INTERACTIVE PARALLEL COMPUTING IN IPYTHON WITH ØMQ

Brian Granger, Cal Poly Physics
Fernando Perez, UC Berkeley Neuroscience
Min Ragan-Kelley, UC Berkeley AS&T

SIAM CSE
Wednesday, March 02, 2011
• First: ØMQ, PyZMQ
• IP network architecture
• Client API
• Performance
• ØMQ Sockets use a pool of C++ IO threads to send, receive and queue messages.

• All of this logic continues even while:
  • GIL releasing or GIL-holding C/C++ extension code runs
  • Any Python code runs
  • Peer may not be ready (or even exist)

• The Socket.send and Socket.recv functions do not block, they simply move pointers to IO threads

• Socket type determines send/recv pattern

Slide content c/o Brian Granger
• Python bindings for ØMQ
• Entirely in Cython
• zero-copy sends
• Supports Python 2.5-3.2
• Integrates Tornado event loop

http://github.com/zeromq/pyzmq
The Python execution model: *code* is executed in a *namespace*

In IPython, object that handles the execution of code in a given namespace is called a *kernel*

IPython is a tool for running code in a namespace *interactively*

The model for Parallel IPython is simply interacting with one or more namespaces *remotely*

Running code in a remote namespace should not differ from doing so locally
What does it look like?
**NETWORK ARCHITECTURE**

- One-to-many **Engines**
- Zero-to-many **Clients**
- Requests/replies relayed via **Schedulers**
- Central **Hub** monitors traffic and tracks state
- Results from other clients or sessions can be retrieved from the Hub
- Cluster persists for the lifetime of a simulation
• Lots of Sockets
• 1 Socket = 1 type of action
• GIL-less pure-C MonitoredQueue for each pattern
• Hub code cannot block Client-Engine communication*

*aside from consuming general machine/network resources
**Life Cycle of a Task**

- **Client creates Task**, sends via socket to Task Scheduler
- Scheduler sends task to Engine then to Monitor
- When the engine is ready, **take Task off ZMQ queue and run**.
- Send back via socket to Scheduler
- Scheduler relays result to Client, Monitor
- When the **client asks for the Result**, it’s waiting in the ZMQ Queue in local memory
- Other clients can request it from the Hub
So how do I use it?
It's all about: \texttt{apply}(f, \ast\texttt{args}, \ast\ast\texttt{kwargs})

\begin{verbatim}
domystuff(a, b, c=numpy.linspace(0,pi), flag=False)

from IPython.zmq.parallel import client
rc = client.Client()

rc.apply(f, \texttt{args=}(), \texttt{kwargs=}{}, \texttt{bound=}True, \texttt{block=}None, \texttt{targets=}None,
\texttt{after=}None, \texttt{follow=}None, \texttt{timeout=}None)

rc.apply(domystuff, (a, b), \texttt{dict}(c=numpy.linspace(0,pi), flag=False))

view = rc.view()
view.apply(f, \ast\texttt{args}, \ast\ast\texttt{kwargs})

view.apply(domystuff, a, b, c=numpy.linspace(0,pi), flag=False)

\end{verbatim}

\texttt{\textcolor{green}{code.replace("domystuff\texttt{"}, "view.apply(domystuff, \texttt{"})}}
It’s all about: `apply(f, *args, **kwargs)`

- View objects provide simpler `apply()` method by having attributes that set keyword args (`block, bound, targets`)
- Tasks are simply Python functions
- Engines are remote namespaces, in which the functions run
- In Python, namespaces are `dicts`
- DirectViews provide the `dict` interface for remote namespaces
EXECUTION PATTERNS

- Multiplexer (MUX): run code on specified engine(s)

  Use a DirectView

- Task Farming: run code in an engine-agnostic way, trusting the Scheduler to load-balance

  Use a LoadBalancedView
  \[ v = rc.view(balanced=True) \]

```python
rc.apply(f, args=(), kwargs={}, bound=True, block=None, targets=None,
         after=None, follow=None, timeout=None)
```
MULTIPLEXING

Specify the engine(s) on which to run index-access to a Client creates a DirectView

In [58]: rc[:].apply_sync(os.getpid)
Out[58]: [64322, 64320, 64323, 64321]

In [65]: rc[:]["a"] = 7
In [66]: rc[::2]["b"] = 6
In [67]: rc[1::2]["b"] = 5

In [68]: rc[:].execute("c=a*b")

In [69]: rc[:]["c"]
Out[69]: [42, 35, 42, 35]

In [70]: rc[0].apply_sync_bound(lambda : a*b)
Out[70]: 42
You can also distribute your data

In [57]: view = rc[:]
In [58]: view.scatter('a',range(16))

In [59]: view['a']
Out[59]: [ [0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], ... ]

In [60]: view.gather('a')
Out[60]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,...]
ASYNC RESULTS

• Non-blocking methods return AsyncResult objects
• These objects represent results that may not have finished
• They can wait for the results
• They also contain metadata about the execution of the task, including times and locations
• superset of stdlib API in multiprocessing.PoolAsyncResult
In [37]: def randwait():
    ...:     import random, time
    ...:     t = random.random()
    ...:     time.sleep(t)
    ...:     return t
In [38]: ar = rc[:].apply(randwait)
In [39]: ar.get()
Out[39]: [0.919, 0.985, 0.720, 0.479]

In [40]: ar.metadata[-1]
Out[40]: {
    'status': 'ok',
    'engine_id': 3,
    'msg_id': 'c431120f-89ab-4c84-9280-5610facc5ff3',
    'submitted': datetime.datetime(2011, 3, 1, 16, 53, 0, 204219),
    'started': datetime.datetime(2011, 3, 1, 16, 53, 0, 206661),
    'completed': datetime.datetime(2011, 3, 1, 16, 53, 1, 125896),
    'received': datetime.datetime(2011, 3, 1, 16, 53, 10, 913960),
    'stderr': '',
    'stdout': '',
    ...
}
Sometimes you have a lot of work to do, but don’t really care where it happens

Use Task Farming

In [43]: v = rc.view()
In [44]: v = rc.view(targets=[1,2], balanced=True)
In [45]: [v.apply_sync(os.getpid) for i in range(3)]
Out[45]: [64323, 64320, 64323]
parallel `map` should as easy as builtin `map`!

In [65]: serial_result = map(lambda x: x**10, range(32))

In [66]: parallel_result = v.map(lambda x: x**10, range(32))

In [67]: serial_result==parallel_result
Out[67]: True
Sometimes you actually do want to exert *some* control over where and when a Task will run.

For this, we have **Dependencies**
DEPENDENCIES

- Two kinds:
  - **DAG** (tasks depending on other tasks)
  - **Functional** (runs on engine)

- DAG dependencies are resolved in the Scheduler, and determine where/when to run based on where/when other tasks ran

- Functional dependencies are evaluated on the engine, and raise an UnmetDependency exception if the engine is inappropriate
• ØMQ gives us very fast LRU load-balancing for free
• But it doesn’t handle more advanced features
• So we also have a Python scheduler that handles advanced logic
But is it fast?
Latency Test

• Send a number of no-op tasks as fast as possible

• wait for all results

```python
def echo(s):
    return s

client.apply(echo, '')
```

All connections via localhost
PERFORMANCE

Latency

Tasks/s

10^4

10^3

10^2

10^1

1
4
16
64
256
1024

Tasks

py
py_ssh
pure
pure_ssh
sent
Throughput Test

- Send 128 echo tasks as fast as possible
- wait for results

```python
def echo(s):
    return s

A = np.random.random(size/8)
client.apply(echo, A)
```

All connections via localhost
Throughput (128 Messages)

- py
- py_ssh
- pure
- pure_ssh
- sent

Tasks/s vs msg size (B)
PERFORMANCE

- ~1ms message latency (roundtrip)
- ~2x penalty for Python scheduler
- Performance *not* a function of the number of messages beyond ~8
- ssh tunnel puts limit at ~1Gbps on the test machine
- without ssh, throughput is ~10Gbps
• IPython magics
• decorators for remote functions and dependencies
• iterate through AsyncResult as tasks finish
• PBS/SSH process launching support (SGE, Condor on the way)
• MongoDB backend to reduce Hub resources
THANKS

AFOSR (funding)
John Verboncoeur (advisor)
GitHub (hosting)
ØMQ (hammer)

Documentation: http://minrk.github.com/ipython-doc
Source: http://github.com/ipython/ipython/tree/newparallel