Python: performance and parallelism

Fernando Pérez
Fernando.Perez@berkeley.edu

Brain Imaging Center, UC Berkeley

CS294 Modern Parallel Languages
Oct 3, 2013
Outline

1. Python
2. Language and standard libraries
3. External Libraries/projects
4. IPython for parallelism
Outline

1. Python
2. Language and standard libraries
3. External Libraries/projects
4. IPython for parallelism
Python: highly dynamic language

Python is strongly but dynamically typed

In [5]: x=42
   ...: print 'x=', x, 'type(x)=', type(x), 'x*2=', x*2
   ...: print
   ...: x="Flexible!"
   ...: print 'x=', x, 'type(x)=', type(x), 'x*2=', x*2
   ...:
x = 42 type(x)= <type 'int'> x*2 = 84

x = Flexible! type(x)= <type 'str'> x*2 = Flexible!Flexible!

- Types are rich but removed from the hardware
  - ints: arbitrary precision
  - floats: wrapped C doubles
  - lists, tuples: far from double*

- Very simple, stack-based Virtual Machine
  - minimal optimization
  - VM overview: http://www.troeger.eu/teaching/pythonvm08.pdf
In [5]: import dis  # Python’s disassembler
    ...: src='''
    ...: s=0
    ...: for i in range(10):
    ...:     s += i
    ...: '''
    ...: code = compile(src, '<input>', 'exec')
    ...: print dis.dis(code)

2 0 LOAD_CONST 0 (0)
 3 STORE_NAME 0 (s)
3 6 SETUP_LOOP 30 (to 39)
 9 LOAD_NAME 1 (range)
12 LOAD_CONST 1 (10)
15 CALL_FUNCTION 1
18 GET_ITER
 19 FOR_ITER 16 (to 38)
22 STORE_NAME 2 (i)
4 25 LOAD_NAME 0 (s)
28 LOAD_NAME 2 (i)
31 INPLACE_ADD
32 STORE_NAME 0 (s)
35 JUMP_ABSOLUTE 19
38 POP_BLOCK
39 LOAD_CONST 2 (None)
42 RETURN_VALUE
In [5]: `import` dis  # Python’s disassembler
   ...: src='''
   ...: s=0
   ...: for i in range(10):
   ...:     s += i
   ...: ''
   ...: code = compile(src, '<input>', 'exec')
   ...: `print` dis.dis(code)

2 0 LOAD_CONST 0 (0)
 3 STORE_NAME 0 (s)
3 6 SETUP_LOOP 30 (to 39)
 9 LOAD_NAME 1 (range)
12 LOAD_CONST 1 (10)
15 CALL_FUNCTION 1
18 GET_ITER
 19 FOR_ITER 16 (to 38)
22 STORE_NAME 2 (i)
4 25 LOAD_NAME 0 (s)
28 LOAD_NAME 2 (i)
31 INPLACE_ADD
32 STORE_NAME 0 (s)
35 JUMP_ABSOLUTE 19
 38 POP_BLOCK
 39 LOAD_CONST 2 (None)
42 RETURN_VALUE
Threading and parallelism in Python: overview

- Multiple implementations of the Virtual Machine:
  - **CPython**: pure C, ‘reference’
  - **IronPython**: .NET
  - **Jython**: Java

- Their threading behaviors differ, I’ll focus on CPython

- Native threads supported, but of limited use.

- **Global interpreter lock (GIL)**: only **one** thread can modify any python data structure

- **No language-specific primitives** for parallelism.
The infamous Global Interpreter Lock in CPython

- Historical reasons, simplicity of implementation
- All attempts at removing it have failed
  - 2x loss of performance is not acceptable
- Threads only good for i/o bound tasks.
- Mostly useless for CPU-bound ones.
- Can operate on pre-allocated arrays, but:
  - code must be in C/C++/Fortran/Cython
  - be very careful with locking if code is not atomic at Python level

The best possible reference on the GIL: David Beazley's work

http://www.dabeaz.com/GIL
The infamous Global Interpreter Lock in CPython

- Historical reasons, simplicity of implementation
- All attempts at removing it have failed
  - 2x loss of performance is not acceptable
- Threads only good for i/o bound tasks.
- Mostly useless for CPU-bound ones.
- Can operate on pre-allocated arrays, but:
  - code must be in C/C++/Fortran/Cython
  - be very careful with locking if code is not atomic at Python level

The best possible reference on the GIL: David Beazley's work
http://www.dabeaz.com/GIL
With these limitations, why should you care?

- **Very dynamic, introspective language**
  
  In [13]: `def f(x, y=1, **kw):
     ...:     """A docstring""
     ...:     return x+y
  
  In [14]: f.func_code
  Out[14]: <code object f at 0xac8cec0, file "<ipython console>", line 1>
  
  In [15]: f.func_defaults
  Out[15]: (1,)

- **It’s open source: the perfect playground**
  - Create a modified VM if you want

- **It’s use in numerical/scientific computing is exploding**
  - There’s a real need and much to be done.
  - Your ideas will have a real impact!
  - GPUs, local multicore, clusters... even large scale supercomputing?
Parallelism in Python

- **In-process (mind the GIL)**
  - Data parallelism with threaded libraries
  - Numpy/scipy can use a threaded ATLAS
  - Numexpr: a 'numpy VM'
  - Theano: a library that thinks it's a compiler
  - GPU-based solutions: PyCuda/PyOpenCL, scikits.cuda.
  - Hand-written threaded code...

- **Out-of-process**
  - The multiprocessing module
  - Python futures
  - Communicating Sequential Processes, ParallelPython, ... many more
  - IPython (I’m obviously biased)
Outline

1. Python
2. Language and standard libraries
3. External Libraries/projects
4. IPython for parallelism
Multiprocessing
Module: multiprocessing

- Built-in since version 2.6 (available for earlier versions)
- An API that closely follows the threading API, but using processes
- Useful high-level objects
  - Process, Process pool, Namespaces, Listeners, ...
- Uses fork() on posix (hence there are some limitations)

A simple example

```python
from multiprocessing import Process

def f(name):
    print 'hello', name

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```
Python futures
In Python 3.2 as concurrent.futures

- High-level interface for asynchronously executing callables.
- Executors and Futures are the key objects

A simple example

```python
from shutil import copy
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(copy, 'src1.txt', 'dest1.txt') # returns a Future
    e.submit(copy, 'src2.txt', 'dest2.txt')
    e.submit(copy, 'src3.txt', 'dest3.txt')
    e.submit(copy, 'src3.txt', 'dest4.txt')
```

Futures have useful methods:

- `f.cancel()`
- `f.running()`
- `f.result(timeout=None)`
- `f.add_done_callback(func)`
Decorators
Dynamic function manipulations

```python
import time

def timed(func):
    def wrapper(n, **kw):
        st = time.clock()
        out = func(n, **kw)
        print "Time used: %.2f s" % (time.clock()-st)
        return out
    return wrapper

@timed
def ssq(n):
    "Sum of squares"
    return sum(i**2 for i in range(n))
```

Produces

In [3]: ssq(100000)
Time used: 0.12 s
Out[3]: 333328333350000L

In [4]: ssq(1000000)
Time used: 1.84 s
Out[4]: 333332833333500000L
Some decorator tricks
For more on this, see: http://fperez.org/py4science/decorators.html

- Decorators **normally** return a modified function...
- But they can do **whatever they want**!

```python
def funnydeco(func):
    return 'Hi, I am a decorator...

@funnydeco
def f(x):
    return x+1
```

This decorator produces:

```python
In [2]: f(10)
Traceback (most recent call last):
  File "<ipython console>", line 1, in <module>
TypeError: 'str' object is not callable

In [3]: print f
Hi, I am a decorator...
```
What does this have to do with parallelism?

Consider a simple pair of 'loop body' and 'loop summary' functions:

```python
def do_work(data, i):
    return data[i]/2

def summarize(results):
    return sum(results)
```

and some 'dataset' (here just a list of 10 numbers)

```python
count = 10
data = [3.0*j for j in range(count)]
```

that has to be processed, done here with a serial function:

```python
def loop_serial():
    results = [None]*count

    for i in range(count):
        results[i] = do_work(data, i)

    return summarize(results)
```
Now let's look for clean syntax to do this in parallel...

```python
def for_each(iterable):
    """This decorator-based loop does a normal serial run. But in principle it could be doing the dispatch remotely""
    def call(func):
        map(func, iterable)  # This could be IPython's parallel map
        # or a gpu dispatch...
    return call
```

This is the actual code of the decorator-based loop:

```python
def loop_deco():
    results = [None]*count

    @for_each(range(count))
    def loop(i):
        results[i] = do_work(data, i)

    return summarize(results)
```

Validate that both versions really do the same thing

```python
In [34]: assert loop_serial() == loop_deco()
...:     print 'OK'
OK
```
Compare normal and decorator based syntax

The serial loop (just the body of the loop)

```python
for i in range(count):
    results[i] = do_work(data, i)
```

The equivalent part in the decorator version

```python
@for_each(range(count))
def loop(i):
    results[i] = do_work(data, i)
```

Decorator benefits

- A named closure
- With controlled access to parameters
- With access to enclosing scope
- With optional return values

This provides semantics extremely similar to Apple’s Grand Central Dispatch (and their GCC extensions that go along with GCD)
Outline

1. Python
2. Language and standard libraries
3. External Libraries/projects
4. IPython for parallelism
Numpy and Scipy: ‘Out of the box’ parallelism?

- Not great...
- Can be built against a threaded ATLAS or the Intel Math Kernel Library (MKL)
  - This can give multithreaded support to many linear algebra operations.
- Manual effort with C/Fortran + OpenMP can give you some gains...
  - but with a fair amount of pain
Numexpr
An expression compiler for numpy

Approach
- Compile Numpy expressions to equivalent Python code...
- Block operations carefully
- execute on a special-purpose mini-VM (written in C)

Benefits
- Reduce the use of temporaries.
- Be cache-friendly.
- Support threads natively for all operations.
- Support Intel Vector Math Library and MKL.
Evaluating simple expressions

```python
>>> import numpy as np
>>> import numexpr as ne

>>> a = np.arange(1e6)  # Choose large arrays for high performance
>>> b = np.arange(1e6)

>>> ne.evaluate("a + 1")  # a simple expression
array([ 1.00000000e+00, 2.00000000e+00, 3.00000000e+00, ..., 9.99998000e+05, 9.99999000e+05, 1.00000000e+06])

>>> ne.evaluate('a*b-4.1*a > 2.5*b')  # a more complex one
array([False, False, False, ..., True, True, True], dtype=bool)
```
Numexpr timings

Comparisons to Numpy and thread usage

```python
>>> timeit a**2 + b**2 + 2*a*b
10 loops, best of 3: 35.9 ms per loop

>>> ne.set_num_threads(1)  # use 1 thread (on a 6-core machine)

>>> timeit ne.evaluate("a**2 + b**2 + 2*a*b")
100 loops, best of 3: 9.28 ms per loop  # 3.9x faster than NumPy

>>> ne.set_num_threads(4)  # use 4 threads (on a 6-core machine)

>>> timeit ne.evaluate("a**2 + b**2 + 2*a*b")
100 loops, best of 3: 4.17 ms per loop  # 8.6x faster than NumPy
```
Note: PiCloud is *not* open-source. I’ve only seen demos of it, I haven’t used it.
A library that thinks it’s the child of a compiler and a Computer Algebra System

- Declare and construct mathematical expressions (including numpy)
- Emit highly optimized code for them:
  - use of GPU for computations
  - constant folding
  - merging of similar subgraphs, to avoid redundant calculation
  - arithmetic simplification (e.g. $x*y/x \rightarrow y$, $-x \rightarrow x$)
  - inserting efficient BLAS operations (e.g. GEMM) in a variety of contexts
  - using inplace operations wherever it does not interfere with aliasing
  - loop fusion for elementwise sub-expressions
  - ...more
import theano
from theano import tensor

# declare two symbolic floating-point scalars
a = tensor.dscalar()
b = tensor.dscalar()

# create a simple expression
c = a + b

# convert the expression into a callable object that takes (a,b) values as input and computes a value for c
f = theano.function([a,b], c)

# bind 1.5 to 'a', 2.5 to 'b', and evaluate 'c'
assert 4.0 == f(1.5, 2.5)

SciPy 2010 presentation:  *Transparent GPU Computing with Theano*

Outline

1. Python
2. Language and standard libraries
3. External Libraries/projects
4. IPython for parallelism
Parallel computing: fully interactive
- development, debugging, testing, execution, monitoring, ...

Easy things should be easy, difficult things possible

Make parallel computing collaborative

More dynamic model for load balancing and fault tolerance

Seamless integration with other tools: plotting/visualization, system shell.

Also want to keep the benefits of traditional approaches:
- Should integrate with threads/MPI if appropriate
- Should be easy to integrate compiled code and libraries
IPython’s parallel architecture
Easy reuse and distribution of existing serial (‘normal’) codes.

High-level abstractions for embarrassingly parallel problems.

- Direct execution of code over the network: multiplexing interface.
- Out-of-the box task farming tools: task interface.

Task farming system is “low-latency” (not in the Myrinet sense...)

- can be integrated into more complex codes.

Implement any approach to parallelism you want:

- Synchronous or asynchronous execution of code on nodes.
- Task farming.
- Traditional Message Passing (MPI).
- Integrate hybrid codes.
- BYO.
Interactive IPython on ØMQ

- Kernel raw_input
- Requests to kernel
- Kernel output broadcast
- Request/Reply direction
- Lots of Sockets
- 1 Socket = 1 type of action
- Complicated picture
- Simple user code
Star Cluster: IPython parallel+Notebook on Amazon EC2
Justin Riley (MIT): http://web.mit.edu/star/cluster
Performance: raw throughput
Send No-op tasks as fast as possible, wait for results
Performance: arrays
Echo 16 random arrays of given size

![Graph showing performance of different methods (zmq, lru, twisted, sent) over varying array sizes (B). The x-axis represents size in bytes (10^2 to 10^7), and the y-axis represents tasks per second (10^0 to 10^4). The graph demonstrates how different methods perform under different array sizes.]