

Python for High Performance Computing

Pawel Pomorski
SHARCNET
University of Waterloo
ppomorsk@sharcnet.ca

May 26, 2015

Outline

- ▶ Speeding up Python code with NumPy
- ▶ Speeding up Python code with Cython
- ▶ Speeding up Python code via multiprocessing
- ▶ Using MPI with Python, via mpi4py

What is Python?

- ▶ Python is a programming language that appeared in 1991
- ▶ compare with Fortran (1957), C (1972), C++ (1983),
- ▶ While the older languages still dominate High Performance Computing (HPC), popularity of Python is growing

Python advantages

- ▶ Designed from the start for better code readability
- ▶ Allows expression of concepts in fewer lines of code
- ▶ Has dynamic type system, variables do not have to be declared
- ▶ Has automatic memory management
- ▶ Has large number of easily accessible, extensive libraries (eg. NumPy, SciPy)
- ▶ All this makes developing new codes easier

Python disadvantages

- ▶ Python is generally slower than compiled languages like C, C++ and Fortran
- ▶ Complex technical causes include dynamic typing and the fact that Python is interpreted, not compiled
- ▶ This does not matter much for a small desktop program that runs quickly.
- ▶ However, this will matter a lot in a High Performance Computing environment.
- ▶ Python use in HPC parallel environments is relatively recent, hence parallel techniques less well known
- ▶ Rest of this talk will describe approaches to ensure your Python code runs reasonably fast and in parallel

1D diffusion equation

To describe the dynamics of some quantity $u(x,t)$ (eg. heat) undergoing diffusion, use:

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2}$$

Problem: given some initial condition $u(x,t=0)$, determine time evolution of u and obtain $u(x,t)$

Use finite difference with Euler method for time evolution

$$u(i\Delta x, (m+1)\Delta t) = u(i\Delta x, m\Delta t) + \frac{\kappa\Delta t}{\Delta x^2} \left[u((i+1)\Delta x, m\Delta t) + u((i-1)\Delta x, m\Delta t) - 2u(i\Delta x, m\Delta t) \right]$$

C code

```
1 #include <math.h>
2 #include <stdio.h>
3
4 int main(){
5
6     int const n=100000, niter=500;
7
8     double x[n], u[n], udt[n];
9     int i, iter;
10    double dx=1.0;
11    double kappa=0.1;
12
13    for (i=0; i<n; i++){
14        u[i]=exp(-pow(dx*(i-n/2.0), 2.0)/100000.0);
15        udt[i]=0.0;
16    }
17
18    ...
```

C code continued :

```
1 ...
2 for (iter=0;iter<niter;iter++){
3     for (i=1;i<n-1;i++){
4         udt[i]=u[i]+kappa*(u[i+1]+u[i-1]-2*u[i]);
5     }
6     for (i=0;i<n;i++){
7         u[i]=udt[i];
8     }
9 }
10 return 0;
11 }
```


Program output

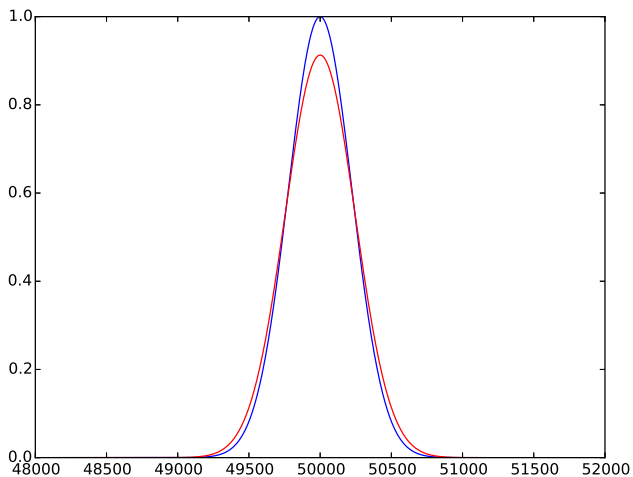


Figure: Evolution of $u(x)$ after 50,000 time steps (blue line initial, red line final)

NumPy

- ▶ To implement the same in Python, will need support for efficient, large numerical arrays
- ▶ These provided by NumPy, an extension to Python
- ▶ NumPy (<http://www.numpy.org/>) along with SciPy (<http://www.scipy.org/>) provide a large set of easily accessible libraries which make Python so attractive to the scientific community

"Vanilla" Python code

```
1 import numpy as np
2 n=100000 ; dx=1.0 ; niter=500 ; kappa=0.1
3 x=np.arange(n, dtype="float64")
4 u=np.empty(n, dtype="float64")
5 udt=np.empty(n, dtype="float64")
6
7 for i in xrange(len(u)):
8     u[i]=np.exp( -(dx*(i-n/2))**2/100000)
9
10 udt[:]=0.0
11
12 for i in xrange(niter):
13
14     for i in xrange(1,n-1):
15         udt[i]=u[i]+kappa*(u[i+1]+u[i-1]-2*u[i])
16
17     for i in xrange(len(u)):
18         u[i]=udt[i]
```

Vanilla code performance

- ▶ 500 iterations, tested on Macbook Pro laptop (2011)
- ▶ C code compiled with GCC takes 0.21 seconds
- ▶ Python "vanilla" code takes 64.75 seconds
- ▶ Python is much slower (by factor 308)
- ▶ Even though we are using Python arrays, code is slow because loops are explicit
- ▶ Must use NumPy array operations instead
- ▶ The difficulty of eliminating loops varies.

Slicing NumPy arrays :

```
1 sharcnet1:~ pawelpomorski$ python
2 Python 2.7.9 (default, Dec 12 2014, 12:40:21)
3 [GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.56)]
   on darwin
4 Type "help", "copyright", "credits" or "license" for
   more information.
5 >>> import numpy as np
6 >>> a=np.arange(10)
7 >>> a
8 array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
9 >>> a[1:-1]
10 array([1, 2, 3, 4, 5, 6, 7, 8])
11 >>> a[0:-2]
12 array([0, 1, 2, 3, 4, 5, 6, 7])
13 >>> a[1:-1]+a[0:-2]
14 array([ 1,  3,  5,  7,  9, 11, 13, 15])
15 >>>
```

NumPy vector operations

Replace explicit loops

```
1 for i in xrange(1, n-1):  
2     udt[i] = u[i] + kappa * (u[i+1] + u[i-1] - 2 * u[i])
```

with NumPy vector operations using slicing

```
1 udt[1:-1] = u[1:-1] + kappa * (u[0:-2] + u[2:] - 2 * u[1:-1])
```

Python code using Numpy operations instead of loops

```
1
2 import numpy as np
3
4 n=100000; dx=1.0; niter=50000; kappa=0.1
5
6 x=np.arange(n, dtype="float64")
7 u=np.empty(n, dtype="float64")
8 udt=np.empty(n, dtype="float64")
9
10 u_init = lambda x: np.exp( -(dx*(x-n/2))**2/100000)
11 u=u_init(x)
12 udt[:]=0.0
13
14 for i in xrange(niter):
15     udt[1:-1]=u[1:-1]+kappa*(u[0:-2]+u[2:]-2*u[1:-1])
16     u[:]=udt[:]
```

Performance

- ▶ 50,000 iterations, tested on Macbook Pro laptop (2011)
- ▶ C code compiled with `gcc -O2` - 12.75 s
- ▶ C code compiled with `gcc` (unoptimized) - 34.31 s
- ▶ Python code with NumPy operations - 40.43 s
- ▶ Python 3.2 times slower than optimized code, only 1.2 times slower than unoptimized code
- ▶ It's likely that GCC can optimize the whole loop over iterations, whereas Numpy vector operations optimize each iteration individually

Numpy libraries

- ▶ For standard operations, eg. matrix multiply, will run at the speed of underlying numerical library
- ▶ Performance will strongly depend on which library is used, can see with **`numpy.show_config()`**
- ▶ If libraries are threaded, python will take advantage of multithreading, with no extra programming (free parallelism)

General approaches for code speedup

- ▶ NumPy does not help with all problems, some don't fit array operations
- ▶ Need a more general technique to speed up Python code
- ▶ As the problem is that Python is not a compiled language, one can try to compile it
- ▶ General compiler: `nuitka` (<http://nuitka.net/>) under active development
- ▶ `PyPy` (<http://pypy.org/>) - Just-in-Time (JIT) compiler
- ▶ **Cython** (<http://cython.org/>)- turns Python program into C and compiles it

Euler problem

If p is the perimeter of a right angle triangle with integral length sides, a, b, c , there are exactly three solutions for $p = 120$.

$(20, 48, 52)$, $(24, 45, 51)$, $(30, 40, 50)$

For which value of $p < N$, is the number of solutions maximized?
Take $N=1000$ as starting point

(from <https://projecteuler.net>)

Get solutions at particular p

```
1 def find_num_solutions(p):  
2     n=0  
3     for a in range(1,p/2):  
4         for b in range(a,p/2):  
5             c=p-a-b  
6             if (a*a+b*b==c*c):  
7                 n=n+1  
8  
9     return n
```

Loop over possible value of p up to N

```
1 nmax=0 ; imax=0
2 N=1000
3
4 for i in range(1,N):
5     print i
6     nsols=find_num_solutions(i)
7     if(nsols>nmax):
8         nmax=nsols ; imax=i
9
10 print "maximum p , number of solutions" ,imax ,nmax
```

Cython

- ▶ The goal is to identify functions in the code where it spends the most time. Python has profiler already built in
- ▶ **python -m cProfile euler37.py**
- ▶ Place those functions in a separate file so they are imported as module
- ▶ Cython will take a python module file, convert it into C code, and then compile it into a shared library
- ▶ Python will import that compiled library module at runtime just like it would import a standard Python module
- ▶ To make Cython work well, need to provide some hints to the compiler as to what the variables are, by defining some key variables

Invoking Cython

- ▶ Place module code (with Cython modifications) in `find_num_solutions.pyx`
- ▶ Create file `setup.py`

```
1 from distutils.core import setup
2 from Cython.Build import cythonize
3
4 setup(
5     ext_modules=cythonize("find_num_solutions.pyx"),
6 )
```

- ▶ Execute: `python setup.py build_ext -inplace`
- ▶ Creates `find_num_solutions.c`, C code implementation of the module
- ▶ From this creates `find_num_solutions.so` library which can be imported as Python module at runtime

Get solutions at particular p, cythonized

```
1 def find_num_solutions(int p): # note definition
2     cdef int a,b,c,n          # note definition
3     n=0
4     for a in range(1,p/2):
5         for b in range(a,p/2):
6             c=p-a-b
7             if (a*a+b*b==c*c):
8                 n=n+1
9
10    return n
```

This code in file find_num_solutions.pyx

Loop over possible value of p up to N, with Cython

Note changes at line 1 and line 9

```
1 import find_num_solutions
2
3 nmax=0 ; imax=0 ; N=1000
4
5 for i in range(1,N):
6     print i
7     nsols=find_num_solutions.find_num_solutions(i)
8     if(nsols>nmax):
9         nmax=nsols ; imax=i
10
11 print "maximum p and , number of solutions",imax ,nmax
```

Speedup with Cython

For $N=1000$, tested on development node of orca cluster

- ▶ vanilla python : 14.158 s
- ▶ Cython without variable definitions : 8.87 s, speedup factor 1.6
- ▶ Cython with integer variables defined : 0.084 s, speedup factor 168

ctypes - a foreign function library for Python

```
1 # compile C library with:  
2 # gcc -shared -o libraryname.so librarycode.c  
3 import ctypes  
4 libraryname = ctypes.CDLL('./libraryname.so')  
5 libraryname.some_function(argument1)
```

Speedup with ctypes

For N=1000, tested on development node of orca cluster

- ▶ vanilla python : 14.158 s
- ▶ Cython without variable definitions : 8.87 s, speedup factor 1.6
- ▶ Cython with integer variables defined : 0.084 s, speedup factor 168
- ▶ ctypes : 0.134, speedup factor 105, 1.6 times slower than cython!
- ▶ pure C code (GCC): 0.134, same as ctypes (as expected)
- ▶ pure C code (icc): 0.0684 s (Intel compiler more efficient)

Parallelizing Python

- ▶ Once the serial version is optimized, need to parallelize Python to do true HPC
- ▶ Threading approach does not work due to Global Interpreter Lock
- ▶ In Python, you can have many threads, but only one executes at any one time, hence no speedup
- ▶ Have to use multiple processes instead

Multiprocessing - apply

```
1 import time,os
2 from multiprocessing import Pool
3
4 def f():
5     start=time.time()
6     time.sleep(2)
7     end=time.time()
8     print "inside f pid",os.getpid()
9     return end-start
10
11 p = Pool(processes=1)
12 result = p.apply(f)
13 print "apply is blocking, total time",result
14
15 result=p.apply_async(f)
16 print "apply_async is non-blocking"
17
18 while not result.ready():
19     time.sleep(0.5)
20     print "could work here while result computes"
21
22 print "total time",result.get()
```

Multiprocessing - Map

```
1 import time
2 from multiprocessing import Pool
3
4 def f(x):
5     return x**3
6
7 y = range(int(1e7))
8 p= Pool (processes=8)
9
10 start= time.time()
11 results = p.map(f,y)
12 end = time.time()
13
14 print "map blocks on launching process , time=",end-
    start
15
16 # map_async
17 start = time.time()
18 results = p.map_async(f,y)
19 print "map_async is non-blocking on launching process"
20 output = results.get()
21 end=time.time()
22 print "time" ,end-start
```

Euler problem with multiprocessing

```
1 from multiprocessing import Pool
2 import find_num_solutions
3
4 p= Pool (processes=4)
5
6 y=range(1,1000)
7 results=p.map(find_num_solutions.find_num_solutions,y)
8 print "answer",y[results.index(max(results))]
```


Multiprocessing performance

timing on orca development node (24 cores)

n=5000 case

Number of processes	time(s)	speedup
1	11.07	1.0
2	7.247	1.52
4	4.502	2.45
8	2.938	3.76
16	2.343	4.72
24	1.885	5.87

Will scale better for larger values of N (for example, for N=10000 get speedup 13.0 with 24 processors)

MPI - Message Passing Interface

- ▶ Approach has multiple processors with independent memory running in parallel
- ▶ Since memory is not shared, data is exchanged via calls to MPI routines
- ▶ Each process runs same code, but can identify itself in the process set and execute code differently

Compare MPI in C and Python with mpi4py - MPI reduce

```
1 int main(int argc, char* argv[]) {
2     int my_rank, imax, imax_in;
3     MPI_Init(&argc, &argv);
4     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
5     imax_in=my_rank;
6     MPI_Reduce(&imax_in, &imax, 1, MPI_INT, MPI_MAX, 0,
7     MPI_COMM_WORLD);
8     if (my_rank == 0) printf("%d \n", imax);
9     MPI_Finalize();
10    return 0;}
```

```
1 from mpi4py import MPI
2 comm = MPI.COMM_WORLD
3 myid = comm.Get_rank()
4 imax_in = myid
5 imax = comm.reduce(imax_in, op=MPI.MAX)
6 if (myid==0):
7     print imax
8 MPI.Finalize
```

Loop over p values up to N distributed among MPI processes

```
1 from mpi4py import MPI
2 import find_num_solutions
3
4 comm = MPI.COMM_WORLD
5 myid = comm.Get_rank()
6 nprocs = comm.Get_size()
7
8 nmax=0 ; imax=0 ; N=5000
9
10 for i in range(1,N):
11
12     if (i%nprocs==myid):
13         nsols=find_num_solutions.find_num_solutions(i)
14         if (nsols>nmax):
15             nmax=nsols ; imax=i
16
17 nmax_global=comm.allreduce(nmax,op=MPI.MAX)
18 if (nmax_global==nmax):
19     print "process ",myid," found maximum at ",imax
20
21 MPI.Finalize
```

MPI performance

timing on orca development node (24 cores)

n=5000 case

MPI processes	time(s)	speedup
1	10.254	1.0
2	6.597	1.55
4	4.015	2.55
8	2.932	3.49
16	2.545	4.02
24	2.818	3.64

Will scale better for larger values of N (for example, for N=10000 get speedup 13.4 with 24 processors)

Conclusion

- ▶ Python is generally slower than compiled languages like C
- ▶ With a bit of effort, can take a Python code which is a great deal slower and make it only somewhat slower
- ▶ The tradeoff between slower code but faster development time is something the programmer has to decide
- ▶ Tools currently under development should make this problem less severe over time