HPC Python Programming

Ramses van Zon

SciNet HPC Consortium

IHPCSS, July 11, 2018

In this session...

- Performance and Python
- Profiling tools for Python
- Fast arrays for Python: Numpy
- Numexpr, Theano, Numba
- Threading
- Multiprocessing
- Mpi4py

Getting Started

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
- numba

Get the code and setup files on Bridges

Code and installation can be copied from my home on Bridges. It can be found in the directory `/home/rzon/hpcpy`.

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 on your own directory (needed only once)

```
$ cp -r /home/rzon/hpcpy $HOME/
$ cd $HOME/hpcpy
$ source setup
```

3. Request an interactive session (needed again if you log off)

```
$ interact -p RM -R ihsswe -n 14 -t 3:00:00
$ source $HOME/hpcpy/activate
```

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Introduction

Performance and Python

- Python is a high-level, interpreted language.
- Those defining features are often at odds with “high performance”.
- But development in Python can be substantially easier (and thus faster) than when using 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
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Why isn't Python "High Performance"?

It's a Interpreted language:

- Translation to machine code 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 of this interfers with optimal streaming of data to processor, which hampers maximum performance.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Example: 2D diffusion equation

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

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

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

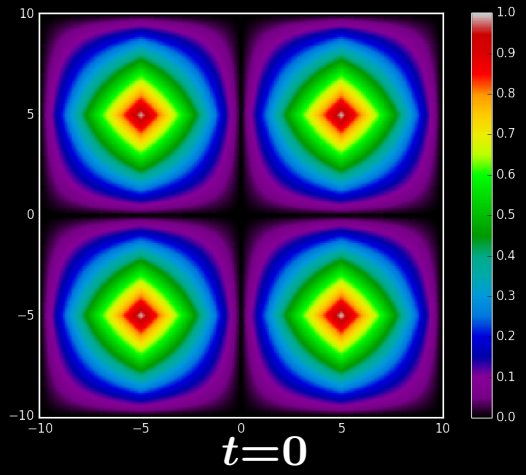
Square domain $[x_1, x_2] \otimes [x_1, x_2]$,

with $p(x,y,t) = 0$ at all times for all points on the domain boundary,

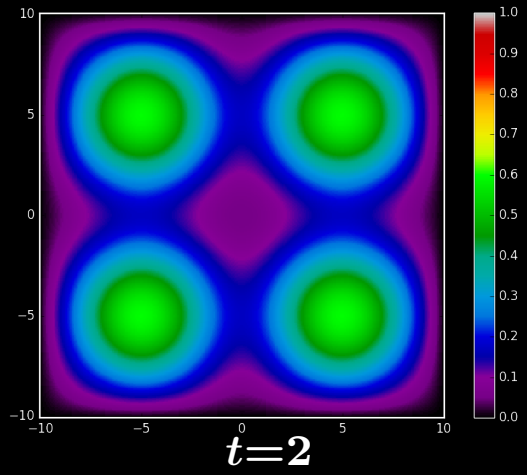
with some given initial condition $p(x,y,t) = p_0(x,y)$.

Example: 2D diffusion, result

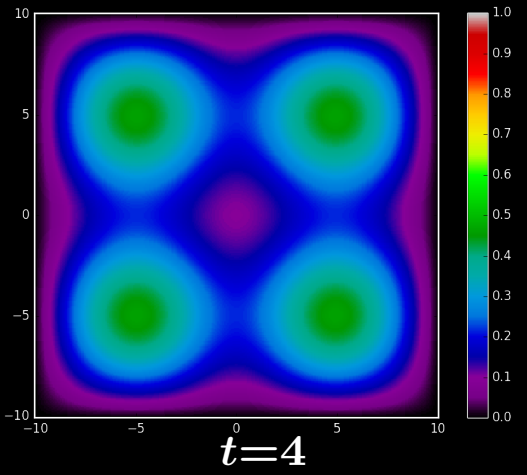
$x_1 = -10$, $x_2 = 10$, $D = 1$, four-peak initial condition.



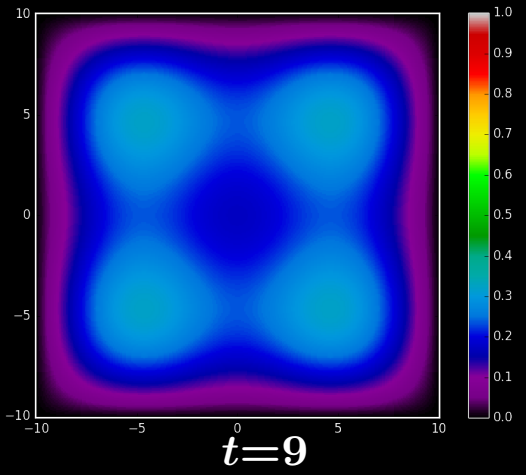
$t=0$



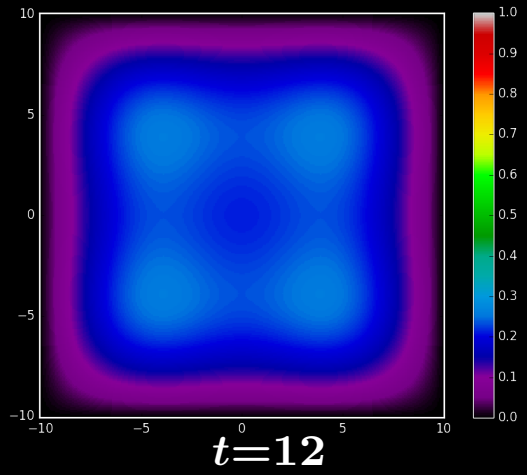
$t=2$



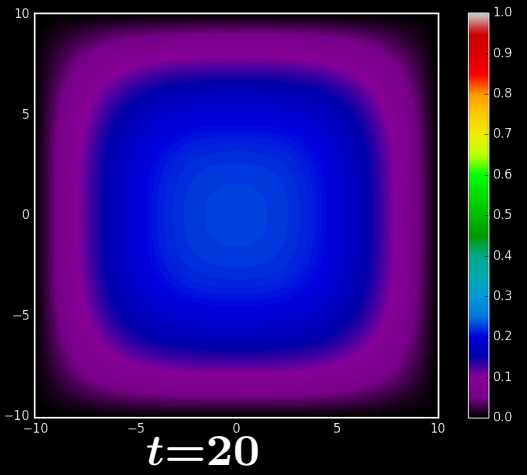
$t=4$



$t=9$



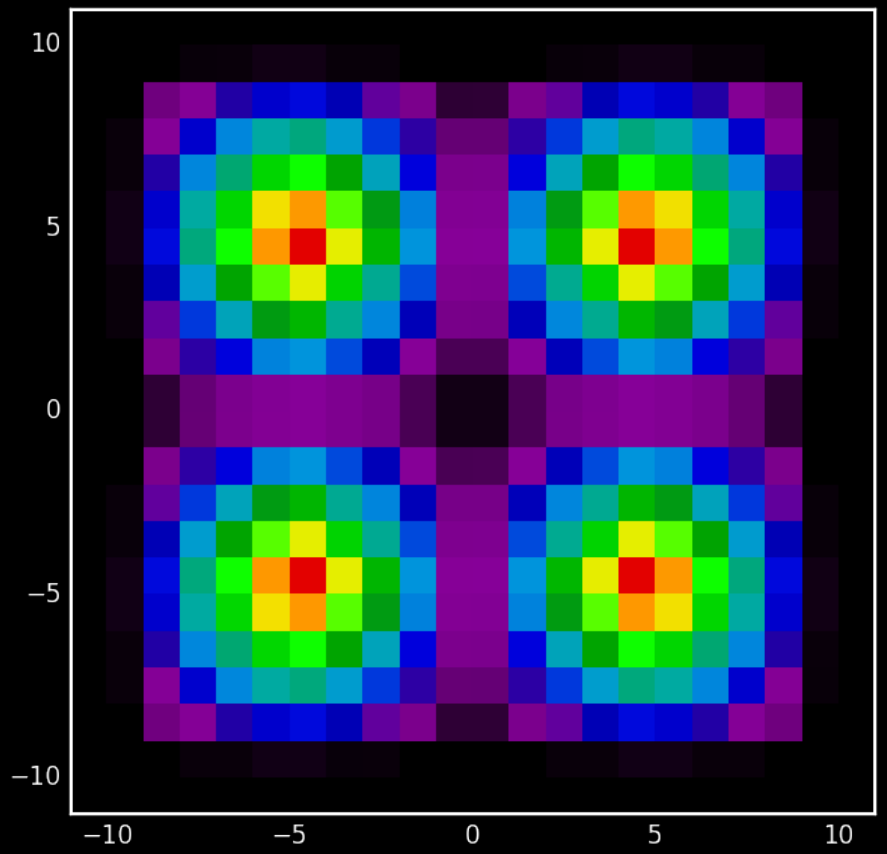
$t=12$



$t=20$

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)
- Discretized Laplacian at grid point (i, j) equals



$$\frac{p(i+1,j) + p(i-1,j) + p(i,j-1) + p(i,j+1) - 4p(i,j)}{dx^2}$$

Example: 2D diffusion, parameters

The fortran, C++, and Python codes all read the same parameter file file (using some tricks).

diff2dparams.py

```
D         =   1.0;
x1        = -10.0;
x2        =  10.0;
runtime   =  15.0;
dx        =  0.1;
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.

```
$ time make diff2d_cpp.ex diff2d_f90.ex
g++ -c -std=c++11 -I. -O3 -o diff2d_cpp.o diff2d.cpp
gfortran -c -o pgplot.o pgplot.f90
...
Elapsed: 4.15 seconds
```

```
$ time ./diff2d_cpp.ex > output_c.txt
Elapsed: 0.40 seconds
$ time ./diff2d_f90.ex > output_f.txt
Elapsed: 0.37 seconds
$ time python diff2d.py > output_p.txt
Elapsed: 173.89 seconds
```

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

Then why do we bother with Python?

Fast development

- Python lends itself easily to writing clear, concise code.
- Python is very flexible: large set of very useful packages.
- Easy of use $\rightarrow$ shorter development time

Performance hit depends on application

- Python's performance hit is most prominent 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, etc.
- Hooks to compiled libraries to remove worst performance pitfalls.

Only once the performance isn't too bad, can we start thinking of parallelization, i.e., using more cpu cores to work on the same problem.

Performance Tuning Tools for Python

Computational 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 **computational performance** first, as measured by the time the computation requires, in two ways:

Time profiling by function

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

Time profiling by line

To find 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>)
```

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 in a function.
- You also need to 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)
```

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
```

Output of line_profiler

```
$ kernprof -l -v profileme.py
1.0
is a one
Wrote profile results to profileme.py.lprof
Timer unit: 1e-06 s
Total time: 0.032287 s
File: profileme.py
Function: profilewrapper at line 2
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     2                                           @profile
     3                                           def profilewrapper():
     4         1      21144.0  21144.0     65.5    x=[1.0]*(2048*2048)
     5         1         28.0     28.0      0.1    a=str(x[0])
     6         1          4.0      4.0      0.0    a+="\nis a one\n"
     7         1      11081.0  11081.0     34.3    del x
     8         1         30.0     30.0      0.1    print(a)
```

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
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

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
```

How would you know that memory usage is problematic?

memory_profiler

- This module 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).

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
```

```
$ python -m memory_profiler profileme.py
1.0
is a one
Filename: profileme.py
Line #    Mem usage    Increment   Line Contents
================================================
     2   33.188 MiB   33.188 MiB   @profile
     3                             def profilewrapper():
     4   65.086 MiB   31.898 MiB     x=[1.0]*(2048*2048)
     5   65.086 MiB    0.000 MiB     a=str(x[0])
     6   65.086 MiB    0.000 MiB     a+="\nis a one\n"
     7   33.223 MiB  -31.863 MiB     del x
     8   33.223 MiB    0.000 MiB     print(a)
```

Hands-on

Profile the diff2d.py code

- Reduce the resolution and runtime in diff2dparams.py, i.e., increase `dx` to 0.5, and decrease `runtime` to 2.0.
- In the same file, ensure that `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?

Numpy: Faster Arrays for Python

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
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

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,  1.16
        1.40816327,  1.48979592,  1.57
        1.81632653,  1.89795918,  1.97
        2.2244898 ,  2.30612245,  2.38
        2.63265306,  2.71428571,  2.79
        3.04081633,  3.12244898,  3.20
        3.44897959,  3.53061224,  3.61
        3.85714286,  3.93877551,  4.02
        4.26530612,  4.34693878,  4.42
        4.67346939,  4.75510204,  4.83
>>> linspace(1,5,6)
array([ 1. ,  1.8,  2.6,  3.4,  4.2,
```

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 f
```

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]])
```

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

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.])
```

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

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
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

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
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

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
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

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:

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

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

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

How to fix the matrix algebra?

- Use the special built-in matrix-multiplication operator of Python 3.5+.
- In combination with numpy arrays.
- Alternatively, use the 'dot' method (in Python < 3.5)

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

```
>>> a.transpose()
array([[1, 4],
       [2, 3]])
>>> np.dot(a, b)
array([[9, 8],
       [16, 17]])
>>> np.dot(b, a.transpose())
array([[ 5, 10],
       [10, 25]])
>>> np.dot(a,c)
array([5, 10])
```

```
>>> a.transpose()
array([[1, 4],
       [2, 3]])
>>> a @ b
array([[9, 8],
       [16, 17]])
>>> b @ a.transpose()
array([[ 5, 10],
       [10, 25]])
>>> a @ c
array([5, 10])
```

Does changing to numpy really help?

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

```
$ time python diff2d.py > output_p.txt
Elapsed: 175.54 seconds
```

Numpy implementation:

```
$ time python diff2d_slow_numpy.py > output_n.txt
Elapsed: 399.43 seconds
```

Hmm, not really (really not!), what gives?

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 indices.
- Numpy will not give much speedup until you use its '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 range(100):
  c[i] = a[i] + b[i]
```

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 range(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]
```

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Hands-on

"Vectorize" the slow numpy code

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

Does changing to numpy really help?

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

```
$ time python diff2d.py > output_p.txt
Elapsed: 175.54 seconds
```

Numpy implementation:

```
$ time python diff2d_numpy.py > output_n.txt
Elapsed: ??? seconds
```

Note: We can call this *"vectorization"* because the code works on whole vectors. But this is different from vectorization which uses the small vector units on the cpu. Here, we're just minimizing the number of lines python needs to interpret.

Numpy vs. compiled

Numpy implementation:

```
$ time python diff2d_numpy.py > output_n.txt
Elapsed: 2.42 seconds
```

However, this is what the compiled versions do:

```
$ time ./diff2d_cpp.ex > output_c.txt
Elapsed: 0.59 seconds
$ time ./diff2d_f90.ex > output_f.txt
Elapsed: 0.47 seconds
```

So python+numpy is still 5× slower than compiled versions.

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.

```
$ make -f Makefile_cython diff2dnumpylib.so
...
$ time python diff2d_numpy.py > output_n.txt
Elapsed: 2.50 seconds
$ time python diff2d_numpy_cython.py > output_nc.txt
Elapsed: 2.63 seconds
```

- The compilation preserves the pythonic nature of the language, i.e, garbage collection, range checking, reference counting, etc, are still done: *no performance enhancement.*
- If you want to get around that, you need to use Cython specific extentions that use c types.

Parallel Python

Parallel Python

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

Package	Functionality
numexpr	threaded parallelization of certain numpy expressions
theano	turns symbolic expressions into compiled code
numba	turns python functions into compiled code
threads	create threads sharing memory
multiprocessing	create processes that behave more like threads
mpi4py	message passing between processes

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.

Numexpr in a nutshell

- Give it an array arithmetic 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`

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: 0.008287844643928111 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: 0.015387782349716873 seconds
>>> old = ne.set_num_threads(1)
>>> etime("ne.evaluate('a**2 + b**2 + 2*a*b',out=c)", "a,b,c")
Elapsed: 0.0039271804969757 8 seconds
>>> old = ne.set_num_threads(2)
>>> etime("ne.evaluate('a**2 + b**2 + 2*a*b',out=c)", "a,b,c")
Elapsed: 0.0025745375896804035 seconds
>>> old = ne.set_num_threads(14)
>>> etime("ne.evaluate('a**2 + b**2 + 2*a*b',out=c)", "a,b,c")
Elapsed: 0.0006712840055115521 seconds
```

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.

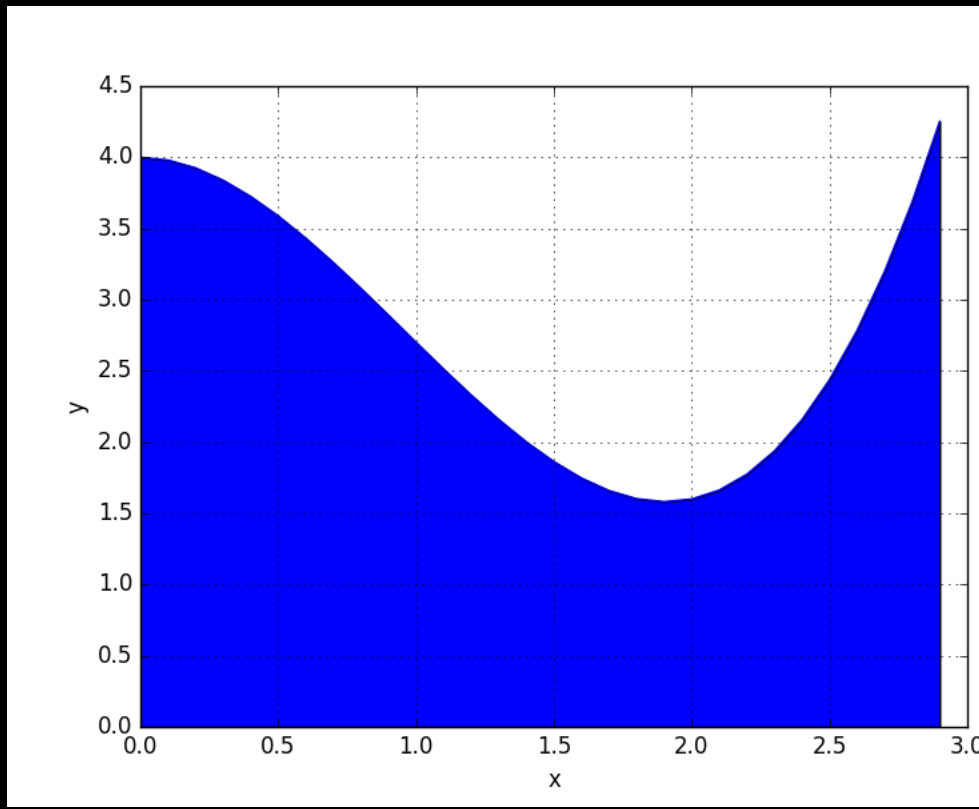
```
$ time python diff2d_numpy.py >diff2d_numpy.out
Elapsed: 2.40 seconds
$ export NUMEXPR_NUM_THREADS=14
$ time python diff2d_numexpr.py >diff2d_numexpr.out
Elapsed: 1.61 seconds
```

Hands-on: 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.

Hands-on: Area under the curve

Serial code is in `auc_serial.py`.

Your test case will be `python auc_serial.py 30000000`

- Reexpress the computation in numpy vectors.
- Measure the speedup.
- Use `numexpr` to parallelize the auc.py code.
- Measure the speed-up using 2, 7, 14 threads.

```
import sys
ntot  = int(sys.argv[1])
dx    = 3.0/ntot
width = 3.0
x = 0
a = 0.0
for i in range(ntot):
    y = 0.7*x**3 - 2*x**2 + 4
    a += y*dx
    x += dx
print("The area is",a)
```

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Numexpr trick for diffusion unraveled

To get the diffusion algorithm in a form that has no slices or offsets, we need to linearize the 2d arrays into 1d arrays, but in a way that avoids copying the data.
This is how this is achieved in `diff2d_numexpr`:

```
dens     = dens.ravel()
densnext = densnext.ravel()
densL = dens[npnts-1:-npnts-1] # same data one cell left
densR = dens[npnts+1:-npnts+1] # same data one cell right
densU = dens[0:-2*npnts]       # same data one cell up
densD = dens[2*npnts:]         # same data one cell down
densC = dens[npnts:-npnts]
ne.evaluate('densC + (D/dx**2)*dt*(densL+densR+densU+densD-4*densC)',
            out=densnext[npnts:-npnts])
dens = dens.reshape((npnts,npnts))
densnext = densnext.reshape((npnts,npnts))
```

Theano

- Theano is a numerical computation library.
- Much like numexpr, it takes an (array) expression and compiles it.
- Theano is frequently used 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 itself, though it may link to multithreaded blas libraries such as MKL.
- Theano can use GPUs, but you're programming them like CUDA, not like OpenACC.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

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?

```
$ export NUMEXPR_NUM_THREADS=14
$ time python diff2d_numpy.py >diff2d_numpy.out
Elapsed: 2.45 seconds
$ time python diff2d_numexpr.py >diff2d_numexpr.out
Elapsed: 1.59 seconds
$ time python diff2d_theano.py  >diff2d_theano.out
Elapsed: 3.82 seconds
```

Numexpr wins.

How about serially?

```
$ export NUMEXPR_NUM_THREADS=1
$ time python diff2d_numexpr.py >diff2d_numexpr_s.out
Elapsed: 2.74 seconds
```

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

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.
- Numba can use GPUs, but you're programming them like CUDA kernels, not like OpenACC.
- While it can also vectorize for multi-core and gpus, it can only do so for specific, independent, non-sliced data.

Numba in the diffusion equation

Before:

```
  # Take one step to produce new density.
  laplacian[1:nrows+1,1:ncols+1] = dens[2:nrows+2,1:ncols+1] + dens[0:nrows
  densnext[:,:] = dens + (D/dx**2)*dt*laplacian
```

```
$ time python diff2d_numpy.py >diff2d_numpy.out
Elapsed: 2.40 seconds
```

After:

```
from numba import autojit
@jit
def timestep(laplacian,dens,densnext,nrows,ncols,D,dx,dt):
    laplacian[1:nrows+1,1:ncols+1] = dens[2:nrows+2,1:ncols+1] + dens[0:nro
    densnext[:,:] = dens + (D/dx**2)*dt*laplacian
...
  timestep(laplacian,dens,densnext,nrows,ncols,D,dx,dt)
```

```
$ time python diff2d_numba.py >diff2d_numba.out
Elapsed: 4.32 seconds
```

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
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

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 ...

How much faster is it using threads?

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

Timings

```
$ python summer_serial.py
Elapsed: 2.85 seconds
$ python summer_threaded.py
Elapsed: 4.64 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 ("Elapsed: %f"%
time.time() - begin,"seconds")
```

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

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
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Processes in Python

Processes in Linux: Forking

- For python, the ancient way of parallel programming is a funny intermediate called "Forking", that can create processes on the same node.
- Only worked on linux.
- We will skip forking, as it is tedious.
- The point of it was to create separate processes, each with its own python interpreter and its own interpreter lock.
- Such processes **can** run in parallel.

Multiprocessing

Multiprocessing

The multiprocessing module tries to strike a balance between forking processes 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.

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
Elapsed: 0.706869
```

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)
```

```
$ time python multiprocessing_shared.py
442
Elapsed: 0.23 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.
- 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)
```

```
$ time 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
from multiprocessing import 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,
```

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

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).

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

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();
}
```

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.

```
# mpi4py_right_rank.py
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
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Mpi4py+numpy: Area-under-the-curve

```
# auc.py
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 range(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)
```

```
# auc_numpy.py
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 range(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?

```
$ time mpirun -n 1 python auc.py 30000000
The area is 8.175000
Elapsed: 18.16 seconds
```

```
$ time mpirun -n 14 python auc.py 30000000
The area is 8.175000
Elapsed: 7.39 seconds
```

```
$ time mpirun -n 14 python auc_numpy.py 30000000
The area is 8.175000
Elapsed: 8.05 seconds
```

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

Hands-on

- The code for the programming challenge has a serial python version: `laplace_serial.py`
- Use mpi to parallelize this code.

or:

- Reflect hard on whether you should use Python for HPC.

Is there no hope, then, Ramses?

Sure there is. . .

When doing data analysis

- If you're reading in data, perform some analysis, and write it out, your performance is likely limited by disk I/O.
- In that case, the cpu penalty of Python may be insignificant.
- If this data is big, consider MPI-IO, NetCDF4, or hdf5 (remember from Parallel I/O session).

When using optimized packages

- Many python modules are actually written in C and just expose an interface to Python; these are as fast as C would be.
- Examples of this include popular **machine learning packages**:
  - sklearn
  - Tensorflow
- More machine learning on Friday.

Conclusion

Summary

- Before thinking about parallelizing your code, ensure the serial version is efficient:
  - profile your script
  - use numpy with "vectorized" programming.
  - Further performance improvements require some sort of compilation (theano, numba, ...)
- Threading in pure python does not improve performance (unless it's a python module that uses a compiled, threaded, c code).
- OpenMP is not possible in python
- MPI is still possible.
- Keep an eye on newer versions of numba and numexpr.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO