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

Quick search

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. `'IPython.parallel.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 =
'IPython.parallel.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' : 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.

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:
'IPython.parallel.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 =
'IPython.parallel.controller.dictdb.DictDB'

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

# and SQLite:
c.HubFactory.db_class =
'IPython.parallel.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 =
'IPython.parallel.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/'
```

[MongoDB] MongoDB database http://www.mongodb.org

[PBS] Portable Batch System http://www.openpbs.org

[SSH] SSH-Agent http://en.wikipedia.org/wiki/ssh-agent