Python Programming for HPC (IHPCSS2024)

Ramses van Zon

July 10, 2024

In this session...

1. Performance and Python
2. Profiling tools for Python
3. Fast arrays for Python
4. Parallel computing in Python

Setting up for this session

To get set up to following along, perform the following steps.

1. Login to Bridges2:

```
$ ssh -Y USERNAME@bridges2.psc.edu
```

2. Copy code for this session:

```
$ cp -r /jet/home/rzon/hpcpycode $HOME
```

3. Request interactive resources:

```
$ $HOME/hpcpycode/interactive8.sh
```

4. Setup the environment:

```
$ cd $HOME/hpcpycode
$ source activate
```

(repeat the last step any time you log back in)

1. Performance and Python

Performance and Python

- Python is a high-level, interpreted language.
- Those defining features are often at odds with “high performance”.
- Python is fairly easy to learn, very expressive, and, not surprisingly, very popular.
- But development in Python can be substantially easier (and thus faster) than when using compiled languages.

Example: 2D Diffusion

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

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

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

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

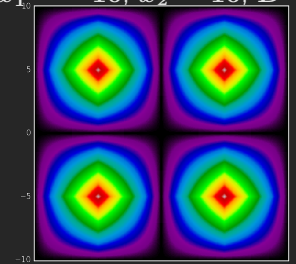
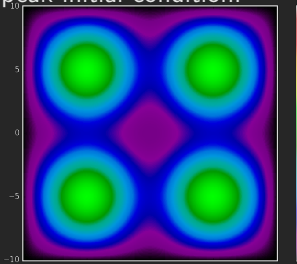
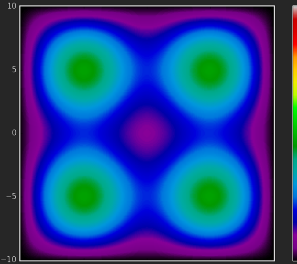
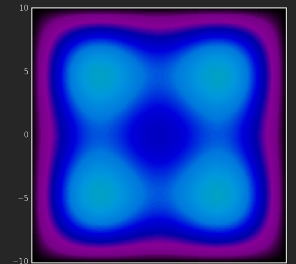
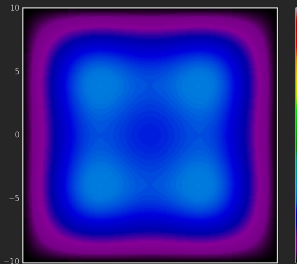
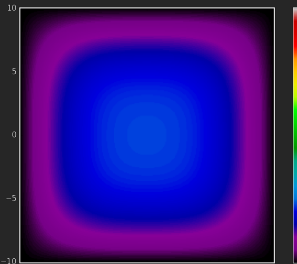
with some given initial condition $\varrho(x,y,t) = \varrho_0(x,y)$.

Here:

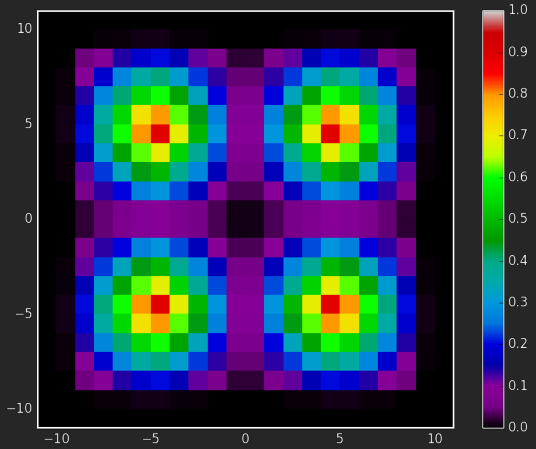
- ϱ: density
- x, y: spatial coordinates
- t: time
- D: diffusion constant

Example 1: 2D Diffusion, Result

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

Example 1: 2D Diffusion, the Algorithm

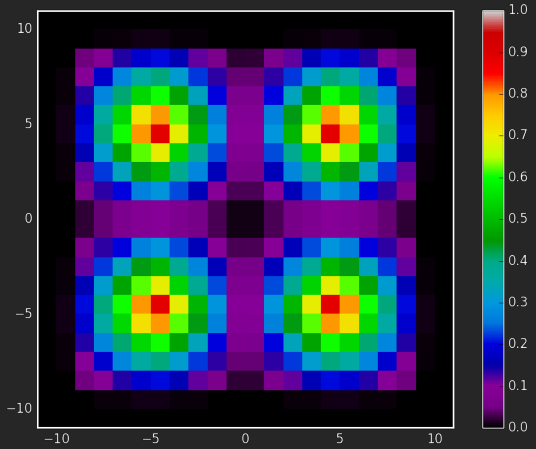


- Discretize space in both directions (points dx apart)
- Replace derivatives with finite differences.
- Explicit finite time stepping scheme (time step set by dx)
- For graphics: Matplotlib for Python, pgplot for C++/Fortran, every `outtime` time units

Parameters in file diff2dparams.py
(also used by C++ and Fortran versions).

```
D         =   1.0;
x1        = -10.0;
x2        =  10.0;
runtime   =  10.0;
dx        =  0.075;
outtime   =  0.5;
graphics  =  True;
```

Example 1: 2D Diffusion, the 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
(also used by C++ and Fortran versions).

```
D         =   1.0;
x1        = -10.0;
x2        =  10.0;
runtime   =  10.0;
dx        =  0.075;
outtime   =  0.5;
graphics  =  False;
```

Example 1: 2D Diffusion, Performance

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

```
$ time make diff2d_cpp.ex diff2d_f90.ex
```

```
g++ -c -O3 -march=native -o diff2d_cpp.o diff2d.cpp
g++ -c -O3 -march=native -o diff2dplot_cpp.o diff2dplot.cpp
g++  -o diff2d_cpp.ex diff2d_cpp.o diff2dplot_cpp.o -lcpgplot -lpgplot -lX11 -lxcb -ldl -lXau -lgfortran
gfortran -c -O3 -march=native -o pgplot90.o pgplot90.f90
gfortran -c -O3 -march=native -o diff2dplot_f90.o diff2dplot.f90
gfortran -c -O3 -march=native -o diff2d_f90.o diff2d.f90
gfortran  -o diff2d_f90.ex diff2d_f90.o diff2dplot_f90.o pgplot90.o -lcpgplot -lpgplot -lX11 -lxcb -ldl -
```

```
Elapsed: 1.18 seconds
```

```
$ time ./diff2d_cpp.ex > output_c.txt
```

```
Elapsed: 0.52 seconds
```

```
$ time ./diff2d_f90.ex > output_f.txt
```

```
Elapsed: 0.43 seconds
```

```
$ time python diff2d.py > output_p.txt
```

```
Elapsed: 212.41 seconds
```

This doesn't look promising for Python for HPC.

The Python version is 400× slower than the compiled versions!

Then why do we bother with Python?

SciNet

```
#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 range(npnts)]
dens     = [[0.0]*npnts for i in range(npnts)]
densnext = [[0.0]*npnts for i in range(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 range(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])
```

```
# 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',
  extent=(x1,x2,x1,x2),vmin=0.0,vmax=1.0,cmap='nipy_s
 if first: plt.colorbar()
 plt.show(); plt.pause(0.1)
```

Then why do we bother with Python?

SciNet

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.
- Some Python packages compile computations on the fly.

2. Profiling Tools for Python

Wall-clock performance

- Performance is about maximizing the utility of a resource.
- This could be cpu processing power, memory, network, file I/O, etc.
- We will focus on **wall-clock performance** here.

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 cpu performance bottlenecks by line of code, there are packages like `line_profiler` and `scalene`.

The cProfile Package

- 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>)
```

The line_profiler package

- 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: 0.018683 s
File: profileme.py
Function: profilewrapper at line 2

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     2                                           @profile
     3                                           def profilewrapper():
     4         1       7816.0   7816.0     41.8    x=[1.0]*(2048*2048)
     5         1         43.0     43.0      0.2    a=str(x[0])
     6         1          3.0      3.0      0.0    a+="\nis a one\n"
     7         1      10783.0  10783.0     57.7    del x
     8         1         38.0     38.0      0.2    print(a)
```

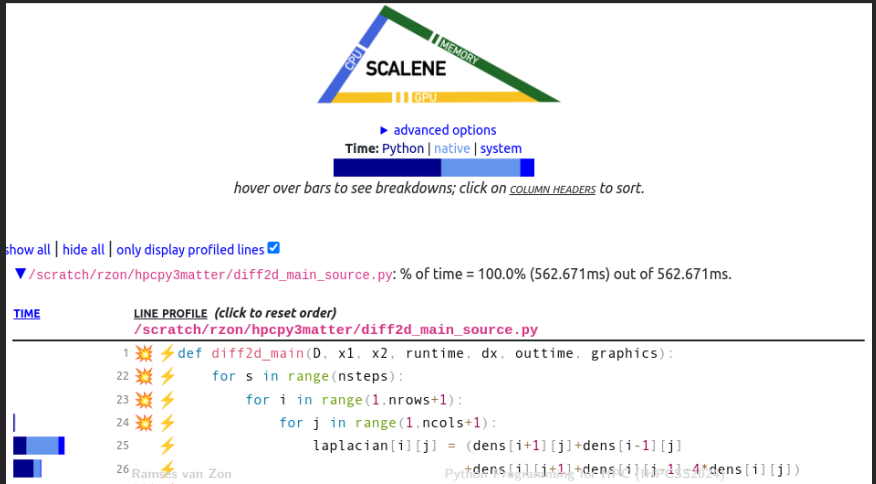
The Scalene Package

- Python Profiler for CPU, memory, and GPU
- Fast
- Accurate
- Distinguished Python from C code
- No decorator required

Scalene Usage

```
$ scalene diff2d.py
```

On your local computer, this would launch the result in a browser:



Scalene Usage on the Command-Line

You can also tell scalene not to generate the HTML but to report the results to the command line instead, as follows:

```
$ scalene --cli diff2d.py
```

3. Fast Arrays for Python

Lists aren't the ideal data type

Python 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]
```

The NumPy Package

- 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
```

```
>>> arange(5)
array([0, 1, 2, 3, 4])
```

```
>>> from numpy import linspace
```

```
>>> linspace(1,5)
array([1.        , 1.08163265, 1.16326531, 1.24489796, 1.326530
       1.40816327, 1.48979592, 1.57142857, 1.65306122, 1.734693
       1.81632653, 1.89795918, 1.97959184, 2.06122449, 2.142857
       2.2244898 , 2.30612245, 2.3877551 , 2.46938776, 2.551020
       2.63265306, 2.71428571, 2.79591837, 2.87755102, 2.959183
       3.04081633, 3.12244898, 3.20408163, 3.28571429, 3.367346
       3.44897959, 3.53061224, 3.6122449 , 3.69387755, 3.775510
       3.85714286, 3.93877551, 4.02040816, 4.10204082, 4.183673
       4.26530612, 4.34693878, 4.42857143, 4.51020408, 4.591836
       4.67346939, 4.75510204, 4.83673469, 4.91836735, 5.
```

```
>>> linspace(1,5,6)
array([1. , 1.8, 2.6, 3.4, 4.2, 5. ])
```

Element-wise arithmetic

vector-vector & vector-scalar multiplication

1-D arrays are often called ‘vectors’.

- When vectors are multiplied with `*`, you get element-by-element multiplication.
- When vectors are multiplied by a scalar

  (a 0-D array), you also get element-by-element multiplication.
- To get an inner product, use `@`.

  (Or use the ‘dot’ method in Python < 3.5)

```
>>> import numpy as np
```

```
>>> a = np.arange(4)
>>> b = np.arange(3., 7.)
>>> c = 2
```

```
>>> a, b, c
(array([0, 1, 2, 3]), array([3., 4., 5., 6.]), 2)
```

```
>>> a * b
array([ 0.,  4., 10., 18.])
```

```
>>> a * c
array([0, 2, 4, 6])
```

```
>>> b * c
array([ 6.,  8., 10., 12.])
```

```
>>> a @ b
32.0
```

Matrix-vector multiplication

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

- Matrix-scalar multiplication with * gives element-by-element multiplication.
- Matrix-vector multiplication with * give a kind-of element-by-element multiplication
- For a linear-algebra-type matrix-vector multiplication, use @.

  (Or use the 'dot' method in Python < 3.5)

```
>>> import numpy as np
```

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

```
>>> b = np.arange(1,4); b
array([1, 2, 3])
```

```
>>> a * b
array([[ 1,  4,  9],
       [ 2,  6, 12]])
```

```
>>> a @ b
array([14, 20])
```

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

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

Matrix-matrix multiplication

Not surprisingly, matrix-matrix multiplication is also element-wise unless performed with @.

```
>>> import numpy as np
```

```
>>> a = np.array([[1,2], [4,3]]) ; a
array([[1, 2],
       [4, 3]])
```

```
>>> b = np.array([[1,2], [4,3]]) ; b
array([[1, 2],
       [4, 3]])
```

```
>>> a * b
array([[ 1,  4],
       [16,  9]])
```

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

```
>>> a @ b
array([[ 9,  8],
       [16, 17]])
```

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

Does changing to NumPy arrays help?

Let's return to our 2D diffusion example.

Note: Restore the original diff2dparam.py!

Pure Python implementation:

```
$ time python diff2d.py > output_p.txt
```

```
Elapsed: 212.41 seconds
```

NumPy implementation:

```
$ time python diff2d_slow_numpy.py > output_p.txt
```

```
Elapsed: 650.86 seconds
```

Hmm, not really

Really not!

So what gives?

Let's inspect the code

```
#diff2d_slow_numpy.py
from diff2dplot import plotdens
from diff2dparams import D,x1,x2,runtime,dx,outtime,graphics
import numpy as np
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        = np.linspace(x1-dx,x2+dx,num=npnts)
dens     = np.zeros((npnts,npnts))
densnext = np.zeros((npnts,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)
```

Look at all those loops and indices!

Look at all those loops and indices!

```
lapl = np.zeros((npnts,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])
```

"Why does that matter?" you ask?

Python overhead

- Python's overhead comes mainly from its interpreted and dynamic nature.
- The `diff2d_slow_numpy.py` code uses NumPy arrays, but still has a loop over indices.
- In each iteration, Python code has to be interpreted and integer manipulation have to be performed, regardless of whether you're using numpy arrays.
- NumPy will not give much speedup until you use its element-wise **'vectorized'** operations.

How to write vectorized Python code

This is easiest explained by example:

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]
```

Vectorization results in

- shorter Python code
- less repeatedly interpreted lines
- calls to C or Fortran functions by NumPy.

Does changing to NumPy really help?

Diffusion example:

Pure Python implementation:

```
$ time python diff2d.py > output_p.txt
```

```
Elapsed: 212.41 seconds
```

NumPy vectorized implementation:

```
$ time python diff2d_numpy.py > output_n.txt
```

```
Elapsed: 2.39 seconds
```

Yeah! 50× speed-up

Reality check: NumPy vs. compiled code

Diffusion example:

NumPy, vectorized implementation:

```
$ time python diff2d_numpy.py > output_n.txt
Elapsed: 2.28 seconds
```

Compiled versions:

```
$ time ./diff2d_cpp.ex > output_c.txt
Elapsed: 0.51 seconds
```

```
$ time ./diff2d_f90.ex > output_f.txt
Elapsed: 0.43 seconds
```

Typically, Python+NumPy is still 5 - 20 $\times$ 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.

Let's look at the timing first:

```
$ make -f Makefile_cython
...
python diff2dnumpylibsetup.py build_ext --inplace
```

```
$ time python diff2d_numpy.py > output_n.txt
Elapsed: 2.26 seconds
```

```
$ time python diff2d_numpy_cython.py > output_nc.txt
Elapsed: 2.46 seconds
```

It is still Python!

- 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 extensions that use C types.
- That would be a whole session in and of itself.

4. Parallel computing in 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
numba	just-in-time compiler for Python functions
multiprocessing	create processes that behave more like threads
mpi4py	message passing between processes
dask	task-based parallelism
ray and dask	task-based parallelism

Unavailable approaches

- Threads in Python: these are like pthreads, but even worse: they do not run simultaneously because of the global interpreted lock.
- OpenMP in Python: compiler directive-based techniques do not work since there is no compiler.

Note about using GPUs in Python

There are roughly two ways that make this possible:

1. By using packages that allow you to write CUDA-like kernels.

   We won't have time to cover that here, but check out **Numba**.

2. Using a formalism that uses GPUs in its implementation, e.g. Tensorflow.

   If a package supports this, great, use it, but it doesn't change how you use it.

The Numexpr Package

The numexpr package is useful if you're doing array 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 its input in as a string.
- In some situations using numexpr can significantly speed up your calculations.
- This is the closest thing to "OpenMP-ing a loop" in Python.

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`

Using the numexpr package

```
>>> from time import time
import numpy as np
a = np.random.rand(30000000)
b = np.random.rand(30000000)
c = np.zeros(30000000)
```

Without numexpr:

```
>>> t = time()
c = a**2 + b**2 + 2*a*b
print("Elapsed: %f seconds" % (time()-t))
```

```
Elapsed: 47.065446 seconds
```

With numexpr:

```
>>> import numexpr as ne
ne.set_num_threads(1);
t = time()
c = ne.evaluate('a**2 + b**2 + 2*a*b'); \
print("Elapsed: %f seconds" % (time()-t))
```

```
Elapsed: 11.294290 seconds
```

```
>>> ne.set_num_threads(4);
t = time();
c = ne.evaluate('a**2 + b**2 + 2*a*b');
print("Elapsed: %f seconds" % (time()-t))
```

```
Elapsed: 3.749861 seconds
```

```
>>> ne.set_num_threads(8);
t = time();
c = ne.evaluate('a**2 + b**2 + 2*a*b');
print("Elapsed: %f seconds" % (time()-t))
```

```
Elapsed: 5.495837 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 (continued)

- 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: 2.56 seconds
```

```
$ export NUMEXPR_NUM_THREADS=8
time python diff2d_numexpr.py > diff2d_numexpr.out
```

```
Elapsed: 2.21 seconds
```

- Nice, 3x speed up.
  (You may get better eve speed-up if you increase the grid, i.e., decrease dx).

Numexpr for the diffusion example (continued more)

SciNet

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))
```

The Numba Package

- 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 (i.e., not like OpenMP).
- While it can also vectorize for multi-core and GPUs with, it can only do so for specific, independent, non-sliced data.

Numba for 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:ncols+1]+dens[0:nrows+0,1:ncols+1]+dens[1:nrows+1,2:ncols
densnext[:,:] = dens + (D/dx**2)*dt*laplacian
```

```
$ time python diff2d_numpy.py >diff2d_numpy.out
```

```
Elapsed: 2.27 seconds
```

After:

```
from numba import jit
@jit(nopython=True)
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:nrows+0,1:ncols+1]+dens[1:nrows+1,2:nco
  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: 6.63 seconds
```

Why the limited performance of Numba here?

- Numba can compile more complicated code than e.g. numexpr, but this compilation takes some time.
- We already optimized the Python code by using vectorized operations.
- So the same numpy routines are called!
- For codes that aren't so easily vectorized, e.g. with complex indexed array operations, Numba can help a lot with very little code changes.

Numba for the Diffusion Equation, 2nd Try

SciNet

```
@jit(nopython=True)
def timestep(laplacian,dens,densnext,nrows,ncols,D,dx,dt):
   for i in range(1,nrows+1):
      for j in range(1,ncols+1):
         laplacian[i][j] = dens[i+1][j]+dens[i-1][j]+dens[i][j+1]+dens[i][j-1]
   for i in range(1,nrows+1):
      for j in range(1,ncols+1):
         densnext[i][j] = dens[i][j]+(D/dx**2)*dt*laplacian[i][j]
```

```
$ time python diff2d_numba_loop.py >diff2d_numba_loop.out
```

```
Elapsed: 3.18 seconds
```

That's better!

Numba for the Diffusion Equation, Parallel

We can ask numba to use multiple cores too.

It can do work-sharing of loops, much in the same way as OpenMP, if you use prange instead of range.

```
from numba import prange
@jit(nopython=True,parallel=True)
def timestep(laplacian,dens,densnext,nrows,ncols,D,dx,dt):
   for i in prange(1,nrows+1):
      for j in range(1,ncols+1):
         laplacian[i][j] = dens[i+1][j]+dens[i-1][j]+dens[i][j+1]+dens[i][j-1]
   for i in prange(1,nrows+1):
      for j in range(1,ncols+1):
         densnext[i][j] = dens[i][j]+(D/dx**2)*dt*laplacian[i][j]
```

```
time python diff2d_numba_par_loop.py >diff2d_numba_par_loop.out
```

```
Elapsed: 1.78 seconds
```

Even (somewhat) better!

Note: You may need to increase the resolution to see some of an effect.

The MPI4PY Package

MPI

- 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 is a C/Fortran Library API

Mpi4py features

- mpi4py is a Python wrapper around the MPI library
- Point-to-point communication (sends, receives)
- Collective (broadcasts, scatters, gathers) communications of any picklable Python object
- Names of functions much the same as in C/Fortran, but are methods of the communicator

Mpi4py in a nutshell

- MPI communication is govered by a communicator:

```
from mpi4py import MPI  # does MPI_Init!
comm = MPI.COMM_WORLD
```

- Every process runs the same code, the full Python script, at the same time.
- Every process has a rank, which is the only feature that distinguishes it from its siblings.

```
rank = comm.Get_rank()
```

- Processes can send values to other ranks:

```
comm.send(variable, dest=torank)
```

- Processes can receive things from other ranks:

```
variable = comm.recv(source=fromrank)
```

- Sends and receives must match or your program will hang. The combined `comm.sendrecv` can help avoid this deadlock.
- Processes can do collective actions, like summing up values:

```
result = comm.reduce(value2sum,
                     op=MPI.SUM, root=0)
```

Mpi4py

- One of the drudgeries of MPI is to have to express the binary layout of your data.
- This arises because C and Fortran don't have *introspection* and the MPI libraries cannot look inside your code.
- With Python, this is different: we can investigate, within Python, what the structure is.
- That means we can send a piece of data without having to specify types and amounts.

```
# mpi4py_right_rank.py
from mpi4py import MPI

comm  = MPI.COMM_WORLD
rank  = comm.Get_rank()
size  = comm.Get_size()
right = (rank+1)%size
left  = (rank+size-1)%size

rankr = comm.sendrecv(rank, left, source=right)

print("I am rank", rank,
      "; my right neighbour is", rankr)
```

```
$ mpirun -np 1 python mpi4py_right_rank.py
I am rank 0 ; my right neighbour is 0
```

```
$ mpirun -n 4 python mpi4py_right_rank.py
I am rank 0 ; my right neighbour is 1
I am rank 1 ; my right neighbour is 2
I am rank 3 ; my right neighbour is 0
I am rank 2 ; my right neighbour is 3
```

It's still slower than C/Fortran!

SciNet

But there is hope:

When throughput matters more

- If you have a reasonable efficient serial Python code (using **NumPy vectorization**, etc.), and you have many independent cases to compute.
- Use **multiprocessing**, or **ray**, or do it in *bash* with **GNU Parallel** O. Tange (2018): GNU Parallel 2018, March 2018, https://doi.org/10.5281/zenodo.1146014.

When doing (big) data analysis

- For reading in data, performing some analysis, and writing it out, performance is likely limited by I/O. E.g. **pyspark**.

When using optimized packages

- Many Python packages are written in C or Fortran, and just expose an interface to Python.
- Examples of this include popular *data science* and *machine learning* packages:
  pandas scipy sklearn tensorflow keras dask ray

The Multiprocessing Package

- Multiprocessing spawns separate processes that run concurrently and have their own memory.
- The Process function launches a separate process.
- The syntax is very similar to spawning threads. This is intentional.
- The details under the hood depend strongly upon the system involved (Windows, Mac, Linux), but are hidden, so your code can be portable.

```
# multiprocessingexample.py
import multiprocessing

def f(x):
   return x*x

processes = []

for x in range(1,50):
   p = multiprocessing.Process(target=f,args=(x,))
   processes.append(p)
   p.start()

for p in processes:
   p.join()
```

Work sharing with multiprocessing

The Pool object from multiprocessing offers a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes (data parallelism).

```
from multiprocessing import Pool, cpu_count

def f(x):
    return x*x

numprocs = cpu_count()

with Pool(numprocs) as p:
    print(p.map(f, range(1,50)))
```

Shared memory with multiprocessing

- `multiprocessing` 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`.

```python
# 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
```

```
485
```

```
Elapsed: 0.20 seconds
```

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.

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

Using shared memory, continued

The solution: be more explicit in your locking.

```
# 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: 0.12 seconds
```

Task-based approaches

- We saw with multiprocessing that if the individual tasks are light, it is hard to get good parallel performance.

  Those worked well with data-parallel approaches like numpy and numexpr.
- Let's consider the case that the tasks are more compute intensive.
- But let's be a bit more general, and allow dependencies between tasks.
- To do task-based parallelizing, how would we describe these dependencies?

We need a way to declare a dependency graph of tasks, and then a way to execute it with multiple workers.

The Dask Package

- An original algorithm may already show the dependencies.
- If we replaced the variables in all the steps of an algorithm with placeholders, we could figure out what could be done in parallel before compute the final result.
- dask.delayed

Example

Immediate, non parallelized:

```
def add(x,y):
   return x+y

x = add(1,2)
y = add(2,3)

z = add(x,y)

print("z is",z)
```

Task-graph, executed in parallel:

```
import dask

def add(x,y):
   return x+y

x = dask.delayed(add)(1,2)
y = dask.delayed(add)(2,3)

z = dask.delayed(add)(x,y)

print("z is",z.compute())
```

Dask, continued

```
import dask

def add(x,y):
   return x+y

x = dask.delayed(add)(1,2)
y = dask.delayed(add)(2,3)

z = dask.delayed(add)(x,y)

print("z is",z.compute())
```

```
z is 8
```

- x is not a number, but a 'Delayed' object
- This just defines what should be done, with arguments that become dependencies
- dask build the dependency tree.
- It does not execute until you use the compute method.

Main features of dask

- Parallel computing
- Providing data structures that are extensions of familiar object: DataFrames, Array, and Bag
- Task scheduling on-node (e.g. using multiprocessing) or distributed
- Scalable
- Dynamics Task Graphs
- Diagnostic Tools
- Works well with numpy, scipy, scikit-learn, etc.

The Ray Package

- Ray is another 'task graph' approach to parallelism.
- Where dask is aimed at data structures, `ray` is more general
- It allows e.g. stateful actors and runtime added tasks.
- It is reportedly optimized for low latency.

Ray Example

```
# ray basics example
import ray

ray.init()

@ray.remote
def add(x,y):
   return x+y

x = add.remote(1,2)
y = add.remote(2,3)

z = add.remote(x,y)

print("z is",ray.get(z))

ray.shutdown()
```

```
z is 8
```

- You need to explicitly start and stop the 'ray cluster'.
- Ray works with decorators.
- The 'delayed' actions are done using `.remote(...)`
- To get the result, you do `ray.get(...)`

This does not really show a difference with dask.

The level of detailed control you need and the presence of specialized functionality, e.g. machine learning for Ray, data manipulation like with numpy or pandas for Dask.

5. Conclusions

Conclusions

Performance

- Getting performance out of Python involves getting out of Python
- Find your performance with scalene or line_profiler before optimizing.
- Numpy, when used with vectorized expressions helps.
  Then numexpr can help even more.
- Numba, when not used with vectorized expressions helps.

Parallel computing

- Numexpr for the simplest cases
- Numba for more complex cases (incl. GPUs)
- For non-lightweight tasks, multiprocessing.
- mpi4py is an option, but not easy with task dependencies.
- Dask or Ray for workflows with dependencies (dask for data analysis and ray for machine learning)