Parallel Computing with IPython

Python in HPC

TACC Training, Oct. 15, 2012

Presenters:
Andy R. Terrel, PhD
Texas Advanced Computing Center
University of Texas at Austin

Yaakoub El Khamra
Texas Advanced Computing Center
University of Texas at Austin

Python in HPC Tutorial by Terrel, and El Khamra is licensed under a Creative Commons Attribution 3.0 Unported License.

0.1 Interacting with the Tutorial Slides

This tutorial is an interactive worksheet designed to encourage you to try out the lessons during the demonstration. If you are looking at the pdf version, we encourage you to download the updated version (see previous slide) and try the interactive version.

To run the interactive version, you need a good Python environment including:

- IPython version >= 13.0
- Numpy version >= 1.5
- Scipy
- Matplotlib

Move to the directory containing the tarball and execute:

```
$ ipython notebook --pylab=inline
```

We heartily endorse the Anaconda distribution and the Free Enthought Python Distribution.

0.2 Presentation mode

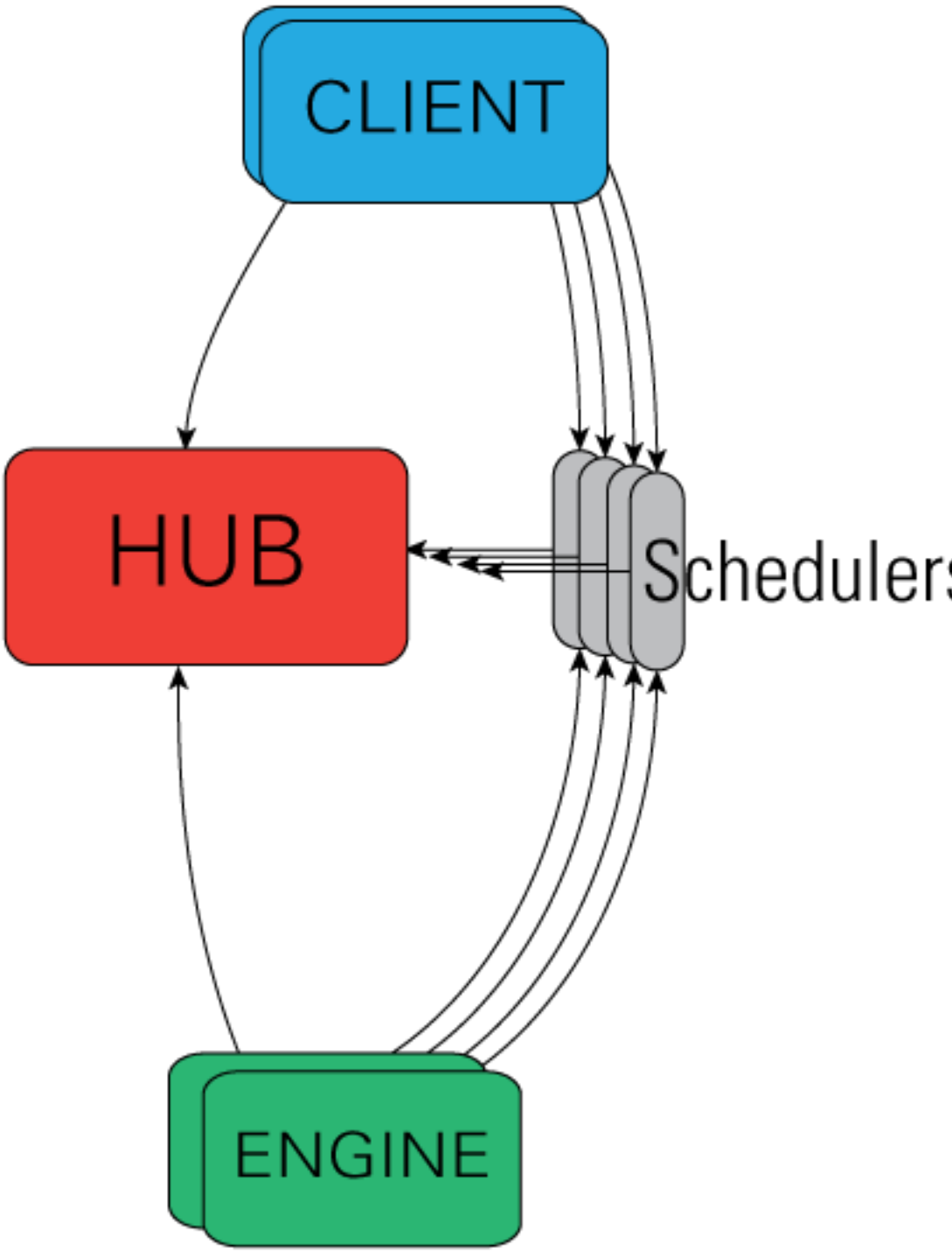
The slide show mode is only supported by an IPython development branch version. To get it I recommend cloning from the official branch, adding Matthias Carreau's remote, fetching and using his branch slideshow_extension2. Here are the commands:

```
git clone git://github.com/ipython/ipython.git # Official clone
cd ipython
git remote add carreau git://github.com/Carreau/ipython.git # Matthias' branch
git fetch carreau # Fetch the branches
git checkout carreau/slideshow_extension2 # Checkout the slideshow extension
python setup.py develop # Install the development version
ipython notebook # Check out the slideshows.
```

1 Architecture Overview

The IPython architecture consists of four components:

- The IPython engine
- The IPython hub
- The IPython schedulers
- The controller client


CLIENT
HUB
Schedulers
ENGINE

1.1 The Engine(s)

The IPython engine is a Python instance that takes Python commands over a network connection. Eventually, the IPython engine will be a full IPython interpreter, but for now, it is a regular Python interpreter. The engine can also handle incoming and outgoing Python objects sent over a network connection. When multiple engines are started, parallel and distributed computing becomes possible. An important feature of an IPython engine is that it blocks while user code is being executed.

1.2 The 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 utilize 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:

- Direct interface, where engines are addressed explicitly.
- 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.

1.3 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.

1.4 The Scheduler(s)

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.

1.5 Starting the IPython Controller and Engines

We already did this. We used: !ipcluster start –engines=MPI –n 4 to launch 4 MPI engines.

There are two ways to start the controller and engines:

- In an automated manner using the ipcluster command (possibly with a profile)
- In a more manual way using the ipcontroller and ipengine commands

Use ipcluster (with a profile) whenever possible. The profile will live in IPYTHONDIR/profile_< profile name >/ (usually $HOME/.ipython...) and that directory needs to be accessible to controller and engines.

ipcluster supports:

- all local: controller and engines are on the same node (we are doing this)
- engines are started with mpiexec (we are also doing this)
- engines are started with a batch system (PBS/SGE)
- controller is on the localhost, the engines are started remotely with ssh

For more information on how to configure a profile for ipcluster, please refer to the IPython documentation

1.6 The 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. There are two default views:

- DirectView class for explicit addressing
- LoadBalanceView class for destination-agnostic scheduling

1.7 The DirectView

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.

```
# import the parallel module and create a Client instance
from IPython.parallel import Client
rc = Client()
# check the id's of the engines
print rc.ids
# Create a DirectView object that will use all of the available engines
direct_view = rc[:]
```

```
[0, 1, 2, 3]
```

1.8 The map() function

Python's builtin map() functions allows a function to be applied to a sequence element-by-element (and is thus trivial to parallelize). The DirectView's version of map() does not do dynamic load balancing

```
# serial version run on the controller
serial_result = map(lambda x:x**10, range(32))
# parallel version run on the engines
parallel_result = direct_view.map_sync(lambda x: x**10, range(32))
# make sure they are equal
serial_result==parallel_result
```

```
True
```

1.9 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:

```
@direct_view.remote(block=True)
def getpid():
    import os
    return os.getpid()
```

```
getpid()
```

```
[8627, 8630, 8629, 8628]
```

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

```
import numpy as np
A = np.random.random((64,48))

@direct_view.parallel(block=True)
def pmul(A,B):
    return A*B

C_local = A*A

C_remote = pmul(A,A)

(C_local == C_remote).all()
```

```
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.

1.10 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.

```
# Setup a blocking view
blocking_direct_view = rc[:]
blocking_direct_view.block = True

# Set the values of a and b on all engines
blocking_direct_view['a'] = 10
blocking_direct_view['b'] = 12

# Apply a function
blocking_direct_view.apply(lambda x: a+b+x, 5)
```

```
[27, 27, 27, 27]
```

Alternatively you can execute Python commands as strings on specific engines

```
# execute on 0 and 2
rc[::2].execute('c=a+b')
# execute on 1 and 3
rc[1::2].execute('c=a-b')
# retrieve c from the engines
blocking_direct_view['c']
```

```
[22, -2, 22, -2]
```

In non-blocking mode, apply() submits the command to the engine and returns immediately with a AsyncResult object. To retrieve the result you can use the get() method from the AsyncResult object. The advantage of using non-blocking execution is that you can submit commands that possibly take a long time to execute without blocking the controller (particularly useful in interactive sessions).

```
non_blocking_direct_view = rc[:]
non_blocking_direct_view.block = False

# create a dummy wait function
def wait(t):
    import time
    tic = time.time()
    time.sleep(t)
    return time.time()-tic

# in non-blocking mode
ar = non_blocking_direct_view.apply_async(wait,2)
# now block for the result
ar.get()
```

```
[2.0009918212890625, 2.000959873199463, 2.001322031021118, 2.00095796585083]
```

```
ar = non_blocking_direct_view.apply_async(wait,10)

# poll to see if the result is ready
ar.ready()
```

```
False
```

```
# Ask for the result but wait for a maximum of 1 second:
ar.get(1)
```

```
[10.00077199935913, 10.000739097595215, 10.00072193145752, 10.0003080368042]
```

1.11 Moving Objects around

Use push() and pull() to send and get objects from engines

```
blocking_direct_view.push(dict(a=1.03234,b=3453))
```

```
[None, None, None, None]
```

```
blocking_direct_view.pull('a')
```

```
[1.03234, 1.03234, 1.03234, 1.03234]
```

```
blocking_direct_view.pull('b', targets=0)
```

```
3453
```

```
blocking_direct_view.pull(('a','b'))
```

```
[[1.03234, 3453], [1.03234, 3453], [1.03234, 3453], [1.03234, 3453]]
```

```
blocking_direct_view.push(dict(c='speed'))
```

```
[None, None, None, None]
```

In non-blocking mode, push() and pull() return AsyncResult objects

```
ar = non_blocking_direct_view.pull('a')
ar.get()
```

```
[1.03234, 1.03234, 1.03234, 1.03234]
```

1.12 Parallel Magic Commands

IPython has a few magic commands that make executing Python commands in parallel a lot more pleasant. The magics will be automatically available when we create a client:

rc = parallel.Client()

```
# Import numpy everywhere, including controller
with rc[:].sync_imports():
    import numpy
```

```
importing numpy on engine(s)
```

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

```
%px a = numpy.random.rand(2,2)
# to see the result:
%px print a
```

```
[stdout:0]
[[ 0.75807677  0.55011869]
 [ 0.06067355  0.23753211]]
[stdout:1]
[[ 0.31290721  0.70553545]
 [ 0.25818139  0.03566729]]
[stdout:2]
[[ 0.39323519  0.98505994]
 [ 0.64079498  0.3318084 ]]
[stdout:3]
[[ 0.44615847  0.66901976]
 [ 0.80198817  0.74741971]]
```

```
%%px --targets ::2
print "I am even"
```

```
[stdout:0] I am even
[stdout:2] I am even
```

```
%%px --targets 1
print "I am number 1"
```

```
I am number 1
```

Important magic notes:

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

- You can change the blocking and targets attributes of the active view with %pxconfig
- You can change the active view by calling the activate method on any view