No GIL - Parallel Python Programming with Cython and OpenMP

EuroSciPy 2012, August 26, 2012

Brussels, Belgium

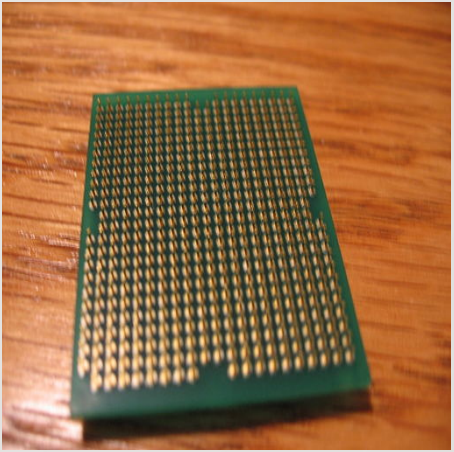
Author: Dr.-Ing. Mike Müller

Email: mmueller@python-academy.de

Python Academy
Training & Consulting

The (Future) Present is Parallel

- CPUs aren't getting (much) faster anymore
- They just get more cores



Shared vs. Distributed Memory

- There are two major memory models for parallel programming

  1. Distributed memory or message passing
  2. Shared memory

MPI

- mpi4py
- pypar
- pyMPI

mpi4py
MPI for Python - Python bindings for MPI

pypar

Non-MPI

- multiprocessing
- Pyro
- ParallelPython
- IPython cluster
- ...

Cython

- Mixture between Python and C
- Standard Python is valid Cython
- Gradually add more C-ish features
- Call existing C/C++ code (source and libs)
- Compile to Python extension (`*.so` or `*.pyd`)

Cython Workflow

- Write Cython code in `*.pyx` file
- Compile, i.e execute your `setup.py`
- Get extension module (`*.so`, `*.pyd`)
- Use your extension from Python

Example - Cython Code

```
# file: cy_101.pyx

# could be used in Python directly
def pure_python_func(a, b):
    return a + b

# cannot be called from Python
cdef double cython_func(double a, double b):
    return a + b

# wrapper to call from Python
def typed_python_func(a, b):
    return cython_func(a, b)
```

Example - Compile

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

setup(
    name = 'cython101',
    ext_modules = cythonize("cy_101.pyx", annotate=True),
)
```

Example - Use

```
# file cy_101_test.py


import cy_101

a = 10
b = 20
print cy_101.pure_python_func(a, b)
print cy_101.typed_python_func(a, b)
```

```
python cy_101_test.py
30
30.0
```

Example - Cython at Work

- 1778 lines of C code
- Annotations indicate Python (yellow) and pure C (white), click to see C source

Generated by Cython 0.17pre on Thu Aug 23 09:30:23 2012

Raw output: cy_101.c

```
# file: cy_101.pyx

# could use in Python directly
def pure_python_func(a, b):
    return a + b

# cannot be called from Python
cdef double cython_func(double a, double b):
    return a + b

# wrapper to call from Python
def typed_python_func(a, b):
    return cython_func(a, b)
```

The Buffer Interface

- NumPy-inspired standard to access C data structures from Python
- Cython supports it
- Fewer conversions between Python and C data types

```
typedef struct bufferinfo {
    void *buf;                 // buffer memory pointer
    PyObject *obj;             // owning object
    Py_ssize_t len;            // memory buffer length
    Py_ssize_t itemsize;       // byte size of one item
    int readonly;              // read-only flag
    int ndim;                  // number of dimensions
    char *format;              // item format description
    Py_ssize_t *shape;         // array[ndim]: length of each dimension
    Py_ssize_t *strides;       // array[ndim]: byte offset to next item in each dimension
    Py_ssize_t *suboffsets;    // array[ndim]: further offset for indirect indexing
    void *internal;            // reserved for owner
} Py_buffer;
```

Example

- `a` and `b` are 2D NumPy arrays with same shape
- `(a + b) * 2 + a * b`
- Size: 2000 x 2000

With Multiprocessing

```
def test_multi(a, b, pool):
    assert a.shape == b.shape
    v = a.shape[0] // 2
    h = a.shape[1] // 2
    quads = [(slice(None, v), slice(None, h)),
             (slice(None, v), slice(h, None)),
             (slice(v, None), slice(h, None)),
             (slice(v, None), slice(None, h))]
    results = [pool.apply_async(test_numpy, [a[quad], b[quad]])
               for quad in quads]
    output = numpy.empty_like(a)
    for quad, res in zip(quads, results):
        output[quad] = res.get()
    return output
```

- multiprocessing solution is 6 times slower than NumPy solution

The Buffer Interface From Cython

```
import numpy
import cython

@cython.boundscheck(False)
@cython.wraparound(False)
def func(object[double, ndim=2] buf1 not None,
         object[double, ndim=2] buf2 not None,
         object[double, ndim=2] output=None,):
    cdef unsigned int x, y, inner, outer
    if buf1.shape != buf2.shape:
        raise TypeError('Arrays have different shapes: %s, %s' % (buf1.shape,
                           buf2.shape))
    if output is None:
        output = numpy.empty_like(buf1)
    outer = buf1.shape[0]
    inner = buf1.shape[1]
    for x in xrange(outer):
        for y in xrange(inner):
            output[x, y] = ((buf1[x, y] + buf2[x, y]) * 2 +
                              buf1[x, y] * buf2[x, y])
    return output
```

Memory Views

```
import numpy
import cython

@cython.boundscheck(False)
@cython.wraparound(False)
cdef add_arrays_2d_views(double[:,:] buf1,
                         double[:,:] buf2,
                         double[:,:] output):
    cdef unsigned int x, y, inner, outer
    outer = buf1.shape[0]
    inner = buf1.shape[1]
    for x in xrange(outer):
        for y in xrange(inner):
            output[x, y] = ((buf1[x, y] + buf2[x, y]) * 2 +
                             buf1[x, y] * buf2[x, y])
    return output
```

Memory Views

```
@cython.boundscheck(False)
@cython.wraparound(False)
def add_arrays_2d(object[double, ndim=2] buf1 not None,
                  object[double, ndim=2] buf2 not None,
                  object[double, ndim=2] output=None,):
    cdef unsigned int v, h
    if buf1.size != buf2.size:
        raise TypeError('Arrays have different sizes: %d, %d' % (buf1.size,
                             buf2.size))
    if buf1.shape != buf2.shape:
        raise TypeError('Arrays have different shapes: %s, %s' % (buf1.shape,
                             buf2.shape))
    if output is None:
        output = numpy.empty_like(buf1)
    v = buf1.shape[0] // 2
    h = buf1.shape[1] // 2
    quad1 = slice(None, v), slice(None, h)
    quad2 = slice(None, v), slice(h, None)
    quad3 = slice(v, None), slice(h, None)
    quad4 = slice(v, None), slice(None, h)
    add_arrays_2d_views(buf1[quad1], buf2[quad1], output[quad1])
```

```python
    add_arrays_2d_views(buf1[quad2], buf2[quad2], output[quad2])
    add_arrays_2d_views(buf1[quad3], buf2[quad3], output[quad3])
    add_arrays_2d_views(buf1[quad4], buf2[quad4], output[quad4])
    return output
```

OpenMP

- De-facto standard for shared memory parallel programming

The OpenMP API supports multi-platform shared-memory parallel programming in C/C++ and Fortran. The OpenMP API defines a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.

-- from openmp.org

OpenMP with Cython - Threads

```
# distutils: extra_compile_args = -fopenmp
# distutils: extra_link_args = -fopenmp

import numpy
import cython
from cython cimport parallel
```

OpenMP with Cython - Threads

```
@cython.boundscheck(False)
@cython.wraparound(False)
cdef int add_arrays_2d_views(double[:,:] buf1, double[:,:] buf2,
                             double[:,:] output) nogil:
    cdef unsigned int x, y, inner, outer
    outer = buf1.shape[0]
    inner = buf1.shape[1]
    for x in xrange(outer):
        for y in xrange(inner):
            output[x, y] = ((buf1[x, y] + buf2[x, y]) * 2 +
                             buf1[x, y] * buf2[x, y])
    return 0
```

OpenMP with Cython - Threads

```
@cython.boundscheck(False)
@cython.wraparound(False)
def add_arrays_2d(double[:,:] buf1 not None,
                  double[:,:] buf2 not None,
                  double[:,:] output=None,):
    cdef unsigned int v, h, thread_id
    if buf1.shape[0] != buf2.shape[0] or buf1.shape[1] != buf2.shape[1]:
        raise TypeError('Arrays have different shapes: (%d, %d) (%d, %d)' % (
                        buf1.shape[0], buf1.shape[1], buf2.shape[0],
                        buf1.shape[1],))
    if output is None:
        output = numpy.zeros_like(buf1)
    v = buf1.shape[0] // 2
    h = buf1.shape[1] // 2
    ids = []
    with nogil, parallel.parallel(num_threads=4):
        thread_id = parallel.threadid()
        with gil:
            ids.append(thread_id)
        if thread_id == 0:
            add_arrays_2d_views(buf1[:v,:h], buf2[:v,:h], output[:v,:h])
```

```python
        elif thread_id == 1:
            add_arrays_2d_views(buf1[:v,h:], buf2[:v,h:], output[:v,h:])
        elif thread_id == 2:
            add_arrays_2d_views(buf1[v:,h:], buf2[v:,h:], output[v:,h:])
        elif thread_id == 3:
            add_arrays_2d_views(buf1[v:,:h], buf2[v:,:h], output[v:,:h])
    print ids
    return output
```

OpenMP with Cython - Parallel Range

```
# distutils: extra_compile_args = -fopenmp
# distutils: extra_link_args = -fopenmp

import numpy

import cython
from cython cimport parallel
```

OpenMP with Cython - Parallel Range

```
@cython.boundscheck(False)
@cython.wraparound(False)
def func(object[double, ndim=2] buf1 not None,
         object[double, ndim=2] buf2 not None,
         object[double, ndim=2] output=None,
         int num_threads=2):
    cdef unsigned int x, y, inner, outer
    if buf1.shape != buf2.shape:
        raise TypeError('Arrays have different shapes: %s, %s' % (buf1.shape,
                               buf2.shape))
    if output is None:
        output = numpy.empty_like(buf1)
    outer = buf1.shape[0]
    inner = buf1.shape[1]
    with nogil, cython.boundscheck(False), cython.wraparound(False):
        for x in parallel.prange(outer, schedule='static',
                                 num_threads=num_threads):
            for y in xrange(inner):
                output[x, y] = ((buf1[x, y] + buf2[x, y]) * 2 +
                                 buf1[x, y] * buf2[x, y])
    return output
```

Speedup

Threads	Speedup
1	1.0
2	1.6
3	1.8
4	2.0

Conclusions

- Cython + OpenMP allow to work without the GIL
- Threads run in parallel for CPU-bound tasks
- There is a price:
  - You need to write more code
  - You loose part of the Python safety net
  - You need to know C and learn Cython