HPC Python Programming

Ramses van Zon

SciNet HPC Consortium, Toronto

IHPCSS, June 2017

SciNet compute | calcul canada

In this session...

1. Performance and Python
2. Profiling tools for Python
3. Fast arrays for Python: Numpy
4. Multicore computations:
   - **Numexpr**
   - **Threading**
   - **Multiprocessing**
   - **Mpi4py**

SciNet compute | calcul canada

Getting started

SciNet compute canada | calcul canada

Packages and code

Requirements for this session

If following along on your own laptop, you need the following packages:

- numpy
- scipy
- numexpr
- matplotlib
- psutil
- line_profiler
- memory_profiler
- theano
- mpi4py
- cython

Get the code and setup files on Bridges

Code and installation can be copied from a Bridges. It can be found in the directory `/home/rzon/hpcpy17`.

SciNet compute canada | calcul canada

Setting up for today's class (Bridges)

To get set up for today's session, perform the following steps.

1. Login to Bridges

```
$ ssh -Y -p 2222 USERNAME@bridges.psc.edu
```

2. Install code and software to your own directory

```
$ cd ~
$ cp -r /home/rzon/hpcpy17 .
$ cd ~/hpcpy17
$ source setup  # very important!
```

The last command will install a few packages into your local account, so as to satisfy the requirements, and will load the correct modules.

3. Request an interactive session on a compute node

```
$ interact -p RM-shared -t 4:00:00 -N 1 --ntasks-per-node=14
```

Introduction

Performance and Python

- Python is a high-level, interpreted language.
- Those defining features are often at odds with "high performance".
- But the development in Python can be substantially easier (and thus faster) than compiled languages.
- In this session, we will explore when using Python still makes sense and how to get the most performance out of it, without loosing the flexibility and ease of development.

SciNet compute | calcul canada

What would make Python not “high performance”?

Interpreted language:

- Translation to machine language happens line-by-line as the script is read.
- Repeated lines are no faster.
- Cross-line optimizations are not possible.

SciNet compute | calcul canada

What would make Python not "high performance"?

Interpreted language:

- Translation to machine language happens line-by-line as the script is read.
- Repeated lines are no faster.
- Cross-line optimizations are not possible.

Dynamic language:

- Types are part of the data: extra overhead
- Memory management is automatic. Behind the scene that means reference counting and garbage collection.
- All this also interfers with optimal streaming of data to processor, which interfers with maximum performance.

Example: 2D diffusion equation

Suppose we are interested in the time evolution of the two-dimension diffusion equation:

$$\frac{\partial p(x,y,t)}{\partial t} = D\left(\frac{\partial^2 p(x,y,t)}{\partial x^2} + \frac{\partial^2 p(x,y,t)}{\partial y^2}\right),$$

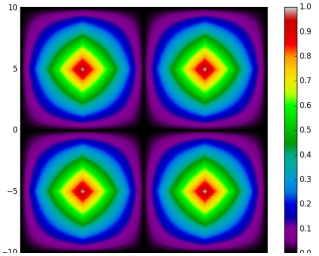
on domain $[x_1, x_2] \otimes [x_1, x_2]$,
with $P(x,y,t) = 0$ at all times for all points on the domain boundary, and for some given initial condition $p(x,y,t) = p_0(x,y)$.

Here:

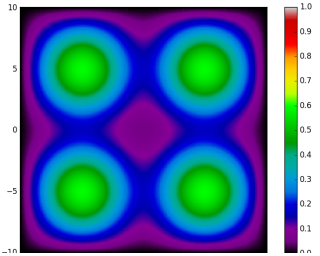
- P: density
- x, y: spatial coordinates
- t: time
- D: diffusion constant

Example: 2D diffusion, result

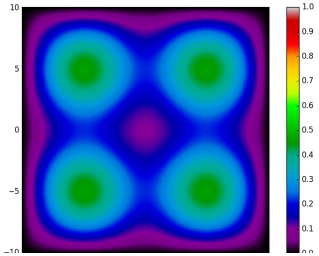
$x_1 = -10$, $x_2 = 10$, $D = 1$, four-peak initial condition.



t=4



t=6

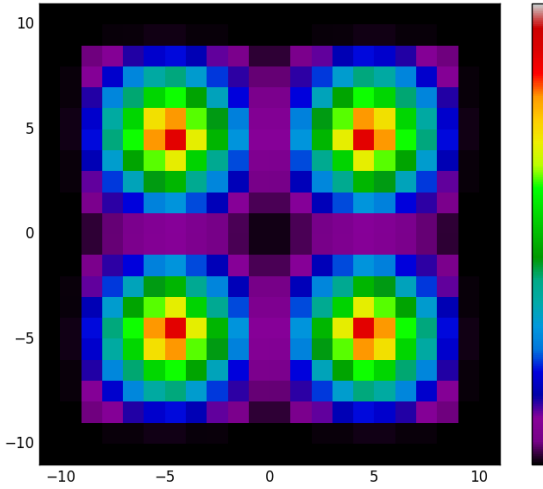


t=10

Example: 2D diffusion, algorithm

- Discretize space in both directions (points dx apart)
- Replace derivatives with finite differences.
- Explicit finite time stepping scheme (time step set by dx)
- For graphics: Matplotlib for python, pgplot for c++/fortran, every `outtime` time units

Parameters in file diff2dparams.py



Example: 2D diffusion, parameters

The fortran, C++ and python codes all read the same files (by some special tricks).

diff2dparams.py

```
D        =   1.0;
x1       = -10.0;
x2       =  10.0;
runtime  =  15.0;
dx       =  0.0667;
outtime  =  0.5;
graphics =  False;
```

Example: 2D diffusion, performance

The files `diff2d.cpp`, `diff2.f90` and `diff2d.py` contain the same algorithm, in C++, Fortran, and Python, respectively.

SciNet compute | calcul canada

Example: 2D diffusion, performance

The files `diff2d.cpp`, `diff2.f90` and `diff2d.py` contain the same algorithm, in C++, Fortran, and Python, respectively.

```
$ etime() { /usr/bin/time -f "Elapsed: %e seconds" $@; }
$ etime make diff2d_cpp.ex diff2d_f90.ex
g++ -c -std=c++11 -O3 -o diff2d_cpp.o diff2d.cpp
gfortran -c -O3 -o pgplot90.o pgplot90.f90
...
Elapsed: 1.80 seconds
```

SciNet compute | calcul canada

Example: 2D diffusion, performance

The files `diff2d.cpp`, `diff2.f90` and `diff2d.py` contain the same algorithm, in C++, Fortran, and Python, respectively.

```
$ etime() { /usr/bin/time -f "Elapsed: %e seconds" $@; }
$ etime make diff2d_cpp.ex diff2d_f90.ex
g++ -c -std=c++11 -O3 -o diff2d_cpp.o diff2d.cpp
gfortran -c -O3 -o pgplot90.o pgplot90.f90
...
Elapsed: 1.80 seconds
```

```
$ etime ./diff2d_cpp.ex > output_c.txt
Elapsed: 0.73 seconds
$ etime ./diff2d_f90.ex > output_f.txt
Elapsed: 0.55 seconds
$ etime python diff2d.py > output_n.txt
Elapsed: 132.79 seconds
```

This doesn't look too promising for Python for HPC...

Then why do we bother with Python?

Then why do we bother with Python?

```
#diff2d.py
from diff2dplot import plotdens
from diff2dparams import D,x1,x2,runtime,dx,outtime,
nrows  = int((x2-x1)/d
ncols  = nrows
npnts  = nrows + 2
dx     = (x2-x1)/nrows
dt     = 0.25*dx**2/D
nsteps = int(runtime/dt)
nper   = int(outtime/dt)
if nper==0: nper = 1
x=[x1+((i-1)*(x2-x1))/nrows for i in xrange(npnts)]
dens     = [[0.0]*npnts for i in xrange(npnts)]
densnext = [[0.0]*npnts for i in xrange(npnts)]
simtime  = 0*dt
for i in range(1,npnts-1):
 a = 1 - abs(1 - 4*abs((x[i]-(x1+x2)/2)/(x2-x1)))
 for j in range(1,npnts-1):
  b = 1 - abs(1 - 4*abs((x[j]-(x1+x2)/2)/(x2-x1)))
  dens[i][j] = a*b
print simtime
if graphics: plotdens(dens,x[0],x[-1],first=True)
lapl = [[0.0]*npnts for i in xrange(npnts)]
```

```
for s in range(nsteps):
 for i in range(1,nrows+1):
  for j in range(1,ncols+1):
   lapl[i][j] = (dens[i+1][j]+dens[i-1][j]
                 +dens[i][j+1]+dens[i][j-1]
                 -4*dens[i][j])
 for i in range(1,nrows+1):
  for j in range(1,ncols+1):
   densnext[i][j]=dens[i][j]+(D/dx**2)*dt*lapl[i][j]
 dens, densnext = densnext, dens
 simtime += dt
 if (s+1)%nper == 0:
  print simtime
  if graphics: plotdens(dens,x[0],x[-1])
```

SciNet compute | calcul canada

Then why do we bother with Python?

```
#diff2d.py
from diff2dplot import plotdens
from diff2dparams import D,x1,x2,runtime,dx,outtime,
nrows  = int((x2-x1)/d
ncols  = nrows
npnts  = nrows + 2
dx     = (x2-x1)/nrows
dt     = 0.25*dx**2/D
nsteps = int(runtime/dt)
nper   = int(outtime/dt)
if nper==0: nper = 1
x=[x1+((i-1)*(x2-x1))/nrows for i in xrange(npnts)]
dens     = [[0.0]*npnts for i in xrange(npnts)]
densnext = [[0.0]*npnts for i in xrange(npnts)]
simtime  = 0*dt
for i in range(1,npnts-1):
 a = 1 - abs(1 - 4*abs((x[i]-(x1+x2)/2)/(x2-x1)))
 for j in range(1,npnts-1):
  b = 1 - abs(1 - 4*abs((x[j]-(x1+x2)/2)/(x2-x1)))
  dens[i][j] = a*b
print simtime
if graphics: plotdens(dens,x[0],x[-1],first=True)
lapl = [[0.0]*npnts for i in xrange(npnts)]
```

```
for s in range(nsteps):
 for i in range(1,nrows+1):
  for j in range(1,ncols+1):
   lapl[i][j] = (dens[i+1][j]+dens[i-1][j]
                 +dens[i][j+1]+dens[i][j-1]
                 -4*dens[i][j])
 for i in range(1,nrows+1):
  for j in range(1,ncols+1):
   densnext[i][j]=dens[i][j]+(D/dx**2)*dt*lapl[i][j]
 dens, densnext = densnext, dens
 simtime += dt
 if (s+1)%nper == 0:
  print simtime
# diff2dplot.py
def plotdens(dens,x1,x2,first=False):
 import os
 import matplotlib.pyplot as plt
 if first:
  plt.clf(); plt.ion()
 plt.imshow(dens,interpolation='none',aspect='equal'
 if first:
  plt.colorbar()
 plt.show();plt.pause(0.1)
```

Then why do we bother with Python?

- Python lends itself easily to writing clear, concise code.
  (2d diffusion fits almost on one slide!)
- Python is very flexible: large set of very useful packages.
- Easy of use $\rightarrow$ shorter development time
- Python's performance hit is most prominant on 'tightly coupled' calculation on fundamental data types that are known to the cpu (integers, doubles), which is exactly the case for the 2d diffusion.
- It does much less worse on file I/O, text comparisons, list manipularions etc.
- Hooks to compiled libraries to remove worst performance pitfalls.
- Once the performance isn't too bad, we can start thinking of parallelization, i.e., using more cpu cores working on the
  same problem.

SciNet compute | calcul canada

Performance tuning tools for Python

CPU performance

- Performance is about maximizing the utility of a resource.
- This could be cpu processing power, memory, network, file I/O, etc.
- Let's focus on **cpu performance** first.

CPU Profiling by function

- To consider the cpu performance of functions, but not of individual lines in your code, there is the package called `cProfile`.

CPU Profiling by line

- To find cpu performance bottlenecks by line of code, there is package called `line_profiler`

cProfile

- Use `cProfile` or `profile` to know in which functions your script spends its time.
- You usually do this on a smaller but representative case.
- The code should be reasonably modular, i.e., with separate functions for different tasks, for cProfile to be useful.

Example

```
$ python -m cProfile -s cumulative diff2d.py
...
         2492205 function calls in 521.392 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.028    0.028  521.392  521.392 diff2d.py:11(<module>)
        1  515.923  515.923  521.364  521.364 diff2d.py:14(main)
  2411800    5.429    0.000    5.429    0.000 {range}
    80400    0.012    0.000    0.012    0.000 {abs}
        1    0.000    0.000    0.000    0.000 diff2dplot.py:5(<module>)
        1    0.000    0.000    0.000    0.000 diff2dparams.py:1(<module>)
```

line_profiler

- Use `line_profiler` to know, line-by-line, where your script spends its time.
- You usually do this on a smaller but representative case.
- First thing to do is to have your code be in a function.
- You also need to include modify your script slightly:
  - Decorate your function with @profile
  - Run your script on the command line with

```
$ kernprof -l -v SCRIPTNAME
```

line_profiler script instrumentation

Script before:

```
x=[1.0]*(2048*2048)
a=str(x[0])
a+="\nis a one\n"
del x
print(a)
```

SciNet compute | calcul canada

line_profiler script instrumentation

Script before:

```
x=[1.0]*(2048*2048)
a=str(x[0])
a+="\nis a one\n"
del x
print(a)
```

Script after:

```
#file: profileme.py
@profile
def profilewrapper():
  x=[1.0]*(2048*2048)
  a=str(x[0])
  a+="\nis a one\n"
  del x
  print(a)
profilewrapper()
```

Run at the command line:

```
$ kernprof -l -v profileme.py
```

SciNet compute | calcul canada

Output of line_profiler

```
1.0
is a one

Wrote profile results to profileme.py.lprof
Timer unit: 1e-06 s

Total time: 0.038648 s
File: profileme.py
Function: profilewrapper at line 2

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     2                                           @profile
     3                                           def profilewrapper():
     4         1        28769  28769.0     74.4    x=[1.0]*(2048*2048)
     5         1           22     22.0      0.1    a=str(x[0])
     6         1            2      2.0      0.0    a+="\nis a one\n"
     7         1         9813   9813.0     25.4    del x
     8         1           42     42.0      0.1    print(a)
```

Memory performance

Why worry about this?

Memory performance

Why worry about this?
Once your script runs out of memory, one of a number of things may happen:

- Computer may start using the harddrive as memory: **very slow**
- Your application crashes
- Your (compute) node crashes

SciNet compute canada | calcul canada

Memory performance

Why worry about this?
Once your script runs out of memory, one of a number of things may happen:

- Computer may start using the harddrive as memory: **very slow**
- Your application crashes
- Your (compute) node crashes

How could you run out of memory?

- You're not quite sure how much memory you program takes.
- Python objects may take more memory that expected.
- Some functions may temporarily use extra memory.
- Python relies on a garbage collector to clean up unused variables.

SciNet compute canada | calcul canada

Garbage collector

- Python uses garbage collector to clean up un-needed variables
- You can force the garbage collection to run at any time by running:

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

- Running gc by hand should only be done in specific circumstances.
- You can also remove objects with del (if object larger than 32MB):

```
>>> x = [0,0,0,0]
>>> del x
>>> print (x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

But how would you know when the memory usage is problematic?

SciNet compute canada | calcul canada

memory_profiler

- This module/utility monitors the python memory usage and its changes throughout the run.
- Good for catching memory leaks and unexpectedly large memory usage.
- Needs same instrumentation as line profiler.
- Requires the psutil module (at least on windows, but helps on linux/mac too).

SciNet compute | calcul canada

memory_profiler, details

Your decorated script is usable by memory profiler.
You run your script through the profiler with the command

```
$ python -m memory_profiler profileme.py
```

memory_profiler, details

Your decorated script is usable by memory profiler.
You run your script through the profiler with the command

```
$ python -m memory_profiler profileme.py
```

```
1.0
is a one

Filename: profileme.py

Line #    Mem usage    Increment   Line Contents
================================================
     2   29.094 MiB    0.000 MiB   @profile
     3                             def profilewrapper():
     4   61.102 MiB   32.008 MiB     x=[1.0]*(2048*2048)
     5   61.105 MiB    0.004 MiB     a=str(x[0])
     6   61.105 MiB    0.000 MiB     a+="\nis a one\n"
     7   29.102 MiB  -32.004 MiB     del x
     8   29.105 MiB    0.004 MiB     print(a)
```

Hands-on

Profile the diff2d.py code

- Reduce the resolution in diff2dparams.py, i.e., increase `dx` to 0.1.
- In the same file, set `graphics=False`.
- Add @profile to the main function
- Run this through both the line and memory profilers.
  - What line(s) cause the most memory usage?
  - What line(s) cause the most cpu usage?

SciNet compute | calcul canada

Numpy: faster numerical arrays for python

SciNet compute canada | calcul canada

Lists aren't the ideal data type

Lists can do funny things that you don't expect, if you're not careful.

- Lists are just a collection of items, of any type.
- If you do mathematical operations on a list, you won't get what you expect.
- These are not the ideal data type for scientific computing.
- Arrays are a much better choice, but are not a native Python data type.

```
>>> a = [1,2,3,4]
>>> a
[1, 2, 3, 4]
>>> b = [3,5,5,6]
>>> b
[3, 5, 5, 6]
>>> 2*a
[1, 2, 3, 4, 1, 2, 3, 4]
>>> a+b
[1, 2, 3, 4, 3, 5, 5, 6]
```

SciNet compute canada | calcul canada

Useful arrays: NumPy

- Almost everything that you want to do starts with NumPy.
- Contains arrays of various types and forms: zeros, ones, linspace, etc.

```
>>> from numpy import zeros, ones
>>> zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])
>>> ones(5, dtype=int)
array([1, 1, 1, 1, 1])
>>> zeros([2,2])
array([[ 0.,  0.],
       [ 0.,  0.]])
```

```
>>> from numpy import arange
>>> from numpy import linspace
>>> arange(5)
array([0, 1, 2, 3, 4])
>>> linspace(1,5)
array([ 1.        ,  1.08163265,  1.16326531,  1.2448
        1.40816327,  1.48979592,  1.57142857,  1.653
        1.81632653,  1.89795918,  1.97959184,  2.061
        2.2244898 ,  2.30612245,  2.3877551 ,  2.469
        2.63265306,  2.71428571,  2.79591837,  2.877
        3.04081633,  3.12244898,  3.20408163,  3.285
        3.44897959,  3.53061224,  3.6122449 ,  3.693
        3.85714286,  3.93877551,  4.02040816,  4.102
        4.26530612,  4.34693878,  4.42857143,  4.510
        4.67346939,  4.75510204,  4.83673469,  4.918
>>> linspace(1,5,6)
array([ 1. ,  1.8,  2.6,  3.4,  4.2,  5. ])
```

compute canada | calcul canada

Accessing array elements

Elements of arrays are accessed using square brackets.

- Python is *row-major* (like C++, Mathematica), NOT *column major* (like Fortran, MATLAB, R)
- This means the first index is the row, not the column.
- Indexing starts at zero.

SciNet compute canada | calcul canada

Accessing array elements

Elements of arrays are accessed using square brackets.

- Python is *row-major* (like C++, Mathematica), NOT *column major* (like Fortran, MATLAB, R)
- This means the first index is the row, not the column.
- Indexing starts at zero.

```
>>> from numpy import *
>>> zeros([2,3])
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> a = zeros([2,3])
>>> a[1,2] = 1
>>> a[0,1] = 2
>>> a
array([[ 0.,  2.,  0.],
       [ 0.,  0.,  1.]])
```

SciNet compute | calcul canada

Accessing array elements

Elements of arrays are accessed using square brackets.

- Python is *row-major* (like C++, Mathematica), NOT *column major* (like Fortran, MATLAB, R)
- This means the first index is the row, not the column.
- Indexing starts at zero.

```
>>> from numpy import *
>>> zeros([2,3])
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> a = zeros([2,3])
>>> a[1,2] = 1
>>> a[0,1] = 2
>>> a
array([[ 0.,  2.,  0.],
       [ 0.,  0.,  1.]])
```

```
>>> a[2,1] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 2 is out of bounds for axis 0 with
```

Copying array variables

Use caution when copying array variables. There's a 'feature' here that is unexpected.

```
>>> a = 10; b = a; a = 20
>>> a, b
(20, 10)
```

```
>>> import numpy as np
>>> a = np.array([[1,2,3],
...               [2,3,4]])
>>> b = a
>>> a[1,0] = -10
>>> a
array([[  1,   2,   3],
       [-10,   3,   4]])
>>> b
array([[  1,   2,   3],
       [-10,   3,   4]])
```

```
>>> import numpy as np
>>> a = np.array([[1,2,3],
...               [2,3,4]])
>>> b = a.copy()
>>> a[1,0] = 16
>>> a
array([[ 1,  2,  3],
       [16,  3,  4]])
>>> b
array([[1, 2, 3],
       [2, 3, 4]])
```

Matrix arithmetic

vector-vector & vector-scalar multiplication

1-D arrays are often called 'vectors'.

- When vectors are multiplied you get element-by-element multiplication.
- When vectors are multiplied by a scalar (a 0-D array), you also get element-by-element multiplication.

```
>>> import numpy as np
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> b = np.arange(4.) + 3
>>> b
array([ 3.,  4.,  5.,  6.])
>>> c = 2
>>> c
2
>>> a * b
array([  0.,   4.,  10.,  18.])
>>> a * c
array([0, 2, 4, 6])
>>> b * c
array([  6.,   8.,  10.,  12.])
```

Matrix-vector multiplication

A 2-D array is sometimes called a 'matrix'.

- Matrix-scalar multiplication gives element-by-element multiplication.
- With numpy, matrix-vector multiplication DOES NOT give the standard result!

```
>>> import numpy as np
>>> a = np.array([[1,2,3],
...               [2,3,4]])
>>> a
array([[1, 2, 3],
       [2, 3, 4]])
>>> b = np.arange(3) + 1
>>> b
array([1, 2, 3])
>>> a * b
array([[ 1,  4,  9],
       [ 2,  6, 12]])
```

Numpy DOES NOT compute this:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} * \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_{11} * b_1 + a_{12} * b_2 + a_{13} * b_3 \\ a_{21} * b_1 + a_{22} * b_2 + a_{23} * b_3 \end{bmatrix}$$

SciNet compute | calcul canada

Matrix-vector multiplication

A 2-D array is sometimes called a 'matrix'.

- Matrix-scalar multiplication gives element-by-element multiplication.
- With numpy, matrix-vector multiplication DOES NOT give the standard result!

```
>>> import numpy as np
>>> a = np.array([[1,2,3],
...               [2,3,4]])
>>> a
array([[1, 2, 3],
       [2, 3, 4]])
>>> b = np.arange(3) + 1
>>> b
array([1, 2, 3])
>>> a * b
array([[ 1,  4,  9],
       [ 2,  6, 12]])
```

Numpy DOES compute this:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} * \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_{11} * b_1 & a_{12} * b_2 & a_{13} * b_3 \\ a_{21} * b_1 & a_{22} * b_2 & a_{23} * b_3 \end{bmatrix}$$

SciNet compute canada | calcul canada

Matrix-matrix multiplication

Not surprisingly, matrix-matrix multiplication doesn't work as expected either, instead doing the same thing as vector-vector multiplication.

```
>>> import numpy as np
>>> a = np.array([[1,2],
...               [4,3]])
>>> b = np.array([[1,2],
...               [4,3]])
>>> a
array([[1, 2],
       [4, 3]])
>>> a * b
array([[ 1,  4],
       [16,  9]])
```

Numpy DOES NOT do this:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} =$$

$$\begin{bmatrix} a_{11} * b_{11} + a_{12} * b_{21} & a_{11} * b_{12} + a_{12} * b_{22} \\ a_{21} * b_{11} + a_{22} * b_{21} & a_{21} * b_{12} + a_{22} * b_{22} \end{bmatrix}$$

SciNet compute | calcul canada

Matrix-matrix multiplication

Not surprisingly, matrix-matrix multiplication doesn't work as expected either, instead doing the same thing as vector-vector multiplication.

```
>>> import numpy as np
>>> a = np.array([[1,2],
...               [4,3]])
>>> b = np.array([[1,2],
...               [4,3]])
>>> a
array([[1, 2],
       [4, 3]])
>>> a * b
array([[ 1,  4],
       [16,  9]])
```

Numpy DOES do this:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} =$$

$$\begin{bmatrix} a_{11} * b_{11} & a_{12} * b_{12} \\ a_{21} * b_{21} & a_{22} * b_{22} \end{bmatrix}$$

How to fix the matrix algebra?

There are two solutions to these matrix multiplication problems.

- The specially built-in array fixes (using 'array' types).
- The matrix module (using 'matrix' types).

The latter option is a bit clunkier, so we recommend the 'fixes'.

```
>>> import numpy as np
>>> a = np.array([[1,2],
...               [4,3]])
>>> b = np.array([[1,2],
...               [4,3]])
>>> a
array([[1, 2],
       [4, 3]])
```

```
>>> a.transpose()
array([[1, 4],
       [2, 3]])
>>> np.dot(a.transpose(), b)
array([[17, 14],
       [14, 13]])
>>> np.dot(b, a.transpose())
array([[ 5, 10],
       [10, 25]])
>>> c = np.arange(2) + 1
>>> np.dot(a,c)
array([5, 10])
```

SciNet compute canada | calcul canada

Does changing to numpy really help?

Let's return to our 2D diffusion example.
Pure python implementation:

```
$ etime python diff2d.py > output_p.txt
Elapsed: 131.87 seconds
```

Does changing to numpy really help?

Let's return to our 2D diffusion example.
Pure python implementation:

```
$ etime python diff2d.py > output_p.txt
Elapsed: 131.87 seconds
```

Numpy implementation:

```
$ etime python diff2d_slow_numpy.py > output_n.txt
Elapsed: 439.40 seconds
```

SciNet compute | calcul canada

Does changing to numpy really help?

Let's return to our 2D diffusion example.
Pure python implementation:

```
$ etime python diff2d.py > output_p.txt
Elapsed: 131.87 seconds
```

Numpy implementation:

```
$ etime python diff2d_slow_numpy.py > output_n.txt
Elapsed: 439.40 seconds
```

Hmm, not really, what gives?

SciNet compute | calcul canada

Python overhead

- Python's overhead comes mainly from it's interpreted nature.
- The `diff2d_slow_numpy.py` code uses numpy arrays, but still has a loop over indiced.
- Numpy will not give much speedup until you use it 'vector' operations.

Python overhead

- Python's overhead comes mainly from it's interpreted nature.
- The `diff2d_slow_numpy.py` code uses numpy arrays, but still has a loop over indiced.
- Numpy will not give much speedup until you use it 'vector' operations.

E.g., instead of

```
a = np.linspace(0.0,1.0,101)
b = np.linspace(1.0,2.0,101)
c = np.ndarray(100)
for i in xrange(100):
  c[i] = a[i] + b[i+1]
```

SciNet compute | calcul canada

Python overhead

- Python's overhead comes mainly from it's interpreted nature.
- The `diff2d_slow_numpy.py` code uses numpy arrays, but still has a loop over indiced.
- Numpy will not give much speedup until you use it 'vector' operations.

E.g., instead of

```
a = np.linspace(0.0,1.0,101)
b = np.linspace(1.0,2.0,101)
c = np.ndarray(100)
for i in xrange(100):
  c[i] = a[i] + b[i+1]
```

You would write:

Python overhead

- Python's overhead comes mainly from it's interpreted nature.
- The `diff2d_slow_numpy.py` code uses numpy arrays, but still has a loop over indiced.
- Numpy will not give much speedup until you use it 'vector' operations.

E.g., instead of

```
a = np.linspace(0.0,1.0,101)
b = np.linspace(1.0,2.0,101)
c = np.ndarray(100)
for i in xrange(100):
  c[i] = a[i] + b[i+1]
```

You would write:

```
a = np.linspace(0.0,1.0,100)
b = np.linspace(1.0,2.0,100)
c = a + b
```

SciNet compute canada | calcul canada

Python overhead

- Python's overhead comes mainly from it's interpreted nature.
- The `diff2d_slow_numpy.py` code uses numpy arrays, but still has a loop over indiced.
- Numpy will not give much speedup until you use it 'vector' operations.

E.g., instead of

```
a = np.linspace(0.0,1.0,101)
b = np.linspace(1.0,2.0,101)
c = np.ndarray(100)
for i in xrange(100):
  c[i] = a[i] + b[i+1]
```

And to deal with shifts, instead of

```
a = np.linspace(0.0,1.0,101)
b = np.linspace(1.0,2.0,101)
c = np.ndarray(100)
for i in xrange(100):
  c[i] = a[i] + b[i+1]
```

You would write:

```
a = np.linspace(0.0,1.0,100)
b = np.linspace(1.0,2.0,100)
c = a + b
```

SciNet compute | calcul canada

Python overhead

- Python's overhead comes mainly from it's interpreted nature.
- The `diff2d_slow_numpy.py` code uses numpy arrays, but still has a loop over indiced.
- Numpy will not give much speedup until you use it 'vector' operations.

E.g., instead of

```
a = np.linspace(0.0,1.0,101)
b = np.linspace(1.0,2.0,101)
c = np.ndarray(100)
for i in xrange(100):
  c[i] = a[i] + b[i+1]
```

You would write:

```
a = np.linspace(0.0,1.0,100)
b = np.linspace(1.0,2.0,100)
c = a + b
```

And to deal with shifts, instead of

```
a = np.linspace(0.0,1.0,101)
b = np.linspace(1.0,2.0,101)
c = np.ndarray(100)
for i in xrange(100):
  c[i] = a[i] + b[i+1]
```

You would write:

SciNet compute | calcul canada

Python overhead

- Python's overhead comes mainly from it's interpreted nature.
- The `diff2d_slow_numpy.py` code uses numpy arrays, but still has a loop over indiced.
- Numpy will not give much speedup until you use it 'vector' operations.

E.g., instead of

```
a = np.linspace(0.0,1.0,101)
b = np.linspace(1.0,2.0,101)
c = np.ndarray(100)
for i in xrange(100):
  c[i] = a[i] + b[i+1]
```

You would write:

```
a = np.linspace(0.0,1.0,100)
b = np.linspace(1.0,2.0,100)
c = a + b
```

And to deal with shifts, instead of

```
a = np.linspace(0.0,1.0,101)
b = np.linspace(1.0,2.0,101)
c = np.ndarray(100)
for i in xrange(100):
  c[i] = a[i] + b[i+1]
```

You would write:

```
a = np.linspace(0.0,1.0,101)
b = np.linspace(1.0,2.0,101)
c = a[0:100] + b[1:101]
```

Hands-on

Vectorize the slow numpy code

- Copy diff2d_slow_numpy.py to diff2d_numpy
- Try to replace the indexed loops with whole-array vector operations

SciNet compute | calcul canada

Does changing to numpy really help?

Let's return to our 2D diffusion example.
Pure python implementation:

```
$ etime python diff2d.py > output_p.txt
Elapsed: 132.19 seconds
```

Does changing to numpy really help?

Let's return to our 2D diffusion example.
Pure python implementation:

```
$ etime python diff2d.py > output_p.txt
Elapsed: 132.19 seconds
```

Numpy implementation:

```
$ etime python diff2d_numpy.py > output_n.txt
Elapsed: 4.46 seconds
```

SciNet compute | calcul canada

Does changing to numpy really help?

Let's return to our 2D diffusion example.
Pure python implementation:

```
$ etime python diff2d.py > output_p.txt
Elapsed: 132.19 seconds
```

Numpy implementation:

```
$ etime python diff2d_numpy.py > output_n.txt
Elapsed: 4.46 seconds
```

However, this is what the compiled versions do:

```
$ etime ./diff2d_cpp.ex > output_c.txt
Elapsed: 0.73 seconds
$ etime ./diff2d_f90.ex > output_f.txt
Elapsed: 0.56 seconds
```

So python+numpy is still 8× slower than compiled.

What about Cython?

- Cython is a compiler for python code.
- Almost all python is valid cython.
- Typically used for packages, to be used in regular python scripts.
- It is important to realize that the compilation preserves the pythonic nature of the language, i.e, garbage collection, range checking, reference counting, etc, are still done: no performance enhancement.

```
$ etime python diff2d_numpy.py        > output_n.txt
Elapsed: 4.46 seconds
$ etime python diff2d_numpy_cython.py > output_nc.txt
Elapsed: 5.49 seconds
```

- If you want to get around that, you need to use Cython specific extentions that essentially use c types.
- From that point on, though, is it still python?

Out-of-core computations

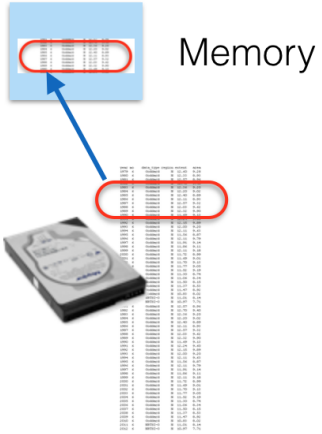
Out-of-core computation

- 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).

SciNet compute | calcul canada

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 well when one's data access involves passing through an entire data sets a small number of times.
- 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 on Bridges)

```
#file: create12GB.py
import numpy as np
n = 40000
f = np.memmap('bigfile', dtype='float64', mode='w+', shape=(n,n))
for i in xrange(n):
  f[i,:] = np.random.rand(n)
```

```
initial fragment:
[ 0.  0.  0.  0.  0.]
random fragment:
[ 0.94379592  0.17967309  0.7169163   0.44854681  0.41266199]
done
```

Out-of-core computation, example

Exit the python prompt, and start over to calculate the mean of the array:

```
#file: average12GB.py
import numpy as np
n = 40000
f = np.memmap('bigfile', mode='r', shape=(n,n))
total = 0.0
for i in xrange(n): total += sum(f[i,:])
average = total / (n*n)
print(average)
```

```
130.186355131
```

This can be very hard on the file system!

Other good use of this technique:

- You have an array that is larger than would fit in memory
- You need only relatively few elements of the array
- But you do not know ahead of time which elements.

Parallel Python

Parallel Python

We will look at a number of approached to parallel programming with Python:

Package	Functionality
numexpr	threaded parallelization of certain numpy expressions
threads	create threads sharing memory
multiprocessing	create processes that behave more like threads
mpi4py	message passing between processes

SciNet compute | calcul canada

Numexpr

The numexpr package

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

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

SciNet compute | calcul canada

Numexpr in a nutshell

- Give it an array artihmetic expression, and it will compile and run it, and return or store the output.
- Supported operators:
  +, -, *, /, **, %, <<, >>, <, <=, ==, !=, >=, >, &, |, ~
- Supported functions:
  `where`, `sin`, `cos`, `tan`, `arcsin`, `arccos` `arctan`, `arctan2`, `sinh`, `cosh`, `tanh`, `arcsinh`, `arccosh` `arctanh`, `log`, `log10`, `log1p`, `exp`, `expm1`, `sqrt`, `abs`, `conj`, `real`, `imag`, `complex`, `contains`.
- Supported reductions:
  `sum`, `product`

SciNet compute | calcul canada

Using the numexpr package

Without numexpr:

```
>>> from etime import etime
>>> import numpy as np
>>> import numexpr as ne
>>> a = np.random.rand(1000000)
>>> b = np.random.rand(1000000)
>>> c = np.zeros(1000000)
>>> etime("c = a**2 + b**2 + 2*a*b", "a,b,c")
Elapsed: 0.00942203998566 seconds
```

Note: The python function `etime` *measures the elapsed time. It is defined in the file* `etime.py` *that is part of the code of this session. The second argument should list the variables used (though some will be picked up automatically).*
Ipython has its own version of this, invoked (without quotes) as

```
In [10]: %time c = a**2 + b**2 +2*a*b
```

Using the numexpr package

With numexpr:

```
>>> from etime import etime
>>> import numpy as np
>>> import numexpr as ne
>>> a = np.random.rand(1000000)
>>> b = np.random.rand(1000000)
>>> c = np.zeros(1000000)
>>> etime("c = a**2 + b**2 + 2*a*b")
Elapsed: 0.0147772550583 seconds
>>> old = ne.set_num_threads(1)
>>> etime("ne.evaluate('a**2 + b**2 + 2*a*b',out=c)", "a,b,c")
Elapsed: 0.00505665540695 seconds
>>> old = ne.set_num_threads(2)
>>> etime("ne.evaluate('a**2 + b**2 + 2*a*b',out=c)", "a,b,c")
Elapsed: 0.0029923081398 seconds
```

SciNet compute | calcul canada

Numexpr for the diffusion example

- Annoyingly, numexpr has no facilities for slicing or offsets, etc.
- This is troubling for our diffusion code, in which we have to do something like

```
laplacian[1:nrows+1,1:ncols+1] = (dens[2:nrows+2,1:ncols+1]
                                + dens[0:nrows+0,1:ncols+1]
                                + dens[1:nrows+1,2:ncols+2]
                                + dens[1:nrows+1,0:ncols+0]
                                - 4*dens[1:nrows+1,1:ncols+1])
```

- We would need to make a copy of `dens[2:nrows+2,1:ncols+1]` etc. into a new numpy array before we can use `numexpr`, but copies are expensive.
- We want `numexpr` to use the same data as in dens, but *viewed* differently.

Numexpr for the diffusion example (cont.)

- We want `numexpr` to use the same data as in dens, but *viewed* differently.
- That is tricky, and requires knowledge of the data's memory structure.
- `diff2d_numexpr.py` shows one possible solution.

```
$ etime python diff2d_numpy.py > diff2d_numpy.out
Elapsed: 4.46 seconds
$ export OMP_NUM_THREADS=14
$ etime python diff2d_numexpr.py  > diff2d_numexpr.out
Elapsed: 2.26 seconds
```

SciNet compute | calcul canada

Theano

- 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.
  (But Tensorflow is quickly gaining ground in this arena.)
- 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).
- Theano can use GPUs, but you're programming them like CUDA, not like OpenACC.

SciNet compute | calcul canada

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 14 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 14 cores?

```
$ etime python diff2d_numpy.py
Elapsed: 4.87 seconds
$ export OMP_NUM_THREADS=14
$ etime python diff2d_numexpr.py
Elapsed: 1.25 seconds
$ etime python diff2d_theano.py
Elapsed: 2.79 seconds
```

Numexpr wins. . . .

How about serially?

SciNet compute canada | calcul canada

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 14 cores?

```
$ etime python diff2d_numpy.py
Elapsed: 4.87 seconds
$ export OMP_NUM_THREADS=14
$ etime python diff2d_numexpr.py
Elapsed: 1.25 seconds
$ etime python diff2d_theano.py
Elapsed: 2.79 seconds
```

Numexpr wins. . . .

How about serially?

```
$ #with "ne.set_num_threads(1)"
$ etime python diff2d_numexpr.py
Elapsed: 3.01 seconds
$ etime python diff2d_theano.py
Elapsed: 2.74 seconds
```

Theano wins serially (just, and not always)

Another compiler-within: Numba

- Numba allows compilation of selected portions of Python code to native code.
- Decorator based: compile a function.
- It can use multi-dimensional arrays and slices, like NumPy.
- Very convenient.
- Downsides:
  - While it can also vectorize, multi-core parallelize and push to a gpu, it can only do so for specific, independent, non-sliced data.
  - Requires LLVM be installed.

SciNet compute | calcul canada

Numba in the diffusion equation

For the diffusion code, we change the time step to a function with a decorator:

Before:

```
  # Take one step to produce new density.
  laplacian[1:nrows+1,1:ncols+1] = dens[2:nrows+2,1:
  densnext[:,:] = dens + (D/dx**2)*dt*laplacian
```

```
$ etime python diff2d_numpy.py
Elapsed: 4.46 seconds
```

After:

```
from number import autojit
@autojit                                                    +1
def timestep(laplacian,dens,densnext,nrows,ncols,D,d
    laplacian[1:nrows+1,1:ncols+1] = dens[2:nrows+2,
    densnext[:,:] = dens + (D/dx**2)*dt*laplacian
...
  # Take one step to produce new density.
  timestep(laplacian,dens,densnext,nrows,ncols,D,dx,
```

```
$ etime python diff2d_numba.py
```

Unfortunately, numba requires llvm, which I couldn't find on Bridges, and for which a local install in your $HOME would exceed your 10GB quote (as I found out yesterday night).

Processes and threads in python

Processes and threads in python

If you've followed the 'mpi/openmp' sessions, you have heard that

- 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.

SciNet compute canada | calcul canada

Threads in Python

Threads in Python

- The good news is: Python has threads.
- The not-so-good news is: No convenient OpenMP launching of threads.
- The worse news: you'll see ...

SciNet compute | calcul canada

How much faster is it using threads?

```
# summer.py - used in all summer*py
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() - begin,"seconds"
```

```
# 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")
```

How much faster is it using threads?

```
# summer.py - used in all summer*py
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() - begin,"seconds"
```

```
# 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")
```

Timings

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

SciNet compute | calcul canada

Not faster at all, slower!

The threading code is no faster than the serial code. 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 canada | calcul canada

Forking

Forking

- For python, the ancient way of parallel programming is a funny intermediate called "Forking", that can create processes on the same node.
- We will skip forking, as it is tedious, but I've included some slides for those interested.

SciNet compute canada | calcul canada

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.
- 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
```

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.

```
# 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
```

compute canada | calcul canada

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.

SciNet compute | calcul canada

Multiprocessing

SciNet compute | calcul canada

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 that run concurrently (like fork), and 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(p)
   p.start()
# Wait for all processes to finish.
for p in processes: p.join()
print ("Elapsed:%f"%
      time.time() - begin)
```

```
$ python summer_multiprocessing.py
Time:0.185396
```

compute canada | calcul canada

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 Process
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)
```

SciNet compute | calcul canada

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 Process
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
475
Elapsed: 0.15 seconds
```

- Did the code behave as expect?

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.

```
# multiprocessing_shared_fixed.py
from multiprocessing import Process
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
```

```
# 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: 0.16 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.

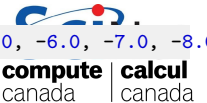
```
# multiprocessing_shared_array.py
from numpy import arange
from multiprocessing import Process,Array
def myfun(a, i):
  a[i] = -a[i]
arr = Array('d', arange(10.))
procs = []
for i in range(10):
  p = Process(target=myfun,
              args=(arr, i))
  procs.append(p)
  p.start()
for proc in procs:
  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 Process,Array
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.
```



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

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.

SciNet compute | calcul canada

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).

SciNet compute canada | calcul canada

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<<"\n";
 MPI_Finalize();
}
```

SciNet compute canada | calcul canada

MPI Fortran recap

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

```
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
```

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.

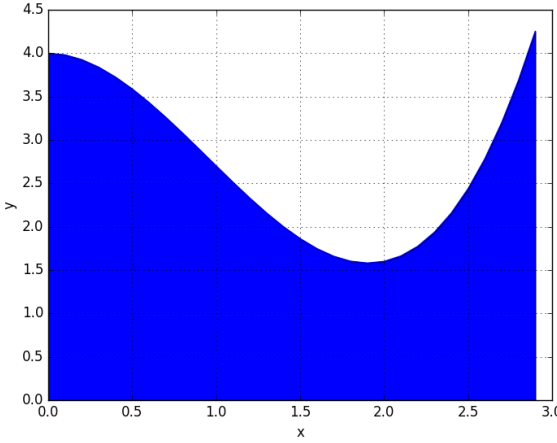
SciNet compute | calcul canada

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

```
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
```

SciNet compute | calcul canada

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]
```

compute canada | calcul canada

Mpi4py Speedup?

```
$ etime mpirun -np 1 python auc.py 300000000
Traceback (most recent call last):
  File "auc.py", line 4, in <module>
    from mpi4py import MPI
ImportError: /home/rzon/.local/lib/python2.7/site-packages/mpi4py/MPI.so: undefined symbol: ompi_mpi_char
Elapsed: 0.23 seconds
```

SciNet compute canada | calcul canada

Mpi4py Speedup?

```
$ etime mpirun -np 1 python auc.py 300000000
Traceback (most recent call last):
  File "auc.py", line 4, in <module>
    from mpi4py import MPI
ImportError: /home/rzon/.local/lib/python2.7/site-packages/mpi4py/MPI.so: undefined symbol: ompi_mpi_char
Elapsed: 0.23 seconds
```

```
$ etime mpirun -np 4 python auc.py 300000000
Traceback (most recent call last):
  File "auc.py", line 4, in <module>
Traceback (most recent call last):
  File "auc.py", line 4, in <module>
    from mpi4py import MPI
ImportError: /home/rzon/.local/lib/python2.7/site-packages/mpi4py/MPI.so: undefined symbol: ompi_mpi_char
    from mpi4py import MPI
ImportError: /home/rzon/.local/lib/python2.7/site-packages/mpi4py/MPI.so: undefined symbol: ompi_mpi_char
Traceback (most recent call last):
  File "auc.py", line 4, in <module>
    from mpi4py import MPI
ImportError: /home/rzon/.local/lib/python2.7/site-packages/mpi4py/MPI.so: undefined symbol: ompi_mpi_char
Traceback (most recent call last):
```

Mpi4py Speedup?

```
$ etime mpirun -np 1 python auc.py 300000000
Traceback (most recent call last):
  File "auc.py", line 4, in <module>
    from mpi4py import MPI
ImportError: /home/rzon/.local/lib/python2.7/site-packages/mpi4py/MPI.so: undefined symbol: ompi_mpi_char
Elapsed: 0.23 seconds
```

```
$ etime mpirun -np 4 python auc.py 300000000
Traceback (most recent call last):
  File "auc.py", line 4, in <module>
Traceback (most recent call last):
  File "auc.py", line 4, in <module>
    from mpi4py import MPI
ImportError: /home/rzon/.local/lib/python2.7/site-packages/mpi4py/MPI.so: undefined symbol: ompi_mpi_char
    from mpi4py import MPI
ImportError: /home/rzon/.local/lib/python2.7/site-packages/mpi4py/MPI.so: undefined symbol: ompi_mpi_char
Traceback (most recent call last):
  File "auc.py", line 4, in <module>
    from mpi4py import MPI
ImportError: /home/rzon/.local/lib/python2.7/site-packages/mpi4py/MPI.so: undefined symbol: ompi_mpi_char
Traceback (most recent call last):
```

Mpi4py Speedup?

```
$ etime mpirun -np 1 python auc.py 300000000
Traceback (most recent call last):
  File "auc.py", line 4, in <module>
    from mpi4py import MPI
ImportError: /home/rzon/.local/lib/python2.7/site-packages/mpi4py/MPI.so: undefined symbol: ompi_mpi_char
Elapsed: 0.23 seconds
```

```
$ etime mpirun -np 4 python auc.py 300000000
Traceback (most recent call last):
  File "auc.py", line 4, in <module>
Traceback (most recent call last):
  File "auc.py", line 4, in <module>
    from mpi4py import MPI
ImportError: /home/rzon/.local/lib/python2.7/site-packages/mpi4py/MPI.so: undefined symbol: ompi_mpi_char
    from mpi4py import MPI
ImportError: /home/rzon/.local/lib/python2.7/site-packages/mpi4py/MPI.so: undefined symbol: ompi_mpi_char
Traceback (most recent call last):
  File "auc.py", line 4, in <module>
    from mpi4py import MPI
ImportError: /home/rzon/.local/lib/python2.7/site-packages/mpi4py/MPI.so: undefined symbol: ompi_mpi_char
Traceback (most recent call last):
```

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?

SciNet compute canada | calcul canada

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, e.g. IPython Parallel or Spark (Friday).

SciNet compute canada | calcul canada

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.
- Examples: IPython Parallel, Spark

SciNet compute | calcul canada