Python Concurrency

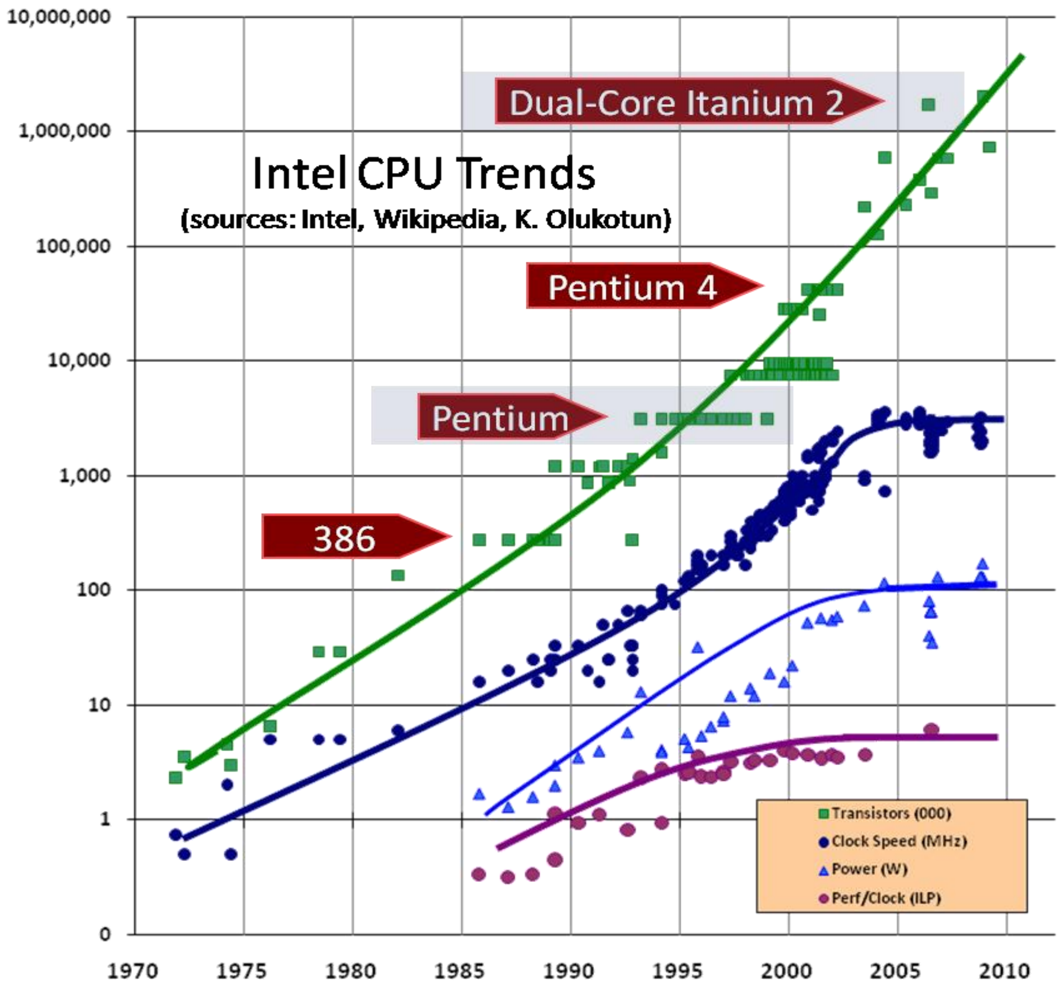
Threading, parallel and GIL adventures

Chris McCafferty, SunGard Global Services

Overview

- The free lunch is over – Herb Sutter
- Concurrency – traditionally challenging
- Threading
- The Global Interpreter Lock (GIL)
- Multiprocessing
- Parallel Processing
- Wrap-up – the Pythonic Way

Reminder - The Free Lunch Is Over



How do we get our free lunch back?

- Herb Sutter's paper at:
  - http://www.gotw.ca/publications/concurrency-ddj.htm
- Clock speed increase is stalled but number of cores is increasing
- Parallel paths of execution will reduce time to perform computationally intensive tasks
- But multi-threaded development has typically been difficult and fraught with danger

Threading

- Use the `threading` module, not `thread`
- Offers usual helpers for making concurrency a bit less risky: `Threads, Locks, Semaphores`…
- Use `logging`, not `print()`
- Don’t start a thread in module import (bad)
- Careful importing from daemon threads

Traditional management view of Threads

Baby pile of snakes, Justin Guyer

Managing Locks with ‘`with`’

- `With` keyword is your friend
- (compare with the ‘with file’ idiom)

```
import threading
rlock = threading.RLock()
with rlock:
     print "code that can only be executed while we acquire rlock"
     #lock is released at end of code block, regardless of exceptions
```

Atomic Operations in Python

- Some operations can be pre-empted by another thread
- This can lead to bad data or deadlocks
- Some languages offer constructs to help
- CPython has a set of atomic operations due to the operation of something called the GIL and the way the underlying C code is implemented
- This is a fortuitous implementation detail – ideally use RLocks to future-proof your code

CPython Atomic Operations

- reading or replacing a single instance attribute
- reading or replacing a single global variable
- fetching an item from a list
- modifying a list in place (e.g. adding an item using **append**)
- fetching an item from a dictionary
- modifying a dictionary in place (e.g. adding an item, or calling the **clear** method)

Example Processing Task

- Maclaurin was an 18th Century Scottish mathematician
- Typical Maclaurin series:

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots , \qquad |x| < 1$$

- This is easily decomposable: split the series up and then just add the results together in any order
- Easy to check the answer, great for testing threads

Threading Example

- See ThreadMaclaurin.py, compare with single-threaded SimpleMaclaurin.py
- Simple single-threaded example takes 4.522s

1 thread	4.623 secs for 12800000 iterations
2 threads	**6.195** secs for 12800000 iterations
4 threads	**6.047** secs for 12800000 iterations
6 threads	**6.357** secs for 12800000 iterations
8 threads	**6.006** secs for 12800000 iterations

The time taken goes **up** not down with more than one thread?!?

The Global Interpreter Lock (GIL)

- Python is an interpreted language
- Only one thread can run in the interpreter at once
- Constant locking and signaling to see which thread gets the GIL next
- Detailed effect of this depends on your operating system
- Heavily affects CPU-bound problems

GIL – not a showstopper

- This is a known problem – brilliant minds are currently working on solutions
- Affects Ruby too and any sensible interpreted language
- Not noticeable on I/O-bound applications
- Lots of other solutions: Jython, multiprocessing, Stackless Python…
- Think in a Pythonic Way.

Threading with Jython

- Jython has many of the CPython modules
- Bytecode compiled, not fully interpreted, runs on the Java Virtual Machine

1 thread	5.855 secs for 12800000 iterations
2 threads	2.836 secs for 12800000 iterations
4 threads	1.581 secs for 12800000 iterations
6 threads	1.323 secs for 12800000 iterations
8 threads	1.139 secs for 12800000 iterations

- That's more like it

Multiprocessing – no more GIL

Snakes on a Plain, by Linda Frost

Multiprocessing

- Jython doesn't have the multiprocessing module
- Each Python process has its own interpreter and GIL
- `multiprocessing` module makes managing processes and interprocess communication easy
- Use modules like `pickle` for passing payloads around
- Less worrying about shared memory and concurrency

Multiprocessing Example

- See MultiprocessMaclaurin.py for a simple example.
- Note use of a Queue to get the results back

1 thread	4.561 secs for 12800000 iterations
2 threads	2.339 secs
4 threads	1.464 secs
6 threads	1.201 secs
8 threads	1.120 secs

Multiprocessing - continued

- Remember there is an overhead associated with processes – don’t fork off thousands
- Full access to Cpython modules
- Be careful spawning processes from a script!
  - Child process needs to be able to import the script or module containing the target function
  - Can lead to recursive behaviour
  - This can lead to processes being spawned until the machine crashes

Avoid multiprocessing recursion

- The ways to avoid recursive behaviour are:
- Have the target method in another module/script
- Protect the executed code with a test for `__main__`:

```
if __name__ == '__main__':
    p = multiprocessing.Process(target=worker, args=(i,))
p.start()
```

- Use a properly object-oriented structure in your code

Parallel Python

- Parallel Python module `pp` supports breaking up into tasks
- Detects number CPUs to decide process pool size for tasks
- No GIL effect
- Easily spread the load onto another machine running a `pp` process

Parallel Python Example

- In ParallelMaclaurin.py we stop caring about the number of processes or threads
- We operate at a higher level of abstraction
- Example breaks the problem into 64 tasks
- Running on an 8 core desktop:
  - Time taken 1.050 secs for 12800000 iterations

Parallel Python for Big Data

- Job management and stats
- Symmetric or asymmetric computing
- Worry about decomposing and parallelising the task, not writing Locks and Semaphores
- Getting our free lunch back

Conclusions

- Python will support sensible `threading` constructs like any decent language
- Watch out for the GIL for CPU-bound tasks
- Switching to `multiprocessing` is easy
- Modules like `pp` support parallel processing and grid computing
- Lots of other options for I/O-bound problems: Stackless Python, Twisted…
- Many modules use threads sensibly behind the scenes
- Ideally, think Pythonicly – only move down the abstraction chain when you need to

Links

- Blog entry on much of this material
  - http://www.christophermccafferty.com/blog/2012/02/threading-in-python/
- David Beazley’s talks:
  - http://blip.tv/rupy-strongly-dynamic-conference/david-beazly-in-search-of-the-perfect-global-interpreter-lock-5727606
  - http://www.slideshare.net/dabeaz/in-search-of-the-perfect-global-interpreter-lock
  - http://blip.tv/carlfk/asynchronous-vs-threaded-python-2243317
- Herb Sutter’s The Free Lunch Is Over:
  - http://www.gotw.ca/publications/concurrency-ddj.htm

Thank you

- Chris McCafferty
  - http://christophermccafferty.com/blog
- Slides will be at:
  - http://christophermccafferty.com/slides
- Contact me at:
  - public@christophermccafferty.com