Introduction to parallel computing with IPython

- Credits for a prior version of this document go to Skipper Seabold
- Most of this presentation is taken from the IPython parallel computing documentation
- And from talks given over the years by the core development team of @minrk, @ellisonbg, and @fperez_org, among many others
- IPython is well documented, including video tutorials
- There is a great support network for IPython on stackoverflow and on their mailing list
- This talk is created using the IPython Notebook, which also support parallelism

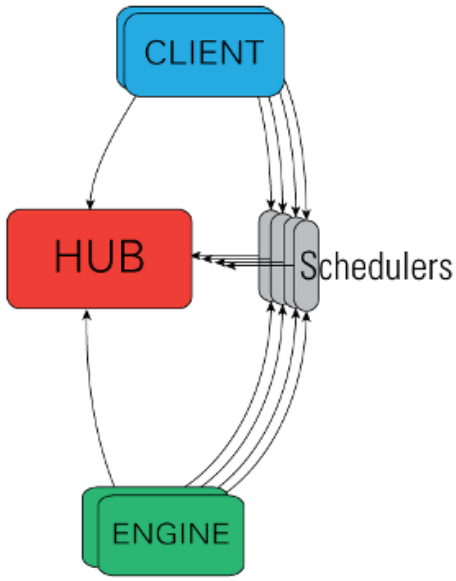
In [1]:
```
%pylab inline
import numpy as np
import matplotlib.pyplot as plt
```

```
Welcome to pylab, a matplotlib-based Python environment [backend:
module://IPython.kernel.zmq.pylab.backend_inline].
For more information, type 'help(pylab)'.
```

Architecture

In [2]:
```
from IPython.core.display import Image
Image("parallel_architecture400.png")
```

Out[2]:



Three Core Parts

I. The **Client**

- This is what you use to run your parallel computations
- You will interact with a **View** on the client
- The type of View depends on the execution model you are using

II. The **Controller**

- The IPython controller consists of 1) the **Hub** and 2) the **schedulers**
- The **Hub** is the central process that monitors everything
- The **schedulers** take care of getting of getting your code where it should go
- The controller is the go between from the Client to the Engines

III. The **Engine**

- An IPython "kernel" where the code is executed
- Listens for instructions over a network connection

IPython client and views

- The **`Client`** object connects to the cluster
- For each execution model there is a corresponding **`View`**

- For example, there are two basic views:
  - The `DirectView` class for explicitly running code on a particular engine(s)
  - The `LoadBalancedView` class for running your code on the 'best' engine(s)
- You can use as many views as you like, many at the same time
- You can read much more about the details of IPython parallel and views in the documentation

Getting Started

- Take advantage of multiple the processors on your local machine
- Say you have 4 processors
- How many processors do I have available?

In [3]:
```
from multiprocessing import cpu_count
print cpu_count()
```

```
4
```

- Start a controller and 4 engines with **ipcluster**
- At the command line type

```
ipcluster start -n 4
```

- Or, in the notebook at the dashboard

Did it work?

In [4]:
```
from IPython import parallel

rc = parallel.Client()
rc.block = True
```

In [5]:
```
def power(a, b):
    return a**b
```

- Create a `direct view` of kernel 0
- Direct views support slicing

In [6]:
```
dv = rc[0]
dv
```

Out[6]:
```
<DirectView 0>
```

In [7]:
```
dv.apply(power, 2, 10)
```

Out[7]:
```
1024
```

Recall that slice notation allows you leave out start and stop steps

In [8]:
```
X = [1, 2, 3, 4]
X
```

Out[8]:
```
[1, 2, 3, 4]
```

In [9]:
```
X[:]
```

Out[9]:
```
[1, 2, 3, 4]
```

Use this to send code to all the engines

In [10]:
```
rc[:].apply_sync(power, 2, 10)
```

Out[10]:
```
[1024, 1024, 1024,
1024]
```

Python's built-in map function allows you to call a sequence a function over a sequences of arguments

In [11]:
```
map(power, [2]*10, range(10))
```

Out[11]:
```
[1, 2, 4, 8, 16, 32, 64, 128, 256,
512]
```

In parallel, you use view.map

In [12]:
```
view = rc.load_balanced_view()
view.map(power, [2]*10, range(10))
```

Out[12]:
```
[1, 2, 4, 8, 16, 32, 64, 128, 256,
512]
```

The Direct Interface

- The direct interface lets the user interact explicitly with each engine
- First, create a direct view

In [13]:
```
from IPython import parallel

rc = parallel.Client()
```

- Above we saw the use of `map` and `apply` in parallel
- You may have also noticed this bit of code

In [14]:
```
rc.block = True
```

- In blocking mode, whenever you call execute code on the engines the controller waits until this code is done executing
- Non-blocking mode is the default

- Get access to all the engines

In [15]:
```
dview = rc[:]
```

- You can block on a call-by-call basis as well, by using apply_sync for synchronous execution

In [16]:
```
dview.block = False

dview["a"] = 5 # shorthand for push
dview["b"] = 7

dview.apply_sync(lambda x: a + b + x, 27)
```

Out[16]:
```
[39, 39, 39,
39]
```

- There is also `apply_async`
- Above you'll notice that the assignments defined these variables on the engines in a dictionary-like manner
- This is shorthand for `push`ing python objects to the engines
- DirectViews provide dictionary-like access by `key` or by using `get` and `update` like built-in dicts
- This can also be done explicitly with `push`
- `push` takes a dictionary

In [17]:
```
dview.push(dict(msg="Hi, there"), block=True)
```

Out[17]:
```
[None, None, None,
None]
```

In [18]:
```
dview.block = True
```

- Python commands can be executed as strings on specific engines

In [19]:
```
dview.execute("x = msg")
```

Out[19]:
```
<AsyncResult:
finished>
```

Or using the interactive `%px` magic, short for "parallel execute"

In [20]:
```
%px y = msg + ' you'
```

In [21]:
```
print dview["x"] # shorthand for pull
print dview["y"]
```

```
['Hi, there', 'Hi, there', 'Hi, there', 'Hi, there']
['Hi, there you', 'Hi, there you', 'Hi, there you', 'Hi, there you']
```

- You can also pull results back from the engine

In [22]:
```
rc[::2].execute("c = a + b")
rc[1::2].execute("c = a - b")
```

Out[22]:
```
<AsyncResult:
finished>
```

In [23]:
```
dview.pull("c")
```

Out[23]:
```
[12, -2, 12,
-2]
```

- If we were working in non-blocking mode, we would get an [`AsyncResult` object] back immediately

In [24]:
```
def wait(t):
    import time
    tic = time.time()
    time.sleep(t)
    return time.time() - tic
```

In [25]:
```
ar = dview.apply_async(wait, 2)
```

In [26]:
```
type(ar)
```

Out[26]:
```
IPython.parallel.client.asyncresult.AsyncResult
```

- We use its get method to get the result
- Calling get blocks

In [27]:
```
ar.get()
```

Out[27]:
```
[2.0022220611572266,
 2.0020439624786377,
 2.0007009506225586,
 2.0022499561309814]
```

- If we weren't quite so patient, we could ask if our tasks are done by using the `ready` method

In [28]:
```
ar = dview.apply_async(wait, 15)
print ar.ready()
```

```
False
```

- Or we can ask for the result, waiting a maximum of, say, 5 seconds

In [29]:
```
ar.get(5)
```

```
---------------------------------------------------------------------------
TimeoutError                              Traceback (most recent call last)
<ipython-input-29-8ed98da2cafb> in <module>()
----> 1 ar.get(5)

/home/fperez/usr/lib/python2.7/site-packages/IPython/parallel/client
/asyncresult.pyc in get(self, timeout)
    126                 raise self._exception
    127         else:
--> 128             raise error.TimeoutError("Result not ready.")
    129 
    130     def _check_ready(self):

TimeoutError: Result not ready.
```

- Often, we can't go on until some results are done
- For this, we can use the `wait` method
- `wait` can take an iterable of `AsyncResults`

In [30]:
```
result_list = [dview.apply_async(wait, 1) for i in range(8)]
```

In [31]:
```
dview.wait(result_list)
```

Out[31]:
```
True
```

In [32]:
```
result_list[0].get()
```

Out[32]:
```
[1.0011539459228516,
 1.0010430812835693,
 1.0004119873046875,
 1.0010390281677246]
```

Connecting directly to your engines

Every IPython engine can be turned into a kernel in one command:

In [33]:
```
%%px
from IPython.parallel import bind_kernel
bind_kernel()
```

Let's send the engine IDs to all of them

In [34]:
```
dview.scatter('eid', rc.ids, flatten=True)
```

And open a console directly on one of them:

In [35]:
```
rc[0].execute("%qtconsole")
```

Out[35]:
```
<AsyncResult:
finished>
```

Scatter and Gather

- You can use `scatter` to partition an iterable across engines
- `gather` pulls the results back
- You can use this to do parallel list comprehensions as below
- Sometimes this is more convenient than map

In [36]:
```
dview.scatter('x', range(64))

%px y = [i**10 for i in x]

y = dview.gather('y')

print y[:10]
```

```
[0, 1, 1024, 59049, 1048576, 9765625, 60466176, 282475249, 1073741824, 3486784401]
```

- The `%` indicates that we are using an IPython 'magic' function
- The availabel parallel magics are listed in the documentation

The Task Interface

- The Task interface allows you to use your the engines as a system of workers
- You no longer have direct access to the individual engines
- If your tasks are easily segmented into pieces that do not depend on each other, the Task Interface may be ideal
- *However*, you can specify complex dependencies to describe task execution order
- You can use many standard scheduling paradigms for how tasks should be run or define your own
- I am not going to discuss the task interface in detail

In [37]:
```
rc = parallel.Client()

lview = rc.load_balanced_view()
```

In [38]:
```
lview.block = True

parallel_result = lview.map(lambda x:x**10, range(32))

print parallel_result[:10]
```

```
[0, 1, 1024, 59049, 1048576, 9765625, 60466176, 282475249, 1073741824, 3486784401]
```

There's much more

- Parallel function decorators
- Using MPI for message passing (or pyzmq)
- Enabling database backends for storing information on running jobs to disk
- DAGs for tasks