Diffusion processes in complex networks

Digression - parallel computing in Python

Janusz Szwabiński

Outlook:

- Multiprocessing
- Parallel computing in IPython
- MPI for Python
- Cython and OpenMP
- Python and OpenCL

References

- J.R. Johansson’s lectures (http://github.com/jrjohansson/scientific-python-lectures (http://github.com/jrjohansson/scientific-python-lectures))
- Multiprocessing: https://docs.python.org/2/library/multiprocessing.html (https://docs.python.org/2/library/multiprocessing.html)
- MPI for Python: http://mpi4py.scipy.org/ (http://mpi4py.scipy.org/)
- PyOpenCL: http://documen.tician.de/pyopencl/ (http://documen.tician.de/pyopencl/)

```plaintext
In [1]:
%matplotlib inline
import matplotlib.pyplot as plt

Multiprocessing

Multiprocessing module is a part of Python's standard library. It may be used for simple parallel tasks.

In [2]:
import multiprocessing
import os
import time
import numpy

In [3]:
#simple function that gets the process id
def task(args):
    print("PID =", os.getpid(), ", args =", args)
    return os.getpid(), args
```
In [4]:

```python
#ordinary call of the function
task("test")
```

```
PID = 11812 , args = test
```

```
Out[4]:
(11812, 'test')
```

In [5]:

```python
#now the same in multiprocessing approach
#create the pool of processes first
pool = multiprocessing.Pool(processes=4)

#then map the function to arguments within the pool
result = pool.map(task, [1,2,3,4,5,6,7,8])
```

```
PID = 11835 , args = 2
PID = 11836 , args = 3
PID = 11837 , args = 4
PID = 11837 , args = 6
PID = 11836 , args = 5
PID = 11837 , args = 7
PID = 11835 , args = 8
PID = 11834 , args = 1
```

In [6]:

```python
#results in a compact form
print(result)
```

```
[(11834, 1), (11835, 2), (11836, 3), (11837, 4), (11836, 5), (11837, 6), (11837, 7), (11835, 8)]
```

The module is useful for problems with little or no need for communication between the tasks. Such a type of problems is called **embarrassingly parallel problems** in parallel computing.

Examples:

- computer simulations (e.g. independent runs in Monte Carlo simulations, parameter sweeping)
- numerical integration
- genetic algorithms
- distributed database queries
- rendering of computer graphics
- brute-force attacks in cryptography
- serving static files on a webserver to multiple users at once
- large scale facial recognition systems

Let us consider now a more useful example than simply getting the process id. We want to calculate a function at many points:
In [7]:
```
def f(x):
    time.sleep(1)  # sleep instead of complicated calculations
    return x*x
```

```
p = multiprocessing.Pool()
p.map(f,[1,2,3,4])
```

Out[7]:
```
[1, 4, 9, 16]
```

In [8]:
```
%timeit p.map(f,range(10))
```

```
1 loop, best of 3: 2 s per loop
```

In [9]:
```
%timeit [f(x) for x in range(10)]
```

```
1 loop, best of 3: 10 s per loop
```

We see that the parallel approach is faster that the traditional one. In general, parallel computing is worth considering if the time needed to proceed a single task is much larger as the overhead related to the multiple processes handling. Otherwise one should rather go for the traditional version:

In [10]:
```
def g(x):
    return x*x  # similar to f, but no delay
```

```
p = multiprocessing.Pool()
```

In [11]:
```
%timeit p.map(g,range(100))
```

```
1000 loops, best of 3: 478 µs per loop
```

In [12]:
```
%timeit [g(x) for x in range(10)]
```

```
1000000 loops, best of 3: 1.17 µs per loop
```

The serial execution of the tasks is indeed much faster in this case.

**Parallel computing in IPython**
IPython environment offers an easy-to-use mechanism for parallel computing based on the concept of engines and controllers, which may proceed some tasks.

First, we have to install the ipyparallel extension,

```
    pip3 install ipyparallel
```

In order to start parallel computing in IPython, we have to create a cluster of engines. We can do that in the command line:

```
$ ipcluster start -n 4
```

Other possibility is the tab 'Clusters' in a jupyter notebook, provided the notebook extension was enabled with the command

```
    ipcluster nbextension enable
```

The above ipcluster command will start 4 engines. If we work on a single machine, such a cluster of engines is useful only if it is a multicore machine. Otherwise it would rather slow down the computations. It should be noted however that the engines may be distributed over several machines (see https://ipyparallel.readthedocs.io/en/latest/ for more details).

After starting the engine cluster we import a client from the ipyparallel module:

```
In [13]:
from ipyparallel import Client

In [14]:
cli = Client()

With ids attribute we can check the ids of the engines:

```
In [15]:
cli.ids
```

```
Out[15]:
[0, 1, 2, 3, 4, 5, 6, 7]
```

Each of these engines may process different tasks. The tasks may be assigned to some of the engines or all of them:

```
In [16]:
def getpid():
    """ Return ID of current process ""
    import os
    return os.getpid()
```

In [17]:

    # we test the function locally
    getpid()

Out[17]:
    11812

In [18]:

    # now we run it on a particular engine
    cli[0].apply_sync(getpid)

Out[18]:
    11732

In [19]:

    # and on all engines in parallel
    cli[:].apply_sync(getpid)

Out[19]:
    [11732, 11734, 11736, 11738, 11740, 11749, 11765, 11774]

We can use this cluster of IPython engines to execute tasks in parallel. The easiest way to dispatch a
function to different engines is to define the function with the decorator

    @view.parallel(block=True)

(look at http://pythonconquerstheuniverse.wordpress.com/2012/04/29/python-decorators/
(http://pythonconquerstheuniverse.wordpress.com/2012/04/29/python-decorators/) for a nice explanation of
decorators).

Here, view is the pool of engines we want to dispatch the function.

Let us check how it works:

In [20]:

    dview = cli[:]

In [21]:

    @dview.parallel(block=True)
    def dummy_task(delay):
        """ do nothing for ‘delay’ seconds and finish ""
        import os, time

        t0 = time.time()
        pid = os.getpid()
        time.sleep(delay)
        t1 = time.time()

        return [pid, t0, t1]
Once our function is defined this way we can dispatch it to the engine using the `map` method in the resulting class:

In [22]:

```python
# random delay times
delay_times = numpy.random.rand(8)
```

In [23]:

```python
#now we simply map the times to the function
dummy_task.map(delay_times)
```

Out[23]:

```
[[11732, 1495541901.9115462, 1495541902.095794],
 [11734, 1495541901.9117837, 1495541902.6235466],
 [11736, 1495541901.911783, 1495541902.079649],
 [11738, 1495541901.9143817, 1495541902.135188],
 [11740, 1495541901.9117632, 1495541902.465705],
 [11749, 1495541901.9135165, 1495541902.2227879],
 [11765, 1495541901.915408, 1495541902.0197375],
 [11774, 1495541901.9134102, 1495541902.8327844]]
```

Let us repeat that with more tasks. We will use matplotlib to visualize the results:

In [24]:

```python
def visualize_tasks(results):
    res = numpy.array(results)
    fig, ax = plt.subplots(figsize=(10, res.shape[1]))

    yticks = []
yticklabels = []
tmin = min(res[:,1])
    for n, pid in enumerate(numpy.unique(res[:,0])):
        yticks.append(n)
yticklabels.append("%d" % pid)
        for m in numpy.where(res[:,0] == pid)[0]:
            ax.add_patch(plt.Rectangle((res[m,1] - tmin, n-0.25),
                                res[m,2] - res[m,1], 0.5, color="green", alpha=0.5))

    ax.set_ylim(-.5, n+.5)
    ax.set_xlim(0, max(res[:,2]) - tmin + 0.)
    ax.set_yticks(yticks)
    ax.set_yticklabels(yticklabels)
    ax.set_xlabel("seconds")
    ax.set_ylabel("PID")

In [25]:

delay_times = numpy.random.rand(64)
```

```
In [26]:

```python
result = dummy_task.map(delay_times)
visualize_tasks(result)
```

We utilized all engines quite well. However, the tasks seem to be not balanced, i.e. one engine might be idle while others still have tasks to work on.

To obtain a load balanced view of the engines we simply use the `load_balanced_view` decorator:

In [27]:

```python
lbview = cli.load_balanced_view()
```

In [28]:

```python
@lbview.parallel(block=True)
def dummy_task_load_balanced(delay):
    """do nothing for `delay` seconds and finish """

    import os, time

    t0 = time.time()
    pid = os.getpid()
    time.sleep(delay)
    t1 = time.time()

    return [pid, t0, t1]
```

In [29]:

```python
result = dummy_task_load_balanced.map(delay_times)
visualize_tasks(result)
```
This time the loads of the engines are similar. Moreover, the time to compilation is now shorter than in the previous case.

And now a more useful example:

In [30]:

```python
import sympy

def factorit(n):
    x = sympy.var('x')
    return sympy.factor(x**n - 1, x)
```

In [31]:

```python
factorit(2)
```

Out[31]:

```
(x - 1)*(x + 1)
```

In [32]:

```python
factorit(5)
```

Out[32]:

```
(x - 1)*(x**4 + x**3 + x**2 + x + 1)
```

In [33]:

```
# let us make it harder
%timeit f = [factorit(i) for i in range(100,110)]
```

10 loops, best of 3: 47.8 ms per loop

In [34]:

```python
cli = Client()
dview = cli[:]
# we have to import sympy on every engine
dview.execute('import sympy')
```

Out[34]:

```
<AsyncResult: execute>
```

In [35]:

```
%timeit f = dview.map(factorit, range(100,110))
```

The slowest run took 4.85 times longer than the fastest. This could mean that an intermediate result is being cached.

100 loops, best of 3: 4.46 ms per loop

In [36]:

```python
f = dview.map(factorit, range(100,110))
```
In [38]:

    f[-1]

Out[38]:

    (x - 1)*(x**108 + x**107 + x**106 + x**105 + x**104 + x**103 + x**10 + x**93 + x**92 + x**91 + x**90 + x**89 + x**88 + x**87 + x**86 + x**85 + x**84 + x**83 + x**82 + x**81 + x**80 + x**79 + x**78 + x**77 + x**76 + x**75 + x**74 + x**73 + x**72 + x**71 + x**70 + x**69 + x**68 + x**67 + x**66 + x**65 + x**64 + x**63 + x**62 + x**61 + x**60 + x**59 + x**58 + x**57 + x**56 + x**55 + x**54 + x**53 + x**52 + x**51 + x**50 + x**49 + x**48 + x**47 + x**46 + x**45 + x**44 + x**43 + x**42 + x**41 + x**40 + x**39 + x**38 + x**37 + x**36 + x**35 + x**34 + x**33 + x**32 + x**31 + x**30 + x**29 + x**28 + x**27 + x**26 + x**25 + x**24 + x**23 + x**22 + x**21 + x**20 + x**19 + x**18 + x**17 + x**16 + x**15 + x**14 + x**13 + x**12 + x**11 + x**10 + x**9 + x**8 + x**7 + x**6 + x**5 + x**4 + x**3 + x**2 + x + 1)

**MPI for Python**

When more communication between processes is required, sophisticated solutions as MPI or OpenMP are often needed.

MPI is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures. It defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran. There are several well-tested and efficient implementations of MPI, many of which are open-source or in the public domain.

MPI can be used in Python programs through the *mpi4py* package ([http://mpi4py.scipy.org/](http://mpi4py.scipy.org/)). To use it we usually include MPI from the corresponding module,

    from mpi4py import MPI

An MPI Python program must be started using

    $mpirun -n N

where N is the number of processes that should be included in the process pool (can be larger as the number of available cores/processors).

**Example 1 - sending data from one process to the other**
In [39]:

```python
%%file mpitest.py

# -*- coding: utf-8 -*-

from mpi4py import MPI

#initialize process pool
comm = MPI.COMM_WORLD
#get process id
rank = comm.Get_rank()

if rank == 0:                          #master
    data = [1.0, 2.0, 3.0, 4.0]         #create some data
    comm.send(data, dest=1, tag=11)     #send data to node 1
elif rank == 1:                        #slave
    data = comm.recv(source=0, tag=11)  #receive data from 0

print("rank =", rank, ", data =", data) #print to screen (all processes)

Writing mpitest.py

In [40]:

!mpirun -n 2 python3 mpitest.py

rank = 0 , data = [1.0, 2.0, 3.0, 4.0]
rank = 1 , data = [1.0, 2.0, 3.0, 4.0]

Przykład 2 - two processes, sending numpy array from one to other

In [41]:

```python
%%file mpi-numpy-array.py

# -*- coding: utf-8 -*-

from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:                       #master
    data = numpy.random.rand(10)     #data to send
    comm.Send(data, dest=1, tag=13)  #(dest - destination id, tag - data label)
elif rank == 1:
    data = numpy.empty(10, dtype=numpy.float64) #container for data
    comm.Recv(data, source=0, tag=13)           #(source - sender id, tag - data label)

print("rank =", rank, ", data =", data) #print to screen (all processes)

Writing mpi-numpy-array.py
Note, that in this case we used Send and Recv instead of send and recv. The latter ones are for built-in Python types only.

In [42]:

```bash
mpiexec -n 2 python3 mpi-numpy-array.py
```

```
rank = 0 , data = [ 0.01115279 0.95229013 0.85597378 0.06067628 0.57811044 0.4065776 
0.0230081 0.38120197 0.44154346 0.85905686]
rank = 1 , data = [ 0.01115279 0.95229013 0.85597378 0.06067628 0.57811044 0.4065776 
0.0230081 0.38120197 0.44154346 0.85905686]
```

**Example 3 - matrix-vector multiplication**

In [43]:

```python
import numpy

# prepare data and save it to files
N = 16
A = numpy.random.rand(N, N)
numpy.save("random-matrix.npy", A)
x = numpy.random.rand(N)
numpy.save("random-vector.npy", x)
```
In [44]:

```python
%%file mpi-matrix-vector.py

# -*- coding: utf-8 -*-

from mpi4py import MPI
import numpy

#initialize MPI cluster
comm = MPI.COMM_WORLD
#get process id
rank = comm.Get_rank()
#get cluster size
p = comm.Get_size()

def matvec(comm, A, x):
    m = A.shape[0] // p
    #every process gets a part of the data
    y_part = numpy.dot(A[rank * m:(rank+1)*m], x)
    #container for the result
    y = numpy.zeros_like(x)
    #collect results from the pool, write them to container y
    comm.Allgather([y_part, MPI.DOUBLE], [y, MPI.DOUBLE])
    return y

A = numpy.load("random-matrix.npy")
x = numpy.load("random-vector.npy")
y_mpi = matvec(comm, A, x)

if rank == 0:
    #test
    y = numpy.dot(A, x)
    print(y_mpi)
    #compare the local and MPI results
    print("sum(y - y_mpi) =", (y - y_mpi).sum())
```

Writing mpi-matrix-vector.py

In [45]:

```bash
mpirun -n 4 python3 mpi-matrix-vector.py
```

```
[ 5.69327703  4.85358127  4.46203991  5.67797315  6.05630264  4.6817
  4254  3.89666749  4.81103283  6.64800639  4.6662954  5.54068971  4.7007
  1352  4.14724973  4.32205529  4.89661218  4.13669822]
sum(y - y_mpi) = 0.0
```

Example 4 - sum of the elements in a vector

In [46]:

```python
# create data and save to file
N = 128
a = numpy.random.rand(N)
numpy.save("random-vector.npy", a)
```
In [47]:

```python
%%file mpi-psum.py

# -*- coding: utf-8 -*-

from mpi4py import MPI
import numpy as np

def psum(a):
    # process id
    r = MPI.COMM_WORLD.Get_rank()
    # cluster size
    size = MPI.COMM_WORLD.Get_size()
    # calculate data size for each process
    m = len(a) // size
    # local sum (different on each host)
    locsum = np.sum(a[r*m:(r+1)*m])
    # container for results
    rcvBuf = np.array(0.0, 'd')
    # collect results and sum them
    MPI.COMM_WORLD.Allreduce([locsum, MPI.DOUBLE], [rcvBuf, MPI.DOUBLE], op=MPI.SUM)
    return rcvBuf

a = np.load("random-vector.npy")
s = psum(a)

if MPI.COMM_WORLD.Get_rank() == 0:
    # mpi vs local
    print("sum =", s, ", numpy sum =", a.sum())
```

Writing mpi-psum.py

In [48]:

```
!mpirun -n 4 python3 mpi-psum.py
```

sum = 59.954215672639656 , numpy sum = 59.9542156726

### Cython i OpenMP

OpenMP (**Open Multi-Processing**) is a standard and widely used thread-based parallel API. It cannot be use directly in Python, because the CPython implementation uses a global interpreter lock, making it impossible to simultaneously run several Python threads. This is clearly a limitation in the Python interpreter, and as a consequence all parallelization in Python must use processes (not threads).

However, when calling out to compiled code the GIL is released, and it is possible to write Python-like code in Cython where we can selectively release the GIL and do OpenMP computations.

In [49]:

```python
import multiprocessing
N_core = multiprocessing.cpu_count()

print("Number of cores: %d" % N_core)
```

Number of cores: 8
Here is a simple example that shows the usage of OpenMP via Cython:

In [50]:
\%load_ext Cython

In [51]:
\%cython -f -c-fopenmp --link-args=-fopenmp -c-g

# -*- coding: utf-8 -*-

cimport cython
cimport numpy
from cython.parallel import prange, parallel
cimport openmp

def cy_openmp_test():
    cdef int n, N

    # release GIL so that we can use OpenMP
    with nogil, parallel():
        N = openmp.omp_get_num_threads()
        n = openmp.omp_get_thread_num()
        with gil:
            print("Number of threads: %d, thread number: %d\n" % (N, n))

In [52]:
cy_openmp_test()

Number of threads: 8, thread number: 0
Number of threads: 8, thread number: 2
Number of threads: 8, thread number: 5
Number of threads: 8, thread number: 4
Number of threads: 8, thread number: 3
Number of threads: 8, thread number: 7
Number of threads: 8, thread number: 6
Number of threads: 8, thread number: 1

Example - matrix-vector multiplication
In [53]:

```python
# prepare some random data
N = 4 * N_core
M = numpy.random.rand(N, N)
x = numpy.random.rand(N)
y = numpy.zeros_like(x)
```

We start with a simple implementation in Cython (without OpenMP):

In [54]:

```cython
#cython
cimport cython
cimport numpy
import numpy

@cython.boundscheck(False) #
@cython.wraparround(False) #

def cy_matvec(numpy.ndarray[numpy.float64_t, ndim=2] M,
              numpy.ndarray[numpy.float64_t, ndim=1] x,
              numpy.ndarray[numpy.float64_t, ndim=1] y):

cdef int i, j, n = len(x)

for i from 0 <= i < n:
    for j from 0 <= j < n:
        y[i] += M[i, j] * x[j]

return y
```

In [55]:

```python
# check that we get the same results
y = numpy.zeros_like(x)
cy_matvec(M, x, y)
numpy.dot(M, x) - y
```

Out[55]:

```python
array([-1.77635684e-15, -1.77635684e-15,  0.00000000e+00,  0.00000000e+00,
       0.00000000e+00, -1.77635684e-15, -8.88178420e-16, -3.55271368e-15,
       -8.88178420e-16,  8.88178420e-16,  0.00000000e+00, -1.77635684e-15,
       -8.88178420e-15, -1.77635684e-15,  0.00000000e+00,  0.00000000e+00,
       8.88178420e-16, -1.77635684e-15, -8.88178420e-16,  0.00000000e+00,
       -8.88178420e-16,  8.88178420e-16, -1.77635684e-15, -8.88178420e-16,
       8.88178420e-16, -1.77635684e-15,  0.00000000e+00,  8.88178420e-16,
       -8.88178420e-16,  8.88178420e-16])
```
In [56]:

```python
#and now some performance tests
%timeit numpy.dot(M, x)
```

The slowest run took 26.68 times longer than the fastest. This could mean that an intermediate result is being cached.
1000000 loops, best of 3: 558 ns per loop

In [57]:

```python
%timeit cy_matvec(M, x, y)
```

The slowest run took 7.56 times longer than the fastest. This could mean that an intermediate result is being cached.
1000000 loops, best of 3: 1.55 µs per loop

The Cython implementation is a bit slower than numpy.dot. We expect to improve the performance by making use of multiple cores:

In [58]:

```python
%%cython -f -c-fopenmp --link-args=-fopenmp -c-g
cimport cython
cimport numpy
from cython.parallel import parallel
cimport openmp

@cython.boundscheck(False)
@cython.wraparound(False)
def cy_matvec_omp(numpy.ndarray[numpy.float64_t, ndim=2] M,
                 numpy.ndarray[numpy.float64_t, ndim=1] x,
                 numpy.ndarray[numpy.float64_t, ndim=1] y):
    cdef int i, j, n = len(x), N, r, m

    # release GIL, so that we can use OpenMP
    with nogil, parallel():
        N = openmp.omp_get_num_threads()
        r = openmp.omp_get_thread_num()
        m = n // N //every thread gets its own data chunk
        for i from 0 <= i < m:
            for j from 0 <= j < n:
                y[r * m + i] += M[r * m + i, j] * x[j]

    return y
```
In [59]:

```python
# check the results
y = numpy.zeros_like(x)
cy_matvec_omp(M, x, y)
print(numpy.dot(M, x) - y)
```

```
[ -1.77635684e-15  -1.77635684e-15   0.00000000e+00   0.00000000e+00
  -1.77635684e-15  -1.77635684e-15   0.00000000e+00   0.00000000e+00
  -8.88178420e-16  -8.88178420e-16  -8.88178420e-16   8.88178420e-16
  -8.88178420e-16   0.00000000e+00  -8.88178420e-16  -8.88178420e-16
  0.00000000e+00  -3.55271368e-15   0.00000000e+00  -8.88178420e-16
  -8.88178420e-16  -8.88178420e-16  -8.88178420e-16  -8.88178420e-16
  -8.88178420e-16  -1.77635684e-15  -1.77635684e-15   8.88178420e-16
```

6]

In [60]:

```python
# performance tests once again
%timeit numpy.dot(M, x)
```

```
The slowest run took 31.11 times longer than the fastest. This could mean that an intermediate result is being cached.
1000000 loops, best of 3: 558 ns per loop
```

In [61]:

```python
%timeit cy_matvec_omp(M, x, y)
```

```
The slowest run took 113.12 times longer than the fastest. This could mean that an intermediate result is being cached.
10000 loops, best of 3: 56 µs per loop
```

The OpenMP implementation is even slower for the given problem size. It is due to the overhead associated with OpenMP and threading. But let us look at larger matrix sizes:

In [62]:

```python
N_vec = numpy.arange(50, 2500, 50) * N_core
```
In [63]:

```python
import time

duration_ref = numpy.zeros(len(N_vec))
duration_cy = numpy.zeros(len(N_vec))
duration_cy_omp = numpy.zeros(len(N_vec))

for idx, N in enumerate(N_vec):
    M = numpy.random.rand(N, N)
    x = numpy.random.rand(N)
    y = numpy.zeros_like(x)

    t0 = time.time()
    numpy.dot(M, x)
    duration_ref[idx] = time.time() - t0

    t0 = time.time()
    cy_matvec(M, x, y)
    duration_cy[idx] = time.time() - t0

    t0 = time.time()
    cy_matvec_omp(M, x, y)
    duration_cy_omp[idx] = time.time() - t0
```

In [64]:

```python
fig, ax = plt.subplots(figsize=(12, 6))

ax.loglog(N_vec, duration_ref, label='numpy')
ax.loglog(N_vec, duration_cy, label='cython')
ax.loglog(N_vec, duration_cy_omp, label='cython+openmp')

ax.legend(loc=2)
ax.set_yscale("log")
ax.set_ylabel("matrix-vector multiplication duration")
ax.set_xlabel("matrix size");
```

For large matrix sizes the OpenMP implementation is faster than `numpy.dot`. The speed-up is about:
In [65]:

((duration_ref / duration_cy_omp)[-10:]).mean()

Out[65]:

1.2226171108325758

However, we are still far away from the theoretical limit of the speed-up:

In [66]:

N_core

Out[66]:

8

Python and OpenCL

OpenCL (Open Computing Language) is an API for heterogenous computing, for example using both CPUs and GPUs for numerical computations. The python package pyopencl allows OpenCL code to be compiled, loaded and executed on the compute units completely from within Python: