Programowanie

Obliczenia równoległe w Pythonie

Janusz Szwabiński

Plan wykładu:

- Multiprocessing
- Obliczenia równoległe w IPythonie
- MPI dla Pythona
- Cython i OpenMP
- Python i OpenCL

Materiały do wykładu

- wykład J.R. Johanssona (http://github.com/jrjohansson/scientific-python-lectures)
- Multiprocessing: https://docs.python.org/2/library/multiprocessing.html
- "Using IPython for parallel computing": http://ipython.org/ipython-doc/dev/parallel/
- MPI for Python: http://mpi4py.scipy.org/
- PyOpenCL: http://documen.tician.de/pyopencl/

In [1]:

```
%matplotlib inline
import matplotlib.pyplot as plt
```

Multiprocessing

Moduł `multiprocessing` to część biblioteki standardowej Pythona, który pozwala na przeprowadzanie obliczeń równoległych.

In [2]:

```
import multiprocessing
import os
import time
import numpy
```

In [3]:

```
#funkcja pobierająca ID procesu
def task(args):
    print("PID =", os.getpid(), ", args =", args)

    return os.getpid(), args
```

In [4]:

```
#"zwykłe" wykonanie funkcji
task("test")

PID = 18603 , args = test
```

Out[4]:

```
(18603, 'test')
```

In [5]:

```
#a teraz to samo w ujęciu wieloprocesowym

#najpierw tworzę pulę procesów do wykorzystania
pool = multiprocessing.Pool(processes=4)

#a następnie wykorzystuję ją do wykonania
#funkcji task na wszystkich argumentach podanych
#w liście
result = pool.map(task, [1,2,3,4,5,6,7,8])

PID = 18752 , args = 3
PID = 18752 , args = 5
PID = 18753 , args = 4
PID = 18752 , args = 6
PID = 18753 , args = 7
PID = 18752 , args = 8
PID = 18750 , args = 1
PID = 18751 , args = 2
```

In [6]:

```
#wynik w zwartej postaci
print(result)

[(18750, 1), (18751, 2), (18752, 3), (18753, 4), (18752, 5), (18752, 6), (18753, 7),
(18752, 8)]
```

Moduł `multiprocessing` jest przydatny wszędzie tam, kiedy poszczególne zadania nie komunikują się ze sobą, lub komunikują się niezwykle rzadko (głównie w celu rozesłania danych do obliczeń i podsumowania wyników). Takie obliczenia nazywa się żargonowo *embarrassingly parallel tasks*.

Przykłady:

- symulacje komputerowe (np. niezależne przebiegi w metodach Monte Carlo, przeszukiwanie przestrzeni parametrów modelu itp)
- całkowanie numeryczne
- algorytmy genetyczne
- przeszukiwanie rozproszonych relacyjnych baz danych
- wystawianie statycznych plików na serwerach www dla wielu użytkowników jednocześnie
- renderowanie klatek w animacjach komputerowych

- ataki siłowe w kryptografii
- rozpoznawanie twarzy (porównanie zdjęcia z już zebranymi zdjęciami)

Rozważmy teraz bardziej życiowy przykład - chcemy policzyć wartość funkcji w wielu punktach:

In [7]:

```
def f(x):
    time.sleep(1) #zamiast skomplikowanych obliczeń
    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: 3.01 s per loop
```

In [9]:

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

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

Jeśli pojedyncze zadania trwają na tyle długo, że można przy nich zaniedbać czas potrzebny na obsługę wielu procesów, obliczenia równoległe mają sens. W przeciwnym wypadku lepiej jest liczyć w tradycyjny sposób:

In [10]:

```
def g(x):
    return x*x #to samo, co f, tylko bez opóźnienia

#funkcja będzie widoczna, jeśli zostanie zdefiniowana
#przed stworzeniem puli procesów

p = multiprocessing.Pool()
```

In [11]:

```
%timeit p.map(g,range(100))

The slowest run took 4.75 times longer than the fastest. This could mean that an
intermediate result is being cached.
1000 loops, best of 3: 818 µs per loop
```

In [12]:

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

```
The slowest run took 8.54 times longer than the fastest. This could mean that an
intermediate result is being cached.
1000000 loops, best of 3: 1.58 µs per loop
```

Obliczenia równoległe w IPythonie

Środowisko IPython oferuje łatwy w użyciu mechanizm do obliczeń równoległych, oparty na pomyśle silników i kontrolerów, którym przypisać można różne zadania.

Ta metoda wymaga doinstalowania rozszerzenia `ipyparallel`, np. poleceniem

```
pip3 install ipyparallel
```

Aby rozpocząć obliczenia równoległe w Pythonie, należy najpierw uruchomik klaster silników. Można to zrobić w konsoli poleceniem:

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

Inna możliwość to skorzystanie z zakładki `"Clusters"` na stronie domowej notanika Jupytera. W tym celu należy wcześniej udostępnić rozszerzenie dla notatnika:

```
ipcluster nbextension enable
```

Powyższe polecenie uruchomi cztery silniki na lokalnym komputerze, co ma sens w przypadku procesorów wielordzeniowych. Możliwe jest jednak tworzenie klastrów rozproszonych na wielu maszynach (więcej szczegółów pod adresem https://ipyparallel.readthedocs.io/en/latest/)

Po uruchomieniu klastra importujemy odpowiednią klasę w notatniku IPythona:

In [15]:

```
from ipyparallel import Client
```

In [22]:

```
cli = Client()
```

Atrybut `ids` pozwoli nam odczytać listę identyfikatorów dostępnych silników w klastrze:

In [23]:

```
cli.ids
```

Out[23]:

```
[0, 1, 2, 3]
```

Każdy z tych silników może wykonywać różne zadania. Możemy je przypisywać wybranym silnikom, lub wykonywać na wszystkich jednocześnie:

In [24]:

```
def getpid():
    """ zwróć ID bieżącego procesu """
    import os
    return os.getpid()
```

In [25]:

```
# testujemy funkcję w ramach procesu notatnika
getpid()
```

Out[25]:

```
18603
```

In [26]:

```
# możemy ją wykonać na wybranym silniku
cli[0].apply_sync(getpid)
```

Out[26]:

```
18919
```

In [27]:

```
# lub na wszystkich silnikach jednocześnie
cli[:].apply_sync(getpid)
```

Out[27]:

```
[18919, 18918, 18921, 18923]
```

Jedynym "wyzwaniem" przy wykorzystaniu klastra silników jest wysłanie zadań na każdy z nich. Najprościej daje się to zrealizować z wykorzystaniem dekoratora (więcej na temat dekoratorów tutaj: http://pythonconquerstheuniverse.wordpress.com/2012/04/29/python-decorators/):

```
@view.parallel(block=True)
```

`view` oznacza tu pulę silników, na której chcemy wykonać zadanie.

Sprawdźmy, jak to działa:

In [28]:

```
dview = cli[:]
```

In [29]:

```
@dview.parallel(block=True)
def dummy_task(delay):
    """ nic nie rób przez 'delay' sekund i zakończ """
    import os, time

    t0 = time.time()
```

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

    return [pid, t0, t1]
```

In [30]:

```
# losowe czasy opóźnień
delay_times = numpy.random.rand(8)
```

In [31]:

```
#teraz wystarczy zmapować funkcję z tymi czasami
dummy_task.map(delay_times)
```

Out[31]:

```
[[18919, 1495095505.7418573, 1495095506.6939182],
 [18919, 1495095506.694015, 1495095507.341749],
 [18918, 1495095505.7523692, 1495095506.4634879],
 [18918, 1495095506.463547, 1495095506.6537924],
 [18921, 1495095505.7529726, 1495095506.4788628],
 [18921, 1495095506.4789178, 1495095507.1548395],
 [18923, 1495095505.7520819, 1495095505.942002],
 [18923, 1495095505.942062, 1495095506.5152273]]
```

Zróbmy to samo z większą liczbą zadań. Możemy przy tym zwizualizować ich wykonanie na poszczególnych silnikach, korzystając z możliwości `matplotlib`:

In [32]:

```
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_ylabel("PID")
```

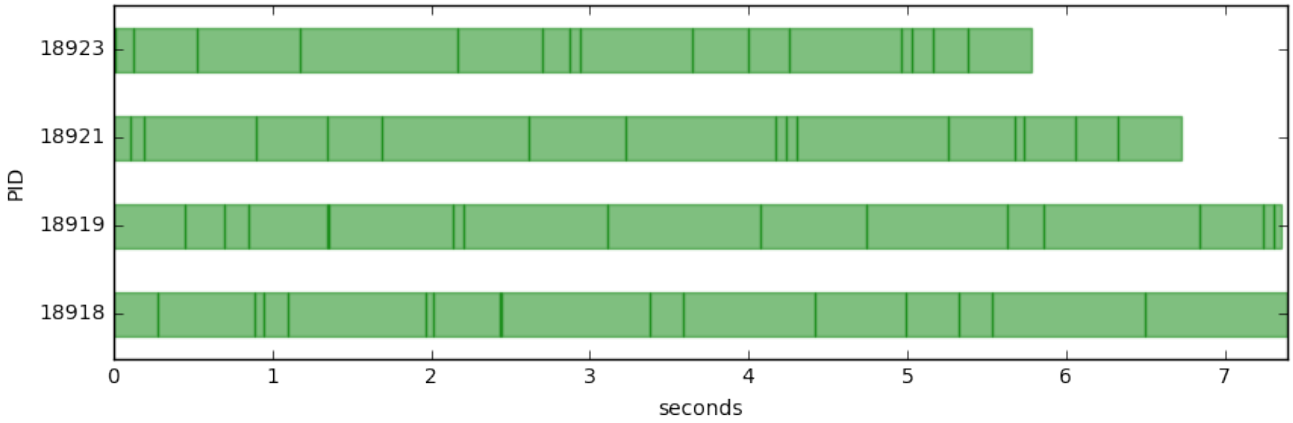
```
    ax.set_xlabel("seconds")
```

In [33]:

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

In [34]:

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



Jak wynika z powyższego wykresu, nie wszystkie silniki pracowały z podobnym obciążeniem.

Możemy to zmienić, korzystając z metody `load_balanced_view` na instancji klastra `cli`:

In [35]:

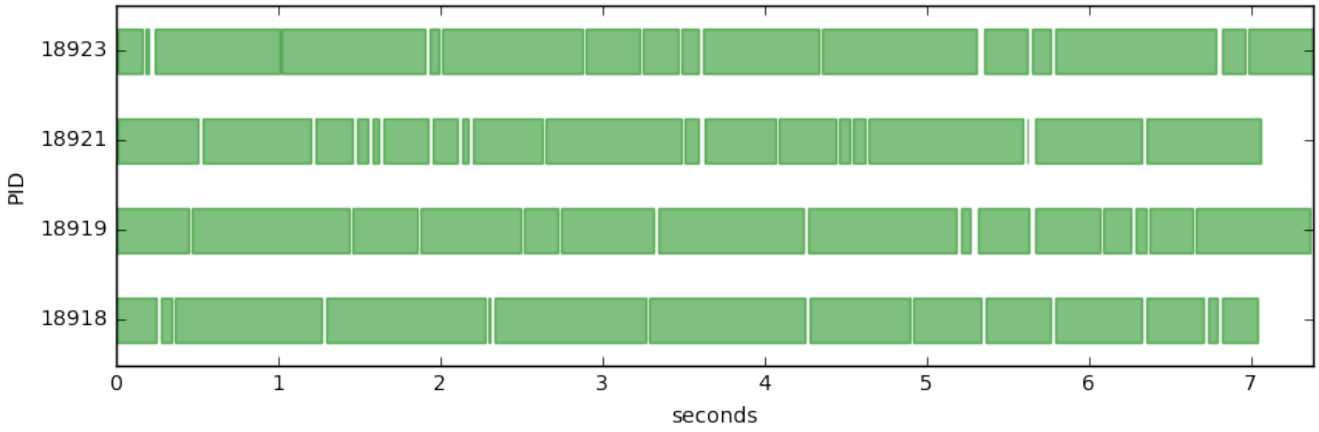
```
lbview = cli.load_balanced_view()
```

In [36]:

```
@lbview.parallel(block=True)
def dummy_task_load_balanced(delay):
    """ nic nie rób przez 'delay' sekund i zakończ """

    import os, time

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

    return [pid, t0, t1]
```

In [37]:

```
result = dummy_task_load_balanced.map(delay_times)
visualize_tasks(result)
```



Tym razem obciążenie poszczególnych silników jest rzeczywiście na podobnym poziomie, a wykonanie całego zadania trwało nieco krócej niż poprzednio.

A teraz coś bardziej pożytecznego:

In [38]:

```
import sympy

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

In [39]:

```
factorit(2)
```

Out[39]:

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

In [40]:

```
factorit(5)
```

Out[40]:

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

In [41]:

```
#trochę większe wyzwanie
%timeit f = [factorit(i) for i in range(100,110)]

10 loops, best of 3: 67.6 ms per loop
```

In [43]:

```
cli = Client()
dview = cli[:]
#musimy zaimportować sympy na każdym z silników
dview.execute('import sympy')
```

Out[43]:

```
<AsyncResult: execute>
```

In [44]:

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

```
The slowest run took 4.65 times longer than the fastest. This could mean that an
intermediate result is being cached.
100 loops, best of 3: 9.96 ms per loop
```

In [48]:

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

In [49]:

```
f[-1]
```

Out[49]:

```
(x - 1)*(x**108 + x**107 + x**106 + x**105 + x**104 + x**103 + x**102 + x**101 +
x**100 + x**99 + x**98 + x**97 + x**96 + x**95 + x**94 + 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 dla Pythona

Oba z proponowanych do tej pory rozwiązań dotyczyły sytuacji, w której nie ma komunikacji między zadaniami. Jeżeli w naszych obliczeniach taka komunikacja jest niezbędna, musimy się uciec do bardziej zaawansowanych rozwiązań, jak np. MPI.

MPI (ang. *Message Passing Interface*) to protokół komunikacyjny stanowiący jeden ze standardów przesyłania komunikatów pomiędzy procesami programów równoległych działających na jednym lub wielu komputerach. Istnieje kilka pakietów oferujących wsparcie dla MPI w Pythonie. Jednym z nich jest `mpi4py` (http://mpi4py.scipy.org/).

Skrypt MPI w Pythonie musi być uruchomiony poleceniem

```
$mpirun -n N
```

gdzie `N` to liczba procesów biorących udział w obliczeniach (może być większa niż liczba dostępnych rdzeni).

Przykład 1 - dwa procesy, jeden wysyła dane do drugiego

In [50]:

```
%%file mpitest.py

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

from mpi4py import MPI

#inicjalizacja puli procesów
comm = MPI.COMM_WORLD
#id poszczególnych procesów
rank = comm.Get_rank()



if rank == 0:                           #master
   data = [1.0, 2.0, 3.0, 4.0]          #stwórz dane
   comm.send(data, dest=1, tag=11)      #wyślij do węzła 1
elif rank == 1:                         #slave
   data = comm.recv(source=0, tag=11)   #odbierz dane z węzła 0

print("rank =", rank, ", data =", data) #wypisz na ekran (wszystkie procesy)
```

```
Overwriting mpitest.py
```

In [51]:

```
!mpirun -n 2 python3 mpitest.py
```

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

Przykład 2 - dwa procesy, jeden wysyła tablicę `numpy` do drugiego

In [52]:

```
%%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)       #dane do wysłania
   comm.Send(data, dest=1, tag=13)    #wysłanie (dest - id odbiorcy, tag - etykieta
danych)
```

```
elif rank == 1:
   data = numpy.empty(10, dtype=numpy.float64) #kontener na odbiór danych
   comm.Recv(data, source=0, tag=13)           #odbieramy dane (source - id nadawcy,
tag - etykieta danych)

print("rank =", rank, ", data =", data)         #wypisujemy na ekran
```

```
Overwriting mpi-numpy-array.py
```

Zwróćmy uwagę, że w przykładzie drugim użyliśmy metod Send i Recv, natomiast w przykładzie pierwszym - send i recv. Te pisane z małej litery dotyczą wbudowanych typów Pythona.

In [53]:

```
!mpirun -n 2 python3 mpi-numpy-array.py
```

```
rank = 0 , data = [ 0.1619755   0.56093871  0.15846384  0.35786488  0.42481524
0.45592277
  0.56990516  0.39924759  0.79135938  0.9851006 ]
rank = 1 , data = [ 0.1619755   0.56093871  0.15846384  0.35786488  0.42481524
0.45592277
  0.56990516  0.39924759  0.79135938  0.9851006 ]
```

Przykład 3 - mnożenie macierzy przez wektor

In [54]:

```
import numpy
# przygotowujemy dane i zapisujemy je do pliku
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 [55]:

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

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

from mpi4py import MPI
import numpy

#inicjalizacja klastra
comm = MPI.COMM_WORLD
#id poszczególnych procesów
rank = comm.Get_rank()
#rozmiar klastra
p = comm.Get_size()
```

```
def matvec(comm, A, x):
    m = A.shape[0] // p
    #każdy proces dostaje swoją "działkę" do policzenia
    y_part = numpy.dot(A[rank * m:(rank+1)*m], x)
    #kontener na wpisanie wyniku
    y = numpy.zeros_like(x)
    #zebranie wszystkich wyników i wpisanie ich do zmiennej 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)
    #porównanie wartości wyliczonej lokalnie i z wykorzystaniem MPI
    print("sum(y - y_mpi) =", (y - y_mpi).sum())

Overwriting mpi-matrix-vector.py
```

In [56]:

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

[ 4.72423389  3.55884908  3.26906473  2.88160532  3.17809718  2.82277663
  3.66217805  3.73236634  3.77571786  2.88770737  3.26158336  3.58158591
  3.87649158  3.00974665  2.95046447  3.58694629]
sum(y - y_mpi) = 0.0
```

Przykład 4 - suma elementów wektora

In [47]:

```
# przygotowujemy dane i zapisujemy je do pliku
N = 128
a = numpy.random.rand(N)
numpy.save("random-vector.npy", a)
```

In [48]:

```
%%file mpi-psum.py

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

from mpi4py import MPI
import numpy as np
```

```
def psum(a):
    #id procesu
    r = MPI.COMM_WORLD.Get_rank()
    #rozmiar klastra
    size = MPI.COMM_WORLD.Get_size()
    #liczba elementów przypadających na jeden proces
    m = len(a) // size
    #suma lokalna (na każdym hoście inna)
    locsum = np.sum(a[r*m:(r+1)*m])
    #kontener do wpisania wyniku
    rcvBuf = np.array(0.0, 'd')
    #zebranie sum cząstkowych
    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 lokalne
    print("sum =", s, ", numpy sum =", a.sum())
```

```
Overwriting mpi-psum.py
```

In [49]:

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

```
sum = 57.60385543476975 , numpy sum = 57.6038554348
```

Cython i OpenMP

OpenMP (ang. *Open Multi-Processing*) to wieloplatformowy interfejs programowania aplikacji (API) umożliwiający tworzenie programów komputerowych dla systemów wieloprocesorowych z pamięcią dzieloną. Niestety, interfejsu tego **nie można** wykorzystywać bezpośrednio w Pythonie, ponieważ jego standardowa implementacja używa globalnej blokady interpretera (ang. *Global Interpreter Lock*, GIL).

Aplikacje napisane w językach używających GIL muszą używać osobnych procesów, aby w pełni wykorzystać potencjał maszyn wieloprocesorowych (wówczas każdy proces ma własną blokadę). Blokada jest jednak zwalniana w momencie, kiedy z poziomu Pythona wywoływany jest kompilowany kod. Dlatego np. Cython umożliwia obejście problemu związanego z blokada i wykonywanie obliczeń z wykorzystaniem OpenMP.

In [57]:

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

print("Liczba rdzeni w systemie: %d" % N_core)
```

```
Liczba rdzeni w systemie: 4
```

Rozważmy prosty przykład wykorzystania OpenMP w Cythonie:

In [58]:

```
%load_ext Cython
```

In [59]:

```
%%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

    # znieś blokadę, aby można było skorzystać z OpenMP
    with nogil, parallel():
        N = openmp.omp_get_num_threads()
        n = openmp.omp_get_thread_num()
        with gil:
            print("Liczba wątków: %d, numer wątku: %d\n" % (N, n))
```

In [60]:

```
cy_openmp_test()
```

```
Liczba wątków: 4, numer wątku: 3
Liczba wątków: 4, numer wątku: 2


Liczba wątków: 4, numer wątku: 1

Liczba wątków: 4, numer wątku: 0
```

Przykład - mnożenie macierzy przez wektor

In [57]:

```
# przygotuj dane do obliczeń
N = 4 * N_core

M = numpy.random.rand(N, N)
```

```
x = numpy.random.rand(N)
y = numpy.zeros_like(x)
```

Najpierw prosta implementacja w Cythonie (bez wykorzystania OpenMP):

In [58]:

```
%%cython

cimport cython
cimport numpy
import numpy

@cython.boundscheck(False) #jesteśmy pewni zakresów
@cython.wraparound(False)  #wyłaczamy sprawdzanie ujemnych indeksów
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 [59]:

```
# sprawdzamy poprawność działania funkcji
y = numpy.zeros_like(x)
cy_matvec(M, x, y)
numpy.dot(M, x) - y
```

Out[59]:

```
array([  8.88178420e-16,  -1.77635684e-15,   0.00000000e+00,
        -1.77635684e-15,  -3.55271368e-15,   8.88178420e-16,
         0.00000000e+00,  -1.77635684e-15,   1.77635684e-15,
         0.00000000e+00,  -1.77635684e-15,   1.77635684e-15,
        -1.77635684e-15,   8.88178420e-16,  -1.77635684e-15,
         1.77635684e-15,   8.88178420e-16,   1.77635684e-15,
        -1.77635684e-15,   1.77635684e-15,  -2.66453526e-15,
         8.88178420e-16,   0.00000000e+00,   0.00000000e+00,
         1.77635684e-15,   0.00000000e+00,   8.88178420e-16,
        -8.88178420e-16,   1.77635684e-15,   0.00000000e+00,
        -1.77635684e-15,  -1.77635684e-15])
```

In [60]:

```
#a teraz testy wydajności
```

```
%timeit numpy.dot(M, x)
```

```
The slowest run took 29.14 times longer than the fastest. This could mean that an
intermediate result is being cached.
1000000 loops, best of 3: 565 ns per loop
```

In [61]:

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

```
The slowest run took 6.74 times longer than the fastest. This could mean that an
intermediate result is being cached.
1000000 loops, best of 3: 1.63 µs per loop
```

Widzimy, że wersja w Cythonie jest wolniejsza od funkcji dostępnej w module NumPy. Spróbujmy poprawić jej wydajność z wykorzystaniem OpenMP:

In [62]:

```
%%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
        #każdy z wątków dostaje swój wycinek macierzy
        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 [63]:

```
# sprawdzenie wyników
y = numpy.zeros_like(x)
cy_matvec_omp(M, x, y)
```

```
print(numpy.dot(M, x) - y)
```

```
[  8.88178420e-16  -1.77635684e-15   0.00000000e+00  -1.77635684e-15
  -3.55271368e-15   8.88178420e-16   0.00000000e+00  -1.77635684e-15
   1.77635684e-15   0.00000000e+00  -1.77635684e-15   1.77635684e-15
  -1.77635684e-15   8.88178420e-16  -1.77635684e-15   1.77635684e-15
   8.88178420e-16   1.77635684e-15  -1.77635684e-15   1.77635684e-15
  -2.66453526e-15   8.88178420e-16   0.00000000e+00   0.00000000e+00
   1.77635684e-15   0.00000000e+00   8.88178420e-16  -8.88178420e-16
   1.77635684e-15   0.00000000e+00  -1.77635684e-15  -1.77635684e-15]
```

In [64]:

```
#testy wydajności raz jeszcze
%timeit numpy.dot(M, x)
```

```
The slowest run took 27.36 times longer than the fastest. This could mean that an
intermediate result is being cached.
1000000 loops, best of 3: 571 ns per loop
```

In [65]:

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

```
The slowest run took 95.63 times longer than the fastest. This could mean that an
intermediate result is being cached.
10000 loops, best of 3: 47.4 µs per loop
```

Tym razem wersja w Cythonie jest dużo wolniejsza od funkcji z modułu NumPy. Przy tak niewielkich macierzach narzut związany z OpenMP i obsługą wątków jest tak duży, że nie opłaca się korzystać z tej technologii. Sprawdźmy jednak, jak będą wyglądały wyniki dla dużo większych macierzy i wektorów:

In [66]:

```
N_vec  = numpy.arange(50, 2500, 50) * N_core
```

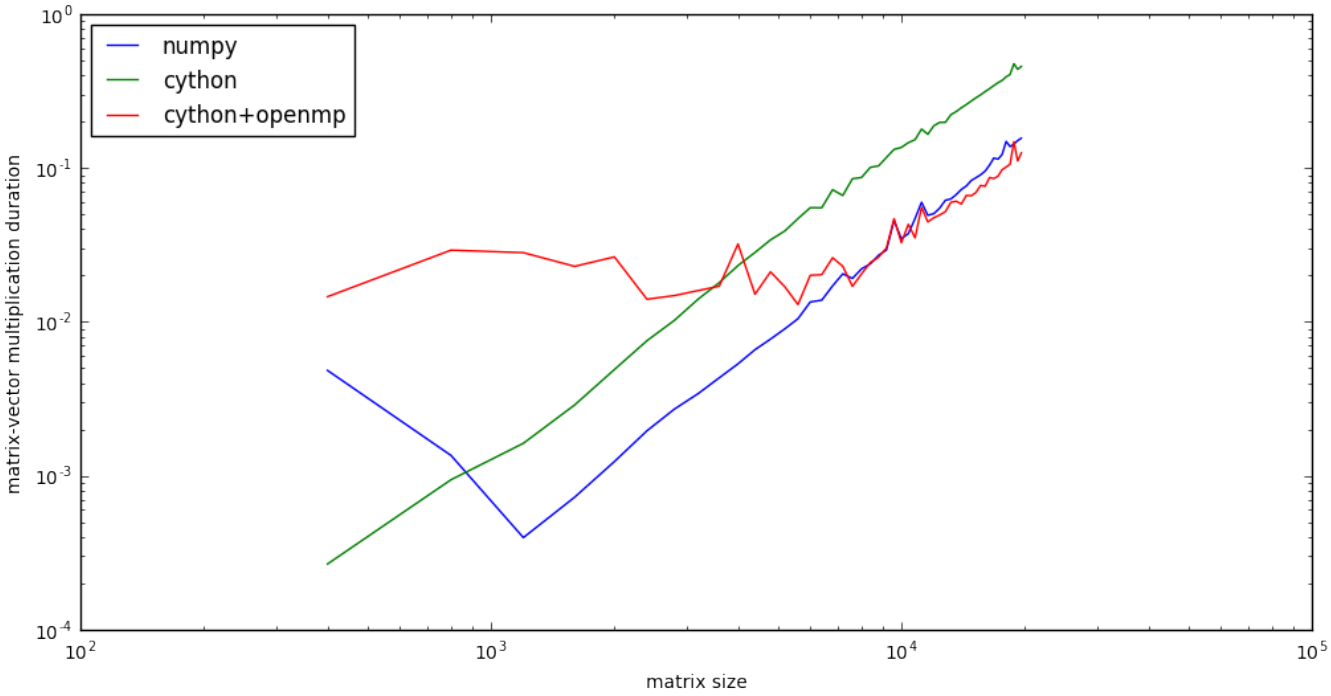
In [67]:

```
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 [68]:

```
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");
```



Dla większych macierzy implementacja w Cython+OpenMP jest już szybsza:

In [69]:

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

Out[69]:

```
1.2702648666068119
```

Mimo to jesteśmy jeszcze bardzo daleko od osiągnięcia przyspieszenia bliskiego teoretycznemu

ograniczeniu:

In [70]:

```
N_core
```

Out[70]:

8

Python i OpenCL

OpenCL (ang. *Open Computing Language*) to API wspomagające pisanie aplikacji działających na platformach heterogenicznych, tzn. składających się z jednostek obliczeniowych różnego rodzaju (m.in. CPU, GPU). Moduł `pyopencl` pozwala na kompilowanie, ładowanie i wykonywanie kodu OpenCL z poziomu Pythona.

In [4]:

```
%%file opencl-dense-mv.py

import pyopencl as cl
import numpy
import time

# problem size
n = 10000

# platform
platform_list = cl.get_platforms()
platform = platform_list[0]

# device
device_list = platform.get_devices()
print(device_list)
device = device_list[0]

if False:
    print("Platform name:" + platform.name)
    print("Platform version:" + platform.version)
    print("Device name:" + device.name)
    print("Device type:" + cl.device_type.to_string(device.type))
    print("Device memory: " + str(device.global_mem_size//1024//1024) + ' MB')
    print("Device max clock speed:" + str(device.max_clock_frequency) + ' MHz')
    print("Device compute units:" + str(device.max_compute_units))

# context
ctx = cl.Context([device]) # or we can use cl.create_some_context()

# command queue
```

```
queue = cl.CommandQueue(ctx)

# kernel
KERNEL_CODE = """
//
// Matrix-vector multiplication: r = m * v
//
#define N %(mat_size)d
__kernel
void dmv_cl(__global float *m, __global float *v, __global float *r)
{
    int i, gid = get_global_id(0);

    r[gid] = 0;
    for (i = 0; i < N; i++)
    {
        r[gid] += m[gid * N + i] * v[i];
    }
}
"""

kernel_params = {"mat_size": n}
program = cl.Program(ctx, KERNEL_CODE % kernel_params).build()

# data
A = numpy.random.rand(n, n)
x = numpy.random.rand(n, 1)

# host buffers
h_y = numpy.empty(numpy.shape(x)).astype(numpy.float32)
h_A = numpy.real(A).astype(numpy.float32)
h_x = numpy.real(x).astype(numpy.float32)

# device buffers
mf = cl.mem_flags
d_A_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_A)
d_x_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_x)
d_y_buf = cl.Buffer(ctx, mf.WRITE_ONLY, size=h_y.nbytes)

# execute OpenCL code
t0 = time.time()
event = program.dmv_cl(queue, h_y.shape, None, d_A_buf, d_x_buf, d_y_buf)
event.wait()
cl.enqueue_copy(queue, h_y, d_y_buf)
t1 = time.time()

print("opencl elapsed time =", t1-t0)
```

```
# Same calculation with numpy
t0 = time.time()
y = numpy.dot(h_A, h_x)
t1 = time.time()

print("numpy elapsed time =", t1-t0)

# see if the results are the same
print("max deviation =", numpy.abs(y-h_y).max())

Overwriting opencl-dense-mv.py
```

In [5]:

```
!python3 opencl-dense-mv.py

[<pyopencl.Device 'Intel(R) HD Graphics Skylake Desktop GT2' on 'Intel Gen OCL
Driver' at 0x7fe7393b62c0>]
opencl elapsed time = 0.3990914821624756
numpy elapsed time = 0.018672466278076172
max deviation = 0.0170898
```

In [ ]: