HPC Python Programming

Ramses van Zon

August 22, 2019

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

In this session...

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

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
- mpi4py
- cython
- numba

Get the code and setup files on Blue Waters, Cori or Stampede2

The hands-on codes have been uploaded to the supercomputers you've been given access to. They can be found in the following directories:

```
Cori:          /global/homes/r/rzon/hpcpython
Blue Waters:   /u/training/instr041/hpcpython
Stampede2:     /home1/06674/tg859732/hpcpython
```

Setting up for this morning's class

Cori

1. Login to Cori:

```
$ ssh USER@cori.nersc.gov
```

2. Copy code:

```
$ cp -r ~rzon/hpcpython $HOME
```

3. Request resources on a compute node:

```
$ salloc -n 8 -t 2:00:00 -q shared
    --reservation=petascale \
    -C haswell
```

4. Setup the environment:

```
$ . ~rzon/hpcpyenv/activate
```

Blue Waters

1. Login to Blue Waters:

```
$ ssh USER@bwbay.ncsa.illinois.edu
```

2. Copy code:

```
$ cp -r ~instr041/hpcpython $HOME
```

3. Request resources on a compute node:

```
$ qsub -I -X -lnodes=1:ppn=8 \
    -lwalltime=2:00:00
```

4. Setup the environment:

```
$ . ~instr041/hpcpyenv/activate
```

Stampede2

1. Login to Stampede2:

```
$ ssh USER@login.xsede.org
$ gsissh stampede2
```

2. Copy code:

```
$ cp -r ~tg859732/hpcpython $HOME
```

3. Request resources on a compute node:

```
$ idev -p skx-dev -N 1 -n 8 -m 120
```

4. Setup the environment:

```
$ . ~tg859732/hpcpyenv/activate
```

We have a reservation for Cori for this session, so that's probably to be preferred. Timings in this slides came from Blue Waters runs.

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.

Why isn't Python "high performance"?

Interpreted language:

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

Dynamic language:

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

Example: 2D diffusion equation

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

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

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

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

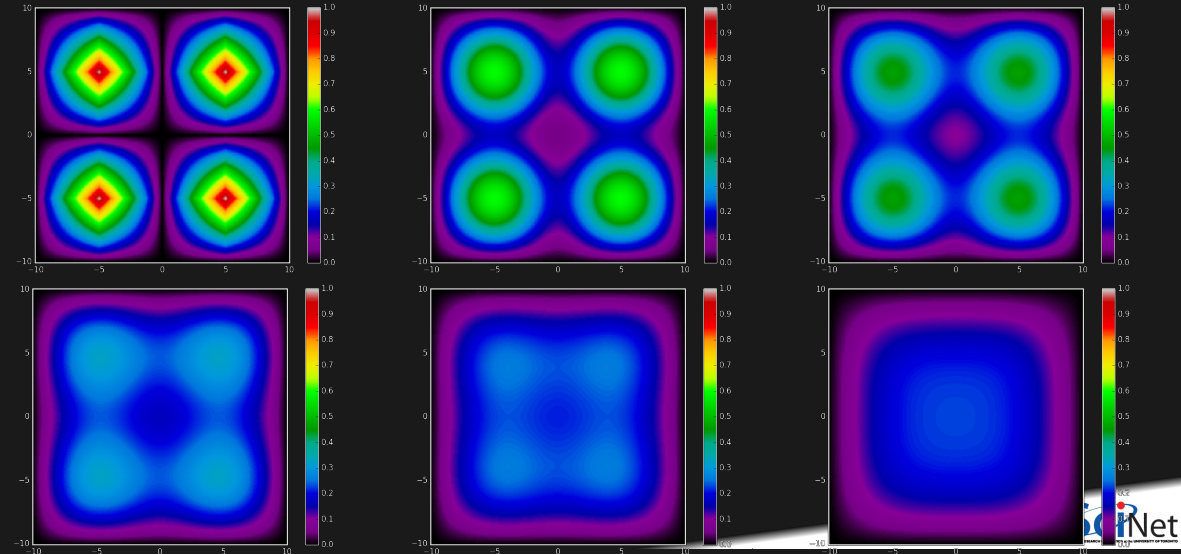
with some given initial condition $p(x,y,t) = p_0(x,y)$.

Here:

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

Example: 2D diffusion, result

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

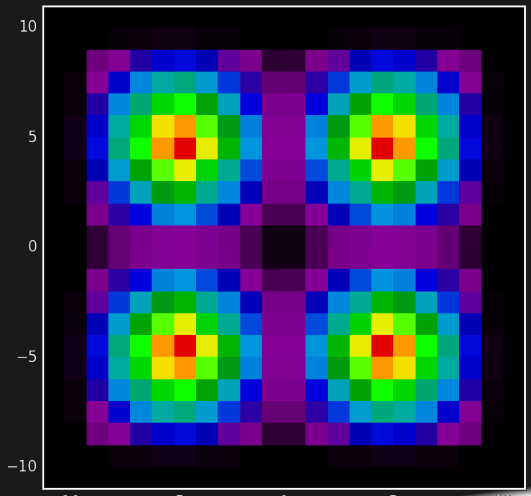


Example: 2D diffusion, algorithm

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

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

```
D        =   1.0;
x1       = -10.0;
x2       =  10.0;
runtime  =  15.0;
dx       =  0.1;
outtime  =  0.5;
graphics =  False;
```



Example: 2D diffusion, performance on Blue Waters

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

```
$ /usr/bin/time make diff2d_cpp.ex diff2d_f90.ex
CC -c -Ihpcpyenv/include -DRA_SKIPINTERMEDIATE -O3 -o diff2d_cpp.o diff2d.cpp
ftn -c -Ihpcpyenv/include -O3 -o pgplot90.o pgplot90.f90
...
ftn -Lhpcpyenv/lib -O3 -o diff2d_f90.ex diff2d_f90.o diff2dplot_f90.o pgplot90.o -lcpgplot -lpgplot -lX11
Elapsed: 9.11 seconds
```

```
$ /usr/bin/time ./diff2d_cpp.ex > out_c.txt
Elapsed: 1.03 seconds
$ /usr/bin/time ./diff2d_f90.ex > out_f.txt
Elapsed: 1.26 seconds
$ /usr/bin/time python diff2d.py > out_n.txt
Elapsed: 642.82 seconds
```

The Python version runs 500× 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_spectral')
 if first: plt.colorbar()
```

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.

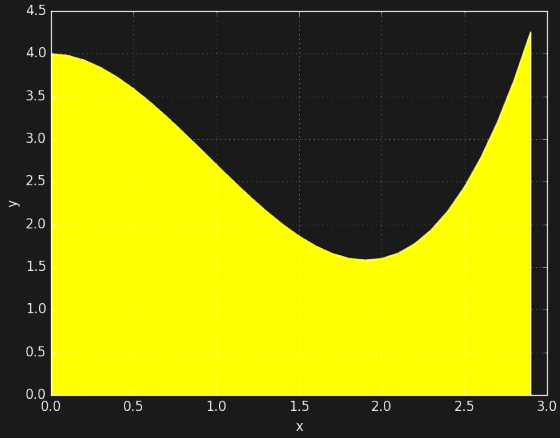
Simpler Example: Area Under the Curve

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

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

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

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



Simpler Example: 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
  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


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)
```

Simpler Example: Area Under the Curve, Initial Timing

```
$ /usr/bin/time make auc_serial_cpp.ex auc_serial_f90.ex
CC -O3  -c -o auc_serial.o auc_serial.cpp
CC -O3 -o auc_serial_cpp.ex auc_serial.o
ftn -c -O3 -o auc_serial_f90.o auc_serial.f90
ftn -O3 -o auc_serial_f90.ex auc_serial_f90.o
Elapsed: 5.28 seconds
```

```
$ /usr/bin/time ./auc_serial_cpp.ex 30000000
The area is 8.175
Elapsed: 0.64 seconds
$ /usr/bin/time ./auc_serial_f90.ex 30000000
 The area is  8.1749998853868746
Elapsed: 0.55 seconds
$ /usr/bin/time python auc_serial.py 30000000
The area is 8.175000
Elapsed: 62.49 seconds
```

Here, Python is about 12× slower than compiled when adding in compilation time.

We want better performance. Where do we start?

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.

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 is package called `line_profiler`

cProfile

- Use `cProfile` 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}
```

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: 0.14838 s
File: profileme.py
Function: profilewrapper at line 2

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     2                                           @profile
     3                                           def profilewrapper():
     4         1     129565.0 129565.0     87.3    x=[1.0]*(2048*2048)
     5         1         27.0     27.0      0.0    a=str(x[0])
     6         1          7.0      7.0      0.0    a+="\nis a one\n"
     7         1      18731.0  18731.0     12.6    del x
     8         1         50.0     50.0      0.0    print(a)
```

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 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.

memory_profiler

- This module/utility monitors the Python memory usage and its changes throughout the run.
- Good for catching memory leaks and unexpectedly large memory usage.
- Needs same instrumentation as line profiler.

memory_profiler, details

Your decorated script is usable by memory profiler.

You run your script through the profiler with the command

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

```
1.0
is a one

Filename: profileme.py

Line #    Mem usage    Increment   Line Contents
================================================
     2   32.664 MiB    0.000 MiB   @profile
     3                             def profilewrapper():
     4   64.629 MiB   31.965 MiB     x=[1.0]*(2048*2048)
     5   64.629 MiB    0.000 MiB     a=str(x[0])
     6   64.629 MiB    0.000 MiB     a+="\nis a one\n"
     7   32.727 MiB  -31.902 MiB     del x
     8   32.727 MiB    0.000 MiB     print(a)
```

Hands-on: Profiling (5-10 mins)

Profile the auc_serial.py code

- Consider the Python code for computing the area under the curve.
- Put the code in a wrapper function.
  Make sure it still works!
- Add @profile to the main function.
- Run this through the line profiler and see what line(s) cause the most cpu usage.

Profile the diff2d.py code (homework)

- 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 the line 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.

```
>>> 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

- Arrays are a much better choice, but are not a native Python data type.
- Almost everything that you want to do starts with NumPy.
- Contains arrays of various types and forms: zeros, ones, linspace, etc.

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

```
>>> from numpy import arange
>>> from numpy import linspace
>>> arange(5)
array([0, 1, 2, 3, 4])
>>> linspace(1,5)
array([1.        , 1.08163265, 1.16326531, 1.2448979
       1.40816327, 1.48979592, 1.57142857, 1.6530612
       1.81632653, 1.89795918, 1.97959184, 2.0612244
       2.2244898 , 2.30612245, 2.3877551 , 2.4693877
       2.63265306, 2.71428571, 2.79591837, 2.8775510
       3.04081633, 3.12244898, 3.20408163, 3.2857142
       3.44897959, 3.53061224, 3.6122449 , 3.6938775
       3.85714286, 3.93877551, 4.02040816, 4.1020408
       4.26530612, 4.34693878, 4.42857143, 4.5102040
       4.67346939, 4.75510204, 4.83673469, 4.9183673
>>> 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-vector multiplication with * give a kind-of element-by-element multiplication
- For a linear-algebra-stype 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 `@`.

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

```
>>> 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]])
```

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

Does changing to numpy really help?

Let's return to our 2D diffusion example.

Pure Python implementation:

```
$ /usr/bin/time python diff2d.py > out_p.txt
```

```
Elapsed: 642.82 seconds
```

Numpy implementation:

```
$ /usr/bin/time python diff2d_slow_numpy.py > out_n.txt
```

```
Elapsed: 1678.97 seconds
```

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

Let's inspect the code

```
#diff2d.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!

```
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])
```

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 element-wise '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 should 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 should write:

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

Hands-on: Vectorizing (10 mins)

Vectorize the auc_serial.py code

- Copy `auc_serial.py` to `auc_vector.py`.
- Remove any `@profile` decorators.
- Reexpress the code using numpy arrays.
- Make sure you are using vectorized operations.
- Measure the speed-up (if any) with `/usr/bin/time`.

Vectorize the slow numpy code (homework)

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:

```
$ /usr/bin/time python diff2d.py > out_p.txt
Elapsed: 642.82 seconds
```

Numpy vectorized implementation:

```
$ /usr/bin/time python diff2d_numpy.py > out_n.txt
Elapsed: 5.63 seconds
```

Yeah! 100× speed-up

Area-under-the-curve example:

Pure Python implementation:

```
$ /usr/bin/time python auc_serial.py 30000000
The area is 8.175000
Elapsed: 63.45 seconds
```

Numpy vectorized implementation:

```
$ /usr/bin/time python auc_vector.py 30000000
The area is 8.175000
Elapsed: 4.76 seconds
```

17× speed-up

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' or 'simd units' on the cpu. We're just minimizing the number of lines Python needs to interpret.

Reality check: NumPy vs. compiled code

Diffusion example:

Numpy, vectorized implementation:

```
$ /usr/bin/time python diff2d_numpy.py > out_n.txt
Elapsed: 5.63 seconds
```

Compiled versions:

```
$ /usr/bin/time ./diff2d_cpp.ex > out_c.txt
Elapsed: 1.08 seconds
$ /usr/bin/time ./diff2d_f90.ex > out_f.txt
Elapsed: 1.30 seconds
```

Area-under-the-curve example:

Numpy, vectorized implementation:

```
$ /usr/bin/time python auc_vector.py 30000000
The area is 8.175000
Elapsed: 4.78 seconds
```

Compiled versions:

```
$ /usr/bin/time ./auc_serial_cpp.ex 30000000
$ /usr/bin/time ./auc_serial_f90.ex 30000000
 The area is  8.1749998853868746
Elapsed: 0.05 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:

```
$ unzip onwiththeshow
$ /usr/bin/time make -f Makefile_cython diff2dnumpylib.so
python diff2dnumpylibsetup.py build_ext --inplace
running build_ext
Elapsed: 1.63 seconds
$ /usr/bin/time python diff2d_numpy.py > out_n.txt
Elapsed: 5.55 seconds
$ /usr/bin/time python diff2d_numpy_cython.py > out_nc.txt
Elapsed: 9.32 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. 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
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.
- In some situations, numexpr can significantly speed up your calculations.

Note: While Python does have threads, there is no convenient OpenMP launching of threads. Event worse: threads running Python do not use multiple cpu cores because of the 'global interpreter lock'.

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`

  (though numpy's `sum` and `product` can often be faster)

Using the numexpr package

Without numexpr:

```
>>> from time import time
>>> def etime(t):
...     print("Elapsed %f seconds" % (time()-t))
...
>>> import numpy as np
>>> 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 0.110790 seconds
```

With numexpr:

```
>>> import numexpr as ne
>>> ne.set_num_threads(1)
>>> t = time();                                  \
... c = ne.evaluate('a**2 + b**2 + 2*a*b'); \
... etime(t)
Elapsed 0.058771 seconds
>>> ne.set_num_threads(4)
>>> t = time();                                  \
... c = ne.evaluate('a**2 + b**2 + 2*a*b'); \
... etime(t)
Elapsed 0.019996 seconds
>>> ne.set_num_threads(8)
>>> t = time();                                  \
... c = ne.evaluate('a**2 + b**2 + 2*a*b'); \
... etime(t)
Elapsed 0.012083 seconds
```

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

Use numexpr to parallelize the area-under-the curve

- Copy `auc_vector.py` to `auc_numexpr.py`.
- Parallelize this code using `numexpr`.
- 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.

```
$ /usr/bin/time python diff2d_numpy.py > diff2d_numpy.out
Elapsed: 5.63 seconds
$ export NUMEXPR_NUM_THREADS=8
$ /usr/bin/time python diff2d_numexpr.py > diff2d_numexpr.out
Elapsed: 4.72 seconds
```

- Check `diff2d_numexpr.py` for details on how to do this.

How about 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 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
```

```
$ /usr/bin/time python diff2d_numpy.py >diff2d_numpy.out
Elapsed: 5.48 seconds
```

After:

```
from numba import njit
@njit
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)
```

```
$ /usr/bin/time python diff2d_numba.py >diff2d_numba.out
Elapsed: 11.85 seconds
```

Multiprocessing

The multiprocessing module in a nutshell

- 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.

```
# multiprocessingexample.py
import multiprocessing

def f(x):
    return x*x

processes = []

for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    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
import os

def f(x):
    return x*x

if 'SLURM_NPROCS' in os.environ:
    numprocs = int(os.environ['SLURM_NPROCS'])
else:
    numprocs = cpu_count()

with Pool(numprocs) as p:
    print(p.map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]))
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

But there's more!

The multiprocessing module is loaded with functionality. Other features include:

- Seamlessly share memory between processes. This is done using 'Value' and 'Array'.

  This makes *race condition* bug possible.
- 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

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.
- https://mpi4py.readthedocs.io/en/stable

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 of that distinguises 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.
```
rvar = comm.sendrecv(svar,dest=torank,
                     source=fromtag)
```
- 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.
- 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)
```

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

Parallelize auc_vector.py

- take your numpy vectorized `auc_vector.py`
- 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.

What about GPUs?

There are several options to use GPUs in Python.

- PyCUDA: you're writing cuda kernels that are callable from CUDA
- Numba: you're writing more pythonic cuda kernels that are callable from CUDA
- Cupy: replace your numpy array with arrays that live on the GPU. If your code uses vectorized numpy expressions, using `cupy` can be an easy gain.
- `Tensorflow` and other AI packages can often use GPUs in the backend.

Is there no hope for Python in HPC, 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 performance penalty of Python may be insignificant.
- If this data is big, consider NetCDF4 or HDF5.
- If your data is more-or-less tabular, consider `pandas`.
- When you have large number of tasks that could be done in parallel, look into `multiprocessing` or `dask` (the latter can deal with task dependencies as well).

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 + keras