Theano (an alternative to numexpr)

- Theano is a numerical computation library.
- Much like numexpr, it takes an (array) expression and compiles it.
- Theano is frequently use in machine learning applications.
- Unlike numexpr, it can use multi-dimensional arrays and slices, like NumPy.
- Unlike numexpr, it does not natively use threads (though it may link to multithreaded blas libraries).
- But it can use GPUs (haven't tried).

Theano in the diffusion equation

For the diffusion code, only the computation of the laplacian changes

```
t_dens = theano.tensor.dmatrix('dens')
t_laplacian = (t_dens[2:nrows+2,1:ncols+1] +
               t_dens[0:nrows+0,1:ncols+1] +
               t_dens[1:nrows+1,2:ncols+2] +
               t_dens[1:nrows+1,0:ncols+0] -
               4*t_dens[1:nrows+1,1:ncols+1])
t_laplacian_fun = theano.function([t_dens], t_laplacian)
laplacian[1:nrows+1,1:ncols+1] = t_laplacian_fun(dens)
```

Theano in the diffusion equation

For the diffusion code, only the computation of the laplacian changes

```
t_dens = theano.tensor.dmatrix('dens')
t_laplacian = (t_dens[2:nrows+2,1:ncols+1] +
               t_dens[0:nrows+0,1:ncols+1] +
               t_dens[1:nrows+1,2:ncols+2] +
               t_dens[1:nrows+1,0:ncols+0] -
               4*t_dens[1:nrows+1,1:ncols+1])
t_laplacian_fun = theano.function([t_dens], t_laplacian)
laplacian[1:nrows+1,1:ncols+1] = t_laplacian_fun(dens)
```

Worth it, using 28 cores?

Theano in the diffusion equation

For the diffusion code, only the computation of the laplacian changes

```
t_dens = theano.tensor.dmatrix('dens')
t_laplacian = (t_dens[2:nrows+2,1:ncols+1] +
               t_dens[0:nrows+0,1:ncols+1] +
               t_dens[1:nrows+1,2:ncols+2] +
               t_dens[1:nrows+1,0:ncols+0] -
               4*t_dens[1:nrows+1,1:ncols+1])
t_laplacian_fun = theano.function([t_dens], t_laplacian)
laplacian[1:nrows+1,1:ncols+1] = t_laplacian_fun(dens)
```

Worth it, using 28 cores?

```
$ etime python diff2d_numpy
Elapsed: 21.03 seconds
$ etime python diff2d_numexpr
Elapsed: 4.16 seconds
$ etime python diff2d_theano
Elapsed: 13.12 seconds
```

Numexpr wins.

Theano in the diffusion equation

For the diffusion code, only the computation of the laplacian changes

```
t_dens = theano.tensor.dmatrix('dens')
t_laplacian = (t_dens[2:nrows+2,1:ncols+1] +
               t_dens[0:nrows+0,1:ncols+1] +
               t_dens[1:nrows+1,2:ncols+2] +
               t_dens[1:nrows+1,0:ncols+0] -
               4*t_dens[1:nrows+1,1:ncols+1])
t_laplacian_fun = theano.function([t_dens], t_laplacian)
laplacian[1:nrows+1,1:ncols+1] = t_laplacian_fun(dens)
```

Worth it, using 28 cores?

```
$ etime python diff2d_numpy
Elapsed: 21.03 seconds
$ etime python diff2d_numexpr
Elapsed: 4.16 seconds
$ etime python diff2d_theano
Elapsed: 13.12 seconds
```

Numexpr wins.

How about serially?

Theano in the diffusion equation

For the diffusion code, only the computation of the laplacian changes

```
t_dens = theano.tensor.dmatrix('dens')
t_laplacian = (t_dens[2:nrows+2,1:ncols+1] +
               t_dens[0:nrows+0,1:ncols+1] +
               t_dens[1:nrows+1,2:ncols+2] +
               t_dens[1:nrows+1,0:ncols+0] -
               4*t_dens[1:nrows+1,1:ncols+1])
t_laplacian_fun = theano.function([t_dens], t_laplacian)
laplacian[1:nrows+1,1:ncols+1] = t_laplacian_fun(dens)
```

Worth it, using 28 cores?

```
$ etime python diff2d_numpy
Elapsed: 21.03 seconds
$ etime python diff2d_numexpr
Elapsed: 4.16 seconds
$ etime python diff2d_theano
Elapsed: 13.12 seconds
```

Numexpr wins.

How about serially?

```
$ #with "ne.set_num_threads(1)"
$ etime python diff2d_numexpr
Elapsed: 14.05 seconds
$ etime python diff2d_theano
Elapsed: 12.91 seconds
```

Theano wins, just.

Forking

Forking (linux specific)

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.

```
# 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.

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

Process forking, continued

What does that look like?

```
$ python firstfork.py
Hello from parent 27089 27090
Hello from 27090
q
$
```

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.

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

```python
# secondfork.py
import os

while (True):
  pid = os.fork()
  if pid == 0:
    os.execlp("python", "python",
              "child.py")
    assert False, "Error starting
  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 diffcult 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, 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.

Threads in Python

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 comprison.
- 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.

How much faster is it using threads?

```
# 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 ("Elasped: %f"%
time.time() - begin,"seconds")
```

```
# summer.py - used in all summer*p
def my_summer(start, stop):
  tot = 0
  for i in xrange(start,stop):
    tot += i
```

```
# summer_serial.py
import time
from summer import my_summer
begin = time.time()
threads = []
for i in range(10):
  my_summer(0, 5000000)
print "Elapsed:", time.time() - be
```

How much faster is it using threads?

```
# 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 ("Elasped: %f"%
time.time() - begin,"seconds")
```

```
# summer.py - used in all summer*p
def my_summer(start, stop):
  tot = 0
  for i in xrange(start,stop):
    tot += i
```

```
# summer_serial.py
import time
from summer import my_summer
begin = time.time()
threads = []
for i in range(10):
  my_summer(0, 5000000)
print "Elapsed:", time.time() - be
```

Timings

```
$ python summer_serial.py
Elapsed: 11.58 seconds
$ python summer_threaded.py
Elapsed: 38.48 seconds
```

Not faster at all, slower!

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.

Multiprocessing

Multiprocessing

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.

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(p)
   p.start()
# Wait for all processes to finis
for p in processes: p.join()
print ("Elapsed:%f"%
      time.time() - begin)
```

```
$ python summer_multiprocessing.py
Elapsed: 0.127079
```

Shared memory with multiprocessing

- `multiprocess` allows one to seamlessly share memory between processes. This is done using 'Value' and 'Array'.
- Value is a wrapper around a strongly typed object called a ctype. When creating a `Value`, the first argument is the variable type, the second is that value.
- Code on the right has 10 processes add 50 increments of 1 to the `Value v`.

```
# multiprocessing_shared.py
from multiprocessing import Proces
from multiprocessing import Value
def myfun(v):
  for i in range(50):
    time.sleep(0.001)
    v.value += 1
v = Value('i', 0);
procs = []
for i in range(10):
  p=Process(target=myfun,args=(v,)
  procs.append(p)
  p.start()
for proc in procs: proc.join()
print(v.value)
```

Shared memory with multiprocessing

- multiprocess allows one to seamlessly share memory between processes. This is done using 'Value' and 'Array'.
- Value is a wrapper around a strongly typed object called a ctype. When creating a Value, the first argument is the variable type, the second is that value.
- Code on the right has 10 processes add 50 increments of

```
# multiprocessing_shared.py
from multiprocessing import Proces
from multiprocessing import Value
def myfun(v):
  for i in range(50):
    time.sleep(0.001)
    v.value += 1
v = Value('i', 0);
procs = []
for i in range(10):
  p=Process(target=myfun,args=(v,)
  procs.append(p)
  p.start()
for proc in procs: proc.join()
print(v.value)
```

```
$ etime python multiprocessing_shared.py
480
Elapsed: 0.12 seconds
```

- Did the code behave as expect?

canada l canada

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.

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

Using shared memory, continued

- The solution, of course, is to be more explicit in your locking.
- If you use shared memory, be sure to test everything thoroughly.

```python
# multiprocessing_shared_fixed.py
from multiprocessing import Proces
from multiprocessing import Value
from multiprocessing import Lock

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

```python
# multiprocessing_shared_fixed.py
# continued
v = Value('i', 0)
lock = Lock()
procs = []
for i in range(10):
 p=Process(target=myfun,
           args=(v,lock))
 procs.append(p)
 p.start()
for proc in procs: proc.join()
print(v.value)
```

```
$ etime python multiprocessing_shared_fixed.py
500
Elapsed: 0.09 seconds
```

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 Proces
def myfun(a, i):
  a[i] = -a[i]
arr = Array('d', arange(10.))
procs = []
for i in range(10):
  p = Process(target=myfun,
              args=(arr, i))
  procs.append(p)
  p.start()
for proc in procs:
  proc.join()
print(arr[:])
```

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 Proces
def myfun(a, i):
  a[i] = -a[i]
arr = Array('d', arange(10.))
procs = []
for i in range(10):
  p = Process(target=myfun,
              args=(arr, i))
  procs.append(p)
  p.start()
for proc in procs:
  proc.join()
print(arr[:])
```

```
[-0.0, -1.0, -2.0, -3.0, -4.0,
 -5.0, -6.0, -7.0, -8.0, -9.0]
```

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.

MPI4PY

Message Passing Interface

The previous parallel techniques used processors on one node.

Using more than one node requires these nodes to communicate.

MPI is one way of doing that communication.

- MPI = Message Passing Interface.
- MPI is a C/Fortran Library API.
- Sending data = sending a message.
- Requires setup of processes through mpirun/mpiexec.
- Requires `MPI_Init(...)` in code to collect processes into a 'communicator'.
- Rather low level.

Mpi4py features

- mpi4py is a wrapper around the mpi library
- Point-to-point communication (sends, receives)
- Collective (broadcasts, scatters, gathers) communications of any picklable Python object,
- Optimized communications of Python object exposing the single-segment buffer interface (NumPy arrays, builtin bytes/string/array objects).
- Names of functions much the same as in C/Fortran, but are methods of the communicator (object-oriented).

MPI C/C++ recap

The following C++ code determines each process' rank and sends that rank to its left neighbor.

```
#include <mpi.h>
#include <iostream>
int main(int argc, char** argv) {
 int rank, size, rankr, right, left;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 right = (rank+1)%size;
 left  = (rank+size-1)%size;
 MPI_Sendrecv(&rank,  1, MPI_INT, left,  13,
              &rankr, 1, MPI_INT, right, 13,
              MPI_COMM_WORLD, MPI_STATUS_IGNORE);
 std::cout<<"I am rank "<<rank<<"; my right neighbour is "<<rankr<<
 MPI_Finalize();
}
```

MPI Fortran recap

The following Fortran code determines each process' rank and sends that rank to its left neighbor.

```fortran
program rightrank
 use mpi
 implicit none
 integer rank, size, rankr, right, left, e
 call MPI_Init(e)
 call MPI_Comm_rank(MPI_COMM_WORLD, rank, e)
 call MPI_Comm_size(MPI_COMM_WORLD, size, e)
 right = mod(rank+1, size)
 left  = mod(rank+size-1, size)
 call MPI_Sendrecv(rank,  1, MPI_INTEGER, left,  13, &
                   rankr, 1, MPI_INTEGER, right, 13, &
                   MPI_COMM_WORLD, MPI_STATUS_IGNORE, e)
 print *, "I am rank ", rank, "; my right neighbour is ", rankr
 call MPI_Finalize(e)
end program rightrank
```

canada canada

Mpi4py

- One of the drudgeries of MPI is to have to express the binary layout of your data.
- The drudgery arises because C and Fortran do not have *introspection* and the MPI libraries cannot look inside your code.
- With Python, this is potentially different: we can investigate, within python, what the structure is.
- That means we should be able to express sending a piece of data without having to specify types and amounts.

```
from mpi4py import MPI
rank  = MPI.COMM_WORLD.Get_rank()
size  = MPI.COMM_WORLD.Get_size()
right = (rank+1)%size
left  = (rank+size-1)%size
rankr = MPI.COMM_WORLD.sendrecv(rank, left, source=right)
print "I am rank", rank, "; my right neighbour is", rankr
```

Mpi4py + numpy

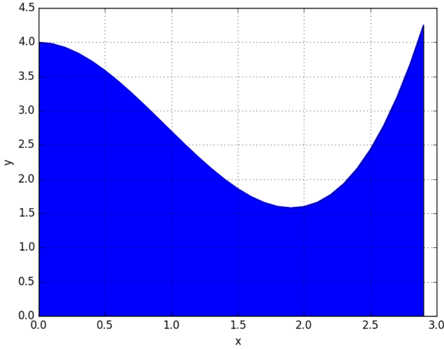
- It turns out that mpi4py's communication is pickle-based.
- Pickle is a *serialization* format which can convert any python object into a bytestream.
- Convenient as any python object can be sent, but conversion takes time.
- For numpy arrays, one can skip the pickling using Uppercase variants of the same communicator methods.
- However, this requires us to preallocate buffers to hold messages to be received.

Example: Area under the curve

- Let's consider a code that numerically computes the following integral:

$$b = \int_{x=0}^{3} \left(\frac{7}{10} x^3 - 2x^2 + 4 \right) dx$$

- Exact answer $b = 8.175$
- It's the area under the curve on the right.



Method: sample $y = \frac{7}{10}x^3 - 2x^2 + 4$ at a uniform grid of x values (using `ntot` number of points), and add the y values.

Mpi4py+numpy: Upper/lowercase example

```python
import sys
from mpi4py import MPI

rank = MPI.COMM_WORLD.Get_rank()
size = MPI.COMM_WORLD.Get_size()
ntot = int(sys.argv[1])
npnts = ntot/size
dx = 3.0/ntot
width = 3.0/size
x = rank*width
a = 0.0
for i in xrange(npnts):
  y = 0.7*x**3 - 2*x**2 + 4
  a += y*dx
  x += dx

b = MPI.COMM_WORLD.reduce(a)
if rank == 0:
  print "The area is", b
```

Mpi4py+numpy: Upper/lowercase example

```
import sys
from mpi4py import MPI

rank = MPI.COMM_WORLD.Get_rank()
size = MPI.COMM_WORLD.Get_size()
ntot = int(sys.argv[1])
npnts = ntot/size
dx = 3.0/ntot
width = 3.0/size
x = rank*width
a = 0.0
for i in xrange(npnts):
  y = 0.7*x**3 - 2*x**2 + 4
  a += y*dx
  x += dx

b = MPI.COMM_WORLD.reduce(a)
if rank == 0:
  print "The area is", b
```

```
import sys
from mpi4py import MPI
from numpy import zeros, asarray
rank = MPI.COMM_WORLD.Get_rank()
size = MPI.COMM_WORLD.Get_size()
ntot = int(sys.argv[1])
npnts = ntot/size
dx = 3.0/ntot
width = 3.0/size
x = rank*width
a = 0.0
for i in xrange(npnts):
  y = 0.7*x**3 - 2*x**2 + 4
  a += y*dx
  x += dx
b = np.zeros(1)
MPI.COMM_WORLD.Reduce(asarray(a),b
if rank == 0:
  print "The area is", b[0]
```

Mpi4py Speedup?

```
$ etime mpirun -np 1 python auc.py 300000000
The area is 8.175000
Elapsed: 15.27 seconds
```

Mpi4py Speedup?

```
$ etime mpirun -np 1 python auc.py 300000000
The area is 8.175000
Elapsed: 15.27 seconds
```

```
$ etime mpirun -np 28 python auc.py 30000000
The area is 8.175000
Elapsed: 3.43 seconds
```

Mpi4py Speedup?

```
$ etime mpirun -np 1 python auc.py 300000000
The area is 8.175000
Elapsed: 15.27 seconds
```

```
$ etime mpirun -np 28 python auc.py 30000000
The area is 8.175000
Elapsed: 3.43 seconds
```

```
$ etime mpirun -np 28 python auc_numpy.py 30000000
The area is 8.175000
Elapsed: 4.74 seconds
```

Mpi4py Speedup?

```
$ etime mpirun -np 1 python auc.py 300000000
The area is 8.175000
Elapsed: 15.27 seconds
```

```
$ etime mpirun -np 28 python auc.py 30000000
The area is 8.175000
Elapsed: 3.43 seconds
```

```
$ etime mpirun -np 28 python auc_numpy.py 30000000
The area is 8.175000
Elapsed: 4.74 seconds
```

Here, there simply isn't enough communication to see the difference between the pickled and non-pickled interface.

Hands-on

1. Use multiprocessing to parallelize the auc.py code.
2. Use numexpr to parallelize the auc.py code.
3. What else could we do to speed up the code?

Map/Reduce variations

Map/reduce

- The diffusion example is, as already admitted, a hard problem to get good performance out of with python.
- That was because it's a tightly coupled problem.
- Other problems aren't, e.g.:
    - Parameter sweeps
    - Reductions
    - Big data
- For such problems, there are some valuable frameworks of the **map/reduce** variety.
- We'll consider two python-enabled map/reduce frameworks:
    - IPython Parallel
    - Apache Spark (in particular `pyspark`)

Common characteristics in map/reduce

- A master process + worker processes
- Master divides or requests work, and collects results
- Overall workflow is data based:
  1. Data is distributed over workers (or already resides there), and workers perform computation on their local data
  2. If reduction: data is moved between workers, and work is done by 'reducers'. This step is iterative.
  3. Result is reported to the master.
- Emphasis on distributing the work and bringing the work to the data. Works well if 'work chuncks' take a good bit of time.

IPython's Parallel Architecture

Ipyparallel

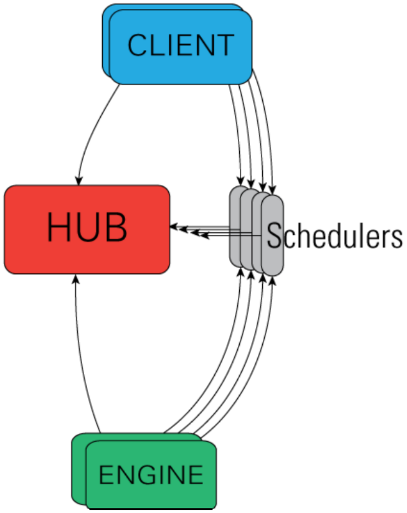
(formerly IPython Parallel)

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:

```
$ hostname
r003.bridges.psc.edu
$ cd ~/hpcpy/code
$ source setup
$ ipcluster start -n 4
a bunch of messages...
```

Do this on the compute node, and remember the hostname. You can minimize this terminal but do not close it.

- Open a second terminal on your laptop and
- Ssh into your compute node.

```
$ ssh -Y bridges.psc.edu
$ ssh -Y r003
```

Using the workers in python

Start python and grab the handles to the clients.

```
$ cd ~/hpcpy/code
$ source setup
$ python
```

```
>>> from ipyparallel import Client
>>> clients = Client()
>>>
```

(older python version? try `from IPython.parallel import Client`)

Accessing the Clients

Let's see what we've got here.

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

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

Accessing the Clients, continued

Here's simple function to execute on the cores:

```
>>> def minus(a, b):
...     return a - b
>>> minus(5, 6)
 -1
>>> #Execution on the first engine only:
>>> clients[0].apply(minus, 5, 6)
 -1
```

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.

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

```
Client().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.

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:

```
>>> clients[:].apply(minus, 5, 6)
[-1, -1, -1, -1]
```

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

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

Examples, continued

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

```
>>> view = clients.load_balanced_view()
>>> view.map(minus, [11, 10, 9, 8], [5, 6, 7, 8])
[6, 4, 2, 0]
```

Direct view

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

```
>>> clients.block = True
>>> dview = clients.direct_view()
>>> dview.block = True
>>> dview.apply(sum, [1, 2, 3])
[6, 6, 6, 6]
```

Direct view, continued

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

```
>>> clients[::2]
<DirectView [0, 2]>
>>> clients[::2].apply(sum, [1, 2, 3])
[6, 6]
```

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:

```
>>> dview = Client().load_balanced_view()
>>> dview.block = True
>>> dview.apply(sum, [1, 2, 3])
6
```

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.

```
>>> v = clients[:]
>>> v.block = True
>>> v.execute('from os import getpid')
<AsyncResult: finished>
>>> v.execute('x = getpid()')
<AsyncResult: finished>
>>> v['x']
[24068, 24067, 24065, 24066]
>>> clients[3]['x']
24066
```

Scatter/Gather

- Some of the parallelization features we saw in the other parallel programming sessions 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:

```
>>> v.scatter('a', np.arange(16))
>>> v['a']
[array([0, 1, 2, 3]),
 array([4, 5, 6, 7]),
 array([8, 9, 10, 11]),
 array([12, 13, 14, 15])]
>>> v.gather('a')
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15])]
```

Spark

Why Spark?

Suppose you find yourself in a situation where your data is ridiculously huge:

- Facebook's daily logs: 60TB.
- 1,000 genomes project: 200TB.
- Google web index: > 10 PB.
- Cost of 1TB disk: \$65.
- Time to read 1TB from disk: 3 hours (100 MB/s).

The scale of these data sets means that we cannot analyse them on a single machine. The analysis must be done in a distributed-memory setting.

In the beginning

A few years ago, if you found yourself in this situation, you would have used Hadoop. Hadoop is essentially a distributed file system, that allows you to take your computation to the data.

But the original Hadoop had some serious issues:

- It was stuck with Map Reduce.
- This meant all data was mapped to a key-value pair and then 'reductions' were performed on this data.
- Each stage of any calculation involved:
  1. Read data from disk.
  2. Perform calculation.
  3. Write result to disk.
  4. Communicate the result.
- Very I/O heavy! And requires a disk available on all nodes.
- Slow! Inefficient!

Spark's approach

Apache Spark does not use I/O so heavily, and has ideal features:

- Spark keeps things in memory instead of disk (though on-disk storage is also supported).
- Map and Reduce are not your only available operations (though this is no longer true for Hadoop 2).
- Many language wrappers available (Scala, Java, R, Python).
- Uses lazy evaluation, which results in improved pipelining.
- Supports batch, interactive and streaming execution models.
- Spark has built-in redundancy; it keeps track of how the data are created, so that if a node fails the data can be rebuilt from scratch (like Hadoop).

Spark's anatomy

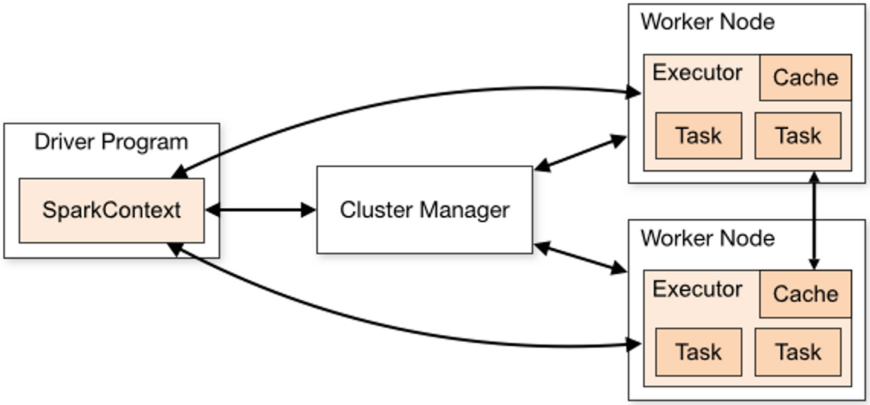
A Spark program consists of two parts:

- Driver program: runs on the driver machine. This is often called the 'master' program.
- Worker programs: run on cluster nodes or in local threads. These are sometimes called "Executors". When RDDs (Resilient Distributed Data sets) are created, they are distributed across the workers.

The first step in a Spark program is to create a SparkContext object.

- This tells Spark how and where to access a cluster.
- This is used to create our RDD.

Spark's anatomy, continued



“Driver program” == “Master”. Notice that only one worker program (executor) runs on each node, but that worker can launch multiple ‘tasks’.

About Spark and pySpark

We will use pySpark to access the running Spark machinery:

- Using Spark 1.6.1, which may still have some issues with Python 3.
- pySpark uses the standard CPython interpreter, so C libraries (like NumPy) can be used.
- Use the “pyspark” command to launch an interactive Python shell.
- The pySpark shell comes with a SparkContext built in, in the variable called “sc”.
- You can also invoke an IPython version of the pySpark shell, which comes with all the usual IPython functionality.

Setting up a Spark session

Note: Bridges has dedicated hadoop/spark nodes, but here, we will simply reuse the interactive node you've already got.

1. Make sure you're on an interactive node and have done `source setup`
2. Setup the Spark session.

```
$ source $SPARK_HOME/scripts/setup_spark.sh
starting org.apache.spark.deploy.master.Master, logging
to /home/rzon/.local/spark/spark-1.6.1-bin-hadoop2.6/logs/spark-rzo
```

This script:

- Detects which nodes are part of your job.
- Starts a master process on the head node.
- Creates `$SCRATCH/temp`, `$SCRATCH/spark-logs`, `$SCRATCH/spark-workers` directories, if necessary.
- Starts a worker process on all of your job's nodes. These are technically Java Virtual Machines.

Setting up a Spark session (cont.)

1. Move to the spark directory.

```
$ cd $HOME/hpcpy/spark
```

2. Launch pySpark with IPython support.

```
$ IPYTHON=1 pyspark
...
SparkContext available as sc.
In [1]:
```

We now have a Spark session running, which we access using pySpark.

Confirming your cluster

To confirm that your cluster is working:

```
>>>
>>> from socket import gethostname
>>>
>>> sc.parallelize(range(5)).map(lambda x:gethostname()).collect()
['r024.pvt.bridges.psc.edu', 'r024.pvt.bridges.psc.edu',
 'r024.pvt.bridges.psc.edu','r024.pvt.bridges.psc.edu',
 'r024.pvt.bridges.psc.edu']
>>>
```

Creating RDDs

Spark is all about RDDs (Resilient Distributed Data sets), which are what we use to hold our data:

- RDDs are objects; all operations on them consist of one of many existing functions.
- RDDs are immutable: they cannot be changed once created.
- To modify data you create new RDDs based on existing RDDs.
- Remember that RDDs are lazily evaluated, meaning not calculated until an ‘action’ is performed.
- RDDs can be ‘cached’ so that the results persist.
- RDDs contain data of two general types, regular (which are single values of any type), and (key, value).

RDD Pipelines

The general procedure for data analysis is a pipeline:

- create an RDD from a data source, file or list.
- apply various transformations to the RDD.
- apply an action to the RDD.

Creating RDDs, continued

So how do I create an RDD?
There are a few options in pySpark:

- parallelize(x): create an RDD out of the data (list or Numpy vector) ‘x’.
- textFile(file): read a text file, and return each line as the data. Will also read all the files in a directory, if a path is given.
- There are also methods for reading in other types of files.

```
In [3]: myverbs = ['run', 'jump',
 'sit', 'laugh', 'run', 'smile']
In [4]: verbRDD =
  sc.parallize(myverbs)
In [5]:
In [5]: myRDD =
  sc.textFile('myfile.txt')
In [6]:
```

Note that the file mentioned above doesn’t (presumably) exist. Why isn’t an error message thrown?

Manipulating RDDs: transformations

Ok, I've got my RDD. Now what? Well, the things you can do fall into two categories. The first is transformations:

- map(func): map each value of the RDD to a function; return a new RDD that contains the return of the function on each RDD value.
- filter(func): return a new dataset formed by applying 'func' to each element, and keeping those which return True.
- flatMap(func): like map, but returns all elements in a single list.
- distinct(): reduce the RDD data points to distinct values only.
- groupByKey(): group key values as lists, return as (key, list) pairs.
- reduceByKey(func): reduce the elements key-by-key, applying func to the elements.
- sortBy(func)/sortByKey(func): sort the RDD, using func.

Transformations return an RDD, with the elements appropriately 'transformed.'

Manipulating RDDs: actions

The second category of RDD functions are actions:

- reduce(func): reduce the elements back to the master process, by applying func to the elements.
- count(): counts the elements.
- collect(): bring all the elements back to the master.
- take(n): bring in ‘n’ elements to the master.
- takeOrdered(n, func): same as take, but re-order based on func.
  max(), min(), mean(), stdev(), sum()

Actions return a value, or list. Read the API if you’re not sure of the name of the function you need.

Manipulating RDDs, continued

```
In [6]: def pastTense(s):
...:        return s + 'ed'
...:
In [7]:
In [7]: myverbs = ['run', 'jump', 'sit', 'laugh', 'run', 'smile']
In [8]:
In [8]: verbRDD = sc.parallelize(myverbs)
In [9]:
In [9]: pastTenseRDD = verbRDD.map(pastTense)
In [10]:
In [10]: pastTenseRDD.collect()
Out[10]: ['runed', 'jumped', 'sited', 'laughed', 'runed', 'smileed'
In [11]:
In [11]: verbRDD.take(1)
Out[11]: ['run']
In [12]:
```

Note that nothing is actually calculated until the 'collect' is called.

Manipulating RDDs: partitioning

By default your RDD is sliced up into 'partitions' and spread across the workers.

- The number of partitions can greatly impact computational efficiency. When individual functions are applied to the data, one task is performed per partition.
- For load balancing you will want at least as many partitions as cores, possibly more.
- Spark documentation suggests 2-4 partitions per CPU. You will often see problems broken up into hundreds or thousands of partitions.
- Be aware that increasing the number of partitions before the data is read in can cause I/O issues. It's better to repartition after the data is read.

Manipulating RDDs: partitioning, cont.

There are built-in commands to adjust your partitions on the fly:

- You can get the system default minimum number of partitions using "sc.defaultMinPartitions".
- You can set the number of partitions as an optional argument when you create your RDD:
  - "numSlices": for parallelize.
  - "minPartitions": for textFile.
- You can get the number of partitions for a given RDD using RDD.getNumPartitions().
- You can change the RDD's number of partitions using "RDD.repartition(num)". This can be an expensive operation, as the data is randomly shuffled.
- "RDD.coalesce(num)" also reduces the number of partitions, without shuffling.

Manipulating RDDs: partitioning, cont.

```
In [12]:
In [12]: sc.defaultMinPartitions
Out[12]: 2
In [13]:
In [13]: myverbs = ['run', 'jump', 'sit', 'laugh', 'run', 'smile']
In [14]: verbRDD = sc.parallelize(myverbs, numSlices = 48)
In [15]: verbRDD.getNumPartitions()
Out[15]: 48
In [16]:
In [16]: myRDD=sc.textFile('data/stopwords.txt',minPartitions=100)
In [17]: myRDD.getNumPartitions()
Out[17]: 101
In [18]:
In [18]: newRDD = myRDD.repartition(200)
In [19]: newRDD.getNumPartitions()
Out[19]: 200
In [20]:
```

Python’s lambda function

Python contains a function called ‘lambda’. Lambda functions

- are small, anonymous functions (not bound to a name).
- have the format “lambda arguments: operation on arguments”.
- are restricted to a single expression.
- example: `lambda a, b:  a - b`.

Who cares? Well, if you combine lambda with the RDD ‘map’ command, you can do some pretty impressive things.

Using lambda

```
In [20]:
In [20]: range(10)
Out[20]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [21]:
In [21]: myRDD = sc.parallelize(range(10))
In [22]:
In [22]: myRDD.map(lambda x: 2 * x)
Out[22]: PythonRDD[1] at RDD at PythonRDD.scala:37
In [23]:
In [23]: myRDD.map(lambda x: 2 * x).collect()
Out[23]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
In [24]:
In [24]: myRDD.filter(lambda x: x % 3 == 0).collect()
Out[24]: [0, 3, 6, 9]
In [25]:
```

Creating (key, value) pairs

The lambda function can be used to create compound variables, as well as to create (key, value) pairs.

```
In [25]: myRDD = sc.parallelize([3, 4, 5])
In [26]:
In [26]: myRDD.map(lambda x: [x, x - 2]).collect()
Out[26]: [[3, 1], [4, 2], [5, 3]]
In [27]:
In [27]: myRDD.flatMap(lambda x: [x, x - 2]).collect()
Out[27]: [3, 1, 4, 2, 5, 3]
In [28]:
In [28]: # create (key, value) pairs
In [28]: myRDD.map(lambda x: (x, 1)).collect()
Out[28]: [(3, 1), (4, 1), (5, 1)]
In [29]:
```

Note that once the data has been scattered to the workers, all operations occur locally, at the workers, until the 'collect' command.

(key, value) example

What is the frequency of words in “Moby Dick”? First prep the data:

```
In [29]: MD = sc.textFile("data/mobydick.txt")
In [30]:
In [30]: # the first 5 lines
In [30]: MD.take(5)
Out[30]: [u’MOBY DICK;’, u’’, u’or, THE WHALE’, u’’, u’’]
In [31]:
In [31]: # We need to clean this data up.
In [31]: import string
In [32]:
In [32]: # convert from unicode to string.
In [32]: strMD = MD.map(lambda x: x.encode(’ascii’, ’ignore’))
In [33]:
In [33]: # Remove the punctuation.
In [33]: MDNoPu=strMD.map(
    ...:    lambda x:x.translate(None,string.punctuation))
In [34]:
In [34]: MDNoPu.take(5)
Out[34]: [’MOBY DICK’, ’’, ’or THE WHALE’, ’’, ’’]
```

(key, value) example, data cleaning

```
In [35]: MDNoPu.take(5)
Out[35]: ['MOBY DICK', '', 'or THE WHALE', '', '']
In [36]:
In [36]: # We want the words, not the whole lines, so split.
In [36]: MDSplit = MDNoPu.flatMap(lambda x: x.split(' '))
In [37]:
In [37]: MDSplit.take(5)
Out[37]: ['MOBY', 'DICK', '', 'or', 'THE']
In [38]:
In [38]: # Make all the words lower case, and remove empty strings.
In [38]: MDlower = MDSPlit.map(
    ...:   lambda x: x.lower()).filter(lambda x: x != '')
In [39]:
In [39]: MDlower.take(5)
Out[39]: ['moby', 'dick', 'or', 'the', 'whale']
In [40]:
```

(key, value) example, data cleaning, cont.

```
In [40]:
In [40]: MDlower.take(5)
Out[40]: ['moby', 'dick', 'or', 'the', 'whale']
In [41]:
In [41]: # stopwords are words which are considered 'uninteresting'
In [41]: stopwords = open('data/stopwords.txt',
    ...:                  'r').read().split('\n')
In [42]:
In [42]: stopwords[0:5]
Out[42]: ['a', 'about', 'above', 'across', 'after']
In [43]:
In [43]: MDReady = MDlower.filter(lambda x: x not in stopwords)
In [44]:
In [44]: MDReady.take(5)
Out[44]: ['moby', 'dick', 'whale', 'herman', 'melville']
In [45]:
```

Ready to start counting.

(key, value) example, counting occurences

Using the (key, value) approach, 1) count the number of times each word appears and 2) give the top 5 words, in terms of frequency.

```
In [45]: kvMD = MDReady.map(lambda word: (word, 1))
In [46]:
In [46]: result = kvMD.reduceByKey(lambda x, y: x + y)
In [47]:
In [47]: result.take(5)
Out[47]:
[('knockers', 1),
('brevet', 1),
('yellow', 19),
('prefix', 1),
('looking', 56)]
In [48]:
In [48]: result.takeOrdered(5, lambda (k, v): -v)
Out[48]: [('whale', 892), ('like', 567), ('old', 436), ('man', 433)
('ahab', 417)]
```

canada canada

Useful web sites for Spark

There is much more too spark, obviously.

One of the nice things is that it comes with a machine learning framework called `mllib`.

There are a number of useful web sites out there to find more information. Here are some of them:

- spark.apache.org/docs/latest/api/python/pyspark.html
- spark.apache.org/docs/latest/programming-guide.html
- spark.apache.org/docs/latest/api/python/pyspark.mllib.html
- spark.apache.org/docs/latest/api/python/pyspark.sql.html