Parallel Python

Interactive parallel computing with IPython

Monte Lunaceck and Thomas Hauser @colorado.edu

Why Python?

- Relatively easy to learn
- Interactive
- Cross platform
  - Linux
  - Windows
  - OS/X
- Large number of packages
  - Numpy – for numerical computation
  - Matplotlib - plotting
  - Ipython – interactive shell

IPython

- Supports different styles of parallelism
  - *Single program, multiple data (SPMD) parallelism*
  - Multiple program, multiple data (MPMD) parallelism
  - Message passing using MPI
  - *Task farming*
  - Combinations of above approaches
  - Custom user defined approaches

- Interactive approach to
  - Development
  - Execution
  - Debugging
  - Monitoring

Overview

- How it works
- Direct interface
  - Executing remote commands
  - Data management
  - Blocking, non-blocking
- Task interface
  - Compare with *pbsdsh*
  - Simple, robust, and load-balanced

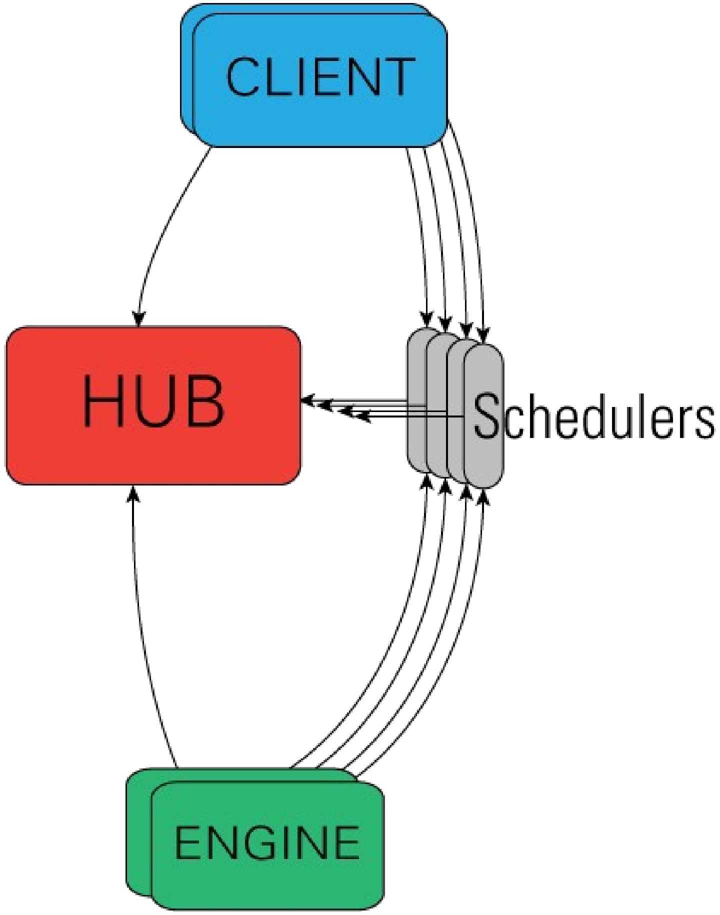
How it works

Architecture

- Engine
  - Takes Python commands over a network connection
  - One engine per core
  - Blocks code: controller has an asynchronous API
- Controller client
  - Interface for working with a set of engines
  - Hub
  - Collection of schedulers
  - View objects represent a subset of engines
    - A **Direct** interface, where engines are addressed explicitly.
    - A **LoadBalanced** interface, where the Scheduler is trusted with assigning work to appropriate engines.

Parallel Architecture

- Hub
  - keeps track of engine connections, schedulers, clients
  - Cluster state
- Scheduler
  - All actions that can be performed on the engine go through a scheduler



Start your Engines

- Simplest method
  - `ipcluster start --n=4`
    - MPIEXEC/MPIRUN mode
    - SSH mode
    - PBS mode for batch processing

- Other methods
  - Ipcontroller
  - ipengine

- Don’t forget to turn off your engine!

Use in HPC environments

- ipython profile create --parallel –profile=mpi

Generating default config file: u'/home/molu8455/.ipython/profile_mpi/ipython_config.py'Generating default config file: u'/home/molu8455/.ipython/profile_mpi/ipython_qtconsole_config.py'Generating default config file: u'/home/molu8455/.ipython/profile_mpi/ipcontroller_config.py'Generating default config file: u'/home/molu8455/.ipython/profile_mpi/ipengine_config.py'Generating default config file: u'/home/molu8455/.ipython/profile_mpi/ipcluster_config.py'Generating default config file: u'/home/molu8455/.ipython/profile_mpi/iplogger_config.py'

- Edit the file `ipcluster_config.py`
  - `c.IPClusterEngines.engine_launcher_class = 'MPIEngineSetLauncher'`

Steps

- Request a resource with qsub:
  - Interactive
  - Batch
- Start the engines
  - `ipcluster start --profile=mpi &`
- Parallel python work …
- Stop the engines
  - `ipcluster stop --profile=mpi`

Example in a PBS setting

```
#PBS -N example
#PBS -j oe
#PBS -l walltime=00:45:00
#PBS -l nodes=2:ppn=12
#PBS -q janus-debug

# any config relevant to your local environment here

# switch to the PBS working directory
cd $PBS_O_WORKDIR

# Start the IPython cluster and wait a little for engines to come up
ipcluster start --profile=mpi &
sleep 90

# Run your own code that uses the ipcluster
python work.py

# Stop the cluster when done
ipcluster stop --profile=mpi
```

A few simple concepts

- The **client:** lightweight handle on all engines of a cluster

```
from IPython.parallel import Client
rc = Client(profile='mpi')
```

- The **views:** “slice” the client with specific execution semantics
  - DirectView: direct execution on *all* engines

```
dview = rc.direct_view()
```

  - LoadBalancedView: run on *any one* engine

```
lview = rc.load_balanced_view()
```

Direct Interface

Direct Interface

- capabilities of each engine are directly and explicitly exposed to the user
- Blocking and non-blocking
  - `AsyncResult`
- Remote execution
  - `apply`
  - `execute`
  - `map`
  - Remote function decorators
- Managing data
  - `push, pull`
  - `scatter, gather`

Blocking vs. Non-blocking

- Blocking
  - waits until all engines are done executing the command

```
dview.block=True
```

- Non-blocking
  - returns an `AsyncResult` immediately
    - Get data later with *get* method
    - Check the status with *ready*
    - `Wait`

```
dview.block=False
```

Apply Example

- Serial

```
import os
pid = os.getpid()
print pid
```

- Parallel

```
with dview.sync_imports():
import os
dview.block=True
pids = dview.apply(os.getpid)
dview.block=False
ar = dview.apply(os.getpid)
dview.wait(ar)
pids = ar.get()
```

Apply

- For convenience
  - apply_sync (block=True)
  - apply_async (block=False)

```
def wait(t):
    import time
    tic = time.time()
    time.sleep(t)
    return time.time()-tic

ar = dview.apply_async(wait,1)
if ar.ready() == True:
    print "ready”
else:
    print "not-ready"
dview.wait(ar)
```

Execute

- Commands can be executed as strings on specific engines

```
dview['a'] = 10
dview['b'] = 20
dview.execute('c=a*b')
rc[1].execute('c=a+b')
print dview['c']
```

[200, 30, 200, 200, 200, 200, 200, 200, 200, 200, 200, 200]

Map

- Applies a sequence of to a function element-by-element
- Does not do dynamic load balancing
- `map_sync`, `map_async`

```python
def fsquare(x):
    return x**2
xvals = range(32)
s_result = map(fsquare, xvals)
dview.block=True
p_result = dview.map(fsquare, xvals)
```

Push and Pull

- Limited to dictionary types

```
dview.block=True
values = dict(a=1.03234,b=3453)
dview.push(values)
a_values = dview.pull('a')
print a_values
dview.push(dict(c='speed'))
print dview.pull('c')
dview['c'] = 'speed'
print dview.pull('c')
```

Scatter and Gather

- Partition a sequence and push/pull

```
dview.scatter('a',range(24))
print dview['a']
all_a = dview.gather('a')
print all_a

# Use
dview.scatter('x',range(64))
dview.execute('y=map(fsquare, x)')
y = dview.gather('y')
print y
```

Task Interface

Task interface

- Presents the engines as a *fault tolerant*, *dynamic load-balanced* system of workers
- No direct access to individual engines
- Simple and powerful

```
lview = rc.load_balanced_view()
```

- Components:
    - `map`
    - Parallel function decoders
    - Dependencies
        - functional
        - graph

Example: parameter study

```
#!/bin/bash
#PBS -N example_mn_1
#PBS -q janus-debug
#PBS -l walltime=0:01:30
#PBS -l nodes=2:ppn=12

cd $PBS_O_WORKDIR

pbsdsh
$PBS_O_WORKDIR/wrapper.sh 0
```

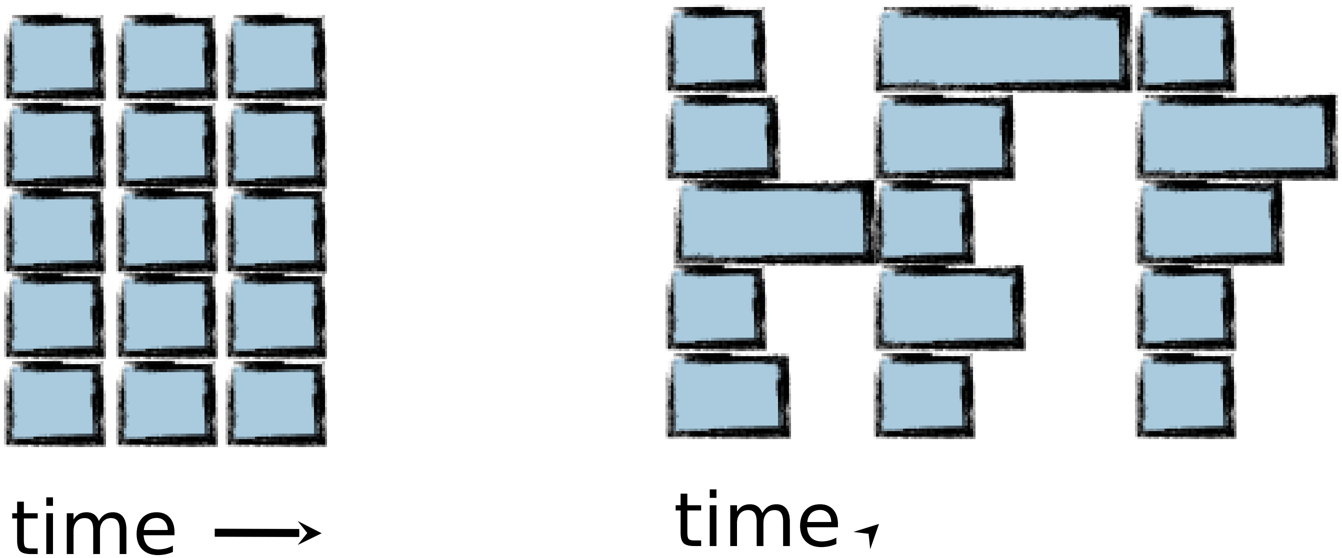
```
#!/bin/bash
PATH=$PBS_O_WORKDIR:$PBS_O_PATH
TRIAL=$(($PBS_VNODENUM + $1))
simulator $TRIAL > $PBS_O_WORKDIR/sim.$TRIAL
```

```
#!/bin/bash
#PBS -N example_mn_3
#PBS -q janus-debug
#PBS -l walltime=0:01:30
#PBS -l nodes=2:ppn=12

cd $PBS_O_WORKDIR

count=0
np=`wc -l < $PBS_NODEFILE`

for i in {1..5}
do
    pbsdsh
$PBS_O_WORKDIR/wrapper.sh $count
    count=$(( $count + $np))
done
```

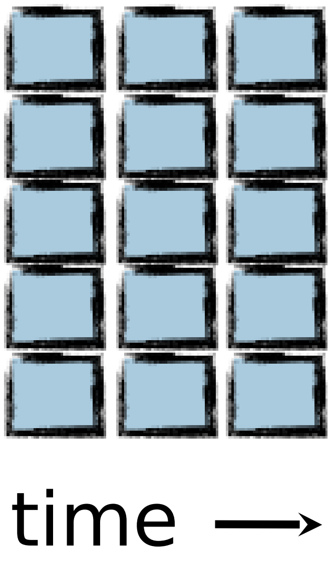
Problems

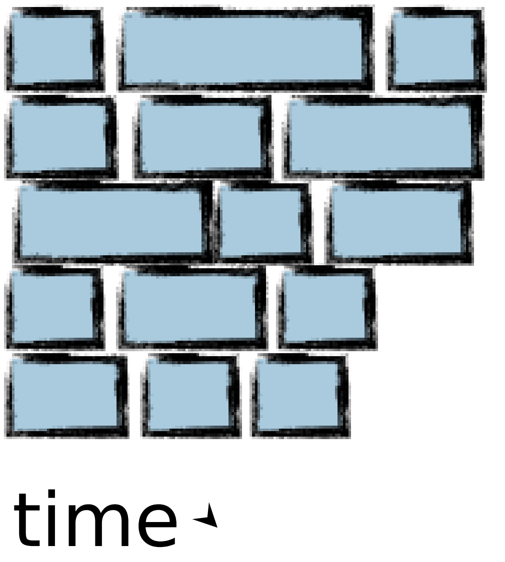
- Core failure to run
- Load-balance

Weaknesses of batch



Load balancing





simulator.py

```
def imulator(x):
    import subprocess
    cmd = "./simulator " + str(x) + " 2 4 > trial." + str(x)
    p = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE)
    (err,out)=p.communicate()
    status = p.returncode
    return "%s, %s" % (out,err)
print simulator(1)
from IPython.parallel import Client
rc = Client(profile='mpi')
lview = rc.load_balanced_view()
r = lview.map_async(simulator, range(100))
print r.get()
```

```bash
#!/bin/bash
#PBS -N example_mn_3
#PBS -q janus-debug
#PBS -l walltime=0:01:30
#PBS -l nodes=2:ppn=12

cd $PBS_O_WORKDIR

ipcluster start --profile=mpi &
sleep 90

python simulator.py

ipcluster stop --profile=mpi
```

Interactive MPI

- Create MPI functions

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

- Call interactively from direct view

Conclusions

- Direct interface
  - Quickly parallelize python code
  - Less control that MPI
- Task interface
  - Fault-tolerant, load-balanced, simple
- MPI interactive
  - Great for developing, debugging

- Tutorials:
  - http://ipython.org/ipython-doc/dev
  - http://minrk.github.com/scipy-tutorial-2011