

GIL Isn't Evil

Dr Russel Winder

Partner, Concertant LLP

russel.winder@concertant.com

GIL Isn't Evil

Just a Little Bit Annoying

Dr Russel Winder

Partner, Concertant LLP

russel.winder@concertant.com

Aims and Objectives of the Session

- Investigate some of the consequences for programming of the “Multicore Revolution”.
- Compare and contrast various features for harnessing parallelism offered by Python and some other programming languages.
- Show that shared-memory multithreading is too low-level a technique for use in applications programming.

*Have a structured “chin wag” that is (hopefully) both illuminating **and** enlightening.*

Structure of the Session

- Introduce GIL.
- Look at some of the concurrency and parallelism support features in Python and give hints of the relationship to C, C++, Fortran, Java, Scala, Erlang, Haskell, Clojure, possibly even **Groovy** . . .
- Mention the Actor Model and Concurrent Sequential Processes (CSP) and why they now matter more than ever.
- Exit stage right.

Gant will not appear in this presentation, but SCons will.

There is an element of dynamic binding to the session so the above is just an initial guide.

Protocol for the Session

- A sequence of slides, interrupted by various bits of code.
- Example executions of code – with the illuminating presence of a system monitor.
- Questions (*and, indeed, answers*) from the audience as and when they crop up.

If an interaction looks like it is getting too involved, we reserve the right to stack it for handling after the session.

NB

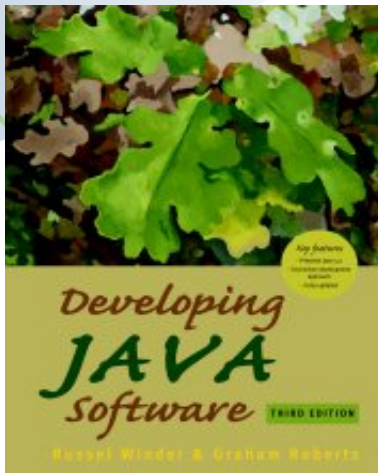
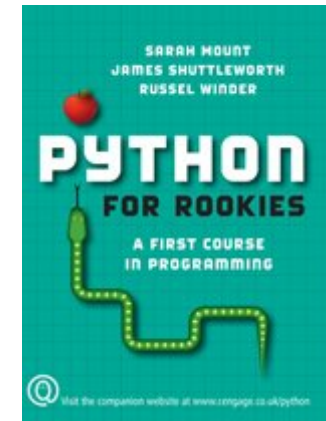
- The session is not about:
 - Algorithms – but they are crucial.
 - Hardware – but it is essential.
- This is more a comparative programming languages session, centred on Python:
 - Looking for “emergent properties” in the directions programming languages and their uses are heading.

Blatant Advertising

Python for Rookies

Sarah Mount, James Shuttleworth and
Russel Winder

Thomson Learning *Now called Cengage Learning.*



Developing Java Software Third Edition

Russel Winder and Graham Roberts

Wiley

Buy these books!



So Who is GIL?

- GIL is not Gil island, Gil in Azerbaijan, any Gilbert or Gilmore, no-one with the surname Gil, any political party.
- GIL is the Global Interpreter Lock:
 - PVM executes one and only one thread of Python code at any one time.
 - GIL is the tool for enforcing sequential activity in the PVM in the presence of multiple threads.

So Python is Single Threading

- Basically yes.
- However, a Python thread can spawn a C or C++ thread and then terminate:
 - The C or C++ code is executing in a native thread not a Python thread and so can be parallel, independent of the state of GIL.
 - Mixing Python and C or C++ activity things get potentially parallel.

This route to parallelism is a definite non-starter – just use C++ in the first place.

Does it Matter?

- Python is so slow compared to Fortran and C++, even Java, that no-one will use it for computations that require parallelism.

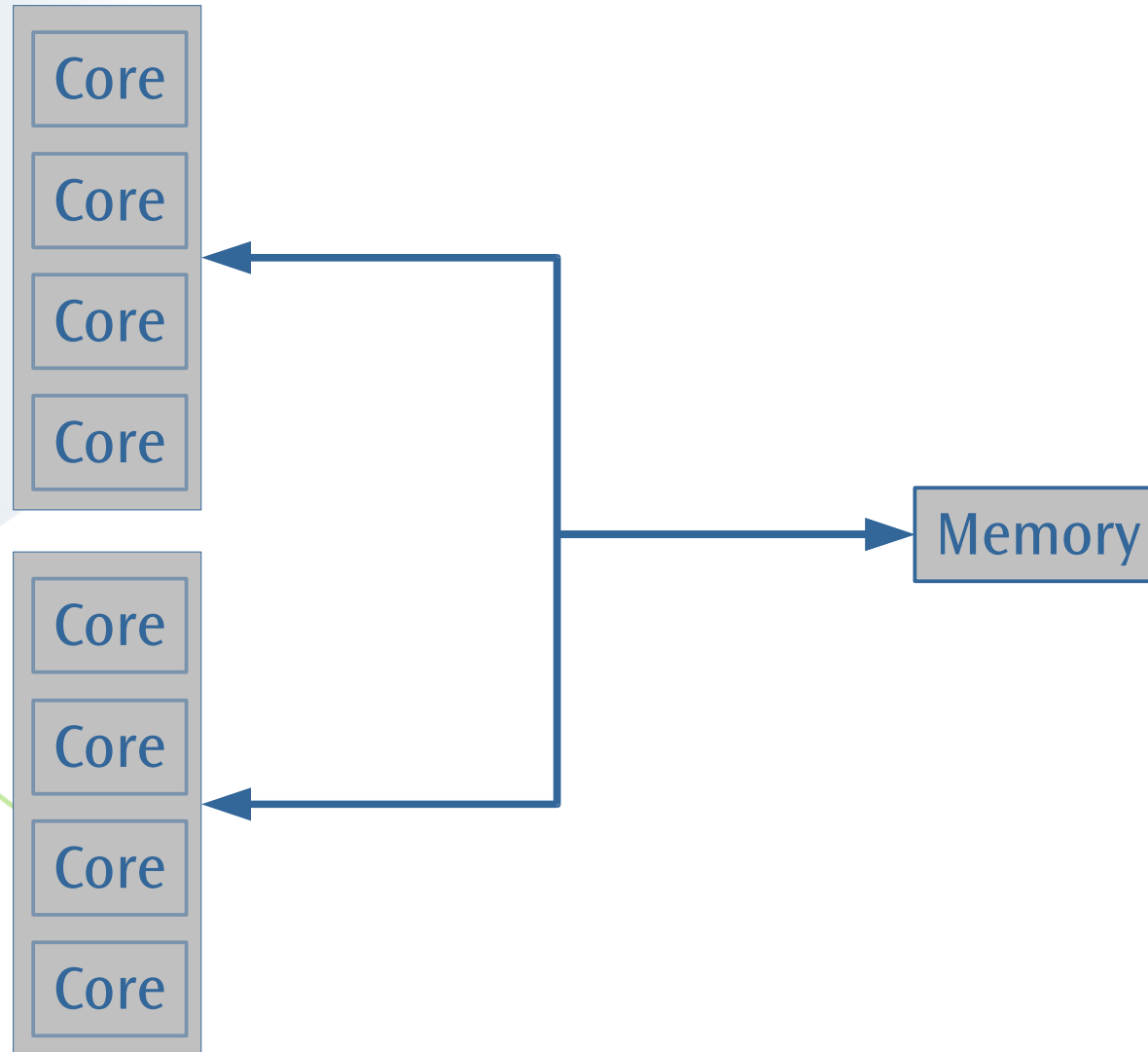
Does it Matter? Yes.

- Python is so slow compared to Fortran and C++, even Java, that no-one will use it for computations that require parallelism.
- True, but Python should not be restricted to GUI and trivial sys. admin. scripting – though Python is exceedingly good for both of those.

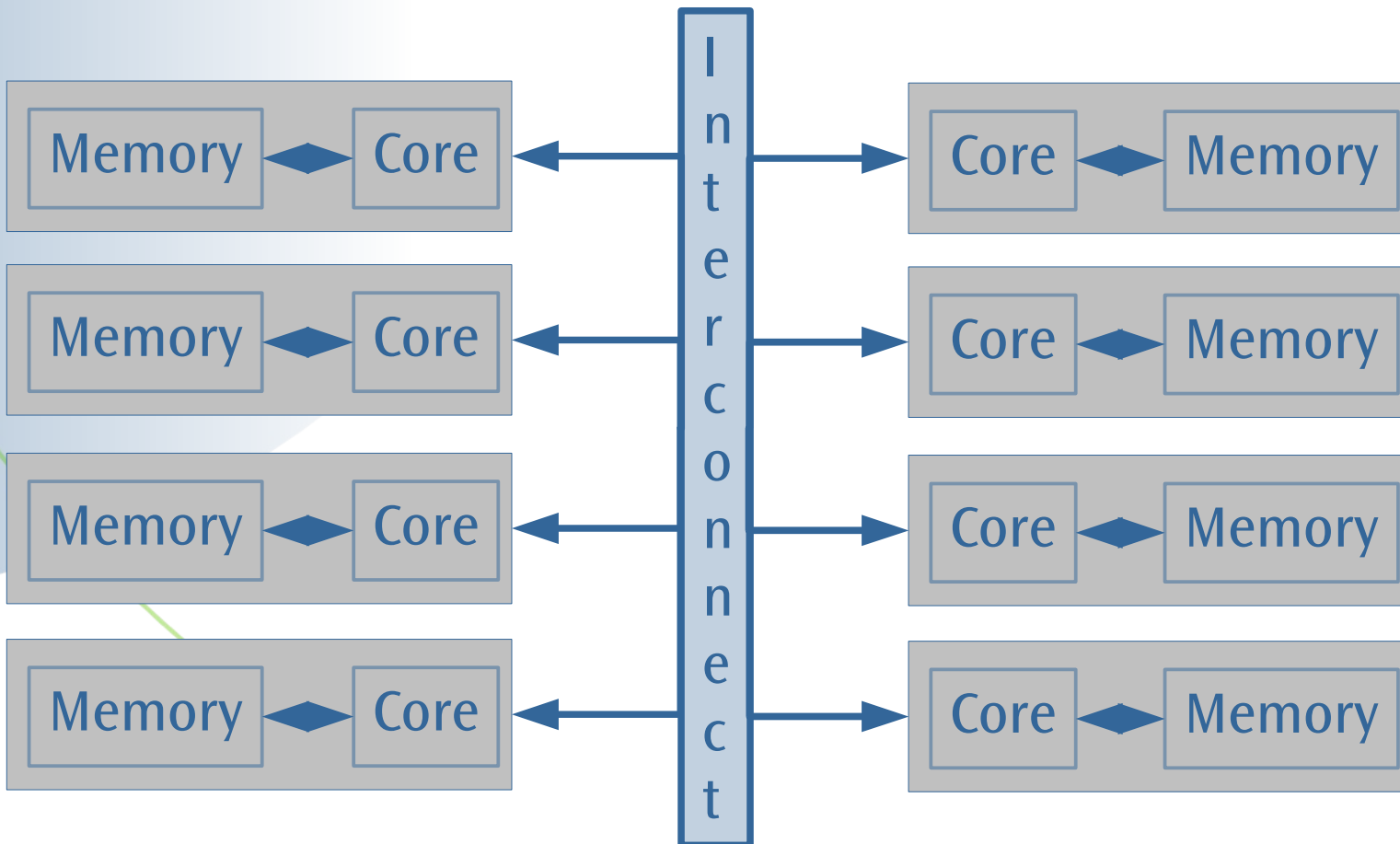
Computers are Now Multicore

- HPC has been massively parallel for years.
- Servers have been parallel systems for almost as long:
 - Rely on being able to use a “one transaction, one thread” model with little or no data sharing other than via a (relational) database.
- Now *every* computer is a parallel computer:
 - 2, 4, 8, 16 core systems everywhere.
 - The revolution has only just begun:
 - Single shared memory cannot support large numbers of cores. Bus structures and memory structures **have** to change to increase computational speeds.

The Shared Memory Problem



The Distributed Memory Solution



Nothing new here, it's been known for decades.

Increasing Application Performance

- Where all applications are single-threaded there isn't a problem – the operating system handles the scheduling of applications over all the processors.
- Where applications must improve their performance, they must harness parallelism and use all the cores and processors in order to achieve more computation per unit time.

Old-style Moore's Law is now completely useless – can no longer improve application performance by waiting.

Programmer's Shouldn't Have to Care?

- Analogy with memory management: processors are a resource that the language should deal with.
- However, Python is an imperative language and so the concurrency/parallelism model is an integral part of the programming model.
- Languages in the future may be able to hide processors as a resource, but that language is not Python.

Functional languages are increasingly seen as being a better solution than imperative languages.

But don't despair, there is hope.

An Example Program

- A simple example – so the code is small.
- An *embarrassingly parallel* problem so that we can look at *scaling*.

Approximating π

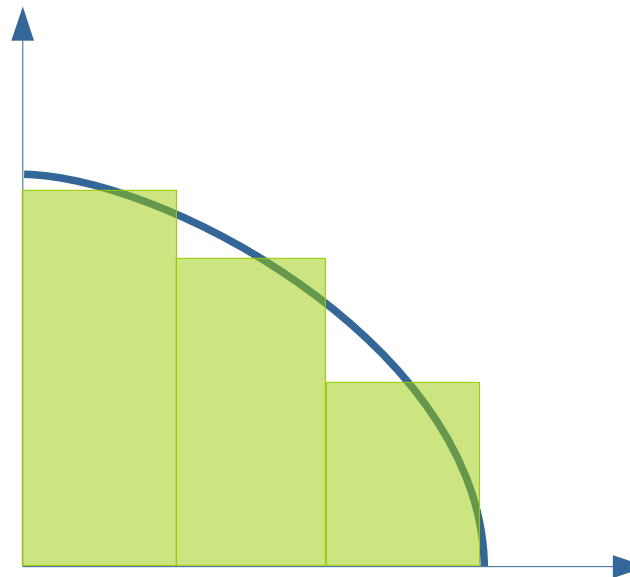
- We know the value of π exactly, it's π (obviously).
- What is its value represented as a floating point number?
 - We can only obtain an approximation.
 - A plethora of possible algorithms to choose from, a popular one is to employ the following integral equation.

$$\frac{\pi}{4} = \int_0^1 \frac{1}{1+x^2} dx$$

A Problem Solved

- Use quadrature to estimate the value of the integral – which is the area under the curve.

$$\pi = \frac{4}{n} \sum_{i=1}^n \frac{1}{1 + \left(\frac{i-0.5}{n}\right)^2}$$



With $n = 3$ not much to do, but potentially lots of error.

Caveat

- The problem is numeric and so should be solved in Fortran or C++ – or possibly Java or Scala.
- Solving this with Python is clearly not really realistic.
- However, it is a simple example of an *embarrassingly parallel* problem and so highlights a class of problem that Python should be able to deal with.

The Python Code

- Sequential.
- Threads.
- Parallel Python.
- Multiprocessing – using processes.
- Multiprocessing – using a process pool.

The Laptop

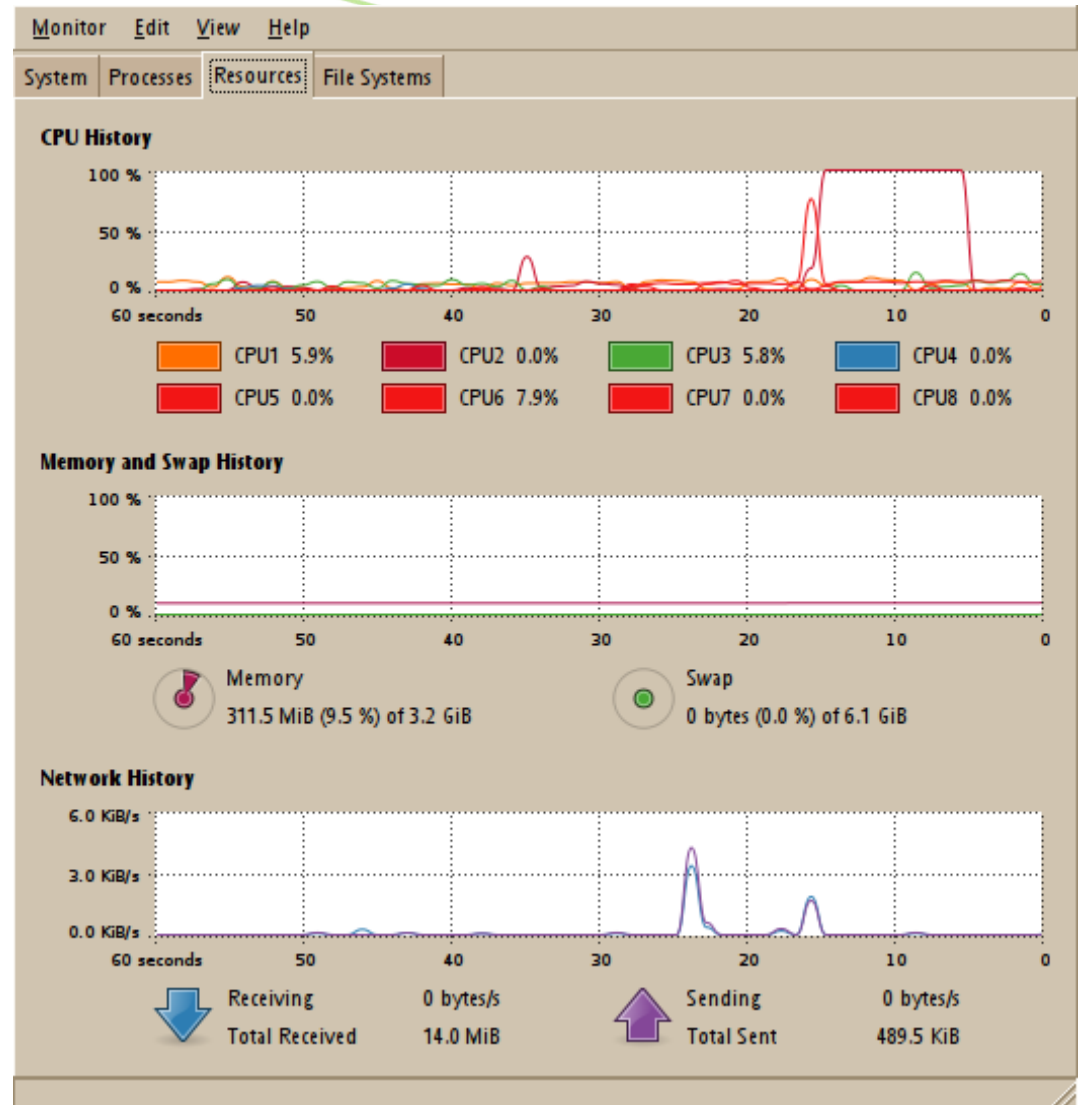
- Core2 Duo T9550 2.66GHz
- 4GB memory
- Ubuntu 9.04 Jaunty Jackalope AMD64.
- Python 2.6.2

The Remote Kit

- Twin Xeon E5410 2.33GHz
- 3GB memory
- Ubuntu 9.04 Jaunty Jackalope – 32-bit (!).
- Python 2.6.2

Sequential

pi = 3.14159265359
 iteration count = 10000000
 elapse = 10.6772670746



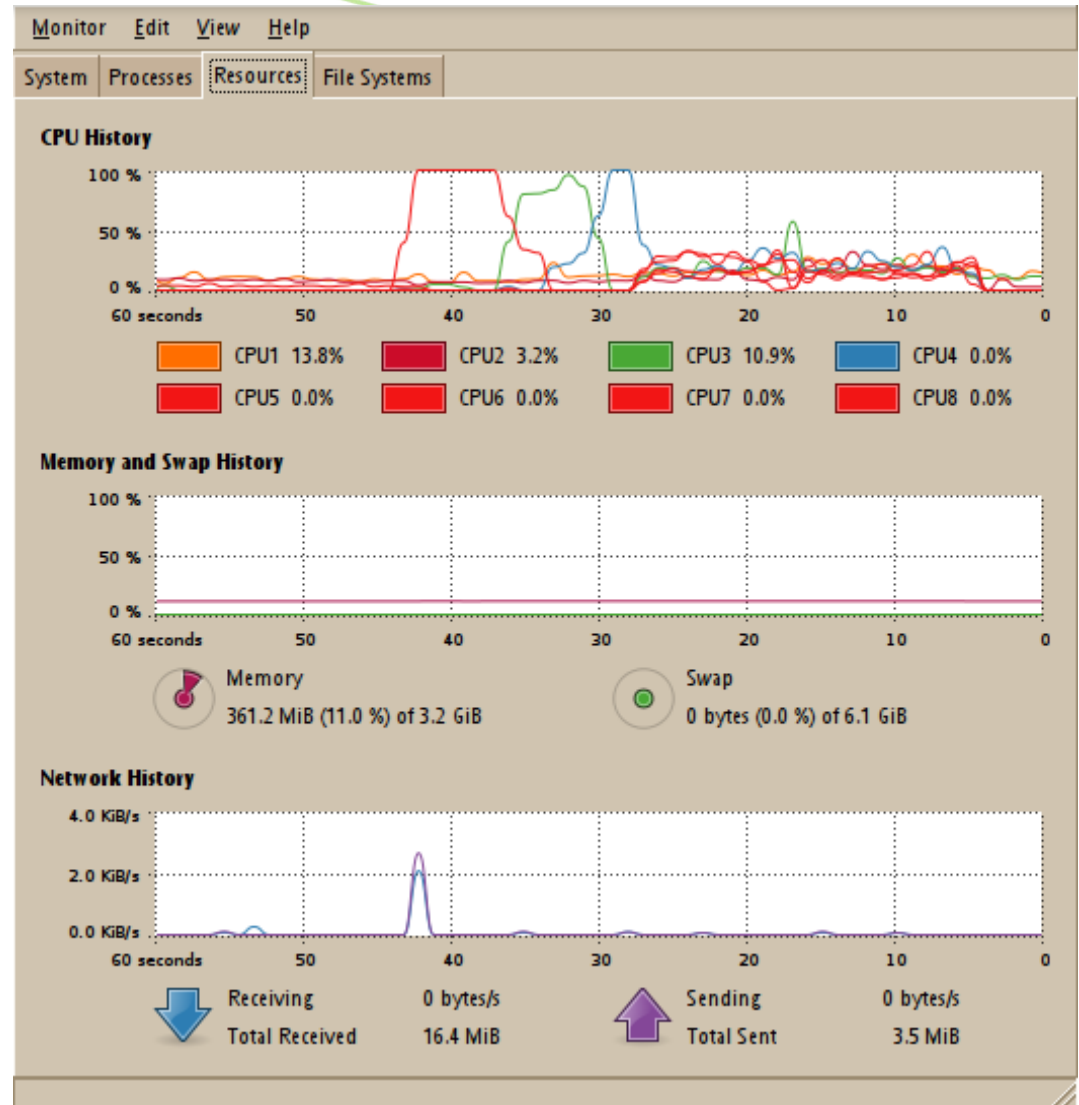
Threads

pi = 3.14159265359
 iteration count = 10000000
 elapse = 7.00223708153
 thread count = 1

pi = 3.14159265359
 iteration count = 10000000
 elapse = 8.75601291656
 thread count = 2

pi = 3.14159265359
 iteration count = 10000000
 elapse = 10.6442840099
 thread count = 8

pi = 3.14159265359
 iteration count = 10000000
 elapse = 11.8278970718
 thread count = 32



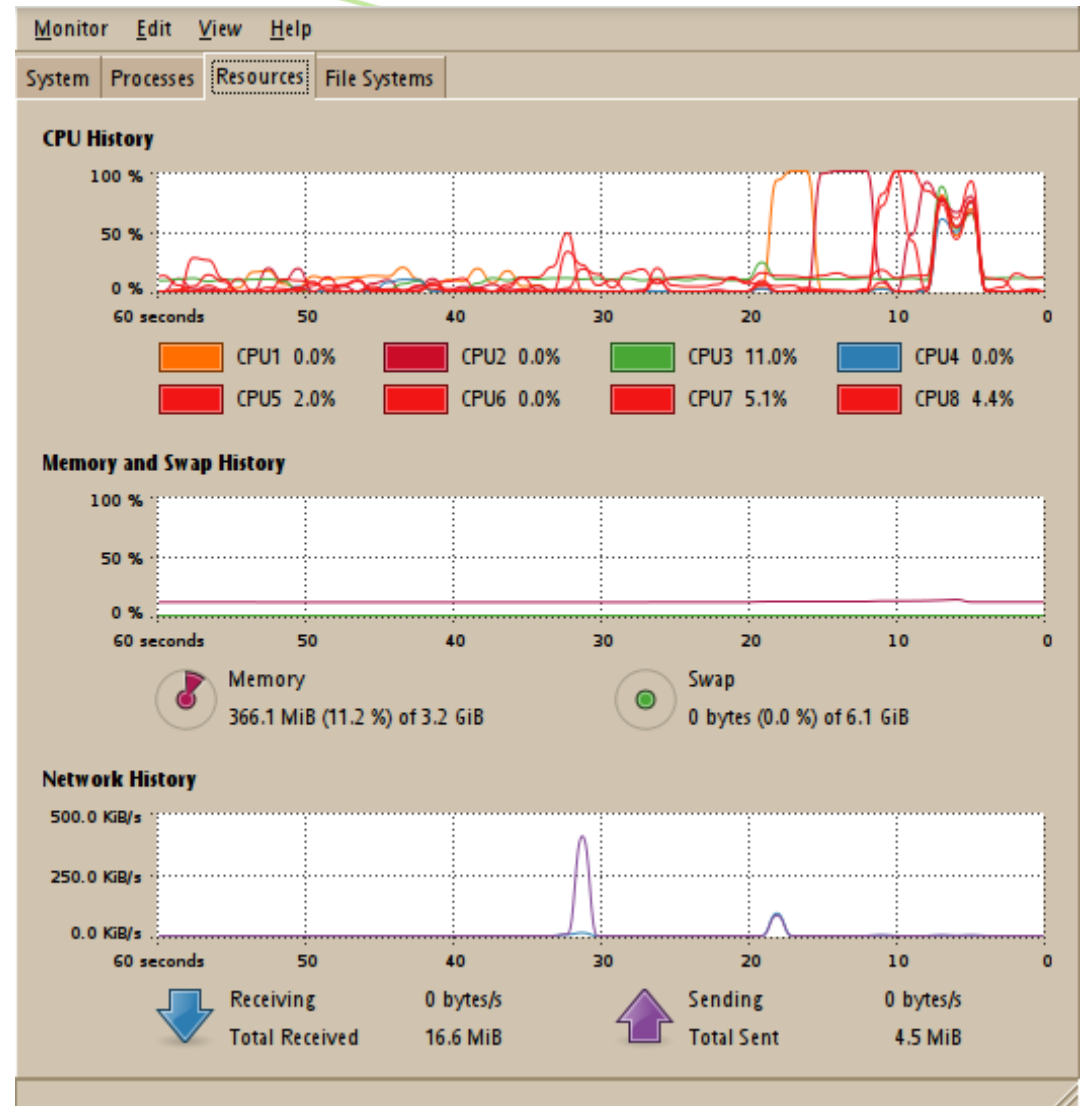
Parallel Python

pi = 3.14159265359
 iteration count = 10000000
 elapse = 7.38268113136
 process count = 1
 processor count = 8

pi = 3.14159265359
 iteration count = 10000000
 elapse = 3.89269590378
 process count = 2
 processor count = 8

pi = 3.14159265359
 iteration count = 10000000
 elapse = 1.4285159111
 process count = 8
 processor count = 8

pi = 3.14159265359
 iteration count = 10000000
 elapse = 1.35900211334
 process count = 32
 processor count = 8



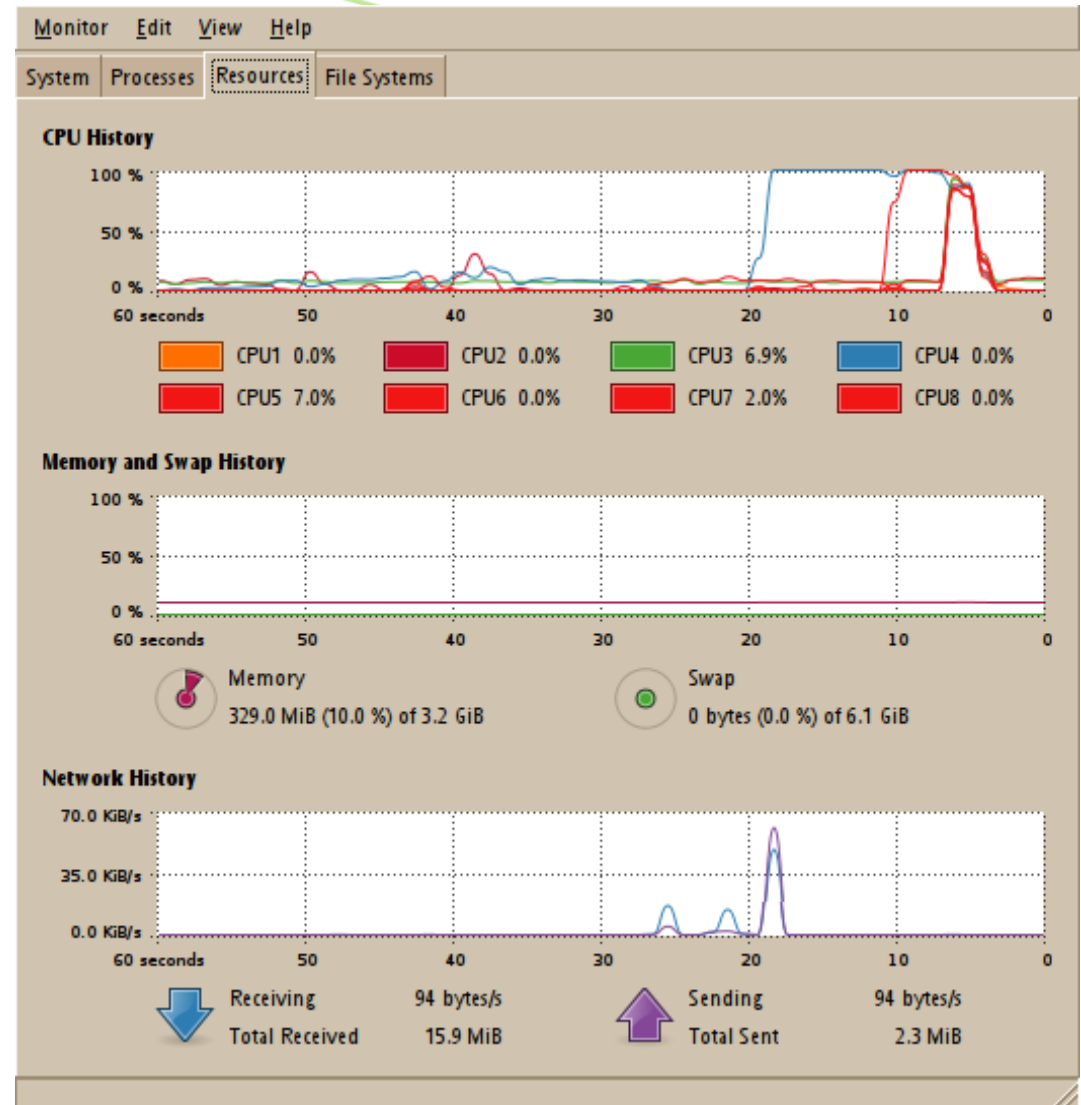
Multiprocessing Process

pi = 3.14159265359
 iteration count = 10000000
 elapse = 8.3213801384
 process count = 1
 processor count = 8

pi = 3.14159265359
 iteration count = 10000000
 elapse = 4.00033593178
 process count = 2
 processor count = 8

pi = 3.14159265359
 iteration count = 10000000
 elapse = 1.05395913124
 process count = 8
 processor count = 8

pi = 3.14159265359
 iteration count = 10000000
 elapse = 1.16570687294
 process count = 32
 processor count = 8



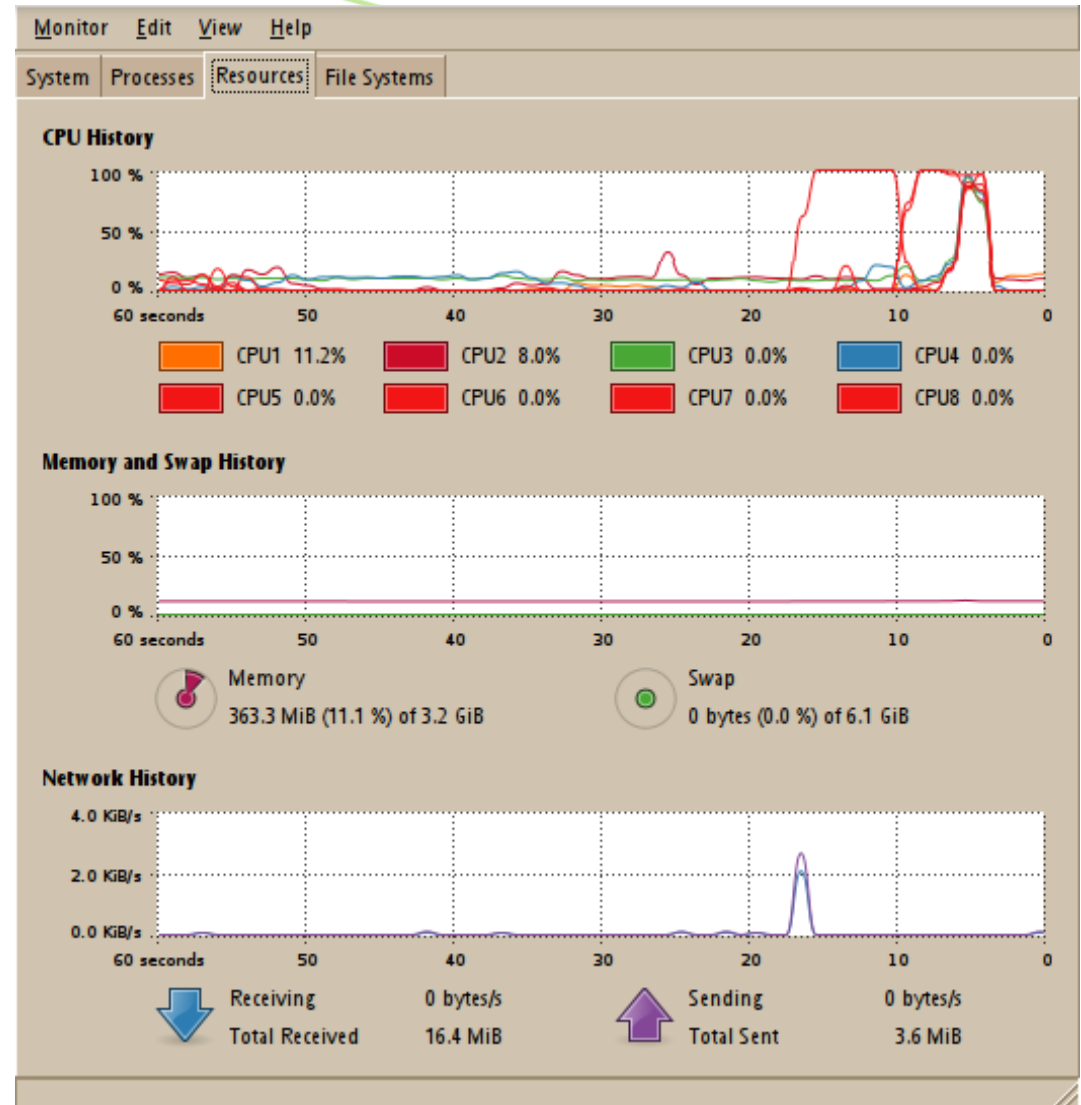
Multiprocessing Pool

pi = 3.14159265359
 iteration count = 10000000
 elapse = 6.91870689392
 process count = 1
 processor count = 8

pi = 3.14159265359
 iteration count = 10000000
 elapse = 3.54036688805
 process count = 2
 processor count = 8

pi = 3.14159265359
 iteration count = 10000000
 elapse = 0.971022844315
 process count = 8
 processor count = 8

pi = 3.14159265359
 iteration count = 10000000
 elapse = 1.01583886147
 process count = 32
 processor count = 8



Results

- Threads in Python are great for co-routine type concurrency.
- Threads in Python are useless for parallelism because of the GIL.
 - Contrast this with C++, Java and Scala where threads map to operating system threads and thence processors. (Sort of.)
- Shared memory and synchronization is the real enemy.

Threads are “dead”, long live processes.

This is Not New

- There is a long history of threads being problematic.
- Synchronization of shared memory is the problem:
 - Semaphores.
 - Locks.
 - Monitors.
- Deadlock and livelock are trivially easy to achieve.

Lock-free programming is not an answer on its own – surprisingly – transactional memory might be, but that is another session.

This is Really Not New

- Theories of concurrency and parallelism have eschewed threads. For example:
 - Communicating Sequential Processes (CSP):
 - An algebra of composing independent (and hence parallel) sequential processes with synchronous communication between them.
 - Actor Model:
 - A model of parallel computation based on asynchronous communication between active objects.
 - Dataflow:
 - A model of parallel computation based on actioning evaluations when data is available.

And the Trick Is . . .

- . . . that the synchronization is inherent in the computational model.
 - CSP: processes are single threaded and the message passing is synchronous.
 - Actor model: the object is in total control of its own synchronization.
 - Dataflow: operators only trigger when the data is available.

The End of Threads . . .

- . . . has been coming for ages.
 - Erlang chose processes and actors for its model of parallelism.
 - Scala chose processes and actors for its model of parallelism.
 - Dataflow architectures are being used for massive data processing applications and showing huge speedups on the same hardware.

It's all About the Memory

- Shared memory and threads based models of parallelism do not scale.
- Distributed memory, message passing approaches will win for applications.

Threads will survive for implementing lightweight processes.

Even though systems programming will involve threads for ever more, applications programmers do not have to suffer the hassles to harness parallelism.

Actually . . .

- . . . its really all about the data:
 - Organizing data so that data parallelism is obvious is the real way forward.

Clearly there are subsystems, e.g. user interfaces, that are event driven. Event-driven approaches are therefore best for these subsystems.

So What about GIL?

- GIL is a problem if you want to do share memory multi-threading and get parallelism.
- The future is process-based, message passing parallelism.



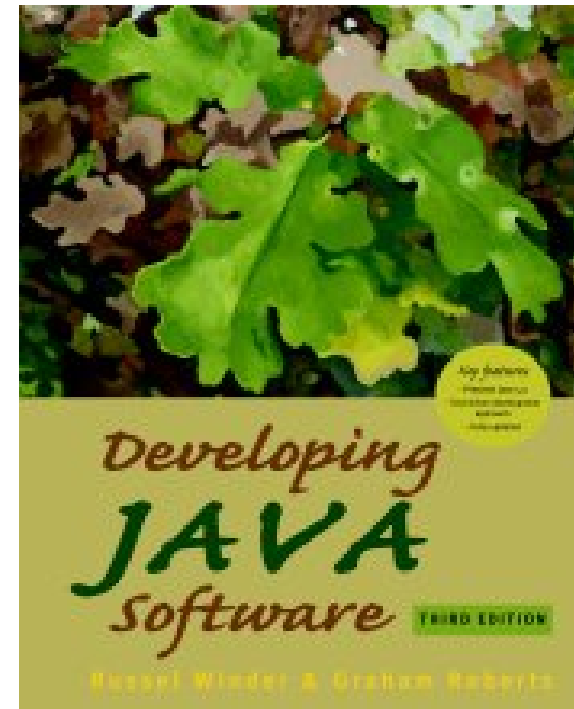
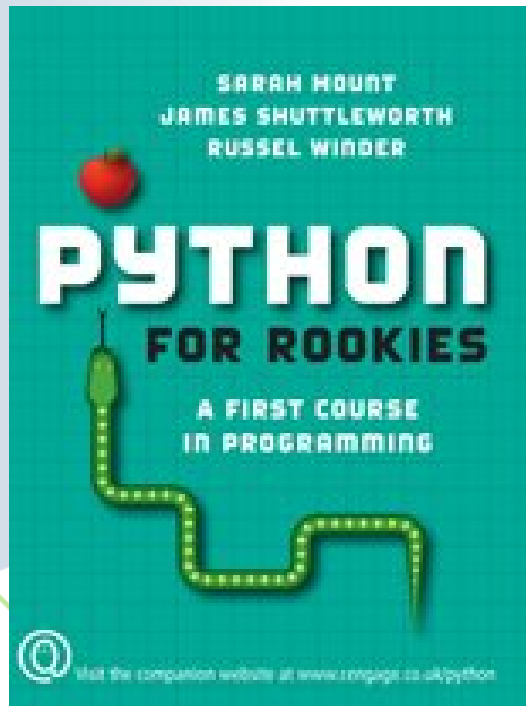
- GIL is not the problem.

Summary

- Harnessing parallelism is about organizing the data and its evolution.
- If you are thinking about threads then you have lost.
- Processes and message passing if you have to, parallel arrays if you can.
- Declarative approaches (cf. functional programming) win over imperative approaches – note how C++ gets more and more declarative with every evolution.
- Event-driven approaches for event-driven systems.

The world is a heterogeneous place, so be heterogeneous. Just avoid sharing the memory.

Unabashed Advertising



You know you want to buy them!

