Shared-memory Multithreading Is The Wrong Way To Do Parallelism

Dr Russel Winder

Partner, Concertant LLP

russel.winder@concertant.com

Copyright © 2009 Russel Winder

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 various 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.*

Copyright © 2009 Russel Winder

Structure of the Session

- Look at some of the concurrency and parallelism support features in various languages such as C, C++, Fortran, Java, Scala, Python, Erlang, Haskell, Clojure, possibly even Groovy . . .
- Mention the Concurrent Sequential Processes (CSP), the Actor Model and dataflow, and why they now matter more than ever.
- Exit stage right.

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

Copyright © 2009 Russel Winder

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.

Copyright © 2009 Russel Winder

NB

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

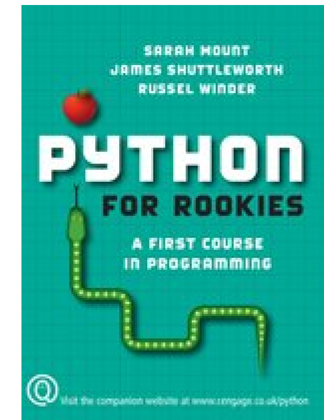
Copyright © 2009 Russel Winder

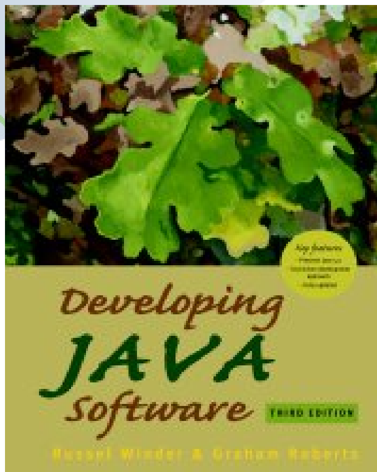
Blatant Advertising

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!

Copyright © 2009 Russel Winder

Ancient History

- Computers ran one sequential program at a time.
- Operators loaded and ran the program.
- Punched cards or tape for the code if you were lucky, toggle in the program via switches if you weren't.

Copyright © 2009 Russel Winder

Utilization Matters

- Create multitasking operating systems.
- Operators can load more than one program at a time.
- Only one program at a time actually executes, but time-division multiplexing gives the appearance of concurrency.

Writing operating systems leads to worrying about interrupts, coroutines, and access to shared memory. Locks, semaphores and monitors get invented.

Copyright © 2009 Russel Winder

Harnessing Concurrency

- Processes and multiprocessing are available – each process has its own address space, guaranteed by the operating system.
- Concurrent systems created using multiple processes and message passing between them.

Communicating Sequential Processes is invented as an underlying theory of concurrency.

Copyright © 2009 Russel Winder

The Slide into the Quagmire

- Inter-process communication, mediated by the operating system, is too slow a way of passing messages between processes.
- Threads are invented.

Shared memory multithreading requires locks, semaphores, monitors, etc. and is shown to be hard in practice.

*People, including programmers, even **Real Programmers**, find getting threaded programs right, **very** hard.*

Copyright © 2009 Russel Winder

Threads become the Norm

- Operating systems support threads.
- Hardware supports threads.
- Programming languages support threads.
  - PThreads
  - Java threads
  - Green Threads
  - C++0x threads

Java the first language to bring threads to the masses.

Copyright © 2009 Russel Winder

Hardware Changes

- Originally a computer had one processor.
- Some applications required parallelism to work at all.
- High Performance Computing (HPC) and supercomputing bring parallelism not just concurrency to the fore.

Copyright © 2009 Russel Winder

An Example Program

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

Copyright © 2009 Russel Winder

Approximating π

- We know the value of π exactly, it's π (obviously).
- What is it's 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.

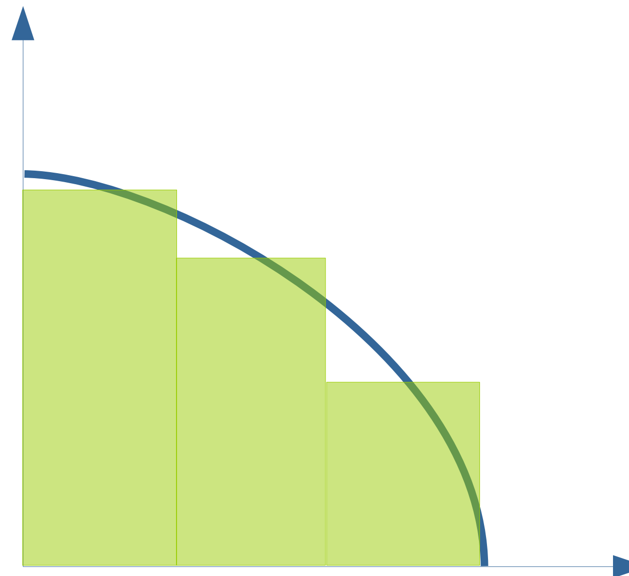
$$\frac{\pi}{4} = \int_0^1 \frac{1}{1+x^2}\,dx$$

Copyright © 2009 Russel Winder

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.

Copyright © 2009 Russel Winder

The Laptop

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

Copyright © 2009 Russel Winder

The Remote Kit

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

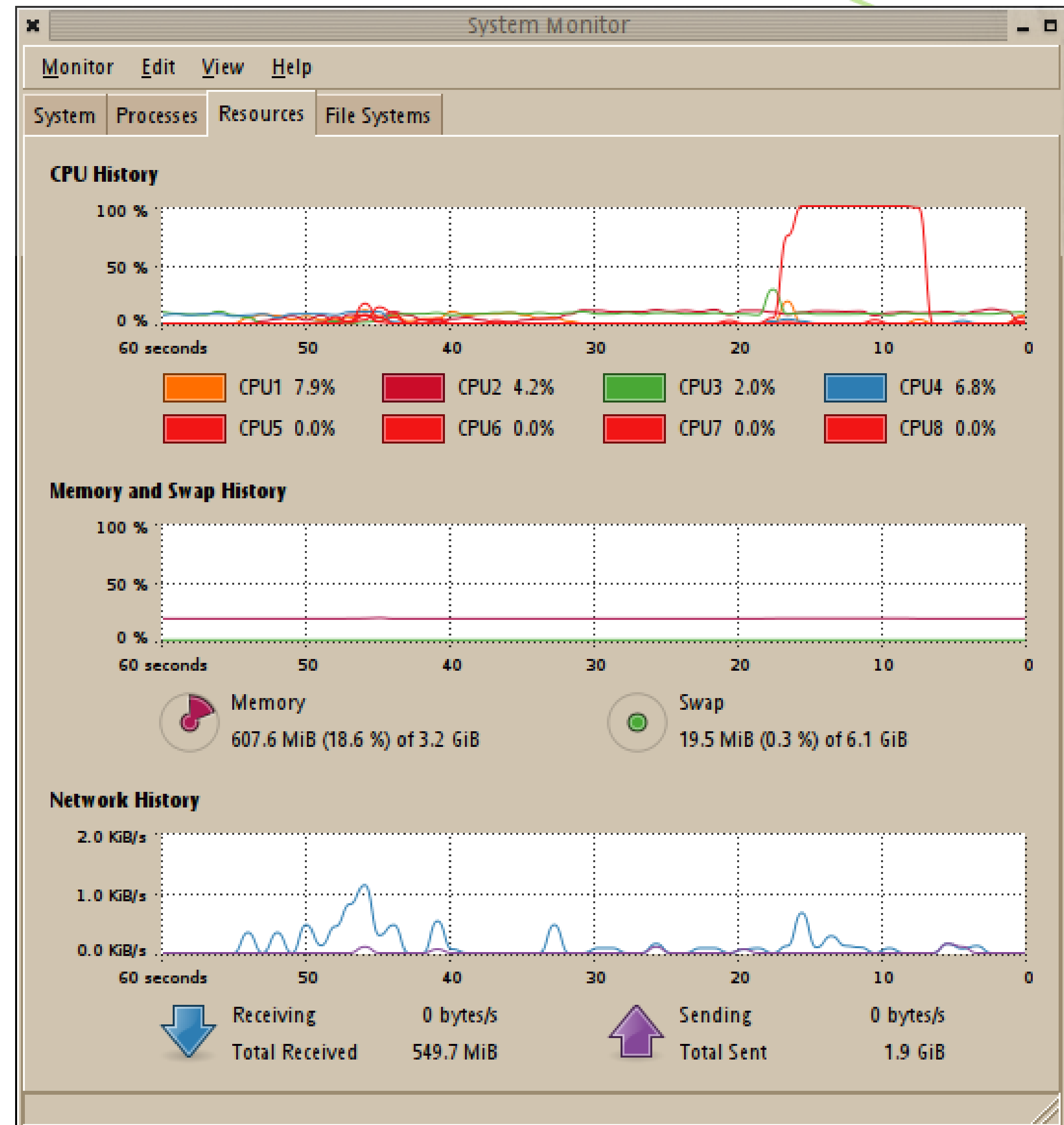
Copyright © 2009 Russel Winder

The Sequential Version

Copyright © 2009 Russel Winder

Sequential Result

pi = 3.14159265358979194159017
iteration count = 1000000000
elapse = 9.46061299999999999864442



Copyright © 2009 Russel Winder

Passing the Parallel Message

- Parallelism means separate processors with separate memory.
- Processors pass messages to each other.
- Fortran, then C, then C++ get the Message Passing Interface (MPI).

SPMD – single program multiple data

Copyright © 2009 Russel Winder

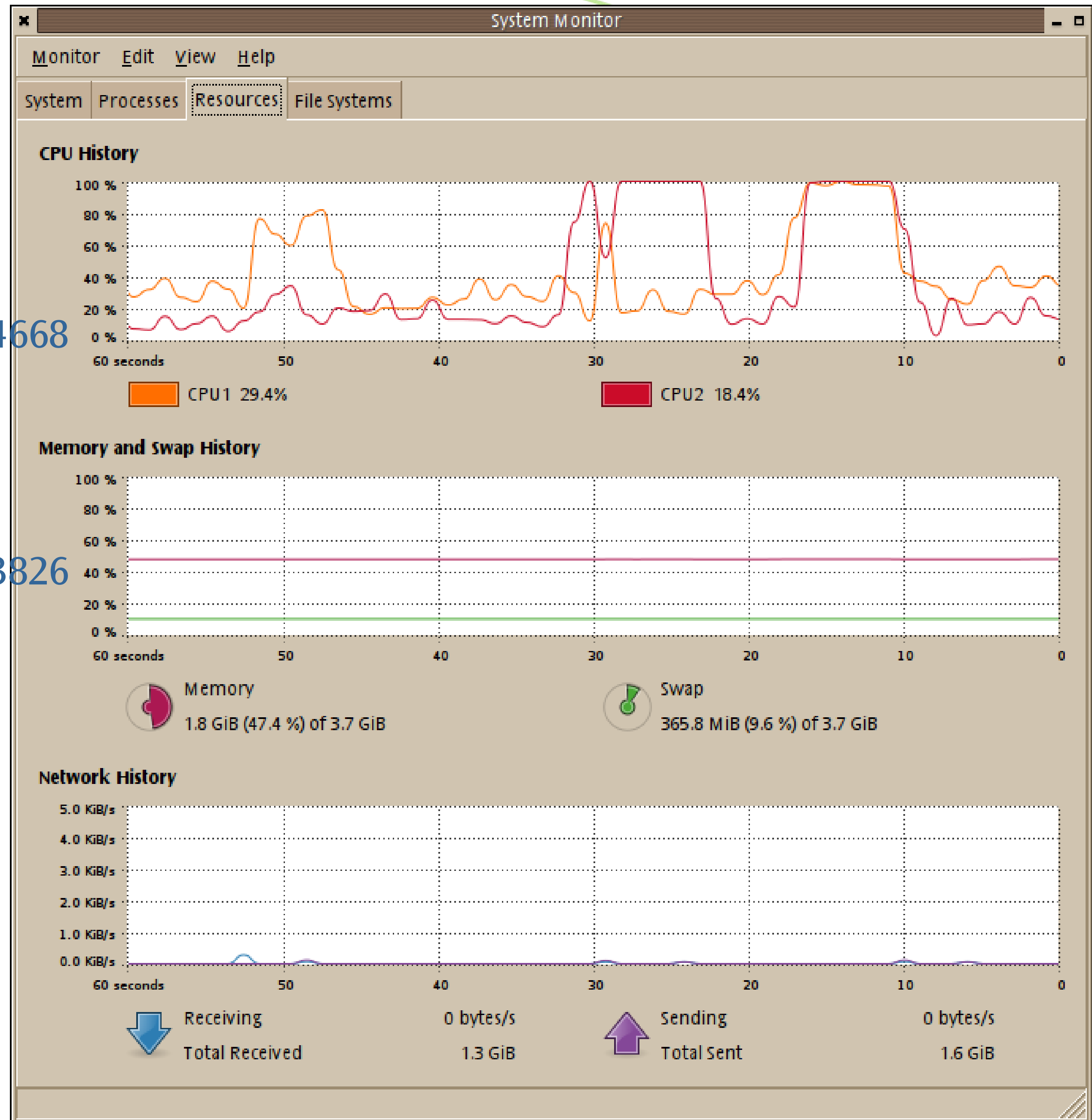
Using MPI

Copyright © 2009 Russel Winder

MPI Result

pi = 3.14159265358979194159017
iteration count = 1000000000
elapse = 9.30110900000000293084668
processor count = 1

pi = 3.141592653589792692725435
iteration count = 1000000000
elapse = 8.30348000000000416093826
processor count = 2



Copyright © 2009 Russel Winder

HPC gets into Shared Memory

- Threads and shared memory are deemed the norm, so HPC invents a better way of handling threads.
- OpenMP.

Copyright © 2009 Russel Winder

Using Threads Explicitly

Copyright © 2009 Russel Winder

C++0x Futures

http://www.stdthread.co.uk/

http://www.justsoftwaresolutions.co.uk/

Copyright © 2009 Russel Winder

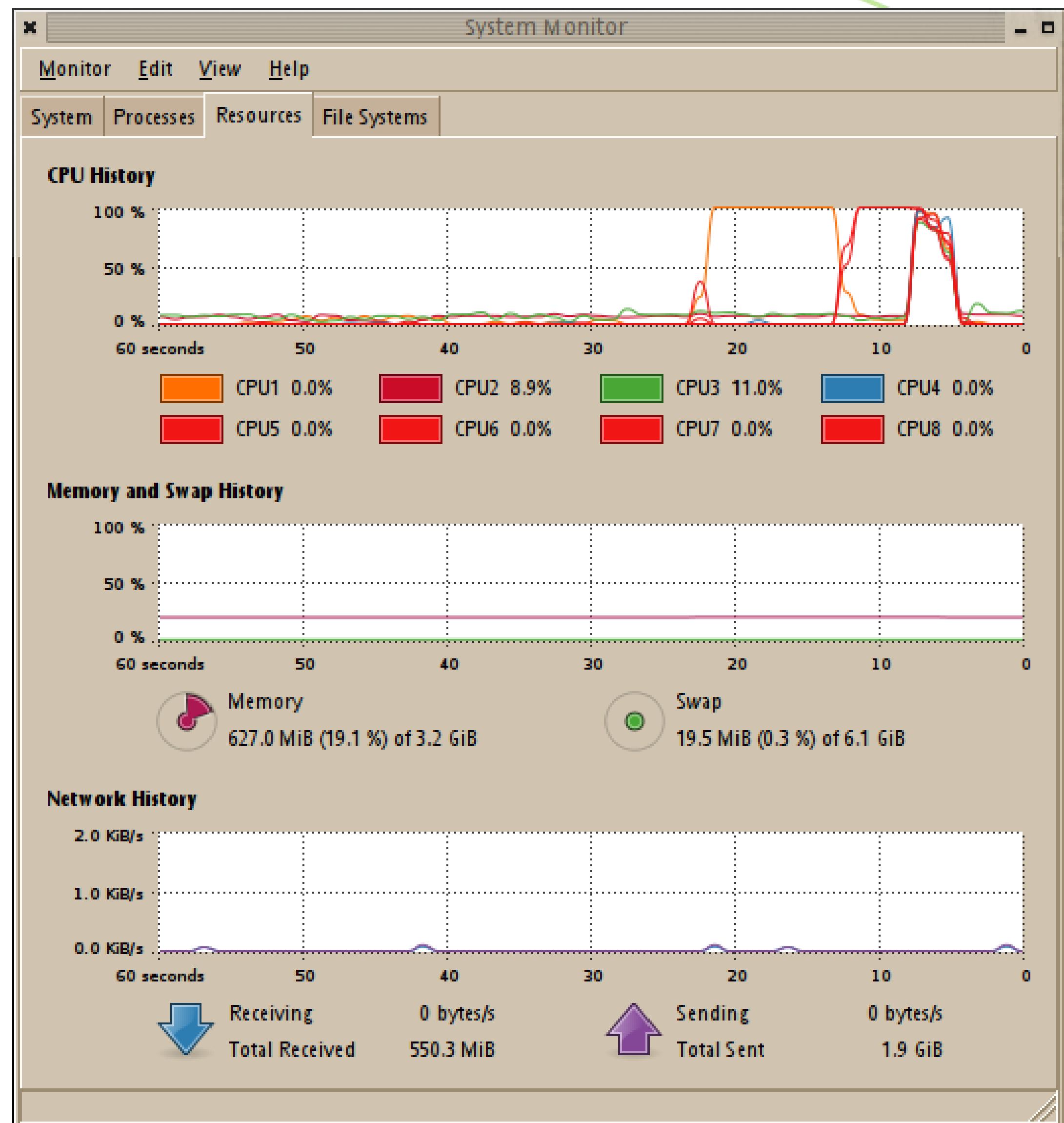
Futures Result

pi = 3.14159265358979194159017
iteration count = 1000000000
elapse = 9.45960900000000000180078
threadCount = 1
processor count = 8

pi = 3.14159265358979269272543 5
iteration count = 1000000000
elapse = 4.73042699999999999879652
threadCount = 2
processor count = 8

pi = 3.14159265358979346099109 5
iteration count = 1000000000
elapse = 1.34772399999999999993387
threadCount = 8
processor count = 8

pi = 3.14159265358979334042781 3
iteration count = 1000000000
elapse = 1.35413500000000000025362
threadCount = 32
processor count = 8



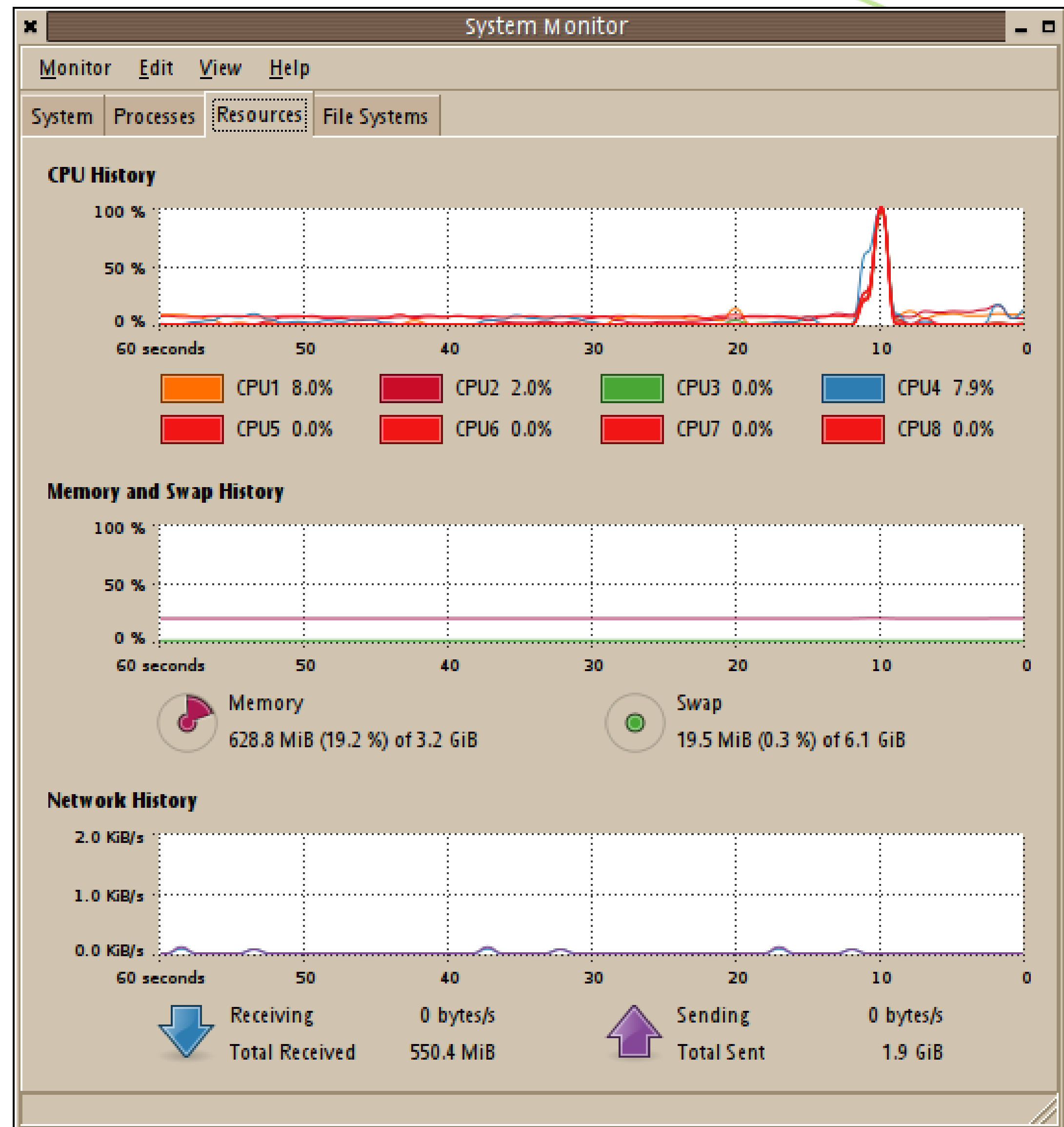
Using OpenMP

Copyright © 2009 Russel Winder

Concertant LLP

OpenMP Result

pi = 3.141592653589793460991095
iteration count = 1000000000
elapse = 1.29603599999999999989474
processor count = 8



Copyright © 2009 Russel Winder

Intel goes One Better

- Threading Building Blocks (TBB).
- C++ only:
  - C++ is a better language that Fortran or C anyway – better mechanisms of abstraction.
  - Fortran and C do not have the right tools.

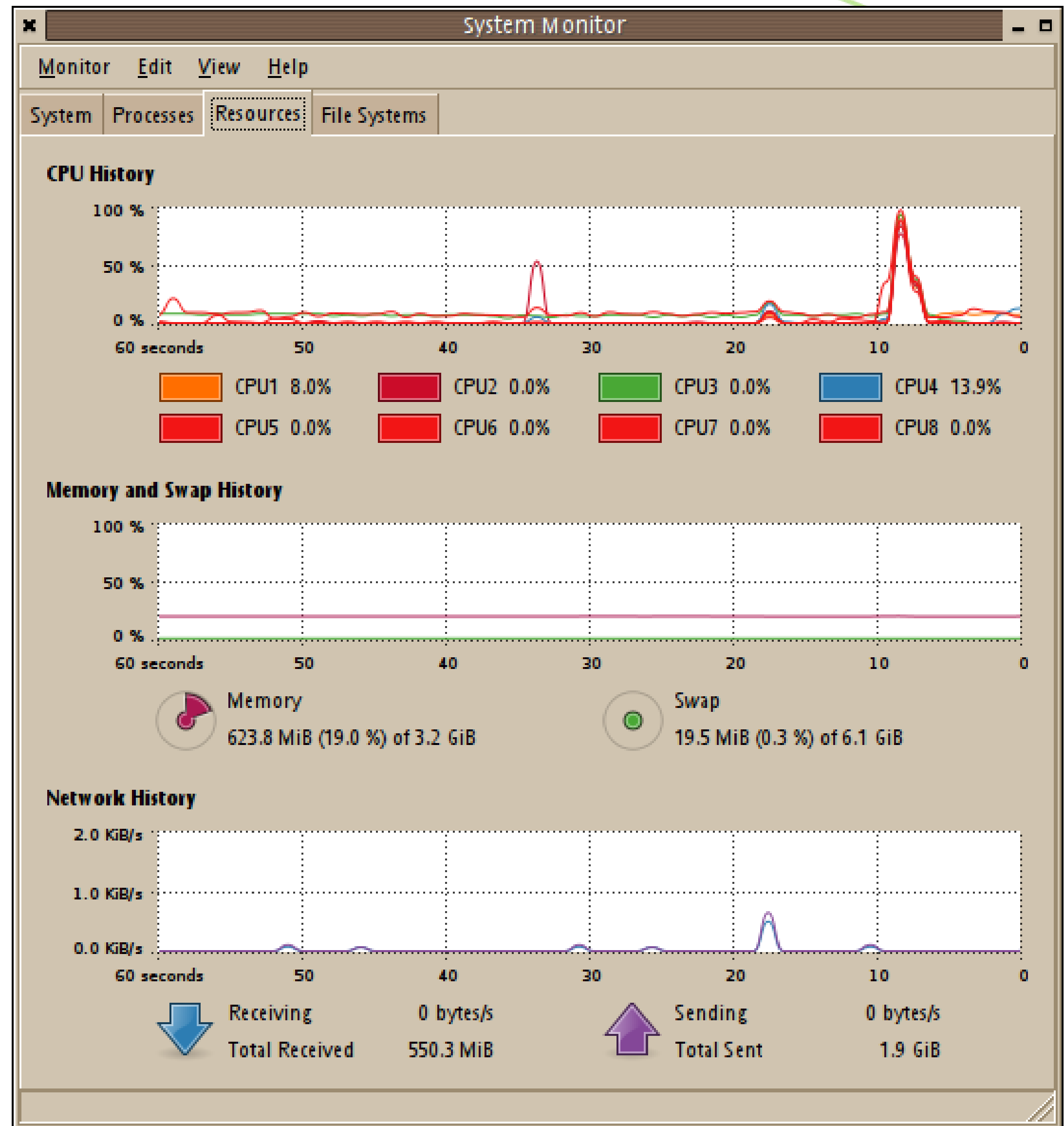
Copyright © 2009 Russel Winder

Using TBB

Copyright © 2009 Russel Winder

TBB Result

pi = 3.141592655589793384694752
iteration count = 1000000000
elapse = 1.28650299999999999965368



Copyright © 2009 Russel Winder

Computers are Now Multicore – 1

- High Performance Computing (HPC) has been massively parallel for years.
  - SPMD, MPI
  - Threads, OpenMP

Copyright © 2009 Russel Winder

Computers are Now Multicore – 2

- Servers have been parallel systems for almost as long:
  - But they rely on being able to use a “one transaction, one thread” model with little or no data sharing other than via a (relational) database.

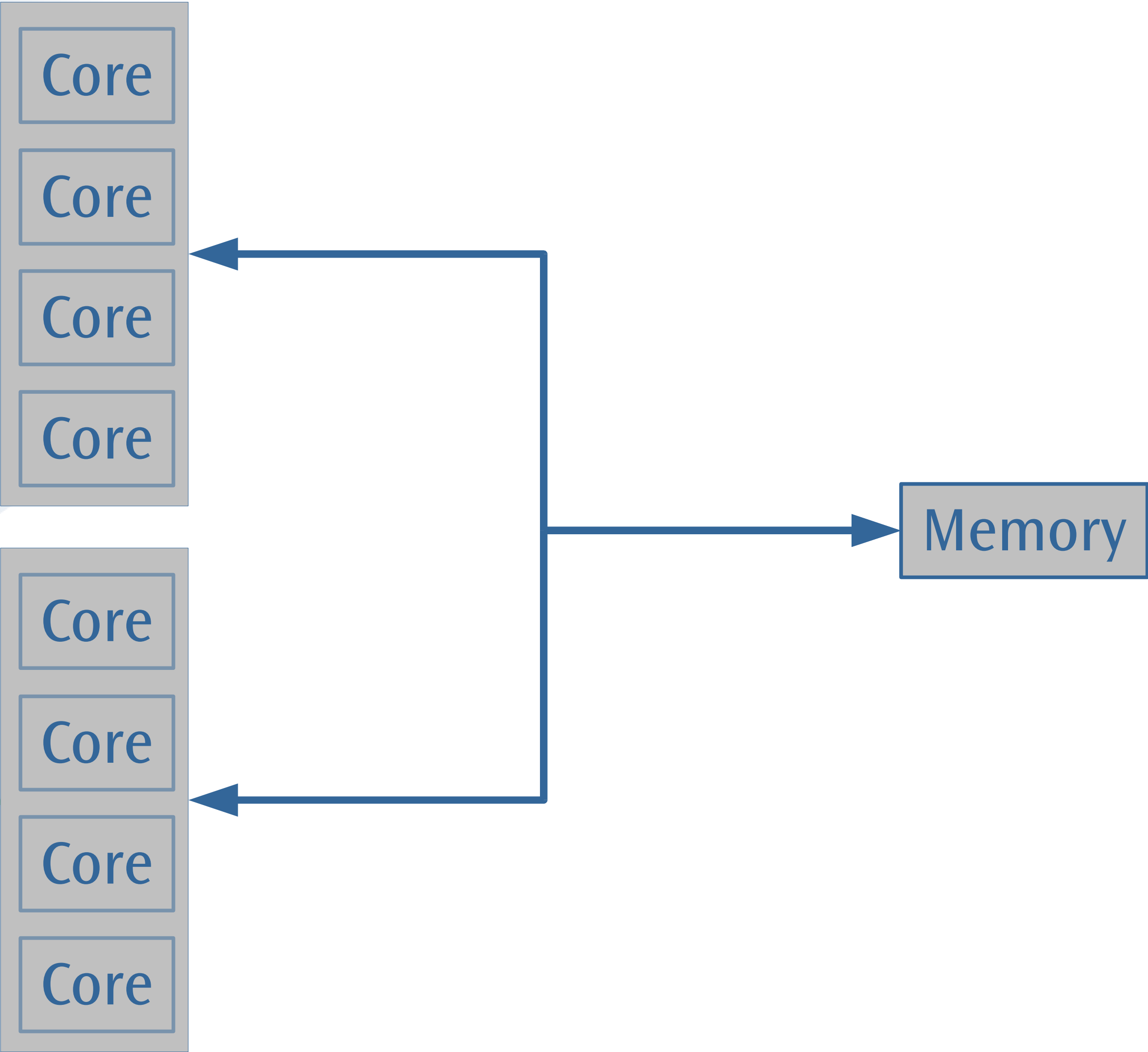
Copyright © 2009 Russel Winder

Computers are Now Multicore – 3

- ***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.

Copyright © 2009 Russel Winder

The Shared Memory Problem



Copyright © 2009 Russel Winder

The Shared Memory Dead End

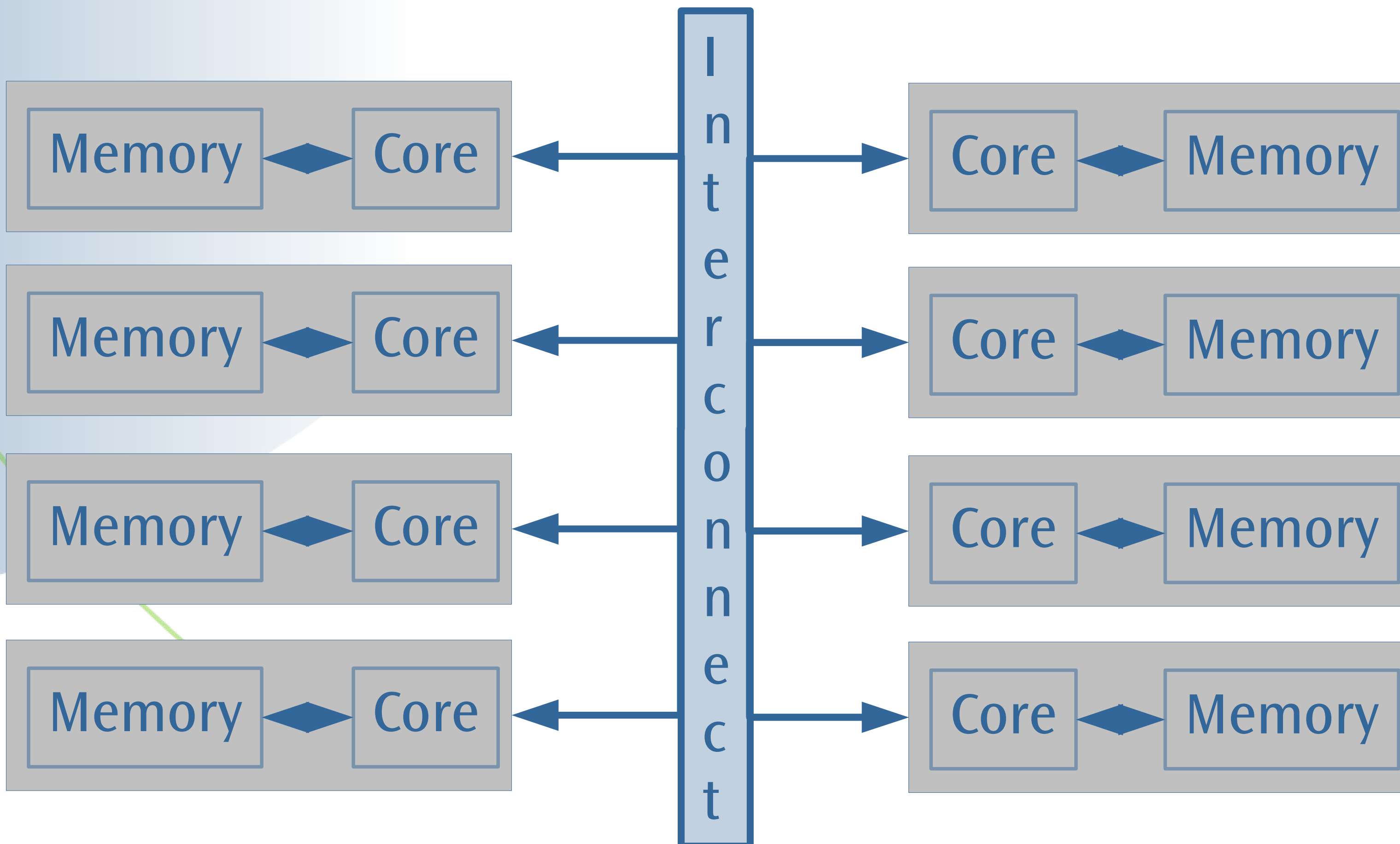
- Processors with large numbers of cores cannot work with a single shared memory – and worse a single shared memory bus – unless you are happy with single processor, single core performance.
- Adding multiple processors, with more cores, to a single bus architecture is simply not tenable.

Current PC and laptop hardware architectures cannot survive
The Multicore Revolution.

Copyright © 2009 Russel Winder

Concertant LLP

The Distributed Memory Solution



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

Copyright © 2009 Russel Winder

Concertant LLP

Systems Programmers Don't Care?

- Operating system have always dealt with concurrency.
- Operating system provide services for applications.
- Parallelism isn't an issue, per se.

- Will need new architectures of operating system though to deal with the distributed memory.

Copyright © 2009 Russel Winder

Distributed Memory Applications

- Where all applications are single-threaded there isn't a problem – the operating system handles the scheduling of applications over all the processors, and maximizes utilization.
- Where applications must improve their performance, they must harness parallelism and use multiple 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.

Copyright © 2009 Russel Winder

Application Programmers Shouldn’t Have to Care

- Analogy with memory management:
  - Memory is a resource that the language, and the compilation tools, deal with.
  - Processors are therefore resources that the language, and the compilation tools, should deal with.

Copyright © 2009 Russel Winder

Application Programmers Have to Care

- C, C++, Fortran, Java, Python, Groovy, are imperative languages:
  - Programmers deal with explicit control flow.
  - The concurrency/parallelism model is explicit and is an integral part of the programming model.

Functional languages (Erlang, Haskell, OCaml, etc.) are increasingly seen as being better than imperative languages for applications programming.

Copyright © 2009 Russel Winder

Functional Programming Gaining Ground?

- Scala is setting itself up to supplant Java:
- Scala is:
  - Procedural.
  - Object-oriented.
  - Functional.
  - Runs on the JVM.
  - Statically typed.
  - Has type inference.

- C++ is clearly not functional, it is procedural and object-oriented.
- C++ templates are a functional language.
- C++0x introduces type inference to C++.

Functional languages are infiltrating their ideas everywhere.

Copyright © 2009 Russel Winder

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.

Copyright © 2009 Russel Winder

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.

Copyright © 2009 Russel Winder

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.

Copyright © 2009 Russel Winder

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.

Copyright © 2009 Russel Winder

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.

Copyright © 2009 Russel Winder

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.

Copyright © 2009 Russel Winder

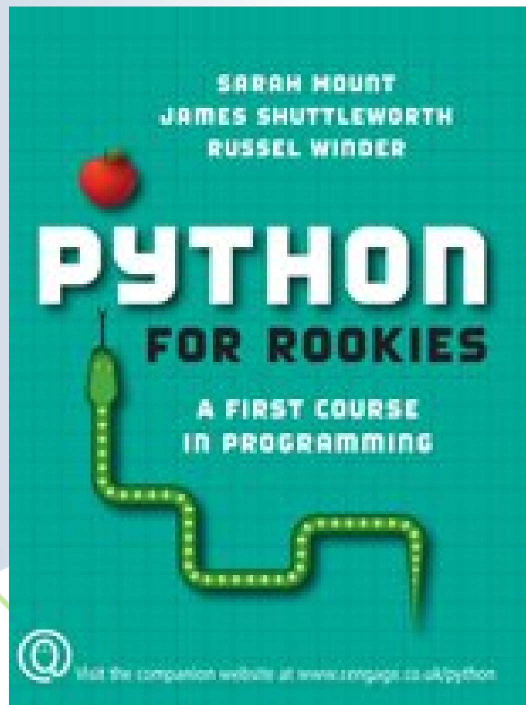
Summary

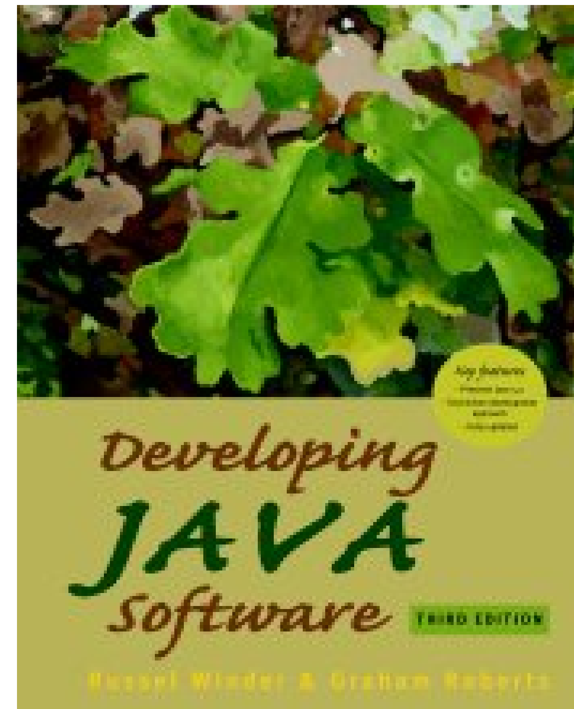
- Harnessing parallelism is about organizing the data and its evolution.
- If you are thinking about threads then you have lost.
- Processes and message passing if you have to, parallel arrays if you can.
- Declarative approaches (cf. functional programming) win over imperative approaches – note how 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.

Copyright © 2009 Russel Winder

Unabashed Advertising





You know you want to buy them!

Copyright © 2008 Russel Winder