the same object as the Kernels used elsewhere in IPython, with the minor exception that engines connect to a controller, while regular kernels bind their sockets, listening for connections from a QtConsole or other frontends.

Sometimes for debugging or inspection purposes, you would like a QtConsole connected to an engine for more direct interaction. You can do this by first instructing the Engine to *also* bind its kernel, to listen for connections:

```
In [50]: %px from ipyparallel import bind_kernel; bind_kernel()
```

Then, if your engines are local, you can start a qtconsole right on the engine(s):

```
In [51]: %px %qtconsole
```

Careful with this one, because if your view is of 16 engines it will start 16 QtConsoles!

Or you can view just the connection info, and work out the right way to connect to the engines, depending on where they live and where you are:

```
In [51]: %px %connect_info
Parallel execution on engine(s): all
[stdout:0]
{
  "stdin_port": 60387,
  "ip": "127.0.0.1",
  "hb_port": 50835,
  "key": "eee2dd69-7dd3-4340-bf3e-7e2e22a62542",
  "shell_port": 55328,
  "iopub_port": 58264
}

Paste the above JSON into a file, and connect with:
    $> ipython <app> --existing <file>
or, if you are local, you can connect with just:
    $> ipython <app> --existing kernel-60125.json
or even just:
    $> ipython <app> --existing
if this is the most recent IPython session you have started.
[stdout:1]
{
  "stdin_port": 61869,
...
```

Note: `%qtconsole` will call `bind_kernel()` on an engine if it hasn't been done already, so you can often skip that first step.

The IPython task interface

The task interface to the cluster presents the engines as a fault tolerant, dynamic load-balanced system of workers. Unlike the multiengine interface, in the task interface the user have no direct access to individual engines. By allowing the IPython scheduler to assign work, this interface is simultaneously simpler and more powerful.

Best of all, the user can use both of these interfaces running at the same time to take advantage of their respective strengths. When the user can break up the user's work into segments that do not depend on previous execution, the task interface is ideal. But it also has more power and flexibility, allowing the user to guide the distribution of jobs, without having to assign tasks to engines explicitly.

Starting the IPython controller and engines

To follow along with this tutorial, you will need to start the IPython controller and four IPython engines. The simplest way of doing this is to use the **`ipcluster`** command:

```
$ ipcluster start -n 4
```

For more detailed information about starting the controller and engines, see our *introduction* to using IPython for parallel computing.

Creating a `LoadBalancedView` instance

The first step is to import the IPython `ipyparallel` module and then create a `Client` instance, and we will also be using a `LoadBalancedView`, here called *lview*:

```
In [1]: from ipyparallel import Client

In [2]: rc = Client()
```

This form assumes that the controller was started on localhost with default configuration. If not, the location of the controller must be given as an argument to the constructor:

```
# for a visible LAN controller listening on an external port:
In [2]: rc = Client('tcp://192.168.1.16:10101')
# or to connect with a specific profile you have set up:
In [3]: rc = Client(profile='mpi')
```

For load-balanced execution, we will make use of a `LoadBalancedView` object, which can be constructed via the client's `load_balanced_view()` method:

```
In [4]: lview = rc.load_balanced_view() # default load-balanced view
```

See also:

For more information, see the in-depth explanation of *Views*.

Quick and easy parallelism

In many cases, you simply want to apply a Python function to a sequence of objects, but *in parallel*. Like the multi-engine interface, these can be implemented via the task interface. The exact same tools can perform these actions in load-balanced ways as well as multiplexed ways: a parallel version of `map()` and `@parallel()` function decorator. If one specifies the argument *balanced=True*, then they are dynamically load balanced. Thus, if the execution time per item varies significantly, you should use the versions in the task interface.

Parallel map

To load-balance `map()` ,simply use a LoadBalancedView:

```
In [62]: lview.block = True

In [63]: serial_result = map(lambda x:x**10, range(32))

In [64]: parallel_result = lview.map(lambda x:x**10, range(32))
In [65]: serial_result==parallel_result
Out[65]: True
```

Parallel function decorator

Parallel functions are just like normal function, but they can be called on sequences and *in parallel*. The multiengine interface provides a decorator that turns any Python function into a parallel function:

```
In [10]: @lview.parallel()
   ....: def f(x):
   ....:     return 10.0*x**4
   ....:

In [11]: f.map(range(32))    # this is done in parallel
Out[11]: [0.0,10.0,160.0,...]
```

Dependencies

Often, pure atomic load-balancing is too primitive for your work. In these cases, you may want to associate some kind of *Dependency* that describes when, where, or whether a task can be run. In IPython, we provide two types of dependencies: *Functional Dependencies* and *Graph Dependencies*

Note: It is important to note that the pure ZeroMQ scheduler does not support dependencies, and you will see errors or warnings if you try to use dependencies with the pure scheduler.

Functional Dependencies

Functional dependencies are used to determine whether a given engine is capable of running a particular task. This is implemented via a special `Exception` class, `UnmetDependency`, found in *ipyparallel.error*. Its use is very simple: if a task fails with an UnmetDependency exception, then the scheduler, instead of relaying the error up to the client like any other error, catches the error, and submits the task to a different engine. This will repeat indefinitely, and a task will never be submitted to a given engine a second time.

You can manually raise the `UnmetDependency` yourself, but IPython has provided some decorators for facilitating this behavior.

There are two decorators and a class used for functional dependencies:

```
In [9]: from ipyparallel import depend, require, dependent
```

@require

The simplest sort of dependency is requiring that a Python module is available. The `@require` decorator lets you define a function that will only run on engines where names you specify are importable:

```
In [10]: @require('numpy', 'zmq')
   ....: def myfunc():
   ....:     return dostuff()
```

Now, any time you apply `myfunc()`, the task will only run on a machine that has numpy and pyzmq available, and when `myfunc()` is called, numpy and zmq will be imported. You can also require specific objects, not just module names:

```
def foo(a):
    return a*a

@parallel.require(foo)
def bar(b):
    return foo(b)

@parallel.require(bar)
def baz(c, d):
    return bar(c) - bar(d)

view.apply_sync(baz, 4, 5)
```

@depend

The `@depend` decorator lets you decorate any function with any *other* function to evaluate the dependency. The dependency function will be called at the start of the task, and if it returns `False`, then the dependency will be considered unmet, and the task will be assigned to another engine. If the dependency returns *anything other than ‘‘False‘‘*, the rest of the task will continue.

```
In [10]: def platform_specific(plat):
   ....:     import sys
   ....:     return sys.platform == plat

In [11]: @depend(platform_specific, 'darwin')
   ....: def mactask():
   ....:     do_mac_stuff()

In [12]: @depend(platform_specific, 'nt')
   ....: def wintask():
   ....:     do_windows_stuff()
```

In this case, any time you apply `mactask`, it will only run on an OSX machine. `@depend` is just like `apply`, in that it has a `@depend(f,*args,**kwargs)` signature.

dependents

You don’t have to use the decorators on your tasks, if for instance you may want to run tasks with a single function but varying dependencies, you can directly construct the `dependent` object that the decorators use:

Graph Dependencies

Sometimes you want to restrict the time and/or location to run a given task as a function of the time and/or location of other tasks. This is implemented via a subclass of `set`, called a `Dependency`. A Dependency is just a set of *msg_ids* corresponding to tasks, and a few attributes to guide how to decide when the Dependency has been met.

The switches we provide for interpreting whether a given dependency set has been met:

any|all Whether the dependency is considered met if *any* of the dependencies are done, or only after *all* of them have finished. This is set by a Dependency’s `all` boolean attribute, which defaults to `True`.

success [default: True] Whether to consider tasks that succeeded as fulfilling dependencies.

failure [default [False]] Whether to consider tasks that failed as fulfilling dependencies. using *failure=True,success=False* is useful for setting up cleanup tasks, to be run only when tasks have failed.

Sometimes you want to run a task after another, but only if that task succeeded. In this case, `success` should be `True` and `failure` should be `False`. However sometimes you may not care whether the task succeeds, and always want the second task to run, in which case you should use *success=failure=True*. The default behavior is to only use successes.

There are other switches for interpretation that are made at the *task* level. These are specified via keyword arguments to the client's `apply()` method.

after,follow You may want to run a task *after* a given set of dependencies have been run and/or run it *where* another set of dependencies are met. To support this, every task has an *after* dependency to restrict time, and a *follow* dependency to restrict destination.

timeout You may also want to set a time-limit for how long the scheduler should wait before a task's dependencies are met. This is done via a *timeout*, which defaults to 0, which indicates that the task should never timeout. If the timeout is reached, and the scheduler still hasn't been able to assign the task to an engine, the task will fail with a `DependencyTimeout`.

Note: Dependencies only work within the task scheduler. You cannot instruct a load-balanced task to run after a job submitted via the MUX interface.

The simplest form of Dependencies is with *all=True,success=True,failure=False*. In these cases, you can skip using Dependency objects, and just pass msg_ids or AsyncResult objects as the *follow* and *after* keywords to `client.apply()`:

```
In [14]: client.block=False

In [15]: ar = lview.apply(f, args, kwargs)

In [16]: ar2 = lview.apply(f2)

In [17]: with lview.temp_flags(after=[ar,ar2]):
   ....:     ar3 = lview.apply(f3)

In [18]: with lview.temp_flags(follow=[ar], timeout=2.5)
   ....:     ar4 = lview.apply(f3)
```

See also:

Some parallel workloads can be described as a Directed Acyclic Graph, or DAG. See *DAG Dependencies* for an example demonstrating how to use map a NetworkX DAG onto task dependencies.

Impossible Dependencies

The schedulers do perform some analysis on graph dependencies to determine whether they are not possible to be met. If the scheduler does discover that a dependency cannot be met, then the task will fail with an `ImpossibleDependency` error. This way, if the scheduler realized that a task can never be run, it won't sit indefinitely in the scheduler clogging the pipeline.

The basic cases that are checked:

- depending on nonexistent messages
- *follow* dependencies were run on more than one machine and *all=True*

- any dependencies failed and *all=True,success=True,failures=False*
- all dependencies failed and *all=False,success=True,failure=False*

Warning: This analysis has not been proven to be rigorous, so it is likely possible for tasks to become impossible to run in obscure situations, so a timeout may be a good choice.

Retries and Resubmit

Retries

Another flag for tasks is *retries*. This is an integer, specifying how many times a task should be resubmitted after failure. This is useful for tasks that should still run if their engine was shutdown, or may have some statistical chance of failing. The default is to not retry tasks.

Resubmit

Sometimes you may want to re-run a task. This could be because it failed for some reason, and you have fixed the error, or because you want to restore the cluster to an interrupted state. For this, the `Client` has a `rc.resubmit()` method. This simply takes one or more msg_ids, and returns an `AsyncHubResult` for the result(s). You cannot resubmit a task that is pending - only those that have finished, either successful or unsuccessful.

Schedulers

There are a variety of valid ways to determine where jobs should be assigned in a load-balancing situation. In IPython, we support several standard schemes, and even make it easy to define your own. The scheme can be selected via the `scheme` argument to **`ipcontroller`**, or in the `TaskScheduler.schemename` attribute of a controller config object.

The built-in routing schemes:

To select one of these schemes, simply do:

```
$ ipcontroller --scheme=<schemename>
for instance:
$ ipcontroller --scheme=lru
```

lru: Least Recently Used

Always assign work to the least-recently-used engine. A close relative of round-robin, it will be fair with respect to the number of tasks, agnostic with respect to runtime of each task.

plainrandom: Plain Random

Randomly picks an engine on which to run.

twobin: Two-Bin Random

Requires numpy

Pick two engines at random, and use the LRU of the two. This is known to be better than plain random in many cases, but requires a small amount of computation.

leastload: Least Load

This is the default scheme

Always assign tasks to the engine with the fewest outstanding tasks (LRU breaks tie).

weighted: Weighted Two-Bin Random

Requires numpy

Pick two engines at random using the number of outstanding tasks as inverse weights, and use the one with the lower load.

Greedy Assignment

Tasks can be assigned greedily as they are submitted. If their dependencies are met, they will be assigned to an engine right away, and multiple tasks can be assigned to an engine at a given time. This limit is set with the `TaskScheduler.hwm` (high water mark) configurable in your `ipcontroller_config.py` config file, with:

```
# the most common choices are:
c.TaskSheduler.hwm = 0 # (minimal latency, default in IPython < 0.13)
# or
c.TaskScheduler.hwm = 1 # (most-informed balancing, default in  0.13)
```

In IPython < 0.13, the default is 0, or no-limit. That is, there is no limit to the number of tasks that can be outstanding on a given engine. This greatly benefits the latency of execution, because network traffic can be hidden behind computation. However, this means that workload is assigned without knowledge of how long each task might take, and can result in poor load-balancing, particularly for submitting a collection of heterogeneous tasks all at once. You can limit this effect by setting hwm to a positive integer, 1 being maximum load-balancing (a task will never be waiting if there is an idle engine), and any larger number being a compromise between load-balancing and latency-hiding.

In practice, some users have been confused by having this optimization on by default, so the default value has been changed to 1 in IPython 0.13. This can be slower, but has more obvious behavior and won't result in assigning too many tasks to some engines in heterogeneous cases.

Pure ZMQ Scheduler

For maximum throughput, the 'pure' scheme is not Python at all, but a C-level `MonitoredQueue` from PyZMQ, which uses a ZeroMQ `DEALER` socket to perform all load-balancing. This scheduler does not support any of the advanced features of the Python `Scheduler`.

Disabled features when using the ZMQ Scheduler:

- **Engine unregistration** Task farming will be disabled if an engine unregisters. Further, if an engine is unregistered during computation, the scheduler may not recover.
- **Dependencies** Since there is no Python logic inside the Scheduler, routing decisions cannot be made based on message content.
- **Early destination notification** The Python schedulers know which engine gets which task, and notify the Hub. This allows graceful handling of Engines coming and going. There is no way to know where ZeroMQ messages have gone, so there is no way to know what tasks are on which engine until they *finish*. This makes recovery from engine shutdown very difficult.

Note: TODO: performance comparisons

More details

The `LoadBalancedView` has many more powerful features that allow quite a bit of flexibility in how tasks are defined and run. The next places to look are in the following classes:

- `LoadBalancedView`
- `AsyncResult`
- `apply()`
- `dependency`

The following is an overview of how to use these classes together:

1. Create a `Client` and `LoadBalancedView`
2. Define some functions to be run as tasks
3. Submit your tasks to using the `apply()` method of your `LoadBalancedView` instance.
4. Use `Client.get_result()` to get the results of the tasks, or use the `AsyncResult.get()` method of the results to wait for and then receive the results.

See also:

A demo of *DAG Dependencies* with NetworkX and IPython.

The AsyncResult object

In non-blocking mode, `apply()` submits the command to be executed and then returns a `AsyncResult` object immediately. The AsyncResult object gives you a way of getting a result at a later time through its `get()` method, but it also collects metadata on execution.

Beyond multiprocessing's AsyncResult

Note: The `AsyncResult` object provides a superset of the interface in `multiprocessing.pool.AsyncResult`. See the official Python documentation for more on the basics of this interface.

Our AsyncResult objects add a number of convenient features for working with parallel results, beyond what is provided by the original AsyncResult.

get_dict

First, is `AsyncResult.get_dict()`, which pulls results as a dictionary keyed by engine_id, rather than a flat list. This is useful for quickly coordinating or distributing information about all of the engines.

As an example, here is a quick call that gives every engine a dict showing the PID of every other engine:

```
In [10]: ar = rc[:].apply_async(os.getpid)
In [11]: pids = ar.get_dict()
In [12]: rc[:]['pid_map'] = pids
```

This trick is particularly useful when setting up inter-engine communication, as in IPython's `examples/parallel/interengine` examples.

Metadata

ipyparallel tracks some metadata about the tasks, which is stored in the `Client.metadata` dict. The AsyncResult object gives you an interface for this information as well, including timestamps stdout/err, and engine IDs.

Timing

IPython tracks various timestamps as `datetime` objects, and the AsyncResult object has a few properties that turn these into useful times (in seconds as floats).

For use while the tasks are still pending:

- `ar.elapsed` is just the elapsed seconds since submission, for use before the AsyncResult is complete.
- `ar.progress` is the number of tasks that have completed. Fractional progress would be:

```
1.0 * ar.progress / len(ar)
```

- `AsyncResult.wait_interactive()` will wait for the result to finish, but print out status updates on progress and elapsed time while it waits.

For use after the tasks are done:

- `ar.serial_time` is the sum of the computation time of all of the tasks done in parallel.
- `ar.wall_time` is the time between the first task submitted and last result received. This is the actual cost of computation, including IPython overhead.

Note: wall_time is only precise if the Client is waiting for results when the task finished, because the *received* timestamp is made when the result is unpacked by the Client, triggered by the `spin()` call. If you are doing work in the Client, and not waiting/spinning, then *received* might be artificially high.

An often interesting metric is the time it actually cost to do the work in parallel relative to the serial computation, and this can be given simply with

```
speedup = ar.serial_time / ar.wall_time
```

Map results are iterable!

When an AsyncResult object has multiple results (e.g. the `AsyncMapResult` object), you can actually iterate through results themselves, and act on them as they arrive:

```
from __future__ import print_function

import time

import ipyparallel as ipp

# create client & view
rc = ipp.Client()
dv = rc[:]
v = rc.load_balanced_view()

# scatter 'id', so id=0,1,2 on engines 0,1,2
dv.scatter('id', rc.ids, flatten=True)
```

```
print("Engine IDs: ", dv['id'])

# create a Reference to `id`. This will be a different value on each engine
ref = ipp.Reference('id')
print("sleeping for `id` seconds on each engine")
tic = time.time()
ar = dv.apply(time.sleep, ref)
for i,r in enumerate(ar):
    print("%i: %.3f"%(i, time.time()-tic))

def sleep_here(t):
    import time
    time.sleep(t)
    return id,t

# one call per task
print("running with one call per task")
amr = v.map(sleep_here, [.01*t for t in range(100)])
tic = time.time()
for i,r in enumerate(amr):
    print("task %i on engine %i: %.3f" % (i, r[0], time.time()-tic))

print("running with four calls per task")
# with chunksize, we can have four calls per task
amr = v.map(sleep_here, [.01*t for t in range(100)], chunksize=4)
tic = time.time()
for i,r in enumerate(amr):
    print("task %i on engine %i: %.3f" % (i, r[0], time.time()-tic))

print("running with two calls per task, with unordered results")
# We can even iterate through faster results first, with ordered=False
amr = v.map(sleep_here, [.01*t for t in range(100,0,-1)], ordered=False, chunksize=2)
tic = time.time()
for i,r in enumerate(amr):
    print("slept %.2fs on engine %i: %.3f" % (r[1], r[0], time.time()-tic))
```

That is to say, if you treat an AsyncMapResult as if it were a list of your actual results, it should behave as you would expect, with the only difference being that you can start iterating through the results before they have even been computed.

This lets you do a dumb version of map/reduce with the builtin Python functions, and the only difference between doing this locally and doing it remotely in parallel is using the asynchronous view.map instead of the builtin map.

Here is a simple one-line RMS (root-mean-square) implemented with Python's builtin map/reduce.

```
In [38]: X = np.linspace(0,100)

In [39]: from math import sqrt

In [40]: add = lambda a,b: a+b

In [41]: sq = lambda x: x*x

In [42]: sqrt(reduce(add, map(sq, X)) / len(X))
Out[42]: 58.028845747399714

In [43]: sqrt(reduce(add, view.map(sq, X)) / len(X))
Out[43]: 58.028845747399714
```

To break that down:

1. `map(sq, X)` Compute the square of each element in the list (locally, or in parallel)
2. `reduce(add, sqX) / len(X)` compute the mean by summing over the list (or AsyncMapResult) and dividing by the size
3. take the square root of the resulting number

See also:

When AsyncResult or the AsyncMapResult don't provide what you need (for instance, handling individual results as they arrive, but with metadata), you can always just split the original result's `msg_ids` attribute, and handle them as you like.

For an example of this, see `examples/customresult.py`

Using MPI with IPython

Often, a parallel algorithm will require moving data between the engines. One way of accomplishing this is by doing a pull and then a push using the multiengine client. However, this will be slow as all the data has to go through the controller to the client and then back through the controller, to its final destination.

A much better way of moving data between engines is to use a message passing library, such as the Message Passing Interface (MPI) *[MPI]*. IPython's parallel computing architecture has been designed from the ground up to integrate with MPI. This document describes how to use MPI with IPython.

Additional installation requirements

If you want to use MPI with IPython, you will need to install:

- A standard MPI implementation such as OpenMPI *[OpenMPI]* or MPICH.
- The mpi4py *[mpi4py]* package.

Note: The mpi4py package is not a strict requirement. However, you need to have *some* way of calling MPI from Python. You also need some way of making sure that `MPI_Init()` is called when the IPython engines start up. There are a number of ways of doing this and a good number of associated subtleties. We highly recommend just using mpi4py as it takes care of most of these problems. If you want to do something different, let us know and we can help you get started.

Starting the engines with MPI enabled

To use code that calls MPI, there are typically two things that MPI requires.

1. The process that wants to call MPI must be started using **`mpiexec`** or a batch system (like PBS) that has MPI support.
2. Once the process starts, it must call `MPI_Init()`.

There are a couple of ways that you can start the IPython engines and get these things to happen.

Automatic starting using `mpiexec` and `ipcluster`

The easiest approach is to use the *MPI* Launchers in **`ipcluster`**, which will first start a controller and then a set of engines using **`mpiexec`**:

```
$ ipcluster start -n 4 --engines=MPIEngineSetLauncher
```

This approach is best as interrupting **`ipcluster`** will automatically stop and clean up the controller and engines.

Manual starting using `mpiexec`

If you want to start the IPython engines using the **`mpiexec`**, just do:

```
$ mpiexec -n 4 ipengine --mpi=mpi4py
```

This requires that you already have a controller running and that the FURL files for the engines are in place. We also have built in support for PyTrilinos *[PyTrilinos]*, which can be used (assuming is installed) by starting the engines with:

```
$ mpiexec -n 4 ipengine --mpi=pytrilinos
```

Automatic starting using PBS and `ipcluster`

The **`ipcluster`** command also has built-in integration with PBS. For more information on this approach, see our documentation on *ipcluster*.

Actually using MPI

Once the engines are running with MPI enabled, you are ready to go. You can now call any code that uses MPI in the IPython engines. And, all of this can be done interactively. Here we show a simple example that uses mpi4py *[mpi4py]* version 1.1.0 or later.

First, lets define a simply function that uses MPI to calculate the sum of a distributed array. Save the following text in a file called `psum.py`:

```
from mpi4py import MPI
import numpy as np

def psum(a):
    locsum = np.sum(a)
    rcvBuf = np.array(0.0,'d')
    MPI.COMM_WORLD.Allreduce([locsum, MPI.DOUBLE],
        [rcvBuf, MPI.DOUBLE],
        op=MPI.SUM)
    return rcvBuf
```

Now, start an IPython cluster:

```
$ ipcluster start --profile=mpi -n 4
```

Note: It is assumed here that the mpi profile has been set up, as described *here*.

Finally, connect to the cluster and use this function interactively. In this case, we create a distributed array and sum up all its elements in a distributed manner using our `psum()` function:

```
In [1]: from ipyparallel import Client

In [2]: c = Client(profile='mpi')

In [3]: view = c[:]

In [4]: view.activate() # enable magics

# run the contents of the file on each engine:
In [5]: view.run('psum.py')

In [6]: view.scatter('a',np.arange(16,dtype='float'))

In [7]: view['a']
Out[7]: [array([ 0.,  1.,  2.,  3.]),
         array([ 4.,  5.,  6.,  7.]),
         array([  8.,   9.,  10.,  11.]),
         array([ 12.,  13.,  14.,  15.])]

In [7]: %px totalsum = psum(a)
Parallel execution on engines: [0,1,2,3]

In [8]: view['totalsum']
Out[8]: [120.0, 120.0, 120.0, 120.0]
```

Any Python code that makes calls to MPI can be used in this manner, including compiled C, C++ and Fortran libraries that have been exposed to Python.

IPython's Task Database

Enabling a DB Backend

The IPython Hub can store all task requests and results in a database. Currently supported backends are: MongoDB, SQLite, and an in-memory DictDB.

This database behavior is optional due to its potential *Cost*, so you must enable one, either at the command-line:

```
$> ipcontroller --dictb # or --mongodb or --sqlitedb
```

or in your `ipcontroller_config.py`:

```
c.HubFactory.db_class = "DictDB"
c.HubFactory.db_class = "MongoDB"
c.HubFactory.db_class = "SQLiteDB"
```

Using the Task Database

The most common use case for this is clients requesting results for tasks they did not submit, via:

```
In [1]: rc.get_result(task_id)
```

However, since we have this DB backend, we provide a direct query method in the `Client` for users who want deeper introspection into their task history. The `db_query()` method of the Client is modeled after MongoDB queries, so if you have used MongoDB it should look familiar. In fact, when the MongoDB backend is in use, the query is relayed directly. When using other backends, the interface is emulated and only a subset of queries is possible.

See also:

MongoDB query docs: https://docs.mongodb.org/manual/tutorial/query-documents/

`Client.db_query()` takes a dictionary query object, with keys from the TaskRecord key list, and values of either exact values to test, or MongoDB queries, which are dicts of The form: `{'operator' : 'argument(s)'}`. There is also an optional *keys* argument, that specifies which subset of keys should be retrieved. The default is to retrieve all keys excluding the request and result buffers. `db_query()` returns a list of TaskRecord dicts. Also like MongoDB, the *msg_id* key will always be included, whether requested or not.

TaskRecord keys:

Key	Type	Description
msg_id	uuid(ascii)	The msg ID
header	dict	The request header
content	dict	The request content (likely empty)
buffers	list(bytes)	buffers containing serialized request objects
submitted	datetime	timestamp for time of submission (set by client)
client_uuid	uuid(ascii)	IDENT of client's socket
engine_uuid	uuid(ascii)	IDENT of engine's socket
started	datetime	time task began execution on engine
completed	datetime	time task finished execution (success or failure) on engine
resubmitted	uuid(ascii)	msg_id of resubmitted task (if applicable)
result_header	dict	header for result
result_content	dict	content for result
result_buffers	list(bytes)	buffers containing serialized request objects
queue	str	The name of the queue for the task ('mux' or 'task')
execute_input	str	Python input source
execute_result	dict	Python output (execute_result message content)
error	dict	Python traceback (error message content)
stdout	str	Stream of stdout data
stderr	str	Stream of stderr data

MongoDB operators we emulate on all backends:

Operator	Python equivalent
'$in'	in
'$nin'	not in
'$eq'	==
'$ne'	!=
'$ge'	>
'$gte'	>=
'$le'	<
'$lte'	<=

The DB Query is useful for two primary cases:

1. deep polling of task status or metadata
2. selecting a subset of tasks, on which to perform a later operation (e.g. wait on result, purge records, resubmit,...)

Example Queries

To get all msg_ids that are not completed, only retrieving their ID and start time:

```
In [1]: incomplete = rc.db_query({'completed' : None}, keys=['msg_id', 'started'])
```

All jobs started in the last hour by me:

```
In [1]: from datetime import datetime, timedelta

In [2]: hourago = datetime.now() - timedelta(1./24)

In [3]: recent = rc.db_query({'started' : {'$gte' : hourago },
                                'client_uuid' : rc.session.session})
```

All jobs started more than an hour ago, by clients *other than me*:

```
In [3]: recent = rc.db_query({'started' : {'$le' : hourago },
                                'client_uuid' : {'$ne' : rc.session.session}})
```

Result headers for all jobs on engine 3 or 4:

```
In [1]: uuids = map(rc._engines.get, (3,4))

In [2]: hist34 = rc.db_query({'engine_uuid' : {'$in' : uuids }, keys='result_header')
```

Cost

The advantage of the database backends is, of course, that large amounts of data can be stored that won't fit in memory. The basic DictDB 'backend' is actually to just store all of this information in a Python dictionary. This is very fast, but will run out of memory quickly if you move a lot of data around, or your cluster is to run for a long time.

Unfortunately, the DB backends (SQLite and MongoDB) right now are rather slow, and can still consume large amounts of resources, particularly if large tasks or results are being created at a high frequency.

For this reason, we have added `NoDB`,a dummy backend that doesn't actually store any information. When you use this database, nothing is stored, and any request for results will result in a KeyError. This obviously prevents later requests for results and task resubmission from functioning, but sometimes those nice features are not as useful as keeping Hub memory under control.

Security details of IPython

Note: This section is not thorough, and IPython.kernel.zmq needs a thorough security audit.

IPython's `IPython.kernel.zmq` package exposes the full power of the Python interpreter over a TCP/IP network for the purposes of parallel computing. This feature brings up the important question of IPython's security model. This document gives details about this model and how it is implemented in IPython's architecture.

Process and network topology

To enable parallel computing, IPython has a number of different processes that run. These processes are discussed at length in the IPython documentation and are summarized here:

- The IPython *engine*. This process is a full blown Python interpreter in which user code is executed. Multiple engines are started to make parallel computing possible.
- The IPython *hub*. This process monitors a set of engines and schedulers, and keeps track of the state of the processes. It listens for registration connections from engines and clients, and monitor connections from schedulers.
- The IPython *schedulers*. This is a set of processes that relay commands and results between clients and engines. They are typically on the same machine as the controller, and listen for connections from engines and clients, but connect to the Hub.
- The IPython *client*. This process is typically an interactive Python process that is used to coordinate the engines to get a parallel computation done.

Collectively, these processes are called the IPython *cluster*, and the hub and schedulers together are referred to as the *controller*.

These processes communicate over any transport supported by ZeroMQ (tcp,pgm,infiniband,ipc) with a well defined topology. The IPython hub and schedulers listen on sockets. Upon starting, an engine connects to a hub and registers itself, which then informs the engine of the connection information for the schedulers, and the engine then connects to the schedulers. These engine/hub and engine/scheduler connections persist for the lifetime of each engine.

The IPython client also connects to the controller processes using a number of socket connections. As of writing, this is one socket per scheduler (4), and 3 connections to the hub for a total of 7. These connections persist for the lifetime of the client only.

A given IPython controller and set of engines engines typically has a relatively short lifetime. Typically this lifetime corresponds to the duration of a single parallel simulation performed by a single user. Finally, the hub, schedulers, engines, and client processes typically execute with the permissions of that same user. More specifically, the controller and engines are *not* executed as root or with any other superuser permissions.

Application logic

When running the IPython kernel to perform a parallel computation, a user utilizes the IPython client to send Python commands and data through the IPython schedulers to the IPython engines, where those commands are executed and the data processed. The design of IPython ensures that the client is the only access point for the capabilities of the engines. That is, the only way of addressing the engines is through a client.

A user can utilize the client to instruct the IPython engines to execute arbitrary Python commands. These Python commands can include calls to the system shell, access the filesystem, etc., as required by the user's application code. From this perspective, when a user runs an IPython engine on a host, that engine has the same capabilities and permissions as the user themselves (as if they were logged onto the engine's host with a terminal).

Secure network connections

Overview

ZeroMQ provides exactly no security. For this reason, users of IPython must be very careful in managing connections, because an open TCP/IP socket presents access to arbitrary execution as the user on the engine machines. As a result, the default behavior of controller processes is to only listen for clients on the loopback interface, and the client must establish SSH tunnels to connect to the controller processes.

Warning: If the controller's loopback interface is untrusted, then IPython should be considered vulnerable, and this extends to the loopback of all connected clients, which have opened a loopback port that is redirected to the controller's loopback port.

SSH

Since ZeroMQ provides no security, SSH tunnels are the primary source of secure connections. A connector file, such as *ipcontroller-client.json*, will contain information for connecting to the controller, possibly including the address of an ssh-server through with the client is to tunnel. The Client object then creates tunnels using either *[OpenSSH]* or *[Paramiko]*, depending on the platform. If users do not wish to use OpenSSH or Paramiko, or the tunneling utilities are insufficient, then they may construct the tunnels themselves, and simply connect clients and engines as if the controller were on loopback on the connecting machine.

Authentication

To protect users of shared machines, *[HMAC]* digests are used to sign messages, using a shared key.

The Session object that handles the message protocol uses a unique key to verify valid messages. This can be any value specified by the user, but the default behavior is a pseudo-random 128-bit number, as generated by *uuid.uuid4()*. This key is used to initialize an HMAC object, which digests all messages, and includes that digest as a signature and part of the message. Every message that is unpacked (on Controller, Engine, and Client) will also be digested by the receiver, ensuring that the sender's key is the same as the receiver's. No messages that do not contain this key are acted upon in any way. The key itself is never sent over the network.

There is exactly one shared key per cluster - it must be the same everywhere. Typically, the controller creates this key, and stores it in the private connection files *ipython-{engine|client}.json*. These files are typically stored in the *~/.ipython/profile_<name>/security* directory, and are maintained as readable only by the owner, just as is common practice with a user's keys in their *.ssh* directory.

Warning: It is important to note that the signatures protect against unauthorized messages, but, as there is no encryption, provide exactly no protection of data privacy. It is possible, however, to use a custom serialization scheme (via Session.packer/unpacker traits) that does incorporate your own encryption scheme.

Specific security vulnerabilities

There are a number of potential security vulnerabilities present in IPython's architecture. In this section we discuss those vulnerabilities and detail how the security architecture described above prevents them from being exploited.

Unauthorized clients

The IPython client can instruct the IPython engines to execute arbitrary Python code with the permissions of the user who started the engines. If an attacker were able to connect their own hostile IPython client to the IPython controller, they could instruct the engines to execute code.

On the first level, this attack is prevented by requiring access to the controller's ports, which are recommended to only be open on loopback if the controller is on an untrusted local network. If the attacker does have access to the Controller's ports, then the attack is prevented by the capabilities based client authentication of the execution key. The relevant authentication information is encoded into the JSON file that clients must present to gain access to the IPython controller. By limiting the distribution of those keys, a user can grant access to only authorized persons, just as with SSH keys.

It is highly unlikely that an execution key could be guessed by an attacker in a brute force guessing attack. A given instance of the IPython controller only runs for a relatively short amount of time (on the order of hours). Thus an attacker would have only a limited amount of time to test a search space of size 2**128. For added security, users can have arbitrarily long keys.

Warning: If the attacker has gained enough access to intercept loopback connections on *either* the controller or client, then a duplicate message can be sent. To protect against this, recipients only allow each signature once, and consider duplicates invalid. However, the duplicate message could be sent to *another* recipient using the same key, and it would be considered valid.

Unauthorized engines

If an attacker were able to connect a hostile engine to a user's controller, the user might unknowingly send sensitive code or data to the hostile engine. This attacker's engine would then have full access to that code and data.

This type of attack is prevented in the same way as the unauthorized client attack, through the usage of the capabilities based authentication scheme.

Unauthorized controllers

It is also possible that an attacker could try to convince a user's IPython client or engine to connect to a hostile IPython controller. That controller would then have full access to the code and data sent between the IPython client and the IPython engines.

Again, this attack is prevented through the capabilities in a connection file, which ensure that a client or engine connects to the correct controller. It is also important to note that the connection files also encode the IP address and port that the controller is listening on, so there is little chance of mistakenly connecting to a controller running on a different IP address and port.

When starting an engine or client, a user must specify the key to use for that connection. Thus, in order to introduce a hostile controller, the attacker must convince the user to use the key associated with the hostile controller. As long as a user is diligent in only using keys from trusted sources, this attack is not possible.

Note: I may be wrong, the unauthorized controller may be easier to fake than this.

Other security measures

A number of other measures are taken to further limit the security risks involved in running the IPython kernel.

First, by default, the IPython controller listens on random port numbers. While this can be overridden by the user, in the default configuration, an attacker would have to do a port scan to even find a controller to attack. When coupled with the relatively short running time of a typical controller (on the order of hours), an attacker would have to work extremely hard and extremely *fast* to even find a running controller to attack.

Second, much of the time, especially when run on supercomputers or clusters, the controller is running behind a firewall. Thus, for engines or client to connect to the controller:

- The different processes have to all be behind the firewall.

or:

- The user has to use SSH port forwarding to tunnel the connections through the firewall.

In either case, an attacker is presented with additional barriers that prevent attacking or even probing the system.

Summary

IPython's architecture has been carefully designed with security in mind. The capabilities based authentication model, in conjunction with SSH tunneled TCP/IP channels, address the core potential vulnerabilities in the system, while still enabling user's to use the system in open networks.

Parallel examples

In this section we describe two more involved examples of using an IPython cluster to perform a parallel computation. We will be doing some plotting, so we start IPython with matplotlib integration by typing:

```
ipython --matplotlib
```

at the system command line. Or you can enable matplotlib integration at any point with:

```
In [1]: %matplotlib
```

150 million digits of pi

In this example we would like to study the distribution of digits in the number pi (in base 10). While it is not known if pi is a normal number (a number is normal in base 10 if 0-9 occur with equal likelihood) numerical investigations suggest that it is. We will begin with a serial calculation on 10,000 digits of pi and then perform a parallel calculation involving 150 million digits.

In both the serial and parallel calculation we will be using functions defined in the `pidigits.py` file, which is available in the `examples/parallel` directory of the IPython source distribution. These functions provide basic facilities for working with the digits of pi and can be loaded into IPython by putting `pidigits.py` in your current working directory and then doing:

```
In [1]: run pidigits.py
```

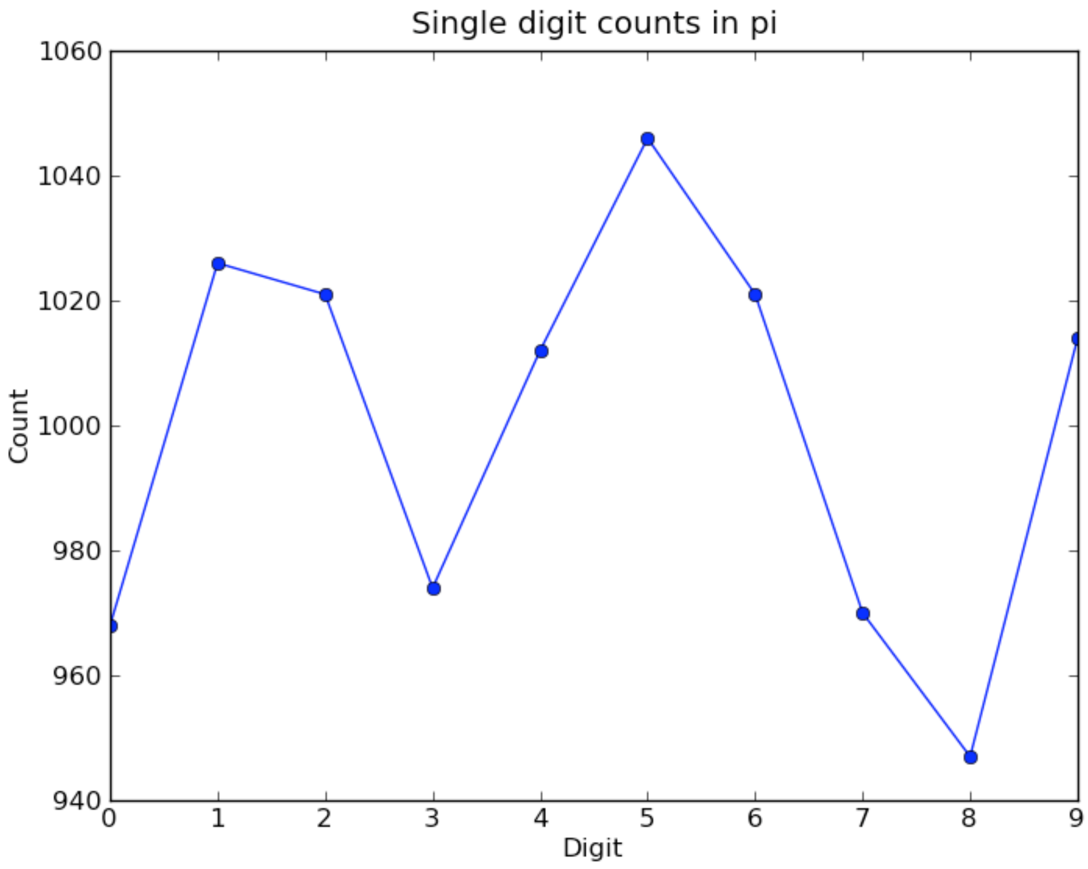
Serial calculation

For the serial calculation, we will use SymPy to calculate 10,000 digits of pi and then look at the frequencies of the digits 0-9. Out of 10,000 digits, we expect each digit to occur 1,000 times. While SymPy is capable of calculating many more digits of pi, our purpose here is to set the stage for the much larger parallel calculation.

In this example, we use two functions from `pidigits.py`: `one_digit_freqs()` (which calculates how many times each digit occurs) and `plot_one_digit_freqs()` (which uses Matplotlib to plot the result). Here is an interactive IPython session that uses these functions with SymPy:

```
In [7]: import sympy

In [8]: pi = sympy.pi.evalf(40)

In [9]: pi
Out[9]: 3.141592653589793238462643383279502884197

In [10]: pi = sympy.pi.evalf(10000)

In [11]: digits = (d for d in str(pi)[2:])  # create a sequence of digits
In [13]: freqs = one_digit_freqs(digits)

In [14]: plot_one_digit_freqs(freqs)
Out[14]: [<matplotlib.lines.Line2D object at 0x18a55290>]
```

The resulting plot of the single digit counts shows that each digit occurs approximately 1,000 times, but that with only 10,000 digits the statistical fluctuations are still rather large:



It is clear that to reduce the relative fluctuations in the counts, we need to look at many more digits of pi. That brings us to the parallel calculation.

Parallel calculation

Calculating many digits of pi is a challenging computational problem in itself. Because we want to focus on the distribution of digits in this example, we will use pre-computed digit of pi from the website of Professor Yasumasa Kanada at the University of Tokyo (http://www.super-computing.org). These digits come in a set of text files (ftp://pi.super-computing.org/.2/pi200m/) that each have 10 million digits of pi.

For the parallel calculation, we have copied these files to the local hard drives of the compute nodes. A total of 15 of these files will be used, for a total of 150 million digits of pi. To make things a little more interesting we will calculate the frequencies of all 2 digits sequences (00-99) and then plot the result using a 2D matrix in Matplotlib.

The overall idea of the calculation is simple: each IPython engine will compute the two digit counts for the digits in a single file. Then in a final step the counts from each engine will be added up. To perform this calculation, we will need two top-level functions from `pidigits.py`, `compute_two_digit_freqs()` and `reduce_freqs()`:

```
def compute_two_digit_freqs(filename):
    """
    Read digits of pi from a file and compute the 2 digit frequencies.
    """
    d = txt_file_to_digits(filename)
    freqs = two_digit_freqs(d)
    return freqs

def reduce_freqs(freqlist):
    """
    Add up a list of freq counts to get the total counts.
    """
    allfreqs = np.zeros_like(freqlist[0])
    for f in freqlist:
        allfreqs += f
    return allfreqs
```

We will also use the `plot_two_digit_freqs()` function to plot the results. The code to run this calculation in parallel is contained in `examples/parallel/parallelpi.py`. This code can be run in parallel using IPython by following these steps:

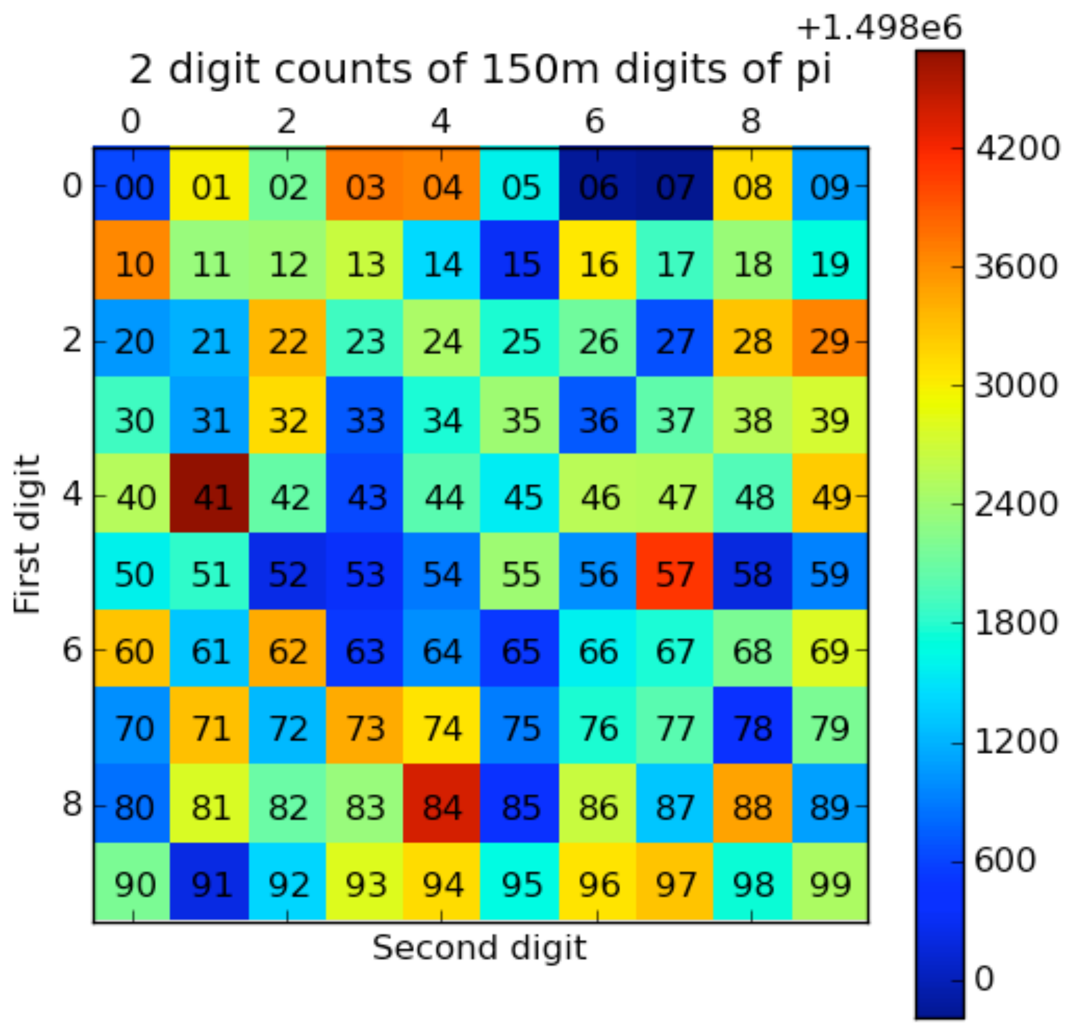
1. Use **`ipcluster`** to start 15 engines. We used 16 cores of an SGE linux cluster (1 controller + 15 engines).
2. With the file `parallelpi.py` in your current working directory, open up IPython, enable matplotlib, and type `run parallelpi.py`. This will download the pi files via ftp the first time you run it, if they are not present in the Engines' working directory.

When run on our 16 cores, we observe a speedup of 14.2x. This is slightly less than linear scaling (16x) because the controller is also running on one of the cores.

To emphasize the interactive nature of IPython, we now show how the calculation can also be run by simply typing the commands from `parallelpi.py` interactively into IPython:

```
In [1]: from ipyparallel import Client

# The Client allows us to use the engines interactively.
# We simply pass Client the name of the cluster profile we
# are using.
In [2]: c = Client(profile='mycluster')
In [3]: v = c[:]

In [3]: c.ids
Out[3]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

In [4]: run pidigits.py

In [5]: filestring = 'pi200m.ascii.%(i)02dof20'

# Create the list of files to process.
In [6]: files = [filestring % {'i':i} for i in range(1,16)]

In [7]: files
Out[7]:
['pi200m.ascii.01of20',
 'pi200m.ascii.02of20',
 'pi200m.ascii.03of20',
 'pi200m.ascii.04of20',
 'pi200m.ascii.05of20',
 'pi200m.ascii.06of20',
 'pi200m.ascii.07of20',
 'pi200m.ascii.08of20',
 'pi200m.ascii.09of20',
 'pi200m.ascii.10of20',
 'pi200m.ascii.11of20',
 'pi200m.ascii.12of20',
 'pi200m.ascii.13of20',
 'pi200m.ascii.14of20',
 'pi200m.ascii.15of20']

# download the data files if they don't already exist:
In [8]: v.map(fetch_pi_file, files)

# This is the parallel calculation using the Client.map method
# which applies compute_two_digit_freqs to each file in files in parallel.
In [9]: freqs_all = v.map(compute_two_digit_freqs, files)

# Add up the frequencies from each engine.
In [10]: freqs = reduce_freqs(freqs_all)

In [11]: plot_two_digit_freqs(freqs)
Out[11]: <matplotlib.image.AxesImage object at 0x18beb110>

In [12]: plt.title('2 digit counts of 150m digits of pi')
Out[12]: <matplotlib.text.Text object at 0x18d1f9b0>
```

The resulting plot generated by Matplotlib is shown below. The colors indicate which two digit sequences are more (red) or less (blue) likely to occur in the first 150 million digits of pi. We clearly see that the sequence “41” is most likely and that “06” and “07” are least likely. Further analysis would show that the relative size of the statistical fluctuations have decreased compared to the 10,000 digit calculation.



Conclusion

To conclude these examples, we summarize the key features of IPython's parallel architecture that have been demonstrated:

- Serial code can be parallelized often with only a few extra lines of code. We have used the `DirectView` and `LoadBalancedView` classes for this purpose.
- The resulting parallel code can be run without ever leaving the IPython's interactive shell.
- Any data computed in parallel can be explored interactively through visualization or further numerical calculations.
- We have run these examples on a cluster running RHEL 5 and Sun GridEngine. IPython's built in support for SGE (and other batch systems) makes it easy to get started with IPython's parallel capabilities.

DAG Dependencies

Often, parallel workflow is described in terms of a Directed Acyclic Graph or DAG. A popular library for working with Graphs is NetworkX. Here, we will walk through a demo mapping a nx DAG to task dependencies.

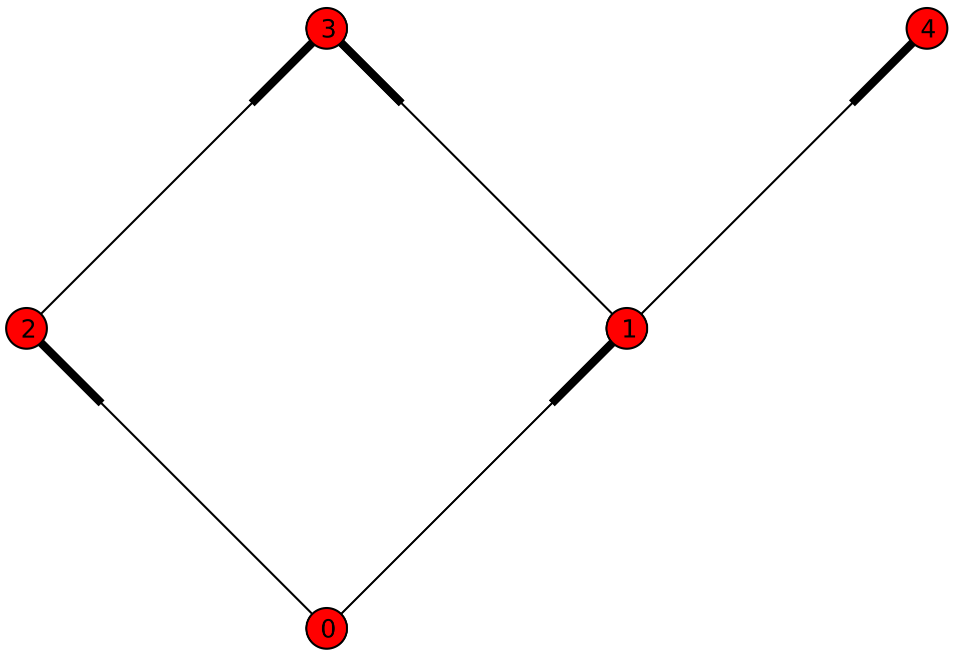
The full script that runs this demo can be found in `examples/parallel/dagdeps.py`.

Why are DAGs good for task dependencies?

The 'G' in DAG is 'Graph'. A Graph is a collection of **nodes** and **edges** that connect the nodes. For our purposes, each node would be a task, and each edge would be a dependency. The 'D' in DAG stands for 'Directed'. This means that each edge has a direction associated with it. So we can interpret the edge (a,b) as meaning that b depends on a, whereas the edge (b,a) would mean a depends on b. The 'A' is 'Acyclic', meaning that there must not be any closed loops in the graph. This is important for dependencies, because if a loop were closed, then a task could ultimately depend on itself, and never be able to run. If your workflow can be described as a DAG, then it is impossible for your dependencies to cause a deadlock.

A Sample DAG

Here, we have a very simple 5-node DAG:



With NetworkX, an arrow is just a fattened bit on the edge. Here, we can see that task 0 depends on nothing, and can run immediately. 1 and 2 depend on 0; 3 depends on 1 and 2; and 4 depends only on 1.

A possible sequence of events for this workflow:

0. Task 0 can run right away
1. 0 finishes, so 1,2 can start
2. 1 finishes, 3 is still waiting on 2, but 4 can start right away
3. 2 finishes, and 3 can finally start

Further, taking failures into account, assuming all dependencies are run with the default *success=True,failure=False*, the following cases would occur for each node's failure:

0. fails: all other tasks fail as Impossible
1. 2 can still succeed, but 3,4 are unreachable
2. 3 becomes unreachable, but 4 is unaffected
3. and 4. are terminal, and can have no effect on other nodes

The code to generate the simple DAG:

```
import networkx as nx

G = nx.DiGraph()

# add 5 nodes, labeled 0-4:
map(G.add_node, range(5))
# 1,2 depend on 0:
G.add_edge(0,1)
G.add_edge(0,2)
# 3 depends on 1,2
G.add_edge(1,3)
G.add_edge(2,3)
# 4 depends on 1
G.add_edge(1,4)

# now draw the graph:
pos = { 0 : (0,0), 1 : (1,1), 2 : (-1,1),
        3 : (0,2), 4 : (2,2)}
nx.draw(G, pos, edge_color='r')
```

For demonstration purposes, we have a function that generates a random DAG with a given number of nodes and edges.

```
def random_dag(nodes, edges):
    """Generate a random Directed Acyclic Graph (DAG) with a given number of nodes␣
↪and edges."""
    G = nx.DiGraph()
    for i in range(nodes):
        G.add_node(i)
    while edges > 0:
        a = randint(0,nodes-1)
        b=a
        while b==a:
            b = randint(0,nodes-1)
        G.add_edge(a,b)
        if nx.is_directed_acyclic_graph(G):
            edges -= 1
        else:
            # we closed a loop!
            G.remove_edge(a,b)
    return G
```

So first, we start with a graph of 32 nodes, with 128 edges:

```
In [2]: G = random_dag(32,128)
```

Now, we need to build our dict of jobs corresponding to the nodes on the graph:

```
In [3]: jobs = {}

# in reality, each job would presumably be different
# randomwait is just a function that sleeps for a random interval
In [4]: for node in G:
   ...:     jobs[node] = randomwait
```

Once we have a dict of jobs matching the nodes on the graph, we can start submitting jobs, and linking up the dependencies. Since we don't know a job's msg_id until it is submitted, which is necessary for building dependencies, it is critical that we don't submit any jobs before other jobs it may depend on. Fortunately, NetworkX provides a `topological_sort()` method which ensures exactly this. It presents an iterable, that guarantees that when you arrive at a node, you have already visited all the nodes it on which it depends:

```
In [5]: rc = Client()
In [5]: view = rc.load_balanced_view()

In [6]: results = {}

In [7]: for node in nx.topological_sort(G):
   ...:    # get list of AsyncResult objects from nodes
   ...:    # leading into this one as dependencies
   ...:    deps = [ results[n] for n in G.predecessors(node) ]
   ...:    # submit and store AsyncResult object
   ...:    with view.temp_flags(after=deps, block=False):
   ...:        results[node] = view.apply(jobs[node])
```

Now that we have submitted all the jobs, we can wait for the results:

```
In [8]: view.wait(results.values())
```

Now, at least we know that all the jobs ran and did not fail (`r.get()` would have raised an error if a task failed). But we don't know that the ordering was properly respected. For this, we can use the `metadata` attribute of each AsyncResult.

These objects store a variety of metadata about each task, including various timestamps. We can validate that the dependencies were respected by checking that each task was started after all of its predecessors were completed:

```
def validate_tree(G, results):
    """Validate that jobs executed after their dependencies."""
    for node in G:
        started = results[node].metadata.started
        for parent in G.predecessors(node):
            finished = results[parent].metadata.completed
            assert started > finished, "%s should have happened after %s"%(node,␣
↪parent)
```

We can also validate the graph visually. By drawing the graph with each node's x-position as its start time, all arrows must be pointing to the right if dependencies were respected. For spreading, the y-position will be the runtime of the task, so long tasks will be at the top, and quick, small tasks will be at the bottom.

```
In [10]: from matplotlib.dates import date2num

In [11]: from matplotlib.cm import gist_rainbow

In [12]: pos = {}; colors = {}

In [12]: for node in G:
   ....:     md = results[node].metadata
   ....:     start = date2num(md.started)
   ....:     runtime = date2num(md.completed) - start
   ....:     pos[node] = (start, runtime)
   ....:     colors[node] = md.engine_id

In [13]: nx.draw(G, pos, node_list=colors.keys(), node_color=colors.values(),
   ....:     cmap=gist_rainbow)
```

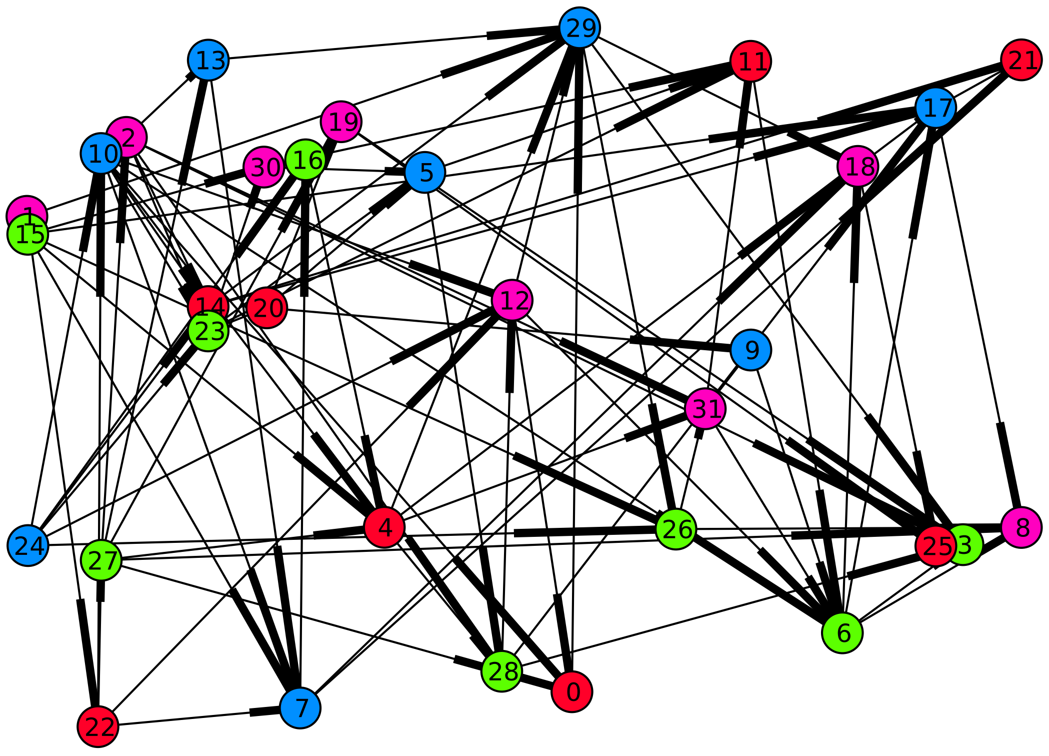


Fig. 2.1: Time started on x, runtime on y, and color-coded by engine-id (in this case there were four engines). Edges denote dependencies.

Details of Parallel Computing with IPython

Note: There are still many sections to fill out in this doc

Caveats

First, some caveats about the detailed workings of parallel computing with 0MQ and IPython.

Non-copying sends and numpy arrays

When numpy arrays are passed as arguments to apply or via data-movement methods, they are not copied. This means that you must be careful if you are sending an array that you intend to work on. PyZMQ does allow you to track when a message has been sent so you can know when it is safe to edit the buffer, but IPython only allows for this.

It is also important to note that the non-copying receive of a message is *read-only*. That means that if you intend to work in-place on an array that you have sent or received, you must copy it. This is true for both numpy arrays sent to engines and numpy arrays retrieved as results.

The following will fail:

```
In [3]: A = numpy.zeros(2)

In [4]: def setter(a):
   ...:   a[0]=1
   ...:   return a

In [5]: rc[0].apply_sync(setter, A)
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)<string>␣
↪in <module>()
<ipython-input-12-c3e7afeb3075> in setter(a)
RuntimeError: array is not writeable
```

If you do need to edit the array in-place, just remember to copy the array if it's read-only. The `ndarray.flags.writeable` flag will tell you if you can write to an array.

```
In [3]: A = numpy.zeros(2)

In [4]: def setter(a):
   ...:     """only copy read-only arrays"""
   ...:     if not a.flags.writeable:
   ...:         a=a.copy()
   ...:     a[0]=1
   ...:     return a

In [5]: rc[0].apply_sync(setter, A)
Out[5]: array([ 1.,  0.])

# note that results will also be read-only:
In [6]: _.flags.writeable
Out[6]: False
```

If you want to safely edit an array in-place after *sending* it, you must use the *track=True* flag. IPython always performs non-copying sends of arrays, which return immediately. You must instruct IPython track those messages *at send time* in order to know for sure that the send has completed. AsyncResults have a `sent` property, and `wait_on_send()` method for checking and waiting for 0MQ to finish with a buffer.

```
In [5]: A = numpy.random.random((1024,1024))

In [6]: view.track=True

In [7]: ar = view.apply_async(lambda x: 2*x, A)

In [8]: ar.sent
Out[8]: False

In [9]: ar.wait_on_send() # blocks until sent is True
```

What is sendable?

If IPython doesn't know what to do with an object, it will pickle it. There is a short list of objects that are not pickled: `buffers/memoryviews`, `bytes` objects, and `numpy` arrays. These are handled specially by IPython in order to prevent extra in-memory copies of data. Sending bytes or numpy arrays will result in exactly zero in-memory copies of your data (unless the data is very small).

If you have an object that provides a Python buffer interface, then you can always send that buffer without copying - and reconstruct the object on the other side in your own code. It is possible that the object reconstruction will become extensible, so you can add your own non-copying types, but this does not yet exist.

Closures

Just about anything in Python is pickleable. The one notable exception is objects (generally functions) with *closures*. Closures can be a complicated topic, but the basic principle is that functions that refer to variables in their parent scope have closures.

An example of a function that uses a closure:

```
def f(a):
    def inner():
        # inner will have a closure
        return a
    return inner

f1 = f(1)
f2 = f(2)
f1() # returns 1
f2() # returns 2
```

`f1` and `f2` will have closures referring to the scope in which *inner* was defined, because they use the variable 'a'. As a result, you would not be able to send `f1` or `f2` with IPython. Note that you *would* be able to send *f*. This is only true for interactively defined functions (as are often used in decorators), and only when there are variables used inside the inner function, that are defined in the outer function. If the names are *not* in the outer function, then there will not be a closure, and the generated function will look in `globals()` for the name:

```
def g(b):
    # note that `b` is not referenced in inner's scope
    def inner():
        # this inner will *not* have a closure
        return a
    return inner
g1 = g(1)
g2 = g(2)
```

```
g1() # raises NameError on 'a'
a=5
g2() # returns 5
```

g1 and *g2 will* be sendable with IPython, and will treat the engine's namespace as globals(). The `pull()` method is implemented based on this principle. If we did not provide pull, you could implement it yourself with *apply*, by simply returning objects out of the global namespace:

```
In [10]: view.apply(lambda : a)

# is equivalent to
In [11]: view.pull('a')
```

You can send functions with closures if you enable using dill or cloudpickle:

```
In [10]: rc[:].use_cloudpickle()
```

which will use a more advanced pickling library, which covers things like closures.

Running Code

There are two principal units of execution in Python: strings of Python code (e.g. 'a=5'), and Python functions. IPython is designed around the use of functions via the core Client method, called *apply*.

Apply

The principal method of remote execution is `apply()`, of `View` objects. The Client provides the full execution and communication API for engines via its low-level `send_apply_message()` method, which is used by all higher level methods of its Views.

f [function] The function to be called remotely

args [tuple/list] The positional arguments passed to *f*

kwargs [dict] The keyword arguments passed to *f*

flags for all views:

block [bool (default: view.block)] Whether to wait for the result, or return immediately.

> **False:** returns AsyncResult
>
> **True:** returns actual result(s) of `f(*args, **kwargs)`
>
> > **if multiple targets:** list of results, matching *targets*

track [bool [default view.track]] whether to track non-copying sends.

targets [int,list of ints, 'all', None [default view.targets]] Specify the destination of the job.

> **if 'all' or None:** Run on all active engines
>
> **if list:** Run on each specified engine
>
> **if int:** Run on single engine

Note: `LoadBalancedView` uses targets to restrict possible destinations. LoadBalanced calls will always execute in just one location.

flags only in LoadBalancedViews:

after [Dependency or collection of msg_ids] Only for load-balanced execution (targets=None) Specify a list of msg_ids as a time-based dependency. This job will only be run *after* the dependencies have been met.

follow [Dependency or collection of msg_ids] Only for load-balanced execution (targets=None) Specify a list of msg_ids as a location-based dependency. This job will only be run on an engine where this dependency is met.

timeout [float/int or None] Only for load-balanced execution (targets=None) Specify an amount of time (in seconds) for the scheduler to wait for dependencies to be met before failing with a DependencyTimeout.

execute and run

For executing strings of Python code, `DirectView` 's also provide an `execute()` and a `run()` method, which rather than take functions and arguments, take simple strings. *execute* simply takes a string of Python code to execute, and sends it to the Engine(s). *run* is the same as *execute*, but for a *file*, rather than a string. It is simply a wrapper that does something very similar to `execute(open(f).read())`.

Note: TODO: Examples for execute and run

Views

The principal extension of the `Client` is the `View` class. The client is typically a singleton for connecting to a cluster, and presents a low-level interface to the Hub and Engines. Most real usage will involve creating one or more `View` objects for working with engines in various ways.

DirectView

The `DirectView` is the class for the IPython *Multiplexing Interface*.

Creating a DirectView

DirectViews can be created in two ways, by index access to a client, or by a client's `view()` method. Index access to a Client works in a few ways. First, you can create DirectViews to single engines simply by accessing the client by engine id:

```
In [2]: rc[0]
Out[2]: <DirectView 0>
```

You can also create a DirectView with a list of engines:

```
In [2]: rc[0,1,2]
Out[2]: <DirectView [0,1,2]>
```

Other methods for accessing elements, such as slicing and negative indexing, work by passing the index directly to the client's `ids` list, so:

```
# negative index
In [2]: rc[-1]
Out[2]: <DirectView 3>

# or slicing:
```

```
In [3]: rc[::2]
Out[3]: <DirectView [0,2]>
```

are always the same as:

```
In [2]: rc[rc.ids[-1]]
Out[2]: <DirectView 3>

In [3]: rc[rc.ids[::2]]
Out[3]: <DirectView [0,2]>
```

Also note that the slice is evaluated at the time of construction of the DirectView, so the targets will not change over time if engines are added/removed from the cluster.

Execution via DirectView

The DirectView is the simplest way to work with one or more engines directly (hence the name).

For instance, to get the process ID of all your engines:

```
In [5]: import os

In [6]: dview.apply_sync(os.getpid)
Out[6]: [1354, 1356, 1358, 1360]
```

Or to see the hostname of the machine they are on:

```
In [5]: import socket

In [6]: dview.apply_sync(socket.gethostname)
Out[6]: ['tesla', 'tesla', 'edison', 'edison', 'edison']
```

Note: TODO: expand on direct execution

Data movement via DirectView

Since a Python namespace is just a `dict`, `DirectView` objects provide dictionary-style access by key and methods such as `get()` and `update()` for convenience. This make the remote namespaces of the engines appear as a local dictionary. Underneath, these methods call `apply()`:

```
In [51]: dview['a']=['foo','bar']

In [52]: dview['a']
Out[52]: [ ['foo', 'bar'], ['foo', 'bar'], ['foo', 'bar'], ['foo', 'bar'] ]
```

Scatter and gather

Sometimes it is useful to partition a sequence and push the partitions to different engines. In MPI language, this is know as scatter/gather and we follow that terminology. However, it is important to remember that in IPython's `Client` class, `scatter()` is from the interactive IPython session to the engines and `gather()` is from the engines back to the interactive IPython session. For scatter/gather operations between engines, MPI should be used:

```
In [58]: dview.scatter('a',range(16))
Out[58]: [None,None,None,None]

In [59]: dview['a']
Out[59]: [ [0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15] ]

In [60]: dview.gather('a')
Out[60]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Push and pull

`push()`

`pull()`

Note: TODO: write this section

LoadBalancedView

The `LoadBalancedView` is the class for load-balanced execution via the task scheduler. These views always run tasks on exactly one engine, but let the scheduler determine where that should be, allowing load-balancing of tasks. The LoadBalancedView does allow you to specify restrictions on where and when tasks can execute, for more complicated load-balanced workflows.

Data Movement

Since the `LoadBalancedView` does not know where execution will take place, explicit data movement methods like push/pull and scatter/gather do not make sense, and are not provided.

Results

AsyncResults

Our primary representation of the results of remote execution is the *`AsyncResult`* object, based on the object of the same name in the built-in `multiprocessing.pool` module. Our version provides a superset of that interface.

The basic principle of the AsyncResult is the encapsulation of one or more results not yet completed. Execution methods (including data movement, such as push/pull) will all return AsyncResults when *block=False*.

The mp.pool.AsyncResult interface

The basic interface of the AsyncResult is exactly that of the AsyncResult in `multiprocessing.pool`, and consists of four methods:

class **`AsyncResult`**
    The stdlib AsyncResult spec

    `wait`([*timeout*])
        Wait until the result is available or until *timeout* seconds pass. This method always returns `None`.

ready()
: Return whether the call has completed.

successful()
: Return whether the call completed without raising an exception. Will raise `AssertionError` if the result is not ready.

get([*timeout*])
: Return the result when it arrives. If *timeout* is not `None` and the result does not arrive within *timeout* seconds then `TimeoutError` is raised. If the remote call raised an exception then that exception will be reraised as a `RemoteError` by *get()*.

While an AsyncResult is not done, you can check on it with its `ready()` method, which will return whether the AR is done. You can also wait on an AsyncResult with its `wait()` method. This method blocks until the result arrives. If you don't want to wait forever, you can pass a timeout (in seconds) as an argument to `wait()`. `wait()` will *always return None*, and should never raise an error.

`ready()` and `wait()` are insensitive to the success or failure of the call. After a result is done, `successful()` will tell you whether the call completed without raising an exception.

If you actually want the result of the call, you can use `get()`. Initially, `get()` behaves just like `wait()`, in that it will block until the result is ready, or until a timeout is met. However, unlike `wait()`, `get()` will raise a `TimeoutError` if the timeout is reached and the result is still not ready. If the result arrives before the timeout is reached, then `get()` will return the result itself if no exception was raised, and will raise an exception if there was.

Here is where we start to expand on the multiprocessing interface. Rather than raising the original exception, a RemoteError will be raised, encapsulating the remote exception with some metadata. If the AsyncResult represents multiple calls (e.g. any time *targets* is plural), then a CompositeError, a subclass of RemoteError, will be raised.

See also:

For more information on remote exceptions, see *the section in the Direct Interface*.

Extended interface

Other extensions of the AsyncResult interface include convenience wrappers for `get()`. AsyncResults have a property, `result`, with the short alias `r`, which simply call `get()`. Since our object is designed for representing *parallel* results, it is expected that many calls (any of those submitted via DirectView) will map results to engine IDs. We provide a `get_dict()`, which is also a wrapper on `get()`, which returns a dictionary of the individual results, keyed by engine ID.

You can also prevent a submitted job from actually executing, via the AsyncResult's `abort()` method. This will instruct engines to not execute the job when it arrives.

The larger extension of the AsyncResult API is the `metadata` attribute. The metadata is a dictionary (with attribute access) that contains, logically enough, metadata about the execution.

Metadata keys:

timestamps

submitted When the task left the Client

started When the task started execution on the engine

completed When execution finished on the engine

received When the result arrived on the Client

> note that it is not known when the result arrived in 0MQ on the client, only when it arrived in Python via `Client.spin()`, so in interactive use, this may not be strictly informative.

Information about the engine

engine_id The integer id

engine_uuid The UUID of the engine

output of the call

error Python exception, if there was one

execute_input The code (str) that was executed

execute_result Python output of an execute request (not apply), as a Jupyter message dictionary.

stderr stderr stream

stdout stdout (e.g. print) stream

And some extended information

status either 'ok' or 'error'

msg_id The UUID of the message

after For tasks: the time-based msg_id dependencies

follow For tasks: the location-based msg_id dependencies

While in most cases, the Clients that submitted a request will be the ones using the results, other Clients can also request results directly from the Hub. This is done via the Client's `get_result()` method. This method will *always* return an AsyncResult object. If the call was not submitted by the client, then it will be a subclass, called `AsyncHubResult`. These behave in the same way as an AsyncResult, but if the result is not ready, waiting on an AsyncHubResult polls the Hub, which is much more expensive than the passive polling used in regular AsyncResults.

The Client keeps track of all results history, results, metadata

Querying the Hub

The Hub sees all traffic that may pass through the schedulers between engines and clients. It does this so that it can track state, allowing multiple clients to retrieve results of computations submitted by their peers, as well as persisting the state to a database.

queue_status

You can check the status of the queues of the engines with this command.

result_status

check on results

purge_results

forget results (conserve resources)

Controlling the Engines

There are a few actions you can do with Engines that do not involve execution. These messages are sent via the Control socket, and bypass any long queues of waiting execution jobs

abort

Sometimes you may want to prevent a job you have submitted from actually running. The method for this is `abort()`. It takes a container of msg_ids, and instructs the Engines to not run the jobs if they arrive. The jobs will then fail with an AbortedTask error.

clear

You may want to purge the Engine(s) namespace of any data you have left in it. After running *clear*, there will be no names in the Engine's namespace

shutdown

You can also instruct engines (and the Controller) to terminate from a Client. This can be useful when a job is finished, since you can shutdown all the processes with a single command.

Synchronization

Since the Client is a synchronous object, events do not automatically trigger in your interactive session - you must poll the 0MQ sockets for incoming messages. Note that this polling *does not* actually make any network requests. It simply performs a *select* operation, to check if messages are already in local memory, waiting to be handled.

The method that handles incoming messages is `spin()`. This method flushes any waiting messages on the various incoming sockets, and updates the state of the Client.

If you need to wait for particular results to finish, you can use the `wait()` method, which will call `spin()` until the messages are no longer outstanding. Anything that represents a collection of messages, such as a list of msg_ids or one or more AsyncResult objects, can be passed as argument to wait. A timeout can be specified, which will prevent the call from blocking for more than a specified time, but the default behavior is to wait forever.

The client also has an `outstanding` attribute - a `set` of msg_ids that are awaiting replies. This is the default if wait is called with no arguments - i.e. wait on *all* outstanding messages.

Note: TODO wait example

Map

Many parallel computing problems can be expressed as a `map`, or running a single program with a variety of different inputs. Python has a built-in `map()`, which does exactly this, and many parallel execution tools in Python, such as the built-in `multiprocessing.Pool` object provide implementations of *map*. All View objects provide a `map()` method as well, but the load-balanced and direct implementations differ.

Views' map methods can be called on any number of sequences, but they can also take the *block* and *bound* keyword arguments, just like `apply()`, but *only as keywords*.

```
dview.map(*sequences, block=None)
```

- iter, map_async, reduce

Decorators and RemoteFunctions

Note: TODO: write this section

`@parallel()`

`@remote()`

`RemoteFunction`

`ParallelFunction`

Dependencies

Note: TODO: write this section

`@depend()`

`@require()`

`Dependency`

Transitioning from IPython.kernel to ipyparallel

We have rewritten our parallel computing tools to use 0MQ and Tornado. The redesign has resulted in dramatically improved performance, as well as (we think), an improved interface for executing code remotely. This doc is to help users of IPython.kernel transition their codes to the new code.

Processes

The process model for the new parallel code is very similar to that of IPython.kernel. There is still a Controller, Engines, and Clients. However, the the Controller is now split into multiple processes, and can even be split across multiple machines. There does remain a single ipcontroller script for starting all of the controller processes.

Note: TODO: fill this out after config system is updated

See also:

Detailed *Parallel Process* doc for configuring and launching IPython processes.

Creating a Client

Creating a client with default settings has not changed much, though the extended options have. One significant change is that there are no longer multiple Client classes to represent the various execution models. There is just one low-level Client object for connecting to the cluster, and View objects are created from that Client that provide the different interfaces for execution.

To create a new client, and set up the default direct and load-balanced objects:

```
# old
In [1]: from IPython.kernel import client as kclient

In [2]: mec = kclient.MultiEngineClient()

In [3]: tc = kclient.TaskClient()

# new
In [1]: from ipyparallel import Client

In [2]: rc = Client()

In [3]: dview = rc[:]
In [4]: lbview = rc.load_balanced_view()
```

Apply

The main change to the API is the addition of the `apply()` to the View objects. This is a method that takes *view.apply(f,*args,**kwargs)*, and calls *f(*args, **kwargs)* remotely on one or more engines, returning the result. This means that the natural unit of remote execution is no longer a string of Python code, but rather a Python function.

- non-copying sends (track)
- remote References

The flags for execution have also changed. Previously, there was only *block* denoting whether to wait for results. This remains, but due to the addition of fully non-copying sends of arrays and buffers, there is also a *track* flag, which instructs PyZMQ to produce a `MessageTracker` that will let you know when it is safe again to edit arrays in-place.

The result of a non-blocking call to *apply* is now an *AsyncResult object*.

MultiEngine to DirectView

The multiplexing interface previously provided by the MultiEngineClient is now provided by the DirectView. Once you have a Client connected, you can create a DirectView with index-access to the client (`view = client[1:5]`). The core methods for communicating with engines remain: *execute*, *run*, *push*, *pull*, *scatter*, *gather*. These methods all behave in much the same way as they did on a MultiEngineClient.

```
# old
In [2]: mec.execute('a=5', targets=[0,1,2])

# new
In [2]: view.execute('a=5', targets=[0,1,2])
# or
In [2]: rc[0,1,2].execute('a=5')
```

This extends to any method that communicates with the engines.

Requests of the Hub (queue status, etc.) are no-longer asynchronous, and do not take a *block* argument.

- `get_ids()` is now the property `ids`, which is passively updated by the Hub (no need for network requests for an up-to-date list).
- `barrier()` has been renamed to `wait()`, and now takes an optional timeout. `flush()` is removed, as it is redundant with `wait()`
- `zip_pull()` has been removed
- `keys()` has been removed, but is easily implemented as:

  ```
  dview.apply(lambda : globals().keys())
  ```

- `push_function()` and `push_serialized()` are removed, as `push()` handles functions without issue.

See also:

Our Direct Interface doc for a simple tutorial with the DirectView.

The other major difference is the use of `apply()`. When remote work is simply functions, the natural return value is the actual Python objects. It is no longer the recommended pattern to use stdout as your results, due to stream decoupling and the asynchronous nature of how the stdout streams are handled in the new system.

Task to LoadBalancedView

Load-Balancing has changed more than Multiplexing. This is because there is no longer a notion of a StringTask or a MapTask, there are simply Python functions to call. Tasks are now simpler, because they are no longer composites of push/execute/pull/clear calls, they are a single function that takes arguments, and returns objects.

The load-balanced interface is provided by the `LoadBalancedView` class, created by the client:

```
In [10]: lbview = rc.load_balanced_view()

# load-balancing can also be restricted to a subset of engines:
In [10]: lbview = rc.load_balanced_view([1,2,3])
```

A simple task would consist of sending some data, calling a function on that data, plus some data that was resident on the engine already, and then pulling back some results. This can all be done with a single function.

Let's say you want to compute the dot product of two matrices, one of which resides on the engine, and another resides on the client. You might construct a task that looks like this:

```
In [10]: st = kclient.StringTask("""
            import numpy
            C=numpy.dot(A,B)
            """,
            push=dict(B=B),
            pull='C'
            )

In [11]: tid = tc.run(st)

In [12]: tr = tc.get_task_result(tid)

In [13]: C = tc['C']
```

In the new code, this is simpler:

```
In [10]: import numpy

In [11]: from ipyparallel import Reference

In [12]: ar = lbview.apply(numpy.dot, Reference('A'), B)

In [13]: C = ar.get()
```

Note the use of `Reference` This is a convenient representation of an object that exists in the engine's namespace, so you can pass remote objects as arguments to your task functions.

Also note that in the kernel model, after the task is run, 'A', 'B', and 'C' are all defined on the engine. In order to deal with this, there is also a *clear_after* flag for Tasks to prevent pollution of the namespace, and bloating of engine memory. This is not necessary with the new code, because only those objects explicitly pushed (or set via *globals()*) will be resident on the engine beyond the duration of the task.

See also:

Dependencies also work very differently than in IPython.kernel. See our *doc on Dependencies* for details.

See also:

Our Task Interface doc for a simple tutorial with the LoadBalancedView.

PendingResults to AsyncResults

With the departure from Twisted, we no longer have the `Deferred` class for representing unfinished results. For this, we have an AsyncResult object, based on the object of the same name in the built-in `multiprocessing.pool` module. Our version provides a superset of that interface.

However, unlike in IPython.kernel, we do not have PendingDeferred, PendingResult, or TaskResult objects. Simply this one object, the AsyncResult. Every asynchronous (*block=False*) call returns one.

The basic methods of an AsyncResult are:

```
AsyncResult.wait([timeout]): # wait for the result to arrive
AsyncResult.get([timeout]): # wait for the result to arrive, and then return it
AsyncResult.metadata: # dict of extra information about execution.
```

There are still some things that behave the same as IPython.kernel:

```
# old
In [5]: pr = mec.pull('a', targets=[0,1], block=False)
In [6]: pr.r
Out[6]: [5, 5]

# new
In [5]: ar = dview.pull('a', targets=[0,1], block=False)
In [6]: ar.r
Out[6]: [5, 5]
```

The `.r` or `.result` property simply calls `get()`, waiting for and returning the result.

See also:

AsyncResult details

Messaging for Parallel Computing

This is an extension of the messaging doc. Diagrams of the connections can be found in the *parallel connections* doc.

ZMQ messaging is also used in the parallel computing IPython system. All messages to/from kernels remain the same as the single kernel model, and are forwarded through a ZMQ Queue device. The controller receives all messages and replies in these channels, and saves results for future use.

The Controller

The controller is the central collection of processes in the IPython parallel computing model. It has two major components:

- The Hub
- A collection of Schedulers

The Hub

The Hub is the central process for monitoring the state of the engines, and all task requests and results. It has no role in execution and does no relay of messages, so large blocking requests or database actions in the Hub do not have the ability to impede job submission and results.

Registration (ROUTER)

The first function of the Hub is to facilitate and monitor connections of clients and engines. Both client and engine registration are handled by the same socket, so only one ip/port pair is needed to connect any number of connections and clients.

Engines register with the `zmq.IDENTITY` of their two `DEALER` sockets, one for the queue, which receives execute requests, and one for the heartbeat, which is used to monitor the survival of the Engine process.

Message type: `registration_request`:

```
content = {
    'uuid'   : 'abcd-1234-...', # the zmq.IDENTITY of the engine's sockets
}
```

Note: these are always the same, at least for now.

The Controller replies to an Engine's registration request with the engine's integer ID, and all the remaining connection information for connecting the heartbeat process, and kernel queue socket(s). The message status will be an error if the Engine requests IDs that already in use.

Message type: `registration_reply`:

```
content = {
    'status' : 'ok', # or 'error'
    # if ok:
    'id' : 0, # int, the engine id
}
```

Clients use the same socket as engines to start their connections. Connection requests from clients need no information:

Message type: `connection_request`:

```
content = {}
```

The reply to a Client registration request contains the connection information for the multiplexer and load balanced queues, as well as the address for direct hub queries. If any of these addresses is *None*, that functionality is not available.

Message type: `connection_reply`:

```
content = {
    'status' : 'ok', # or 'error'
}
```

Heartbeat

The hub uses a heartbeat system to monitor engines, and track when they become unresponsive. As described in messaging, and shown in *connections*.

Notification (PUB)

The hub publishes all engine registration/unregistration events on a `PUB` socket. This allows clients to have up-to-date engine ID sets without polling. Registration notifications contain both the integer engine ID and the queue ID, which is necessary for sending messages via the Multiplexer Queue and Control Queues.

Message type: `registration_notification`:

```
content = {
    'id' : 0, # engine ID that has been registered
    'uuid' : 'engine_id' # the IDENT for the engine's sockets
}
```

Message type : `unregistration_notification`:

```
content = {
    'id' : 0 # engine ID that has been unregistered
    'uuid' : 'engine_id' # the IDENT for the engine's sockets
}
```

Client Queries (ROUTER)

The hub monitors and logs all queue traffic, so that clients can retrieve past results or monitor pending tasks. This information may reside in-memory on the Hub, or on disk in a database (SQLite and MongoDB are currently supported). These requests are handled by the same socket as registration.

`queue_request()` requests can specify multiple engines to query via the *targets* element. A verbose flag can be passed, to determine whether the result should be the list of *msg_ids* in the queue or simply the length of each list.

Message type: `queue_request`:

```
content = {
    'verbose' : True, # whether return should be lists themselves or just lens
    'targets' : [0,3,1] # list of ints
}
```

The content of a reply to a `queue_request()` request is a dict, keyed by the engine IDs. Note that they will be the string representation of the integer keys, since JSON cannot handle number keys. The three keys of each dict are:

```
'completed' :  messages submitted via any queue that ran on the engine
'queue' : jobs submitted via MUX queue, whose results have not been received
'tasks' : tasks that are known to have been submitted to the engine, but
            have not completed.  Note that with the pure zmq scheduler, this will
            always be 0/[].
```

Message type: `queue_reply`:

```
content = {
    'status' : 'ok', # or 'error'
    # if verbose=False:
    '0' : {'completed' : 1, 'queue' : 7, 'tasks' : 0},
    # if verbose=True:
    '1' : {'completed' : ['abcd-...','1234-...'], 'queue' : ['58008-'], 'tasks' : []},
}
```

Clients can request individual results directly from the hub. This is primarily for gathering results of executions not submitted by the requesting client, as the client will have all its own results already. Requests are made by msg_id, and can contain one or more msg_id. An additional boolean key 'statusonly' can be used to not request the results, but simply poll the status of the jobs.

Message type: `result_request`:

```
content = {
    'msg_ids' : ['uuid','...'], # list of strs
    'targets' : [1,2,3], # list of int ids or uuids
    'statusonly' : False, # bool
}
```

The `result_request()` reply contains the content objects of the actual execution reply messages. If *statusonly=True*, then there will be only the 'pending' and 'completed' lists.

Message type: `result_reply`:

```
content = {
    'status' : 'ok', # else error
    # if ok:
    'acbd-...' : msg, # the content dict is keyed by msg_ids,
                     # values are the result messages
                    # there will be none of these if `statusonly=True`
    'pending' : ['msg_id','...'], # msg_ids still pending
    'completed' : ['msg_id','...'], # list of completed msg_ids
}
buffers = ['bufs','...'] # the buffers that contained the results of the objects.
                        # this will be empty if no messages are complete, or if
                        # statusonly is True.
```

For memory management purposes, Clients can also instruct the hub to forget the results of messages. This can be done by message ID or engine ID. Individual messages are dropped by msg_id, and all messages completed on an engine are dropped by engine ID. This may no longer be necessary with the mongodb-based message logging backend.

If the msg_ids element is the string `'all'` instead of a list, then all completed results are forgotten.

Message type: `purge_request`:

```
content = {
    'msg_ids' : ['id1', 'id2',...], # list of msg_ids or 'all'
    'engine_ids' : [0,2,4] # list of engine IDs
}
```

The reply to a purge request is simply the status 'ok' if the request succeeded, or an explanation of why it failed, such as requesting the purge of a nonexistent or pending message.

Message type: `purge_reply`:

```
content = {
    'status' : 'ok', # or 'error'
}
```

Schedulers

There are three basic schedulers:

- Task Scheduler
- MUX Scheduler
- Control Scheduler

The MUX and Control schedulers are simple MonitoredQueue ØMQ devices, with `ROUTER` sockets on either side. This allows the queue to relay individual messages to particular targets via `zmq.IDENTITY` routing. The Task scheduler may be a MonitoredQueue ØMQ device, in which case the client-facing socket is ROUTER, and the engine-facing socket is DEALER. The result of this is that client-submitted messages are load-balanced via the DEALER socket, but the engine's replies to each message go to the requesting client.

Raw DEALER scheduling is quite primitive, and doesn't allow message introspection, so there are also Python Schedulers that can be used. These Schedulers behave in much the same way as a MonitoredQueue does from the outside, but have rich internal logic to determine destinations, as well as handle dependency graphs Their sockets are always ROUTER on both sides.

The Python task schedulers have an additional message type, which informs the Hub of the destination of a task as soon as that destination is known.

Message type: `task_destination`:

```
content = {
    'msg_id' : 'abcd-1234-...', # the msg's uuid
    'engine_id' : '1234-abcd-...', # the destination engine's zmq.IDENTITY
}
```

`apply()`

In terms of message classes, the MUX scheduler and Task scheduler relay the exact same message types. Their only difference lies in how the destination is selected.

The Namespace model suggests that execution be able to use the model:

```
ns.apply(f, *args, **kwargs)
```

which takes *f*, a function in the user's namespace, and executes `f(*args, **kwargs)` on a remote engine, returning the result (or, for non-blocking, information facilitating later retrieval of the result). This model, unlike the execute message which just uses a code string, must be able to send arbitrary (pickleable) Python objects. And ideally, copy as little data as we can. The *buffers* property of a Message was introduced for this purpose.

Utility method `build_apply_message()` in `IPython.kernel.zmq.serialize` wraps a function signature and builds a sendable buffer format for minimal data copying (exactly zero copies of numpy array data or buffers or large strings).

Message type: `apply_request`:

```
metadata = {
    'after' : ['msg_id',...], # list of msg_ids or output of Dependency.as_dict()
    'follow' : ['msg_id',...], # list of msg_ids or output of Dependency.as_dict()
}
content = {}
buffers = ['...'] # at least 3 in length
                # as built by build_apply_message(f,args,kwargs)
```

after/follow represent task dependencies. 'after' corresponds to a time dependency. The request will not arrive at an engine until the 'after' dependency tasks have completed. 'follow' corresponds to a location dependency. The task will be submitted to the same engine as these msg_ids (see `Dependency` docs for details).

Message type: `apply_reply`:

```
content = {
    'status' : 'ok' # 'ok' or 'error'
    # other error info here, as in other messages
}
buffers = ['...'] # either 1 or 2 in length
```

```
                  # a serialization of the return value of f(*args,**kwargs)
                  # only populated if status is 'ok'
```

All engine execution and data movement is performed via apply messages.

Raw Data Publication

`display_data` lets you publish *representations* of data, such as images and html. This `data_pub` message lets you publish *actual raw data*, sent via message buffers.

data_pub messages are constructed via the `ipyparallel.datapub.publish_data()` function:

```
from ipyparallel.datapub import publish_data
ns = dict(x=my_array)
publish_data(ns)
```

Message type: `data_pub`:

```
content = {
    # the keys of the data dict, after it has been unserialized
    'keys' : ['a', 'b']
}
# the namespace dict will be serialized in the message buffers,
# which will have a length of at least one
buffers = [b'pdict', ...]
```

The interpretation of a sequence of data_pub messages for a given parent request should be to update a single namespace with subsequent results.

Control Messages

Messages that interact with the engines, but are not meant to execute code, are submitted via the Control queue. These messages have high priority, and are thus received and handled before any execution requests.

Clients may want to clear the namespace on the engine. There are no arguments nor information involved in this request, so the content is empty.

Message type: `clear_request`:

```
content = {}
```

Message type: `clear_reply`:

```
content = {
    'status' : 'ok' # 'ok' or 'error'
    # other error info here, as in other messages
}
```

Clients may want to abort tasks that have not yet run. This can by done by message id, or all enqueued messages can be aborted if None is specified.

Message type: `abort_request`:

```
content = {
    'msg_ids' : ['1234-...', '...'] # list of msg_ids or None
}
```

Message type: abort_reply:

```
content = {
    'status' : 'ok' # 'ok' or 'error'
    # other error info here, as in other messages
}
```

The last action a client may want to do is shutdown the kernel. If a kernel receives a shutdown request, then it aborts all queued messages, replies to the request, and exits.

Message type: shutdown_request:

```
content = {}
```

Message type: shutdown_reply:

```
content = {
    'status' : 'ok' # 'ok' or 'error'
    # other error info here, as in other messages
}
```

Implementation

There are a few differences in implementation between the *StreamSession* object used in the newparallel branch and the *Session* object, the main one being that messages are sent in parts, rather than as a single serialized object. *StreamSession* objects also take pack/unpack functions, which are to be used when serializing/deserializing objects. These can be any functions that translate to/from formats that ZMQ sockets can send (buffers,bytes, etc.).

Split Sends

Previously, messages were bundled as a single json object and one call to `socket.send_json()`. Since the hub inspects all messages, and doesn't need to see the content of the messages, which can be large, messages are now serialized and sent in pieces. All messages are sent in at least 4 parts: the header, the parent header, the metadata and the content. This allows the controller to unpack and inspect the (always small) header, without spending time unpacking the content unless the message is bound for the controller. Buffers are added on to the end of the message, and can be any objects that present the buffer interface.

Connection Diagrams of The IPython ZMQ Cluster

This is a quick summary and illustration of the connections involved in the ZeroMQ based IPython cluster for parallel computing.

All Connections

The IPython cluster consists of a Controller, and one or more each of clients and engines. The goal of the Controller is to manage and monitor the connections and communications between the clients and the engines. The Controller is no longer a single process entity, but rather a collection of processes - specifically one Hub, and 4 (or more) Schedulers.

It is important for security/practicality reasons that all connections be inbound to the controller processes. The arrows in the figures indicate the direction of the connection.

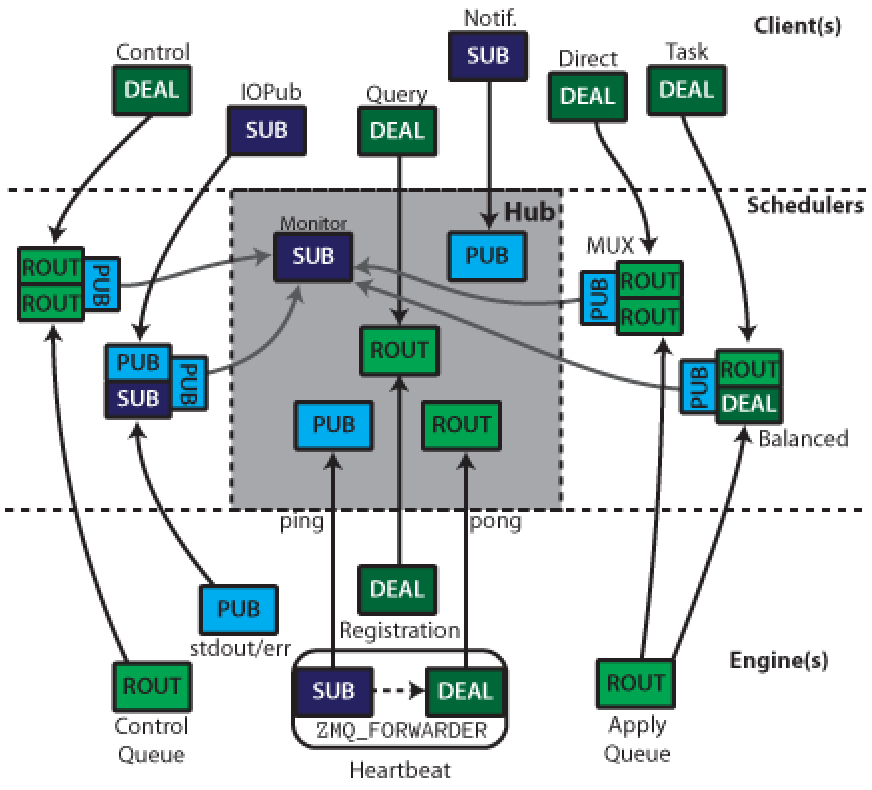


Fig. 2.2: All the connections involved in connecting one client to one engine.

The Controller consists of 1-5 processes. Central to the cluster is the **Hub**, which monitors engine state, execution traffic, and handles registration and notification. The Hub includes a Heartbeat Monitor for keeping track of engines that are alive. Outside the Hub are 4 **Schedulers**. These devices are very small pure-C MonitoredQueue processes (or optionally threads) that relay messages very fast, but also send a copy of each message along a side socket to the Hub. The MUX queue and Control queue are MonitoredQueue ØMQ devices which relay explicitly addressed messages from clients to engines, and their replies back up. The Balanced queue performs load-balancing destination-agnostic scheduling. It may be a MonitoredQueue device, but may also be a Python Scheduler that behaves externally in an identical fashion to MQ devices, but with additional internal logic. stdout/err are also propagated from the Engines to the clients via a PUB/SUB MonitoredQueue.

Registration

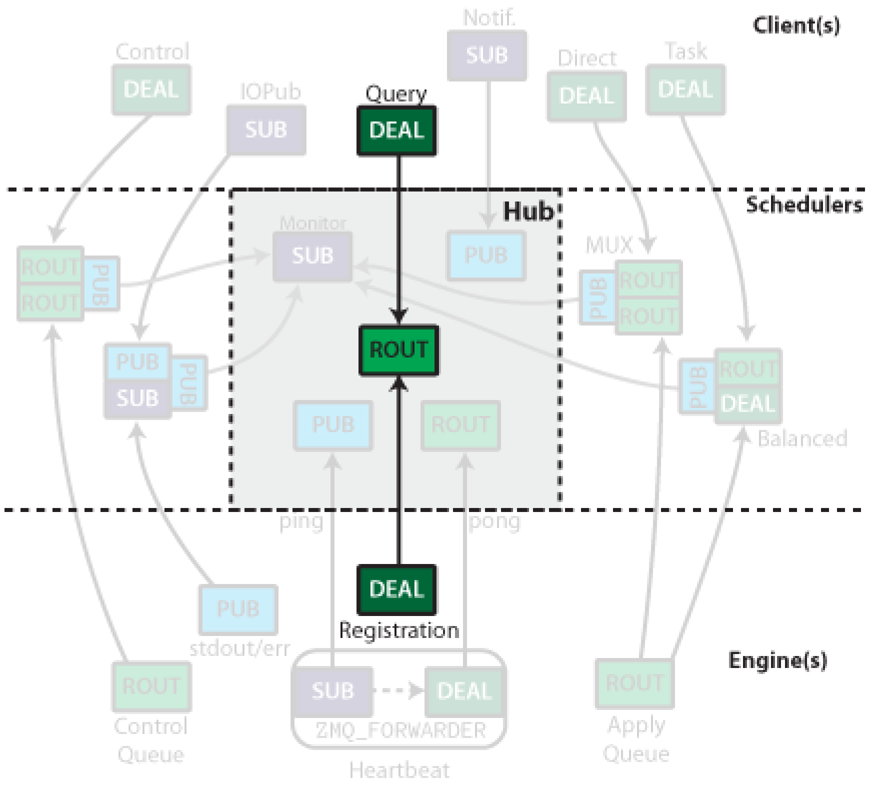


Fig. 2.3: Engines and Clients only need to know where the Query ROUTER is located to start connecting.

Once a controller is launched, the only information needed for connecting clients and/or engines is the IP/port of the Hub's ROUTER socket called the Registrar. This socket handles connections from both clients and engines, and replies with the remaining information necessary to establish the remaining connections. Clients use this same socket for querying the Hub for state information.

Heartbeat

The heartbeat process has been described elsewhere. To summarize: the Heartbeat Monitor publishes a distinct message periodically via a PUB socket. Each engine has a zmq.FORWARDER device with a SUB socket for input, and DEALER socket for output. The SUB socket is connected to the PUB socket labeled *ping*, and the DEALER is connected

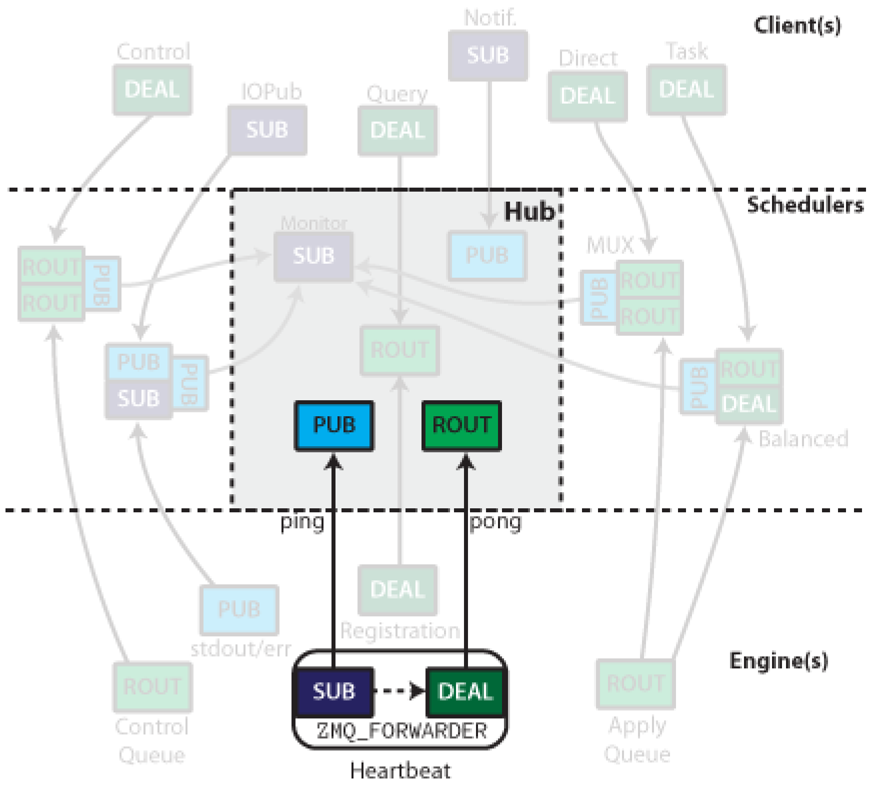


Fig. 2.4: The heartbeat sockets.

to the ROUTER labeled *pong*. This results in the same message being relayed back to the Heartbeat Monitor with the addition of the DEALER prefix. The Heartbeat Monitor receives all the replies via an ROUTER socket, and identifies which hearts are still beating by the zmq.IDENTITY prefix of the DEALER sockets, which information the Hub uses to notify clients of any changes in the available engines.

Schedulers

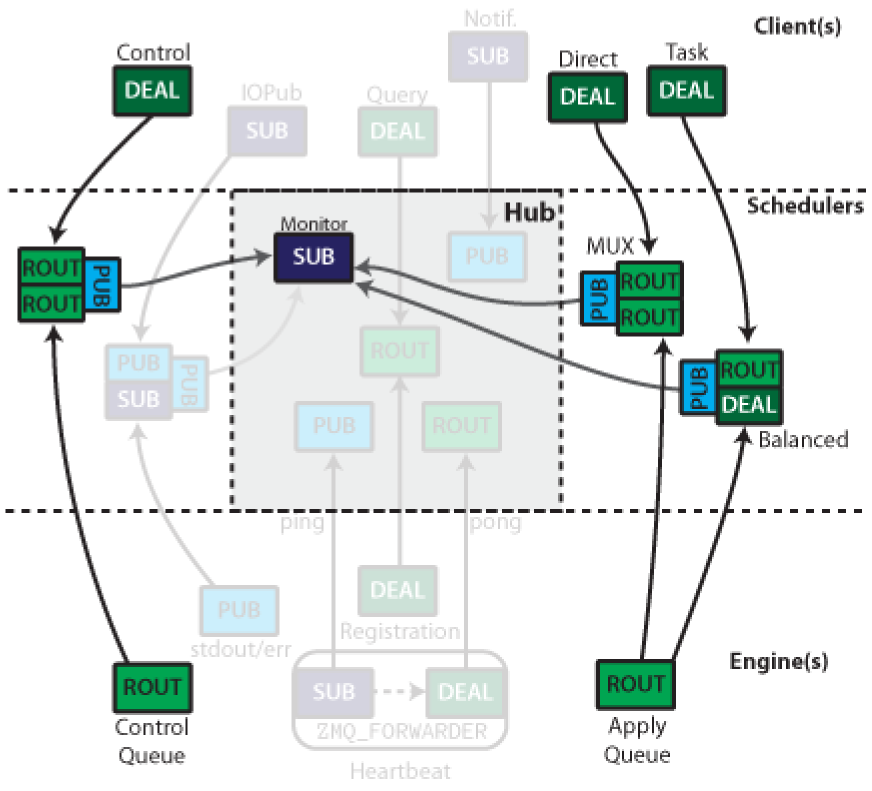


Fig. 2.5: Control message scheduler on the left, execution (apply) schedulers on the right.

The controller has at least three Schedulers. These devices are primarily for relaying messages between clients and engines, but the Hub needs to see those messages for its own purposes. Since no Python code may exist between the two sockets in a queue, all messages sent through these queues (both directions) are also sent via a PUB socket to a monitor, which allows the Hub to monitor queue traffic without interfering with it.

For tasks, the engine need not be specified. Messages sent to the ROUTER socket from the client side are assigned to an engine via ZMQ's DEALER round-robin load balancing. Engine replies are directed to specific clients via the IDENTITY of the client, which is received as a prefix at the Engine.

For Multiplexing, ROUTER is used for both in and output sockets in the device. Clients must specify the destination by the zmq.IDENTITY of the ROUTER socket connected to the downstream end of the device.

At the Kernel level, both of these ROUTER sockets are treated in the same way as the REP socket in the serial version (except using ZMQStreams instead of explicit sockets).

Execution can be done in a load-balanced (engine-agnostic) or multiplexed (engine-specified) manner. The sockets on the Client and Engine are the same for these two actions, but the scheduler used determines the actual behavior. This routing is done via the zmq.IDENTITY of the upstream sockets in each MonitoredQueue.

IOPub

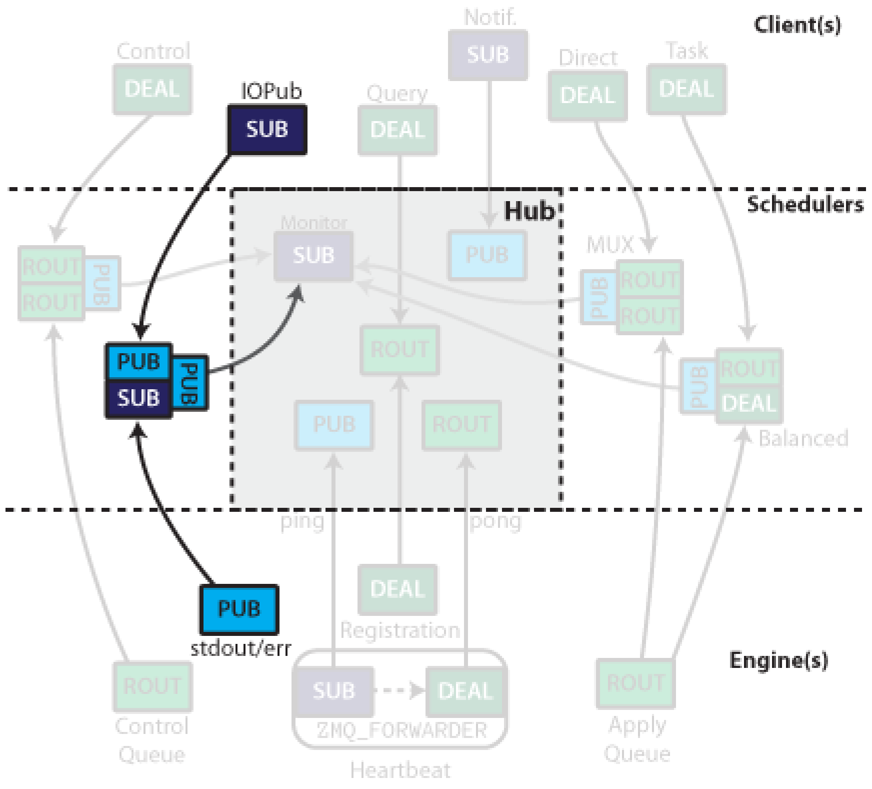


Fig. 2.6: stdout/err are published via a `PUB/SUB` MonitoredQueue

On the kernels, stdout/stderr are captured and published via a `PUB` socket. These `PUB` sockets all connect to a `SUB` socket input of a MonitoredQueue, which subscribes to all messages. They are then republished via another `PUB` socket, which can be subscribed by the clients.

Client connections

The hub's registrar `ROUTER` socket also listens for queries from clients as to queue status, and control instructions. Clients connect to this socket via an `DEALER` during registration.

The Hub publishes all registration/unregistration events via a `PUB` socket. This allows clients to stay up to date with what engines are available by subscribing to the feed with a `SUB` socket. Other processes could selectively subscribe to just registration or unregistration events.

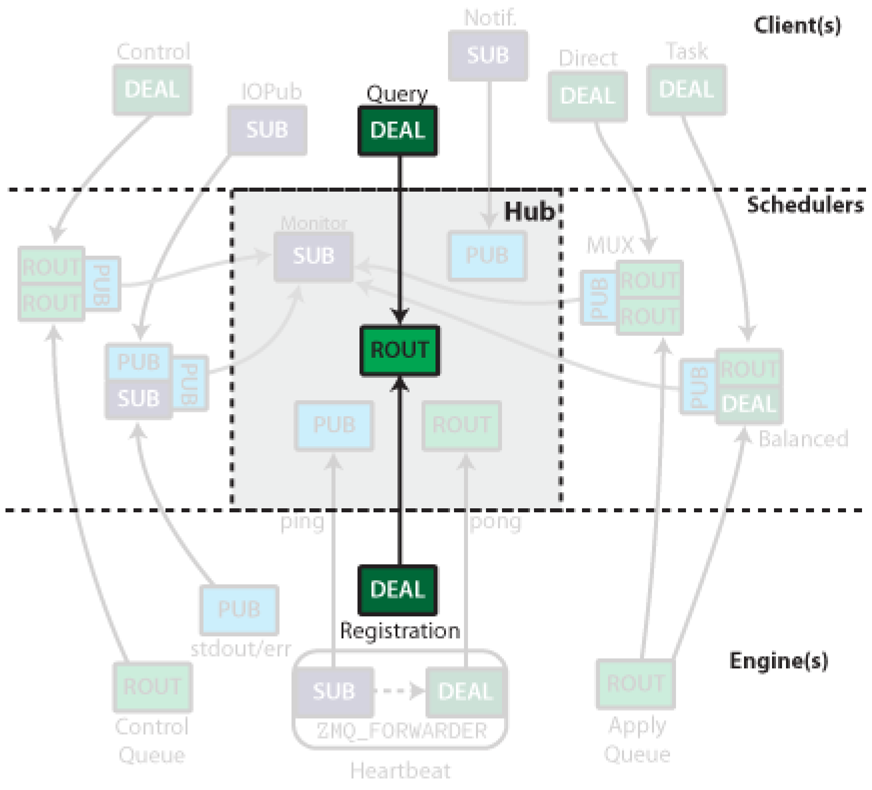


Fig. 2.7: Clients connect to an ROUTER socket to query the hub.

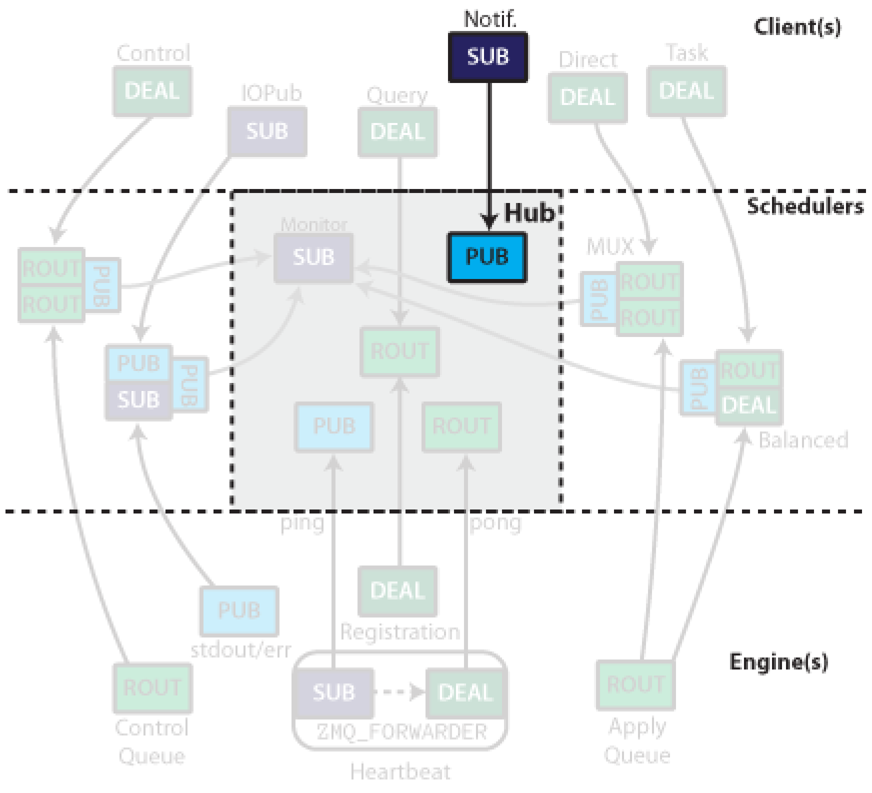


Fig. 2.8: Engine registration events are published via a `PUB` socket.

CHAPTER 3

ipyparallel API

ipyparallel

Classes

Decorators

IPython parallel provides some decorators to assist in using your functions as tasks.

Exceptions

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Bibliography

[ZeroMQ] http://zeromq.org/

[MongoDB] MongoDB database https://www.mongodb.org/

[PBS] Portable Batch System http://www.mcs.anl.gov/research/projects/openpbs/

[SSH] SSH-Agent https://en.wikipedia.org/wiki/Ssh-agent

[MPI] Message Passing Interface. http://www-unix.mcs.anl.gov/research/projects/mpi/

[mpi4py] MPI for Python. mpi4py: http://mpi4py.scipy.org/

[OpenMPI] Open MPI. http://www.open-mpi.org/

[PyTrilinos] PyTrilinos. https://trilinos.org/

[RFC5246] <http://tools.ietf.org/html/rfc5246>

[OpenSSH] <http://www.openssh.com/>

[Paramiko] <https://www.lag.net/paramiko/>

[HMAC] <http://tools.ietf.org/html/rfc2104.html>

Python Module Index

Index