Python Programming for HPC

Ramses van Zon

IHPCSS, Atlanta, GA - July 12, 2023

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Preliminaries

In this session...

- Performance and Python
- Profiling tools for Python
- Fast arrays for Python: Numpy
- Numexpr
- Numba
- Mpi4py

Packages and code

Requirements for this session

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

- numpy
- scipy
- matplotlib
- numexpr
- scalene
- mpi4py
- cython
- numba

But you should work on Bridges2

Code and installation can be copied from my folder on Bridges2. It can be found in the directory `/jet/home/rzon/hpcpycode`

Setting up for this session

To get set up for today's session, 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:

```
$ interact -n 8 -t 2:00:00
```

4. Setup the environment:

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

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

Introduction

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 lean, very expressive, and, not surprisingly, very popular.
- But development in Python can be substantially easier (and thus faster) than when using compiled languages.

Why isn't Python 'high performance'?

Python is interpreted

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

Python is dynamically typed

- 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 interferes with optimal streaming of data to processor, which interferes with maximum performance.

Example 1: 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 $\varrho(x,y,t) = \varrho_0(x,y)$.

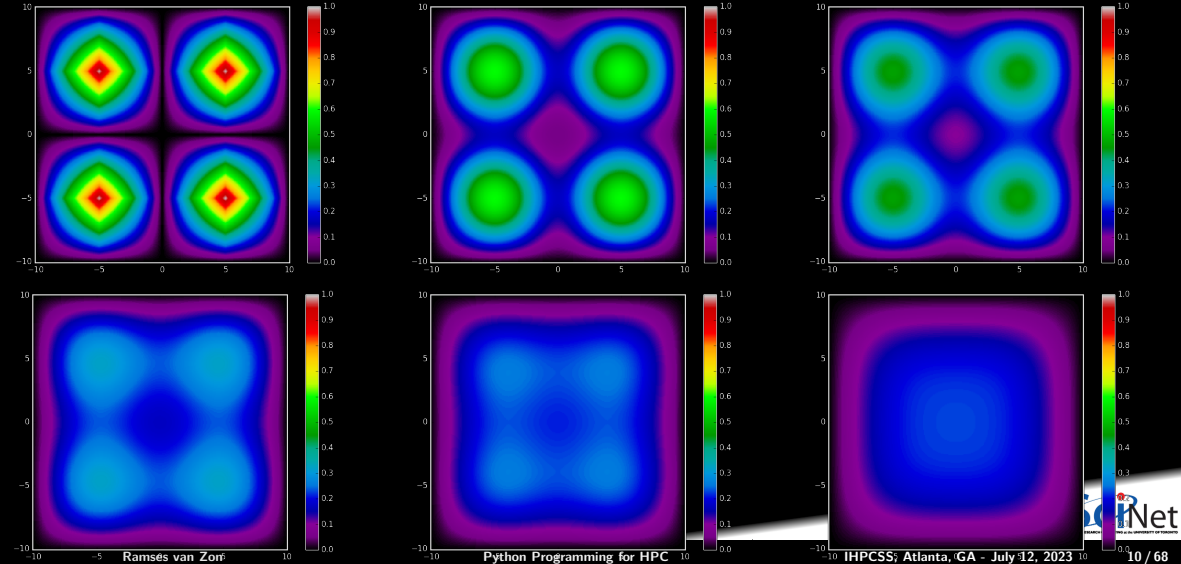
Here:

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

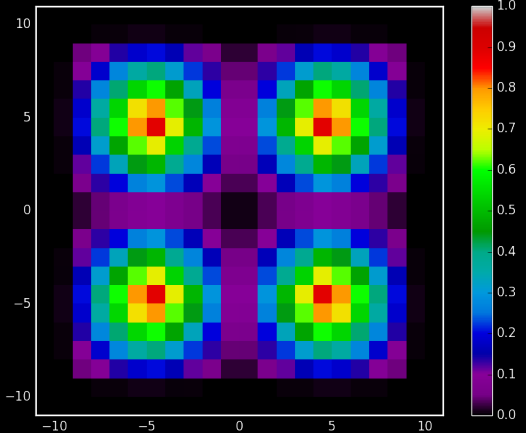
By the way: the programming challenge solves the stationary state of this same equation, but with a different initial and "boundary" condition.

Example 1: 2D Diffusion, Result

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



Example 1: 2D Diffusion, the Algorithm

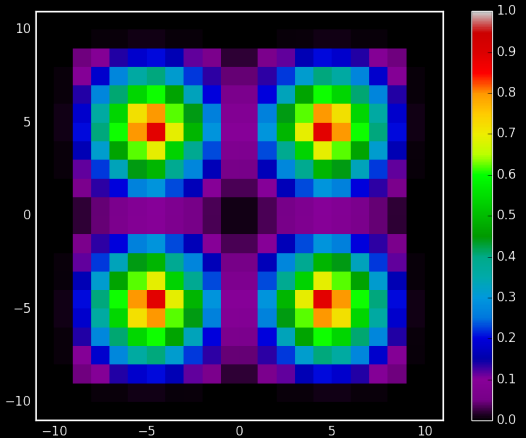


- Discretize space in both directions (points dx apart)
- Replace derivatives with finite differences.
- Explicit finite time stepping scheme (time step set by dx)
- For graphics: Matplotlib for Python, pgplot for C++/Fortran, every `outtime` time units

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
gfortran -c -O3 -march=native -o pgplot90.o pgplot90.f90
...
gfortran -O3 -march=native -o diff2d_f90.ex diff2d_f90.o diff2dplot_f90.o pgplot90.o -lcpgplot -lpgplot -
Elapsed: 1.19 seconds
```

```
$ time ./diff2d_cpp.ex > output_c.txt
Elapsed: 0.49 seconds
$ time ./diff2d_f90.ex > output_f.txt
Elapsed: 0.41 seconds
$ time python diff2d.py > output_p.txt
Elapsed: 395.00 seconds
```

The Python version is 240× slower than the compiled versions.

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

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 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_
 if first: plt.colorbar()
 plt.show();plt.pause(0.1)
```

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.

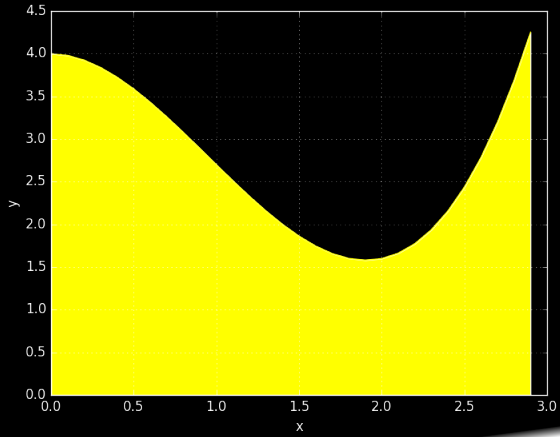
Example 2: Area Under the Curve

- 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 $b = 327/40 = 8.175$
- 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.



Example 2: Area Under the Curve, Codes

C++

```cpp
// auc_serial.cpp
#include <iostream>
#include <cmath>
int main(int argc, char** argv)
{
  size_t ntot = atoi(argv[1]);
  double width = 3.0;
  double dx = width/ntot;
  double x = 0, y;
  double a = 0.0;
  for (size_t i=0; i<ntot; ++i) {
    y = 0.7*x*x*x - 2*x*x + 4;
    a += y*dx;
    x += dx;
  }
  std::cout << "The area is "
            << a << std::endl;
}
```

Fortran

```fortran
program auc_serial
 implicit none
 integer :: i, ntot
 character(64) :: arg
 double precision :: dx,width,x,y,a
 call get_command_argument(1,arg)
 read (arg,'(i40)') ntot
 width = 3.0
 dx = width/ntot
 x = 0.0
 a = 0.0
 do i = 1,ntot
     y = 0.7*x**3 - 2*x**2 + 4
     a = a + y*dx
     x = x + dx
 end do
 print *, "The area is " , a
end program
```

Python

```python
# auc_serial.py
import sys

def main():
    ntot  = int(sys.argv[1])
    width = 3.0
    dx    = width/ntot
    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 %f"%a)

if __name__ == "__main__":
  main()
```

Example 2: Area Under the Curve, Initial Timing

```
$ time make auc_serial_cpp.ex auc_serial_f90.ex
g++ -O3 -march=native   -c -o auc_serial.o auc_serial.cpp
g++ -O3 -march=native -o auc_serial_cpp.ex auc_serial.o
gfortran -c -O3 -march=native -o auc_serial_f90.o auc_serial.f90
gfortran -O3 -march=native -o auc_serial_f90.ex auc_serial_f90.o -lcpgplot -lpgplot -lX11 -lxcb -ldl -lXa
Elapsed: 0.29 seconds
```

```
$ time ./auc_serial_cpp.ex 70000000
The area is 8.175
Elapsed: 0.10 seconds
$ time ./auc_serial_f90.ex 70000000
 The area is    8.1749993888506776
Elapsed: 0.11 seconds
$ time python auc_serial.py 70000000
The area is 8.175000
Elapsed: 35.01 seconds
```

Here, Python is about 80× slower than compiled when adding in compilation time.

We want better performance. Where do we start?

Performance Tuning Tools for Python

Wallclock 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 **wallclock performance** first, as measured by the time the computation requires.

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`.

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>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Memory usage

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 unexpectedly 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.

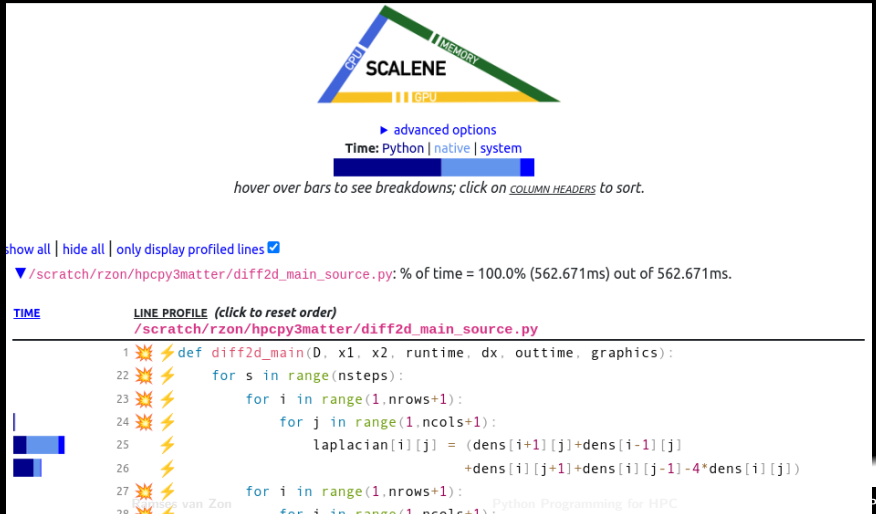
Scalene

- Python Profiler for CPU, memory, and GPU.
- Fast
- Accurate
- Distinguished Python from C code.

Scalene Usage

```
$ scalene diff2d.py
```

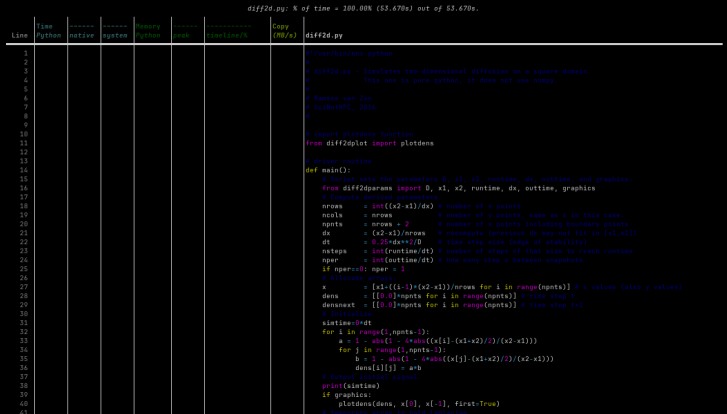
On your local computer, this would launch the result in a browser:



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
0.0
...
```



Hands-on: Profiling (10 mins)

Profile the auc_serial.py code

- Consider the Python code for computing the area under the curve, `auc_serial.py`.
- Run this through the Scalene profiler and see what line(s) cause the most cpu usage.
- Is there any memory usage?

Profile the diff2d.py code (optional)

- 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`.
- Run this through the Scalene profiler and see what line(s) cause the most cpu usage.

NumPy: Faster 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]
```

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

```
>>> from numpy import linspace
>>> linspace(1,5)
array([1.        , 1.08163265, 1.16326531, 1.24489796, 1.32653061,
       1.40816327, 1.48979592, 1.57142857, 1.65306122, 1.73469388,
       1.81632653, 1.89795918, 1.97959184, 2.06122449, 2.14285714,
       2.2244898 , 2.30612245, 2.3877551 , 2.46938776, 2.55102041,
       2.63265306, 2.71428571, 2.79591837, 2.87755102, 2.95918367,
       3.04081633, 3.12244898, 3.20408163, 3.28571429, 3.36734694,
       3.44897959, 3.53061224, 3.6122449 , 3.69387755, 3.7755102 ,
       3.85714286, 3.93877551, 4.02040816, 4.10204082, 4.18367347,
       4.26530612, 4.34693878, 4.42857143, 4.51020408, 4.59183673,
       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)
>>> 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.])
>>> 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]])
>>> 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]])
>>> 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]])
>>> b = np.array([[1,2],
...               [4,3]])
>>> a
array([[1, 2],
       [4, 3]])
>>> a * b
array([[ 1,  4],
       [16,  9]])
>>> a @ b
array([[ 9,  8],
       [16, 17]])
```

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

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

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: 395.00 seconds
```

NumPy implementation:

```
$ time python diff2d_slow_numpy.py > output_n.txt
Elapsed: 1027.45 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 Fortan functions by NumPy.

Hands-on: Vectorizing Python code (15 minutes)

Vectorize the auc_serial.py code

- Copy the Python code for computing the area under the curve to a new file `auc_numpy.py`.
- Reexpress the code using NumPy arrays.
- Make sure you are using vectorized operations.
- Measure the speed-up (if any) with the `time` command.

Vectorize the slow NumPy code (optional)

If you are done with the auc example, try this:

- 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?

Diffusion example:

Pure Python implementation:

```
$ time python diff2d.py > output_p.txt
Elapsed: 395.00 seconds
```

NumPy vectorized implementation:

```
$ time python diff2d_numpy.py > output_n.txt
Elapsed: 2.20 seconds
```

Yeah! 180× speed-up

Area-under-the-curve example:

Pure Python implementation:

```
$ time python auc_serial.py 70000000
The area is 8.175000
Elapsed: 33.84 seconds
```

NumPy vectorized implementation:

```
$ time python auc_numpy.py 70000000
The area is 8.175000
Elapsed: 3.29 seconds
```

11× speed-up

Note: Is this really vectorization?

- We call this vectorization because the code works on whole vectors.
- But this is different from 'vectorization' which uses the 'small vector units' or 'simd units' on the cpu.
- We're just minimizing the number of lines Python needs to interpret and trusting NumPy to call C or Fortran functions.
- Those functions may or may not use simd, this depends on how NumPy was compiled and linked.
- Regardless, the benefits of using Python vectorization with NumPy are impressive.

Reality check: NumPy vs. compiled code

Diffusion example:

NumPy, vectorized implementation:

```
$ time python diff2d_numpy.py > output_n.txt
Elapsed: 2.20 seconds
```

Compiled versions:

```
$ time ./diff2d_cpp.ex > output_c.txt
Elapsed: 0.51 seconds
$ time ./diff2d_f90.ex > output_f.txt
Elapsed: 0.41 seconds
```

Area-under-the-curve example:

NumPy, vectorized implementation:

```
$ time python auc_numpy.py 70000000
The area is 8.175000
Elapsed: 3.29 seconds
```

Compiled versions:

```
$ time ./auc_serial_cpp.ex 70000000
The area is 8.175
Elapsed: 0.10 seconds
$ time ./auc_serial_f90.ex 70000000
 The area is    8.1749993888506776
Elapsed: 0.11 seconds
```

So 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:

```
$ time make -f Makefile_cython
make[1]: Entering directory '/jet/home/rzon/python-
python diff2dnumpylibsetup.py build_ext --inplace
running build_ext
make[1]: Leaving directory '/jet/home/rzon/python-i
Elapsed: 0.27 seconds
$ time python diff2d_numpy.py > output_n.txt
Elapsed: 2.17 seconds
$ time python diff2d_numpy_cython.py > output_nc.tx
Elapsed: 2.38 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 extentions that use C types.
- That would be a whole session in and of itself.

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
numba	just-in-time compiler for Python functions
mpi4py	message passing between processes

Unavailable approaches

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

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.

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 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`

- Supported reductions (but often slow):

  `sum, product`

Using the numexpr package

Without numexpr:

```
>>> from time import time
>>> import numpy as np
>>> def etime(t):
...     print("Elapsed %f seconds" % (time()-t))
...
>>> a = np.random.rand(3000000)
>>> b = np.random.rand(3000000)
>>> c = np.zeros(3000000)
>>> t = time();                \
... c = a**2 + b**2 + 2*a*b; \
... etime(t)
Elapsed 0.016991 seconds
```

With numexpr:

```
>>> from time import time
>>> import numpy as np, numexpr as ne
>>> def etime(t):
...     print("Elapsed %f seconds" % (time()-t))
...
>>> a = np.random.rand(3000000)
>>> b = np.random.rand(3000000)
>>> c = np.zeros(3000000)
>>> old = ne.set_num_threads(1)
>>> t = time();                                \
... c = ne.evaluate('a**2 + b**2 + 2*a*b'); \
... etime(t)
Elapsed 0.014807 seconds
>>> old = ne.set_num_threads(4)
>>> t = time();                                \
... c = ne.evaluate('a**2 + b**2 + 2*a*b'); \
... etime(t)
Elapsed 0.004122 seconds
>>> old = ne.set_num_threads(8)
>>> t = time();                                \
... c = ne.evaluate('a**2 + b**2 + 2*a*b'); \
... etime(t)
Elapsed 0.004093 seconds
```

Hands-on: Parallelize Area-under-the-curve (10 mins)

- Take your NumPy vectorized `auc_numpy.py` and copy it to a new Python script `auc_numexpr.py`.
- Use `numexpr` to parallelize the `auc_numexpr.py` code.
- Measure the speed-up using up to 8 threads.

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: 2.22 seconds
$ export NUMEXPR_NUM_THREADS=8
$ time python diff2d_numexpr.py > diff2d_numexpr.out
Elapsed: 1.38 seconds
```

- You'll get better speed-up if you increase the grid (i.e., decrease dx).

Numexpr for the diffusion example (continued more)

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))
```

Numba

Another compiler-within-an-interpreter: 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 (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: 2.26 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: 6.27 seconds
```

Why the low performance of Numba for this case?

- 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.
- 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

```
@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: 4.37 seconds
```

That's better!

Numba for the Diffusion Equation, Parallel

We can ask numba to use multiple cores too.

It can do worksharing 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: 1.93 seconds
```

Even better!

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'.
- Need to be explicit for all data movement.

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,
- 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).

Mpi4py in a nutshell

- MPI communication is govered by a communicator:

```
from mpi4py import MPI
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:

```
comm.recv(variable, 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:

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

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

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,"; my right neighbour is",rankr)
```

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

Hands-on: MPI area under the curve (10-15 mins)

Parallelize auc_numpy.py

- Take your NumPy vectorized `auc_numpy.py` and copy it to a new Python script `auc_mpi4.py`.
- In this new script, divide the work over mpi tasks
- Measure speed-up for up to 8 processes.

Note: 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.
- For the area-under-the curve, it turns out there is no advantage.

It's still slower than C/Fortran. Is there no hope for Python in HPC?

That's right, it is hopeless.

Just kidding!

There is hope, for certain fairly common cases

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. See *Big Data* session, **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 sklearn tensorflow keras dask