High Performance Computing with Python

Pawel Pomorski
SHARCNET
University of Waterloo
ppomorsk@sharcnet.ca

April 29, 2015
Outline

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

Please attend our Summer School (next one held May 25-29, 2015) for more in depth coverage of this material.
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 (e.g., 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
#include <math.h>
#include <stdio.h>

int main()
{
  int const n = 100000, niter = 500;

double x[n], u[n], udt[n];
int i, iter;
double dx = 1.0;
double kappa = 0.1;

for (i = 0; i < n; i++)
{
  u[i] = exp(-pow(dx*(i-n/2.0), 2.0)/100000.0);
  udt[i] = 0.0;
}
...
```
C code continued:

```c
... 
for (iter = 0; iter < niter; iter++){
    for (i = 1; i < n - 1; i ++){
        udt[i] = u[i] + kappa * (u[i + 1] + u[i - 1] - 2 * u[i]);
    }
    for (i = 0; i < n; i ++){
        u[i] = udt[i];
    }
}
return 0;
```
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
import numpy as np
n=100000 ; dx=1.0 ; niter=500 ; kappa=0.1
x=np.arange(n,dtype="float64")
u=np.empty(n,dtype="float64")
udt=np.empty(n,dtype="float64")

for i in xrange(len(u)):
    u[i]=np.exp(-dx*(i-n/2)**2/100000)
udt[:]=0.0

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

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

Replace explicit loops

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

with NumPy vector operations using slicing

```python
udt[1:-1] = u[1:-1] + kappa * (u[0:-2] + u[2:] - 2 * u[1:-1])
```
Python code using Numpy operations instead of loops

```python
import numpy as np

n=100000; dx=1.0; niter=50000; kappa=0.1

x=np.arange(n, dtype="float64")
u=np.empty(n, dtype="float64")
udt=np.empty(n, dtype="float64")

u_init = lambda x: np.exp(-dx*(x-n/2)**2/100000)
u=u_init(x)
udt[:]=0.0

for i in xrange(niter):
    udt[1:-1]=u[1:-1]+kappa*(u[0:-2]+u[2:]-2*u[1:-1])
    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
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 $p$

```python
def find_num_solutions(p):
    n=0
    # a+b+c=p
    for a in range(1, p/2):
        for b in range(a, p):
            c=p-a-b
            if (c>0):
                if (a*a+b*b==c*c):
                    n=n+1
    return n
```
Loop over possible value of p up to N

```python
nmax=0 ; imax=0
N=1000

for i in range(1,N):
    print i
    nsols=find_num_solutions(i)
    if (nsols>nmax):
        nmax=nsols ; imax=i

print "maximum p, number of solutions",imax,nmax
```
Cython

- The goal is to identify functions in the code where it spends the most time
- 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`

```python
from distutils.core import setup
from Cython.Build import cythonize

setup(
    ext_modules=cythonize("find_num_solutions.pyx"),
)
```

- 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

```python
def find_num_solutions(int p):  # note definition
cdef int a, b, c, n  # note definition
    n = 0
    # a+b+c=p
    for a in range(1, p/2):
        for b in range(a, p):
            c = p - a - b

            if (c > 0):
                if (a*a+b*b==c*c):
                    n = n + 1

    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

```python
import find_num_solutions

nmax=0 ; imax=0 ; N=1000

for i in range(1,N):
    print i
    nsols=find_num_solutions.find_num_solutions(i)
    if (nsols>nmax):
        nmax=nsols ; imax=i

print "maximum $p$ and , number of solutions",imax,nmax
```
Speedup with Cython

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

- vanilla python: 38.324 s
- Cython without variable definitions: 26.18 s, speedup factor 1.5
- Cython with integer variables defined: 0.416 s, speedup factor 92
Parallelizing Python

- One 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
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

```c
int main(int argc, char* argv[]) {
    int my_rank, imax, imax_in;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    imax_in=my_rank;
    MPI_Reduce(&imax_in,&imax,1,MPI_INT,MPI_MAX,0,
               MPI_COMM_WORLD);
    if (my_rank == 0) printf("%d \n",imax);
    MPI_Finalize();
    return 0;}
```

```python
from mpi4py import MPI
comm = MPI.COMM_WORLD
myid = comm.Get_rank()
imax_in = myid
imax = comm.reduce(imax_in,op=MPI.MAX)
if (myid==0):
    print imax
MPI.Finalize
```
Loop over p values up to N distributed among MPI processes

```python
from mpi4py import MPI
import find_num_solutions

comm = MPI.COMM_WORLD
myid = comm.Get_rank()
nprocs = comm.Get_size()

nmax=0 ; imax=0 ; N=5000

for i in range(1,N):
    if (i%nprocs==myid):
        nsols=find_num_solutions.find_num_solutions(i)
        if (nsols>nmax):
            nmax=nsols ; imax=i

imax_global=comm.reduce(imax,op=MPI.MAX)
MPI.Finalize

if (myid == 0 ):
    print "maximum p ",imax_global
```
MPI performance

timing on orca development node (24 cores)
n=5000 case

<table>
<thead>
<tr>
<th>MPI processes</th>
<th>time(s)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>34.32</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>18.43</td>
<td>1.86</td>
</tr>
<tr>
<td>4</td>
<td>9.88</td>
<td>3.47</td>
</tr>
<tr>
<td>8</td>
<td>5.78</td>
<td>5.93</td>
</tr>
<tr>
<td>16</td>
<td>3.80</td>
<td>9.03</td>
</tr>
<tr>
<td>24</td>
<td>3.63</td>
<td>9.45</td>
</tr>
</tbody>
</table>

Will scale better for larger values of N
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.