Parallel Python

Erik Spence

SciNet HPC Consortium

14 July 2015

Material for this class

All the material for the HPC Summer School can be found here:

`https://wiki.scinet.utoronto.ca/wiki/index.php/2015_Ontario_Summer_School_for_High_Performance_Computing_Central`

The slides for this class can be found here:

`http://www.scinethpc.ca/~ejspence/Parallel-Python.pdf`

and at the SciNet education website:

`http://support.scinet.utoronto.ca/education`

An introduction to parallel Python

This afternoon we will cover the following approaches to parallelizing Python code:

- Getting setup on SciNet.
- Memory management.
- Out-of-core computation.
- numexpr package.
- process forks.
- Spawned threads (Threading module).
- Spawned processes (multiprocessing module).
- Pools.
- MPI and Python.
- ipython cluster.

SciNet compute • calcul CANADA

Getting setup on SciNet

Please perform the following steps to get yourself setup for today's class.

```
ejspence@mycomp ~>
ejspence@mycomp ~> ssh ejspence@login.scinet.utoronto.ca -X
ejspence@scinet01-ib0 ~>
ejspence@scinet01-ib0 ~> ssh -X gpc03
ejspence@gpc-f103n084-ib0 ~>
ejspence@gpc-f103n084-ib0 ejspence> type the command below

            qsub -l nodes=1:ppn=8,walltime=4:00:00 -X -q teach -I

ejspence@gpc-f108n045-ib0 ~>
```

It should only take a moment to get your compute node. Raise your hand if it takes more than a minute.

Getting setup on SciNet, continued

You now have your own compute node on SciNet. This is where you will run the code for today's class.

```
ejspence@gpc-f108n045-ib0 ~>
ejspence@gpc-f108n045-ib0 ~> pwd
/home/s/scinet/ejspence
ejspence@gpc-f108n045-ib0 ~> cd /scinet/course/ss2015/Python/code
ejspence@gpc-f108n045-ib0 code>
ejspence@gpc-f108n045-ib0 code> pwd
/scinet/course/ss2015/Python/code
ejspence@gpc-f108n045-ib0 code>
ejspence@gpc-f108n045-ib0 code> source ../setup
ejspence@gpc-f108n045-ib0 code>
```

Why are we here?

What is the modivation for this class? The basic problem is one of data analysis:

- Datasets can be huge.
- Data analysis is often done using languages that are not generally considered "high performance", such as Python and R.
- Parallelization of the data analysis process can greatly speed things up.
- The means by which such parallelization is accomplished is often not well known, or the pitfalls involved are not well-understood.

We're here to try to help you understand how to parallelize code written in Python, to speed up data analysis in particular.

SciNet compute • calcul CANADA

Python and memory management

Python, like R, relies on a "garbage collector" to clean up un-needed variables and limit memory usage.

- "Every so often" a garbage collection task runs and deletes variables that won't be used anymore.
- You can force the garbage collector to run at any time by running the command:

```
import gc
collect = gc.collect()
```

- That being said, the garbage collector knows how to do its job, so running it by hand should only be done in specific circumstances.

Running the garbage collector

Here are some recommendations for runnning the garbage collector (GC) by hand:

- Run the GC after your application has finished starting up and transitions to a 'steady state', if appropriate.
- Run the GC after running infrequently-run sections of code which use and then free large amounts of memory.
- Do not run the GC very often; it can be a slow function to run. As a general rule, it knows how to do its job.

```
import gc, numpy

a = numpy.arange(10000000)
# Do some stuff.
a = 0
collect = gc.collect()
```

Deleting variables

Can't you just delete a variable?

- You probably noticed the lack of an analogy to the "rm(variable)" function in R.
- There is no such function in Python. At least not that can be run within scripts.
- The "del" command can be run at the interactive prompt, but not from within a script.

```
In [1]:
```

```
In [1]: a = numpy.arange(10000000)
```

```
Do some stuff.
```

```
In [2]: del a
```

```
In [3]:
```

Out-of-core computation

Some problems require doing fairly simple analysis on data that is too large to fit into memory

- Min/mean/max.
- Data cleaning.
- Even linear fitting is pretty simple.

In this case, one processor may be enough; you just want a way to not run out of memory.

"Out of core" or "external memory" computation leaves the data on disk, bringing into memory only what is needed, or what fits, at any given time.

For some computations, this works out well (but note: disk access is always much slower than memory access).

Out-of-core computation, continued

The NumPy.memmap class creates a memory-map to an array stored in a binary file on disk. This allows a file-backed out-of-memory computation, but only on NumPy arrays.

This approach works fairly well when one's data access involves passing through the entire data set a very small number of times, either combining data or extracting a subset.

There are other techniques for Python out-of-core computations, involving the combined use of pytables, hdf5 and numpy, but we won't cover them today.

Out-of-core computation, example

First, let us create a large array on file (don't actually perform these steps).

```
In [3]: import numpy as np
In [4]:
In [5]: # this creates a 12G file
In [6]: fp = np.memmap('bigfile', dtype = 'float64', mode = 'w+',
....:  shape = (40000, 40000))
In [7]:
In [7]: fp[0,0:5]
Out[7]: memmap([ 0., 0., 0., 0., 0.])
In [8]: fp[0,:] = np.random.rand(40000)
In [9]: fp[0,0:5]
Out[9]: memmap([ 0.78326485, 0.15310125, 0.57022429, 0.39744545, 0.15487935])
In [10]:
In [10]: for i in xrange(40000): fp[i,:] = np.random.rand(40000)
```

Note: this is really hard on the filesystem (the last command took over an hour to complete).

Out-of-core computation, example

Now start over and calculate the mean of the array.

```
In [11]: exit
ejspence@gpc-f108n045-ib0 ejspence>
ejspence@gpc-f108n045-ib0 ejspence> ipython --pylab
In [1]:
In [1]: import numpy as np
In [2]: fp = np.memmap('bigfile', mode = 'r', shape = (40000, 40000))
In [3]:
In [3]: total = 0.0
In [3]:
In [3]: for i in xrange(40000): total += sum(fp[i,:])
In [4]:
In [4]: average = total / (40000 * 40000)
In [5]:
In [5]: print average
0.0001204345
```

Using multiple processors with Python

The rest of today we will cover using multiple processors and/or nodes to do large-scale computations using Python.

- numexpr package.
- process forks.
- Spawned threads (Threading module).
- Spawned processes (multiprocessing module).
- Pools.
- MPI and Python.
- ipython cluster.

SciNet compute • calcul CANADA

The numexpr package

The numexpr package is useful if you're doing matrix algebra:

- it's essentially a just-in-time compiler for NumPy.
- it takes matrix expressions, breaks things up into threads, and does the calculation in parallel.
- Somewhat awkwardly, it takes it's input in as a string.
- "%timeit" is an IPython "magic command". It accesses the "timeit.timeit" package to time calculations.

In some situations using numexpr can significantly speed up your calculations.

Using the numexpr package

```
In [6]:
```

Using the numexpr package

```
In [6]: import numpy as np, numexp as ne
```

```
In [7]:
```

```
In [7]:
```

Using the numexpr package

```
In [6]: import numpy as np, numexp as ne
In [7]:
In [7]: a = np.random.rand(1e6)
In [8]: b = np.random.rand(1e6)
In [9]: c = np.zeros(1e6)
In [10]:
In [10]:
```

Using the numexpr package

```
In [6]: import numpy as np, numexp as ne
In [7]:
In [7]: a = np.random.rand(1e6)
In [8]: b = np.random.rand(1e6)
In [9]: c = np.zeros(1e6)
In [10]:
In [10]: %timeit c = a**2 + b**2 + 2 * a * b
10 loops, best of 3:  46.6 ms per loop
In [11]:
In [11]:
```

Using the numexpr package

```
In [6]: import numpy as np, numexp as ne
In [7]:
In [7]: a = np.random.rand(1e6)
In [8]: b = np.random.rand(1e6)
In [9]: c = np.zeros(1e6)
In [10]:
In [10]: %timeit c = a**2 + b**2 + 2 * a * b
10 loops, best of 3:  46.6 ms per loop
In [11]:
In [11]: old = ne.set_num_threads(1)
In [12]:
```

Using the numexpr package

```
In [6]: import numpy as np, numexp as ne
In [7]:
In [7]: a = np.random.rand(1e6)
In [8]: b = np.random.rand(1e6)
In [9]: c = np.zeros(1e6)
In [10]:
In [10]: %timeit c = a**2 + b**2 + 2 * a * b
10 loops, best of 3:  46.6 ms per loop
In [11]:
In [11]: old = ne.set_num_threads(1)
In [12]: %timeit ne.evaluate("a**2 + b**2 + 2 * a * b", out = c)
10 loops, best of 3:  11.7 ms per loop
In [13]:
In [13]:
```

Using the numexpr package

```
In [6]: import numpy as np, numexp as ne
In [7]:
In [7]: a = np.random.rand(1e6)
In [8]: b = np.random.rand(1e6)
In [9]: c = np.zeros(1e6)
In [10]:
In [10]: %timeit c = a**2 + b**2 + 2 * a * b
10 loops, best of 3:  46.6 ms per loop
In [11]:
In [11]: old = ne.set_num_threads(1)
In [12]: %timeit ne.evaluate("a**2 + b**2 + 2 * a * b", out = c)
10 loops, best of 3:  11.7 ms per loop
In [13]:
In [13]: old = ne.set_num_threads(2)
In [14]: %timeit ne.evaluate("a**2 + b**2 + 2 * a * b", out = c)
10 loops, best of 3:  7.8 ms per loop
In [15]:
```

Process forking

Another simple way to run code in parallel is to 'fork' the process.

- The system call fork() creates a copy of the process that called it, and runs it as a child process.
- The child gets ALL the data of the parent process.
- The child gets its own process number (PID), and as such runs independently of the parent.
- We use the return value of fork() to determine which process we are; 0 means we're the child.
- Probably doesn't work in Windows.

```
# firstfork.py
import os

# Our child process.
def child():
  print "Hello from", os.getpid()
  os._exit(0)

# The parent process.
while (True):
  newpid = os.fork()
  if newpid == 0:
    child()
  else:
    print "Hello from parent",
      os.getpid(), newpid

  if raw_input() == "q": break
```

SciNet compute • calcul CANADA

Process forking, continued

What does that look like?

```
ejspence@gpc-f108n045-ib0 code>
ejspence@gpc-f108n045-ib0 code> python firstfork.py
Hello from parent 27089 27090
Hello from 27090
q
ejspence@gpc-f108n045-ib0 code>
```

If a python script is given as an argument to the "python" command, then Python will run the script.

forking/executing

What if we prefer to run a completely different code, rather than copying the existing code to the child?

- we can run one of the os.exec series of functions.
- The os.execlp call replaces the currently running program with the new one specified, in the child process only.
- If os.execlp is successful at lauching the program, it never returns. Hence the assert statement is only invoked if something goes wrong.

```
# child.py
import os
print "Hello from", os.getpid()
os._exit(0)
```

```
# secondfork.py
import os

while (True):
  pid = os.fork()
  if pid == 0:
    os.execlp("python", "python",
      "child.py")
    assert False,
      "Error starting program"
  else:
    print "The child is", pid
    if raw_input() == "q": break
```

Notes about fork()

Fork was an early implementation used to spawn sub-processes, and is no longer commonly used. Some things to remember if you try to use this approach:

- use os.waitpid(child_pid) if you need to wait for the child process to finish. Otherwise the parent will exit and the child will live on.
- fork() is a Unix command. It doesn't work on Windows, except under Cygwin.
- This must be used very carefully, ALL the data is copied to the child process, including file handles, open sockets, database connections...
- Be sure to exit using os._exit(0) rather than os.exit(0), or else the child process will try to clean up resources that the parent process is still using.
- Because of the above, fork() can lead to code that is difficult to maintain long-term.

Using fork in data analysis

Some notes about using forks in the context of data analysis:

- Something you may have noticed the about fork examples thus far is the lack of return from the functions.
- Forked processes, being processes and not threads, do not share anything with the parent process.
- As such, the only way they can return anything to the parent function is through inter-process communication.
- This is possible (as we saw yesterday with R), though a bit tricky. We'll look at one way to do this later in the class.
- Your best bet, from a data processing point of view, is to just use fork for one-time functions that do not return anything to the parent.

SciNet compute • calcul CANADA

Processes versus threads

There is often confusion on the difference between threads and processes.

- A process provides the resources needed to execute a program. A thread is a path of execution *within* a process. As such, a process contains at least one thread, possibly many.
- A process contains a considerable amount of state information (handles to system objects, PID, address space, ...). As such they are more resource-intensive to create. Threads are very light weight in comparison.
- Threads within the same process share the same address space. This means they can share the same memory and can easily communicate with each other.
- Different processes do not share the same address space. Different processes can only communicate with each other through OS-supplied mechanisms.

Notes about threads

Are there advantages to using threads, versus processes?

- As noted about, threads are light-weight compared to processes. As a result, they start up more quickly.
- Threads can be simpler to program, especially when the threads need to communicate with each other.
- Threads share memory, which can simplify (as well as obfuscate) programming.
- Threads are more portable than forked processes, as they are fully supported by Windows.

These points aside, there are downsides to using threads in a data-analysis application, as we'll see in a moment.

SciNet compute • calcul CANADA

Spawning threads, which is faster?

```python
# summer.py
def my_summer(start, stop):
  tot = 0
  for i in xrange(start,stop):
    tot += i
```

```python
# summer.serial.py
import time
from summer import my_summer

begin = time.time()
threads = []

for i in range(10):
  my_summer(0, 5000000)

print "Time:", time.time() - begin
```

```python
# summer.threaded.py
import time, threading
from summer import my_summer

begin = time.time()
threads = []

for i in range(10):
  t = threading.Thread(target = my_summer,
    args = (0, 5000000))
  threads.append(t)
  t.start()

# Wait for all threads to finish.
for t in threads: t.join()

print "Time:", time.time() - begin
```

SciNet compute • calcul CANADA

Not even close

The threading code is no faster than the serial code, even on my computer with two cores. Why?

- The Python Interpreter uses the Global Interpreter Lock (GIL).
- To prevent race conditions, the GIL prevents threads from the same Python program from running simultaneously. As such, only one core is used at any given time.
- Consequently the threaded code is no faster than the serial code, and is generally slower due to thread-creation overhead.
- As a general rule, threads are not used for most Python applications (GUIs being one important exception). This example is for demonstration purposes only.
- Instead, we will use one of several other modules, depending on the application in question. These modules will launch subprocesses, rather than threads.

SciNet compute • calcul CANADA

The multiprocessing module

The multiprocessing module tries to strike a balance between forks and threads:

- Unlike fork, multiprocessing works on Windows (better portability).
- Slightly longer start-up time than threads.
- Multiprocessing spawns separate processes, like fork, and as such they each have their own memory.
- Multiprocessing requires pickleability for its processes on Windows, due to the way in which it is implemented. As such, passing non-pickleable objects, such as sockets, to spawned processes is not possible.

SciNet compute • calcul CANADA

The multiprocessing module, continued

A few notes about the multiprocessing module:

- The Process function launches a separate process.
- The syntax is very similar to the threading module. This is intentional.
- The details under the hood depend strongly upon the system involved (Windows, Mac, Linux), thus the portability of code written with this module.

```
# summer.multiprocessing.py
import time, multiprocessing
from summer import my_summer

begin = time.time()
processes = []

for i in range(10):
  p = multiprocessing.Process(
    target = my_summer,
    args = (0, 5000000))
  processes.append(t)
  p.start()

# Wait for all processes to finish.
for p in processes: p.join()

print "Time:", time.time() - begin
```

Multiprocessing example

How does it perform?

```
ejspence@gpc-f108n045-ib0 code>
ejspence@gpc-f108n045-ib0 code> python summer.serial.py
Time: 4.01846909523
ejspence@gpc-f108n045-ib0 code>
ejspence@gpc-f108n045-ib0 code> python summer.multiprocessing.py
Time: 0.57817697525
ejspence@gpc-f108n045-ib0 code>
```

Much improved.

Multiprocessing pools

The process of assigning tasks can be automated using Pools. The number of jobs to be run is given by the number of entries in the input which is passed.

Note that Pool.map is a blocking function.

```
# summer.py
def my_summer2(data):

  # Only one argument may be
  # passed using Pool.
  start, stop = data
  tot = 0
  for i in xrange(start,stop):
    tot += i
```

```
# summer.multiprocesing.pool.py
import time, multiprocessing
from summer import my_summer2

begin = time.time()
numjobs = 10
numprocs = multiprocessing.cpu_count()

# The arguments are the same for all.
input = [(0, 5000000)] * numjobs

p = multiprocessing.Pool(
  processes = numprocs)

p.map(my_summer2, input)

print "Time:", time.time() - begin
```

SciNet compute • calcul CANADA

The multiprocess module, shared memory

The multiprocess module allows one to seamlessly share memory between processes. This is done using the 'Value' and 'Array' objects.

Value is a wrapper around a ctype object. The first argument is the variable type, the second is that value.

Does this code behave as expect? Why not?

```
# multiprocessing.shared.py
import time
from multiprocessing import Process,
  Value

def myfunc(v):
  for i in range(50):
    time.sleep(0.001)
    v.value += 1

if __name__ == "__main__":
  v = Value('i', 0);      procs = []
  for i in range(10):
    p = Process(target = myfunc,
      args = (v,))
    procs.append(p)
    p.start()

  for proc in procs: proc.join()
  print v.value
```

Race conditions

What went wrong?

- Race conditions occur when program instructions are executed in an order not intended by the programmer. The most common cause is when multiple processes are given access to a resource.
- In the example here, we've modified a location in memory that is being accessed by multiple processes.
- Note that it need not only be processes or threads that can modify a resource, anything can modify a resource, hardware or software.
- Bugs caused by race conditions are extremely hard to find.
- Disasters can occur (Therac-25).

Be very very careful when sharing resources between multiple processes or threads!

SciNet compute • calcul CANADA

Using shared memory, continued

The solution, of course, is to be more explicit in your locking.

If you must use shared memory, be sure to test everything very thoroughly.

```
# multiprocessing.shared.fixed.py
import time
from multiprocessing import Process,
  Value, Lock

def myfunc(v, lock):
  for i in range(50):
    time.sleep(0.001)
    with lock:
      v.value += 1
```

```
# multiprocessing.shared.fixed.py,
# continued

if __name__ == "__main__":
  v = Value('i', 0)
  lock = Lock()
  procs = []
  for i in range(10):
    p = Process(target = myfunc,
      args = (v, lock))
    procs.append(p)
    p.start()

  for proc in procs: proc.join()
  print v.value
```

Using shared memory, arrays

Multiprocessing also allows you to share a block of memory through the Array ctypes wrapper.

Only 1-D arrays are permitted. Note that multiprocessing.Process must be used; shared memory does not work with multiprocessing.Pool.map.

Note that, since arr is actually a ctypes object, you must print the contents of arr to see the result.

```
# multiprocessing.shared.array.py
from numpy import arange
from multiprocessing import Process,
Array

def myfuncf(a, i): a[i] = -a[i]

if __name__ == "__main__":
  arr = Array('d', arange(10.))
  procs = []
  for i in range(10):
    p = Process(target = myfunc,
      args = (arr, i))
    procs.append(p)
    p.start()

  for proc in procs: proc.join()
  print arr[:]
```

But there's more!

The multiprocessing module is loaded with functionality. Other features include:

- Inter-process communciation, using Pipes and Queues.
- multiprocessing.manager, which allows jobs to be spread over multiple 'machines' (nodes).
- subclassing of the Process object, to allow further customization of the child process.
- multiprocessing.Event, which allows event-driven programming options.
- multiprocess.condition, which is used to synchronize processes.

We're not going to cover these features today.

SciNet compute • calcul CANADA

The multiprocessing package is ubiquitous

The multiprocessing module is the module used my many other packages to handle parallel functionality. Some of these include:

- scikit-learn, a machine-learning suite.
- fabric, a package used to streamline SSH commands in applications.
- rufus, used for simplifying pipelines.
- PyCAM, a toolpath generator for 3-axis CNC machining.
- Nipype, a toolkit for neuroimaging pipelines.
- pydoit, a task dependency and execution manager.
- Pythics, laboratory instrument control software.
- depparse, a dependency parser.

And many others.

SciNet compute • calcul CANADA

The focus of parallel Python

As you may have noticed, the parallel capabilities of Python are kinda clunky, at least for data analysis:

- The approach of the multiprocessing package is rather low-level as compared to what we saw yesterday. It lacks the nice detail-management which is baked into the parallel R functionality.
- The approach is not nearly as compatible with interactive programming as parallel R.
- The approach does not lend itself as well to data analysis as parallel R functionality. Most of what we've seen is better suited to systems administration.
- Nonetheless, with some work multiple processors could be worked into your workflow. In particular, analyses that are not exploratory, but rather simply repetitive number crunching.

SciNet compute • calcul CANADA

However, there are other options

The multiprocessing module offers an interface for spawning independent processes. As such, it's pretty handy and easy to use. However, there are other packages out there that also attempt to fill this role:

- Parallel Python
- pprocess
- joblib
- Celery
- and others.

One advantage of multiprocessing is that it's part of the standard Python distribution, and thus is the most commonly used.

Python and MPI

There are other approaches available to parallelize your Python data analysis. You may be wondering where MPI is in all of this?

- Python interfaces to MPI functionality have been written. There are several implementations available.
- These interfaces have all of the usual MPI functionality that you've come to know and love.
- However, I've never seen anyone use Python MPI for data processing.
- Why?
  - Python MPI interfaces are not nearly as fast as compiled MPI codes (C, C++, Fortran).
  - There are easier ways to parallelize your Python workflows (as we've seen).
  - MPI is better suited to communication-heavy problems; data analysis usually doesn't fall into this category.

For completeness, we'll cover one example.

Python MPI

Several interfaces to MPI have been implemented in Python.

- mpi4py, pypar, MaroonMPI, pyMPI, pupyMPI, ...
- We will use pypar for our example.
- Note that, on GPC, pypar has been compiled with IntelMPI, so that module must be loaded for pypar to work.

Area-under-the-curve using pypar

```
# AUC.parallel.py
import pypar, sys
from numpy import zeros

numprocs = pypar.size()
myid = pypar.rank()

# buffers for communication
msg = zeros(1)
answer = zeros(1)

if (len(sys.argv) == 2):
  n = int(sys.argv[1])
else: n = 10

dx = 3.0 / n    # Width of each bar.
area = 0.0

# Width of each processor's block.
width = 3.0 / numprocs
```

```
# Number of bars for each processor.
numbars = n / numprocs

# My starting x value.
x = myid * width

# Each proc. just works on numbars.
for i in range(numbars):
  y = 0.7 * x**3 - 2 * x**2 + 4
  area = area + y * dx
  x = x + dx

msg[0] = area

pypar.reduce(msg, pypar.SUM, 0,
  buffer = answer)

if (myid == 0):
  print "The area is", answer
pypar.finalize()
```

What's the output?

```
ejspence@gpc-f108n045-ib0 code>
```

```
ejspence@gpc-f108n045-ib0 code> mpirun -np 2 python AUC.parallel.py
Pypar (version 2.1.5) initialised MPI OK with 2 processors
The area is 8.09175
```

```
ejspence@gpc-f108n045-ib0 code>
```

```
ejspence@gpc-f108n045-ib0 code> mpirun -np 2 python AUC.parallel.py 100
Pypar (version 2.1.5) initialised MPI OK with 2 processors
The area is 8.1620175
```

```
ejspence@gpc-f108n045-ib0 code>
```

```
ejspence@gpc-f108n045-ib0 code> mpirun -np 10 python AUC.parallel_args.py
20000000
Pypar (version 2.1.5) initialised MPI OK with 10 processors
The area is [ 8.17499993]
```

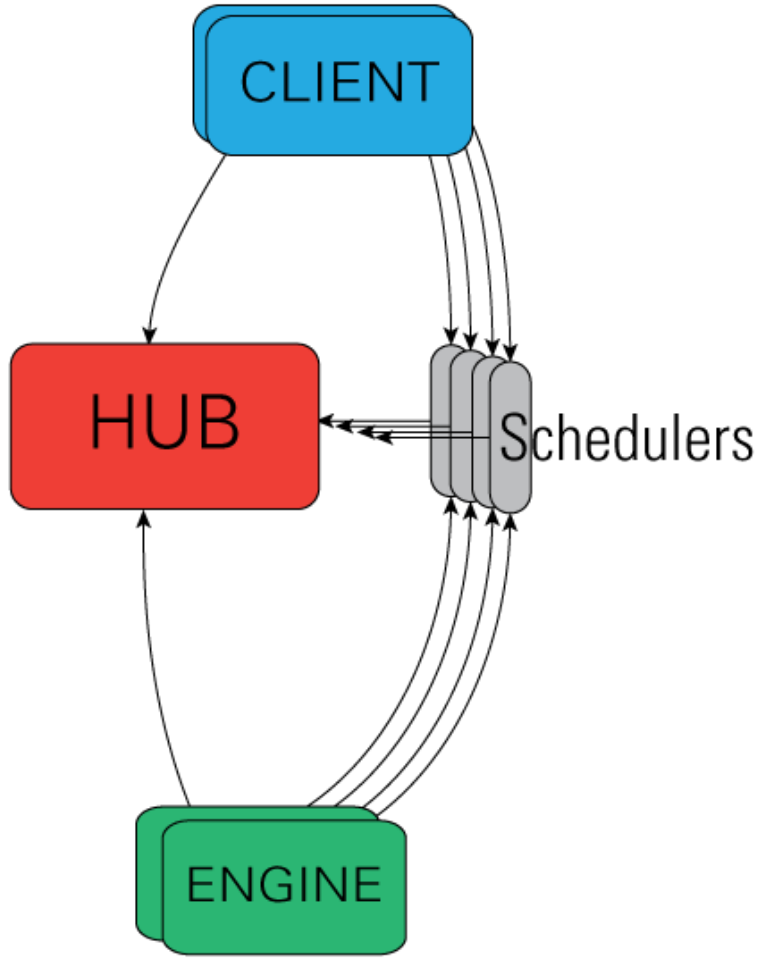
```
ejspence@gpc-f108n045-ib0 code>
```

The exact area for this problem is 8.175.

IPython's Parallel Architecture

Interestingly, IPython comes with a built-in parallel engine. It consists of four components:

- Engines: Do the work. One core, one engine.
- Schedulers: Deliver and divide the work.
- Hub: Coordinates and logs the engine and schedule activity.
- Clients: Request work to be done on engines.



Schedulers + Hub + Engine = Cluster
Schedulers + Hub = Controller

Starting an IPython cluster

Starting up an IPython cluster is not difficult. But there are some steps that you need to go through carefully:

1. In your current SciNet terminal, on the compute node:
   1. Get the hostname of your node. You'll need this on for the next slide.
   2. Change to the ipython directory.
   3. Source the setup file.
   4. Start the IPython cluster.

```
ejspence@gpc-f108n045-ib0 code> hostname
gpc-f108n045
```

```
ejspence@gpc-f108n045-ib0 code> pwd
/scratch/s/scinet/ejspence/Python
```

```
ejspence@gpc-f108n045-ib0 code> cd ipython
```

```
ejspence@gpc-f108n045-ib0 ipython> source setup
```

```
ejspence@gpc-f108n045-ib0 ipython> ipcluster start -n 4
a bunch of messages...
```

You can minimize this terminal, but don't close it.

Starting an IPython cluster, continued

Now for step 2:

2. Open a second terminal on your laptop.
   1. ssh into a SciNet login node.
   2. From the login node, ssh directly into YOUR compute node.

```
ejspence@mycomp ~>
ejspence@mycomp ~> ssh ejspence@login.scinet.utoronto.ca -X
ejspence@scinet01-ib0 ~>
ejspence@scinet01-ib0 ~> ssh gpc-f108n045
ejspence@gpc-f108n045-ib0 ~>
```

Starting an IPython cluster, continued

We are now ready to access the running cluster. In your new terminal, on your compute node:

- Move to the correct directory.
- Start IPython.
- Then grab the handles to the clients.

```
ejspence@gpc-f108n045-ib0 ~>
ejspence@gpc-f108n045-ib0 ~> cd /scinet/course/ss2015/Python/ipython
ejspence@gpc-f108n045-ib0 ipython>
ejspence@gpc-f108n045-ib0 ipython> source setup
ejspence@gpc-f108n045-ib0 ipython>
ejspence@gpc-f108n045-ib0 ipython> ipython --pylab
In [1]:
In [1]: from IPython.parallel import Client
In [2]: clients = Client()
In [3]:
```

Accessing the Clients

Let's see what we've got here.

```
In [3]:
In [3]: # use synchronous computations
In [3]: clients.block = True
In [4]:
In [4]: print len(clients)
4
In [5]:
In [5]: print clients.ids
[0, 1, 2, 3]
In [6]:
```

Each client has been assigned an id, starting at 0.

Accessing the Clients, continued

Here's simple function to execute on the cores:

```
In [6]:
In [6]: def minus(a, b): return a - b
In [7]:
In [7]: minus(5, 6)
Out[7]: -1
In [8]:
```

Execution on the first engine only:

```
In [8]:
In [8]: clients[0].apply(minus, 5, 6)
Out[9]: -1
In [10]:
```

Interfaces to the Engines

Some notes about the engines:

- The python environment that holds the 'Clients' part is completely separate from that of the 'Engines'.
- As such, you need to move data and code to the Engines.
- You also need to request to execute code on the Engines.

The Controller (Schedulers + Hub) is the single point of contact for the clients.

SciNet compute • calcul CANADA

Views

Views are a layer over sets of engines that allow access to engine variables through a dictionary, storing settings, and scheduling tasks.

There are two kinds of views:

- A Direct interface, where engines are addressed explicitly. You get this view by using square brackets. For example: Client()[1:8:2].
- A LoadBalanced interface, where the Scheduler is trusted with assigning work to appropriate engines. You get this from Clients.load_balanced_view().

View is selected by the client.

Parallel Execution

There are a number of ways to invoke the Engines:

- clients[:].run takes a script and runs in on the engine(s).
- clients[:].execute takes a command, as a string, to run on the engine(s).
- clients[:].apply takes a function and arguments, to run on the engine(s).
- clients[:].map takes a function and a list, to distribute over the engine(s).

In the last two, the function and arguments get shipped to the engine.

SciNet compute • calcul CANADA

Blocking/Nonblocking

There are two modes in which execution of code can run:

- In blocking mode ("synchronous"), all execution must be finished before results are recorded.
- In non-blocking mode, an "AsyncResult" is returned, which we can ask if it is done (.ready()), and what the result is (.get()).

The latter is potentially faster, but requires a bit more 'infrastructure'.

Examples

Execute `minus` in parallel on all the engines at once:

```
In [10]:
```

```
In [10]: clients[:].apply(minus, 5, 6)
Out[10]: [-1, -1, -1, -1]
```

```
In [11]:
```

What if we want different arguments to each engine? In normal Python we could use "map":

```
In [11]:
```

```
In [11]: map(minus [11, 10, 9, 8], [5, 6, 7, 8])
Out[11]: [6, 4, 2, 0]
```

```
In [12]:
```

Examples, continued

The client view's "map" function executes in parallel. Using a load-balanced view, this would look like this:

```
In [12]:
In [12]: view = clients.load_balanced_view()
In [13]:
In [13]: view.map(minus, [11, 10, 9, 8], [5, 6, 7, 8])
Out[13]: [6, 4, 2, 0]
In [14]:
```

Direct view

Recall that the "Direct View" allows you to directly command the engines. To execute a command on all the engines:

```
In [14]:
In [14]: clients.block = True
In [14]:
In [14]: dview = clients.direct_view()
In [15]:
In [15]: dview.block = True
In [16]:
In [16]: dview.apply(sum, [1, 2, 3])
Out[16]: [6, 6, 6, 6]
In [17]:
```

Direct view, continued

Slicing a Client's object gets you a Direct view as well:

```
In [17]:
```

```
In [17]: clients[::2]
Out[17]: <DirectView [0, 2]>
```

```
In [18]: clients[::2].apply(sum, [1, 2, 3])
Out[18]: [6, 6]
```

```
In [19]:
```

Which we saw previously, when we used clients[:].apply(minus, 5, 6).

Load Balanced View

To execute a command on all the engines, using the Load Balanced View:

```
In [19]:
In [19]: dview = Clients().load_balanced_view()
In [20]:
In [20]: dview.block = True
In [21]:
In [21]: dview.apply(sum, [1, 2, 3])
Out[21]: 6
In [22]:
```

This view is useful if you're going to execute tasks one by one, or if the tasks take a varying amount of time.

We will focus on direct view in the remainder, which is a bit more flexible.

Data movement to and from engines

Moving data around is straighforward.

- You can access variables though a dictionary-like interface.
- Indexing a client gives access to the dictionary for a particular engine.
- A view has a dictionary interface too, which gives you a list of the values in all the engines.

```
In [22]:
```

```
In [22]: v = clients[:]
```

```
In [23]: v.block = True
```

```
In [24]: v.execute('from os import getpid')
Out[24]: <AsyncResult:  finished>
```

```
In [25]: v.execute('x = getpid()')
Out[25]: <AsyncResult:  finished>
```

```
In [26]: v['x']
Out[26]: [24068, 24067, 24065, 24066]
```

```
In [27]: c[3]['x']
Out[27]: 24066
```

Scatter/Gather

Some of the parallelization features we saw yesterday are built in as well.

- Sometimes you want to explicitly divide a list or array on the engines: Scatter.
- or reconstruct a larger list on the client from local lists on the engines: Gather.

This is quite simple in IPython.parallel:

```
In [28]: v.scatter('a', np.arange(16))
```

```
In [29]: v['a']
Out[29]:
[array([0, 1, 2, 3]),
array([4, 5, 6, 7]),
array([8, 9, 10, 11]),
array([12, 13, 14, 15])]
```

```
In [30]: v.gather('a')
Out[30]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15])]
```

IPython Hands-on

For each month in 1988-2012, list airports with longest departure delay.
Output should look like this

```
1988 01  1389.0 ['LGA']
1988 02  1320.0 ['TUS', 'DCA']
```

- Re-use the python function "maxdelay", which is in the ipython directory.
- Run different months on different IPython engines.
- Also do a scaling test, meaning determine how long this takes use 2, 4, 6 and 8 cores.

```
from csv import reader
def maxdelay(file):
  rd = reader(open(file,'rb'))
  hd = rd.next()
  dl = hd.index('DEP_DELAY')
  on = hd.index('ORIGIN')
  ls = [[float(r[dl]), r[on]]
    for r in rd if r[dl] != '']
  dt = max(ls)[0]
  ps = [r[1] for r in ls if r[0] == dt]
  return dt, list(set(ps))
```