High Performance Computing with Python

Pawel Pomorski
SHARCNET
University of Waterloo
ppomorsk@sharcnet.ca

March 15,2017

Outline

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

Please attend our Summer School (next one held May 29 - June 2, 2017) 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 (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

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

C code

```c
#include<math.h>
#include<stdio.h>

int main(){

int const n=100000, niter=2000;

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 :

```
...
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;
}
```

Program output

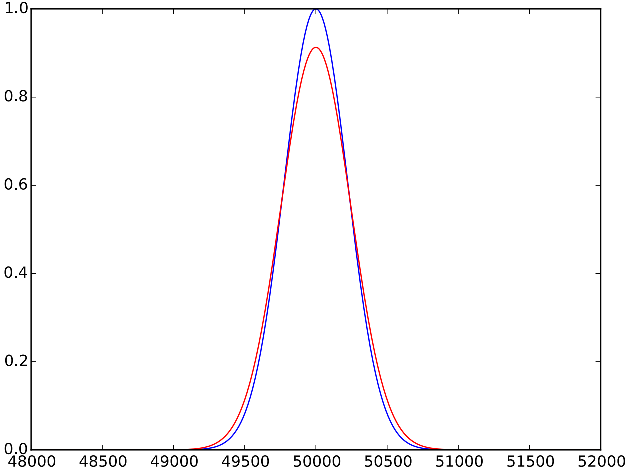

Figure: Evolution of u(x) after 50,000 time steps (blue line initial, red line final)

"Vanilla" Python code

```
import math
n=100000 ; dx=1.0 ; niter=2000 ; kappa=0.1

x=n*[0.0,]
u=n*[0.0,]
udt=n*[0.0,]

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

fac=(1-2.0*kappa)
for itern in xrange(niter):

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

    for i in xrange(n):
        u[i]=udt[i]
```

Vanilla code performance

- 2000 iterations, tested on node in "dusky" cluster (Intel Xeon "Haswell")
- C code compiled with Intel compiler (icc) takes 0.55 seconds
- Python "vanilla" code takes 96.11 seconds
- Python is much slower (by factor 175)
- Code is slow because loops are explicit

NumPy

- To achieve reasonable efficiency 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
- The goal is to eliminate costly explicit loops and replace them with numpy operations instead
- Numpy functions invoke efficient libraries written in C
- The difficulty of eliminating costly explicit loops varies.

Slicing NumPy arrays :

```
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

```
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

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

Python code using Numpy operations instead of loops

```

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 itern in xrange(niter):
    udt[1:-1]=fac*u[1:-1]+kappa*(u[0:-2]+u[2:])
    u[:]=udt[:]
```

Performance

- 50,000 iterations, tested on node in "dusky" cluster (Intel Xeon "Haswell")
- C code compiled with icc -xCORE-AVX2 -fma - 6.71 s
- C code compiled with icc (unoptimized) - 36.70 s
- Python code with NumPy operations - 38.33 s
- Python 6.4 times slower than optimized code, only 1.05 times slower than unoptimized code
- It's likely that the compiler can optimize the entire loops more efficiently.

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

```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

```
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. 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

```
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

```
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 7

```
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

```python
# compile C library with:
# icc -shared -o findnumsolutions.so findnumsolutions.c
import ctypes
findnumsolutions = ctypes.CDLL('./findnumsolutions.so')
# ...
nsols=findnumsolutions.findnumsolutions(i)
```

C code : findnumsolutions.c

```c
int findnumsolutions(int p){
    int a,b,c,n;
    n=0;
    for(a=1;a<p/2;a++){
        for (b=a;b<p/2;b++){
            c=p-a-b;
            if(a*a+b*b==c*c){
                n=n+1;
            }
        }
    }
return n;
}
```

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 : 0.065 s , speedup factor 218, 1.3 times faster than cython
- pure C code (icc): 0.068 s (almost same as ctypes)
- It's important to choose the best compiler (Intel more efficient than GCC)
- pure C code (GCC): 0.134

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
- Python has multiprocessing module but that only works within one node. Have to use MPI to achieve parallelism over many nodes

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

```
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;}
```

```
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

```
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

nmax_global=comm.allreduce(nmax,op=MPI.MAX)
if(nmax_global==nmax):
    print "process ",myid," found maximum at ",imax

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)

Python on GPUs

- PyCUDA - Python wrapper for CUDA (https://mathema.tician.de/software/pycuda/)
- GPU Kernels must still be written in CUDA C
- Aside from that, more convenient to use than CUDA
- Popular software implemented in Python with GPU acceleration
- Theano (http://deeplearning.net/software/theano/)
- TensorFlow (https://www.tensorflow.org/)

PyCUDA example:

```
import pycuda.driver as drv
import pycuda.tools
import pycuda.autoinit
import numpy
import numpy.linalg as la
from pycuda.compiler import SourceModule

mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")
```

...

PyCUDA example - continued:

```
a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1))

print dest-a*b
```

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