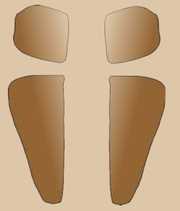
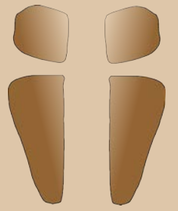


Parallelism can be Groovy

Dr Russel Winder

It'z Interactive Ltd

russel@itzinteractive.com



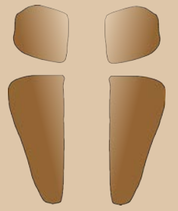
Aims and Objectives of this Session

- Investigate some of the issues for programming, and programming languages raised by the **next** phase of *The Multicore Revolution*.
- Show that parallel computing is as viable on virtual machines (including the JVM) as it is using native code systems.
- Introduce GPars and Python-CSP.
- Prepare people to go down the pub.

Structure of the Session

- Introduce some stuff.
- Present about some stuff.
- Promote discussion about some stuff.
- Summarize some stuff.
- Relocate (to an hostelry) for some other stuff.

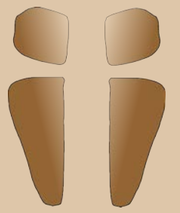
Stuff is what it's all about!



Nota Bene

- This session is not about:
  - Algorithms – but they are crucial.
  - Hardware – but it is essential.

In fact, various abstract hardware models will be introduced since hardware architecture is important in determining software architecture.



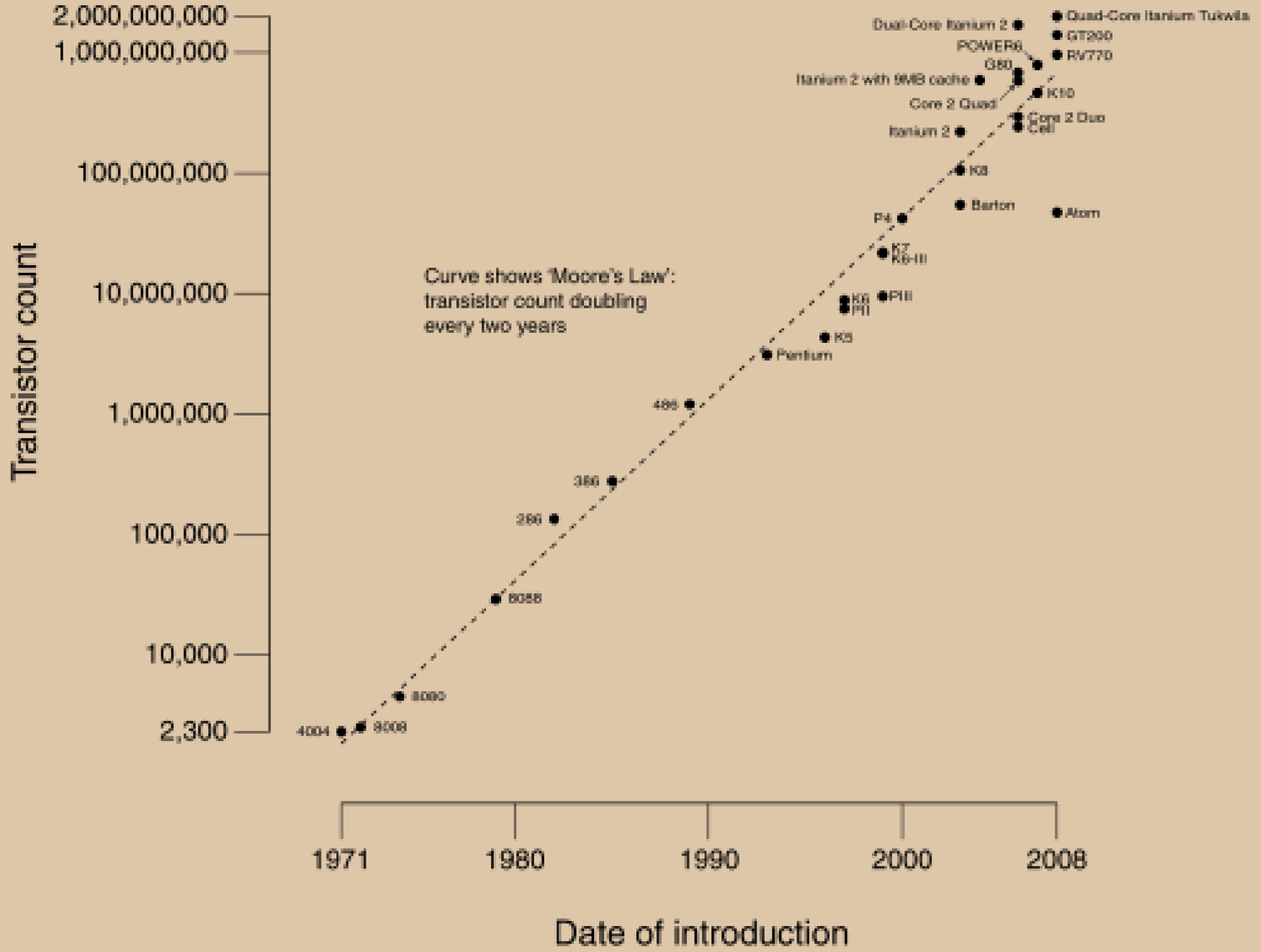
The Multicore Revolution

Moore's Law – The Original Statement

The complexity for minimum component costs has increased at a rate of roughly a factor of two per year... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer.

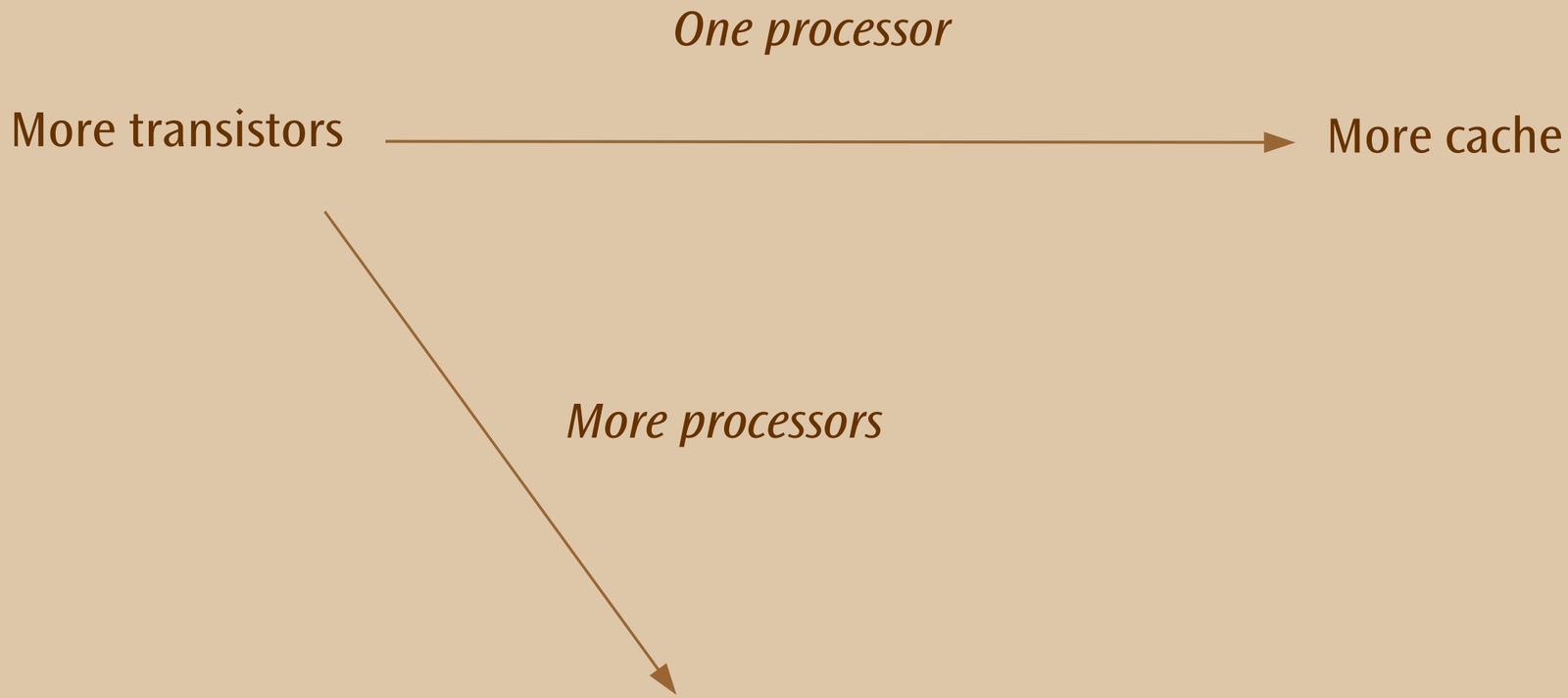
"Cramming more components onto integrated circuits", Electronics Magazine 19 April 1965.

Moore's Law – Some Data

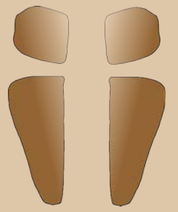


http://en.wikipedia.org/wiki/File:Transistor_Count_and_Moore's_Law_-_2008.svg

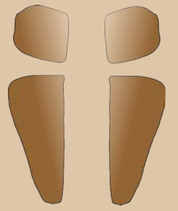
What to do?



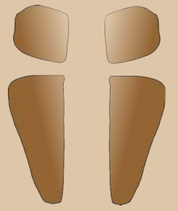
1950 – 2005 (ish)



Same processor means same transistor count, so use all the extra transistors to add more cache.



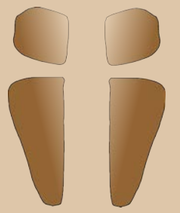
But there is only so much cache you can add.



Alternative:

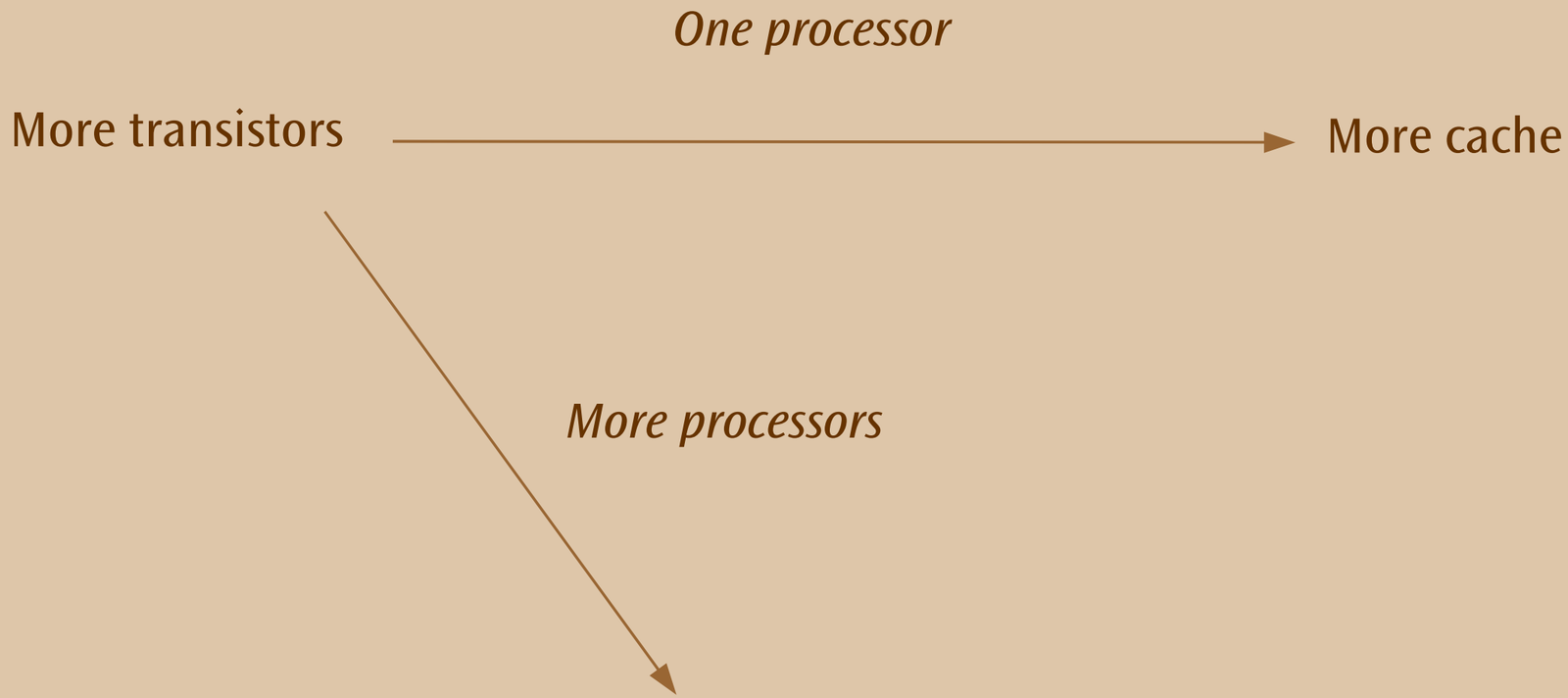
put all the memory on the chip with the processor and dispense with the bus for memory access;

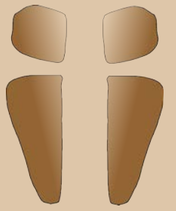
and get rid of all the cache.



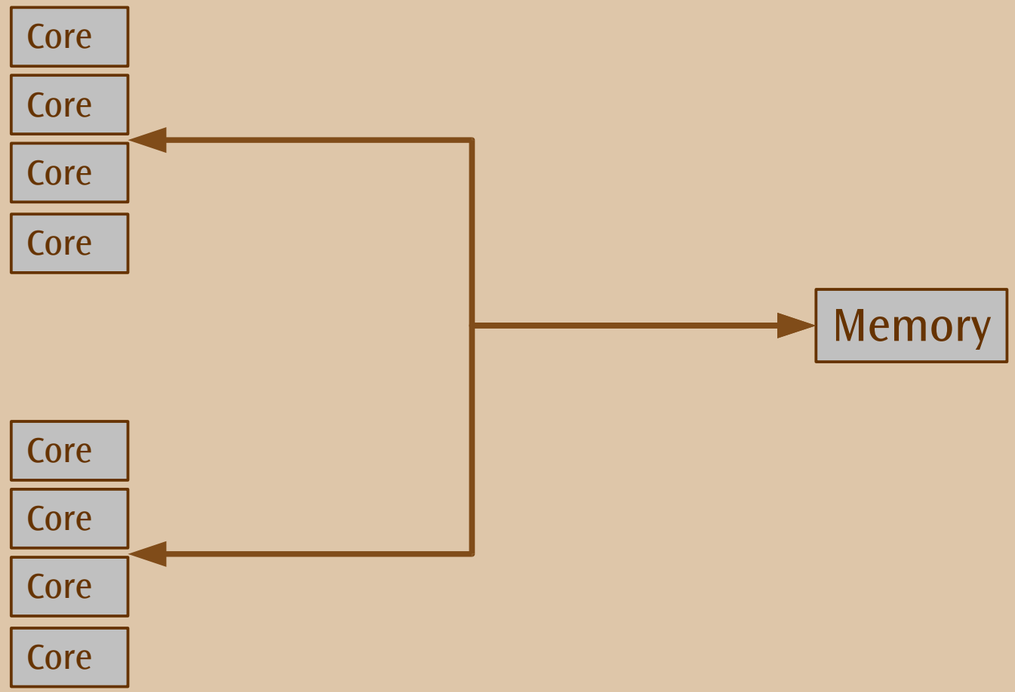
This direction gives no increase in instructions executed per unit time,
so do something else, something more marketable.

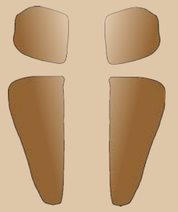
What to do?



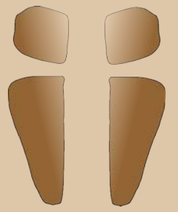


2005 (ish) – 2012 (ish)

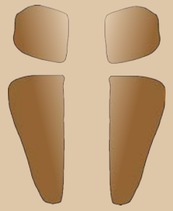




Significant increase in instructions executed per unit time – very marketable.

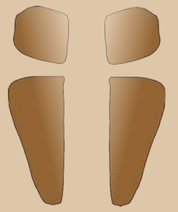


The hardware is executing many more instructions per unit time than ever before, therefore computing is faster because speed is measured in instructions per unit time.

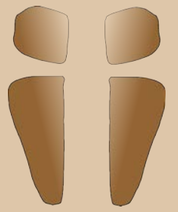


Parallelism,
not concurrency,
is now the norm.

The universe has always been a very parallel place, but early computers insisted on doing only one thing at once, very sequentially at that . . .



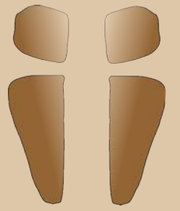
. . . sadly this has had a serious sequentializing effect on our notion of algorithm, computation and programming in general.



Utilization Matters

- Early computers ran one program at a time to completion. Programs were loaded before they could be run.
- Every microsecond of processor time was expensive.
- Operating systems provide abstraction of hardware, but more importantly a way of loading the next program whilst the current program is running.
- Extend this to multiprogramming: Have many programs loaded in memory at the same time, and use time-division multiplexing to give time to the various runnable programs. Create the appearance of running multiple programs concurrently.
- Programs were invariable still single flow of control, run to completion.
- Operating system had to manage multiple tasks, a very concurrent system.

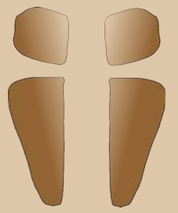
All problems in concurrency – interrupts, coroutines, shared memory, locks, semaphores, monitors – stem from writing operating systems.



Harnessing Concurrency

- The problems of concurrency led to lots of theory:
  - Actor Model
  - Communicating Sequential Processes (CSP)

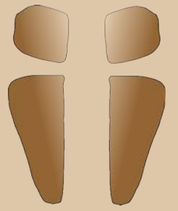
Note the commonality here, small independent processes passing messages to each other. No shared memory.



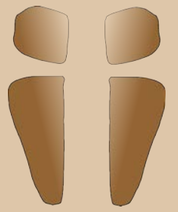
Being Side-tracked

- Operating system processes too heavyweight.
- People invent threads:
  - In a process run a miniature operating system.
  - Pull in all the operating system concurrency ideas into applications building.
- Chip manufacturer decide to support threads in hardware.
- Operating systems (well some of them) efficiently handle hardware threading.
- Applications developers conned into believing that using shared-memory multi-threading connected to kernel threads in their applications is a good thing and that coping with locks etc. is just something that has to be done..
  - PThreads
  - Green Threads
  - Java Threads
  - C++0x Threads

The assumption is: what is good for systems programming is good for application programming.



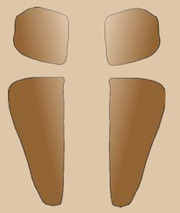
Threads are the problem not the solution.



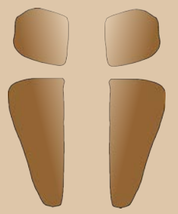
Threads are . . .

- . . . the problem not the solution:
  - Multi-threaded programming generally uses shared memory.
  - Shared memory use required locks, etc.
  - Programmers generally find it hard to reason about locks.
- Transactional memory?
  - Some consider it the answer.
  - Some consider it evil.
  - Hardware manufacturers do not support it.
  - Software transactional memory . . . ?

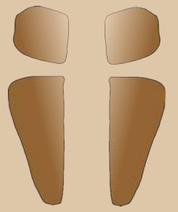
. . . also part of the solution . . .



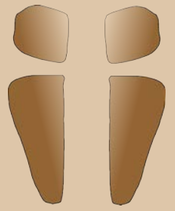
So what?



Word, OpenOffice.org, indeed all the applications people actually use, are inherently ***sequential***.



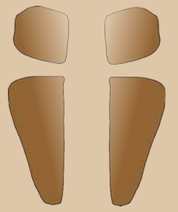
Well they are event driven anyway, so use an event loop approach.



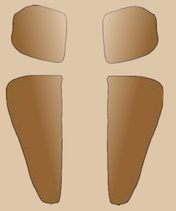
Event loop + threads

$\Rightarrow$

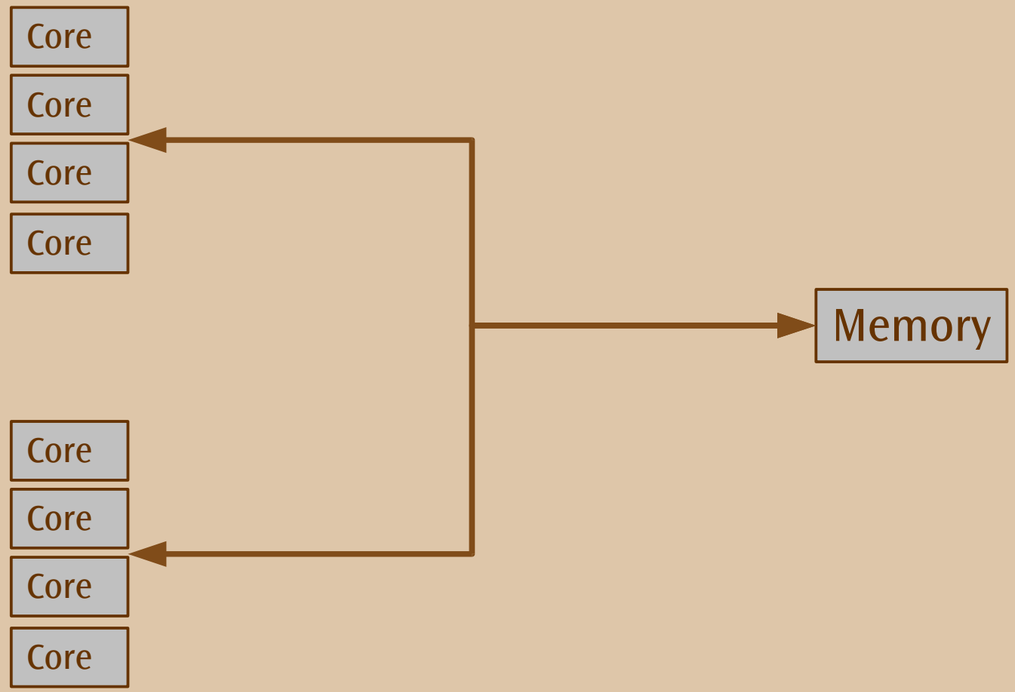
Solved problem.



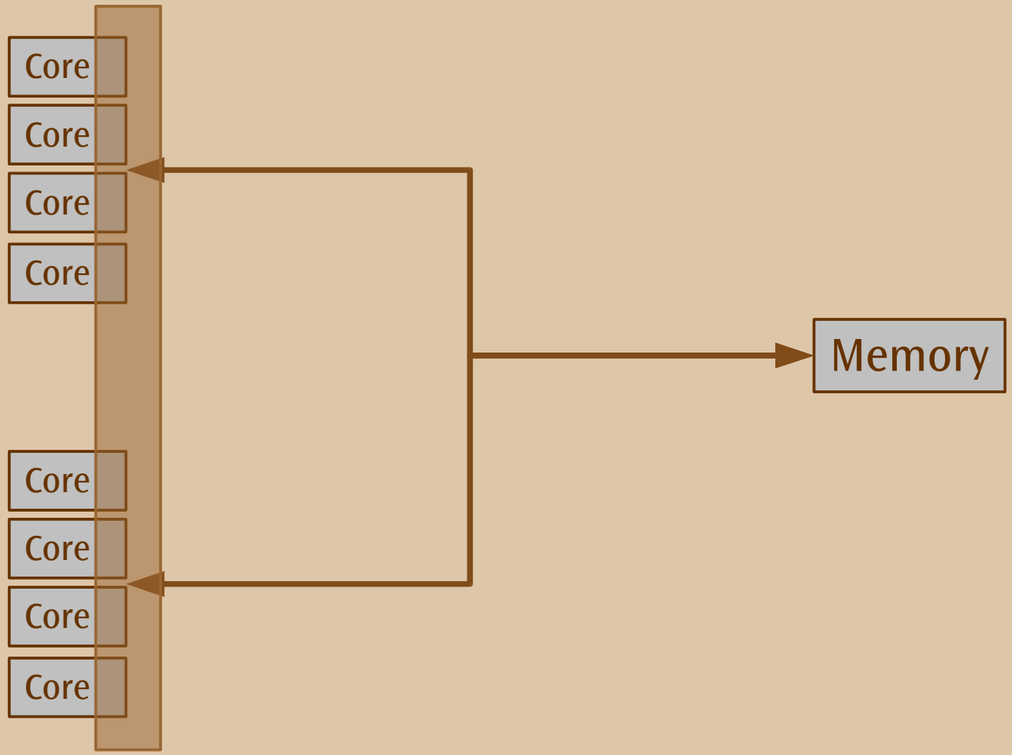
So what's the problem?



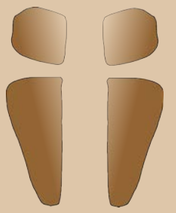
2005 (ish) – 2012 (ish)



2005 (ish) – 2012 (ish)

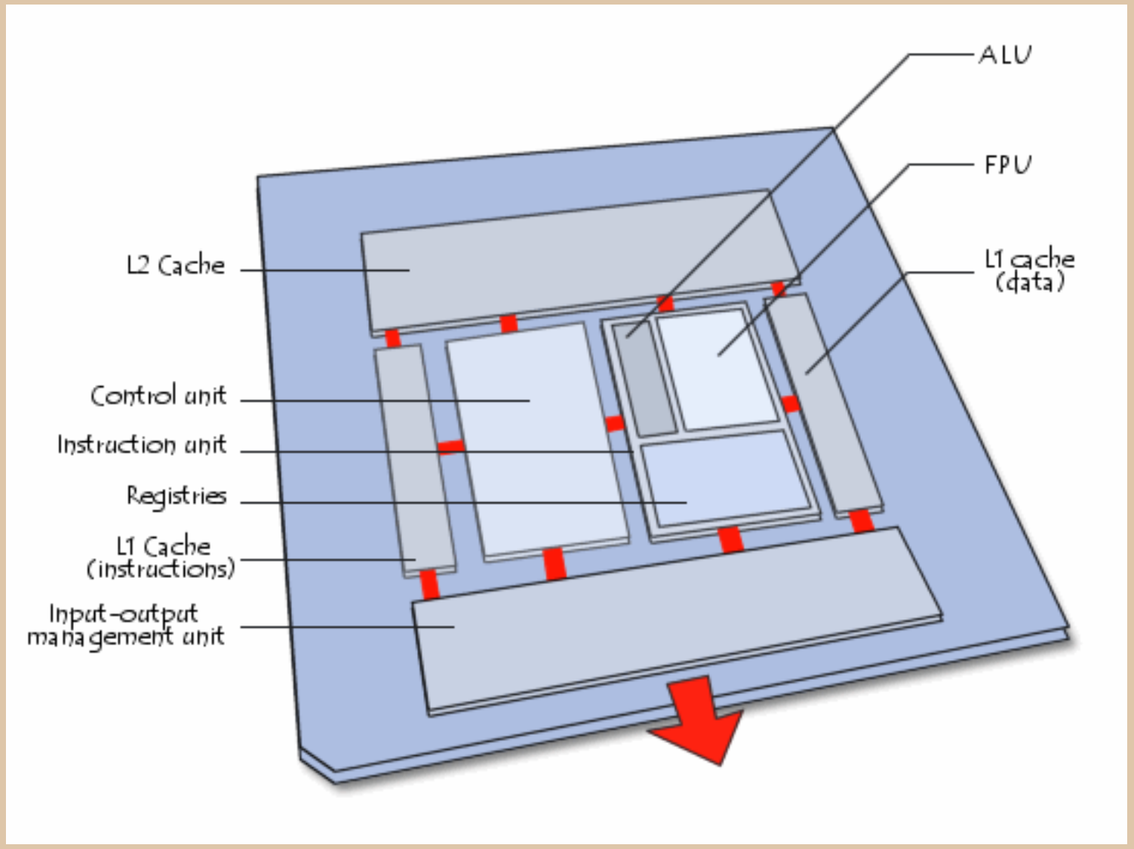


Cache coherence.

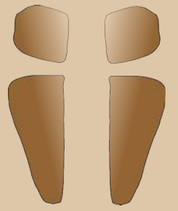


Cache is . . .

- Level 1 – L1
- Level 2 – L2
- Level 3 – L3



http://static.commentcamarche.net/en.kioskea.net/faq/images/KjBt8WVH4Epro7ik.png

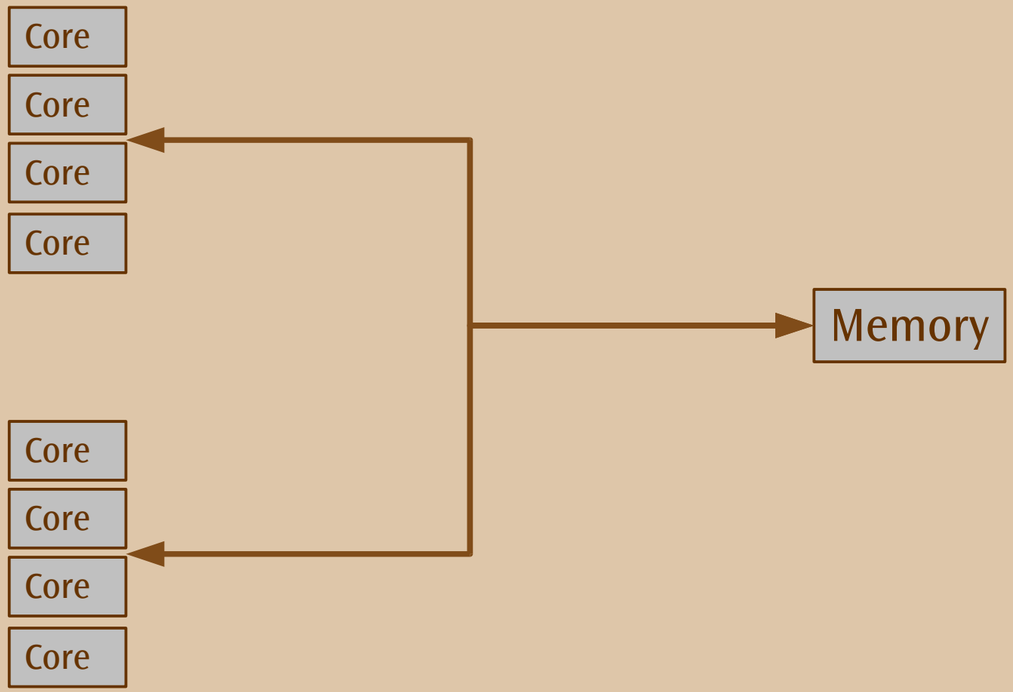


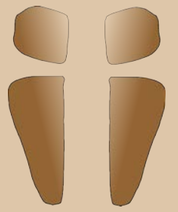
Cache is . . . the Problem

- Level 1 – L1
- Level 2 – L2
- Level 3 – L3

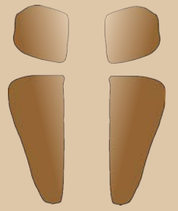
Just “sticking plaster” to avoid addressing the real issue.

2005 (ish) – 2012 (ish)

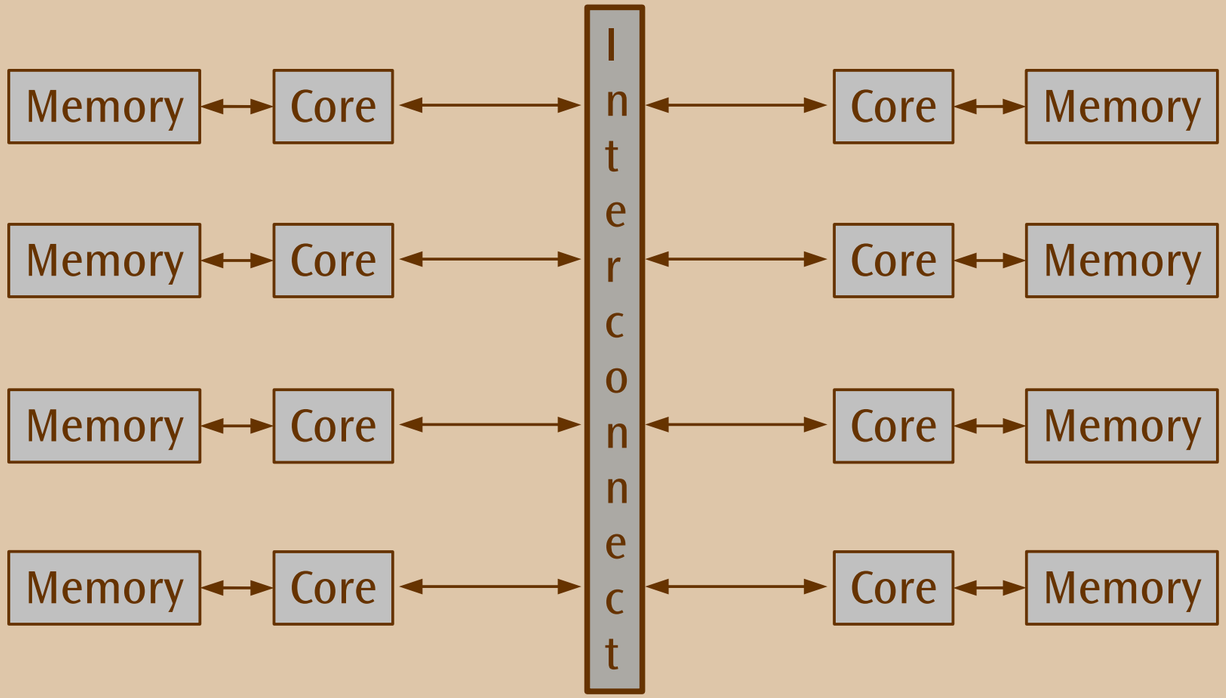


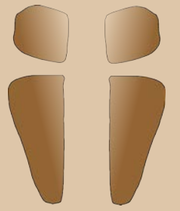


Solution?



Change the model!

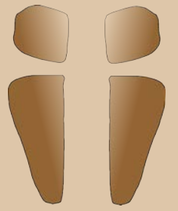




There's Nothing New

- Standard model of a cluster system.
- Standard model of a blade system.

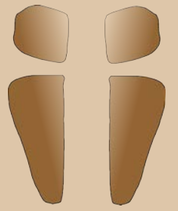
*Remember this is **parallelism***
*not **concurrency**.*



So What Is New?

- It's the standard model of Intel's future many-core (aka multicore) processors:
  - Multi-level storage model.
  - No off-chip memory bus.
  - Cache transformed to primary memory.
  - A return to zero wait state processing.
  - No hugely complex instruction scheduling system.

Reputedly, anyway.

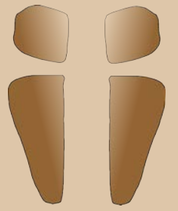


A return to a simpler world?

Can Current Languages Cope?

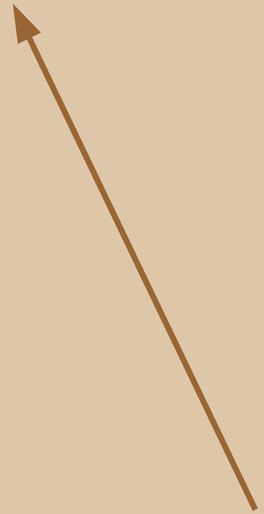
- Of course they can:
  - Fortran, C, C++ with OpenMP, MPI.
  - C++ with Threading Building Blocks (TBB).
  - Java – cf. blade server systems.

OpenMP and MPI in the Java world?

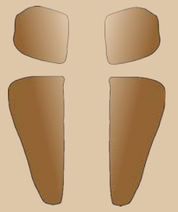


Can Current Languages Cope?

- Of course not:
  - No coherent programming model for the single program in Fortran, C, and C++.
    - SPMD isn't likely to scale.
  - Vast amount of uniprocessor legacy code.
    - Obsession with auto-parallelization so as to avoid rewriting all the code from the 1950s and 1960s that every still uses.

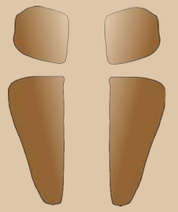


Problem or opportunity?



Examples Required

- Arguments about programming and programming languages need example programs to talk over.
- The core (!) issue of parallelism is scaling: does the performance of a program increase linearly with the number of processors:
  - Depends on the algorithm – obviously an inherently sequential program cannot benefit from parallelization.
  - Only *embarrassingly parallel* problems can ever approach linear scaling.



A Problem – π

- We know the value of π exactly, its π (obviously).
- But 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.

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

This problem is embarrassingly parallel.

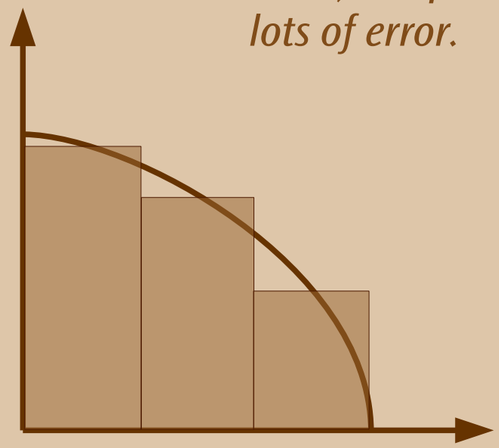
- Estimate the value of the integral using a summation.

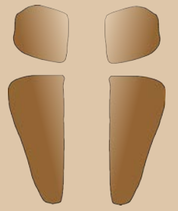
$$\pi = \frac{4}{n} \sum_{i=1}^{n} \frac{1}{1 + \left(\frac{i - 0.5}{n}\right)^2}$$

Quadrature

This is not actually a rendering of the problem being solved!

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

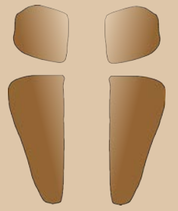




Less Structured Problems

- Most problems are less open to “perfect parallelization”.
- Synchronization problems from classic concurrency make for some fun.

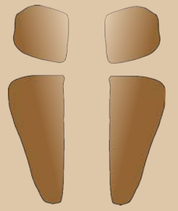
“Dining Philosophers” is the problem most people think of?



A Totally Different Problem – The Sleeping Barber

- A problem from Edsgar Dijksttra – as was “Dining Philosophers”, but “Sleeping Barber” is different and far more fun.
- In a barber's shop:
  - One barber sleeps in the cutting chair if there are no customers.
  - A customer enters the shop and:
    - If the barber is asleep in the chair, the customer wakens the barber, sits in the chair and gets a hair cut.
    - If the barber is cutting another customers hair, the new customer checks to see if there is a vacant waiting chair and if there is the customer takes the chair and waits a turn to have a hair cut.
    - If there is no waiting space, the customer leaves to seek a hair cut elsewhere.
- The “Sleeping Barber” problem (just like the “Dining Philosphers” problem, is an issue in process management within operating systems, but is far more fun when treated as a simulation problem.

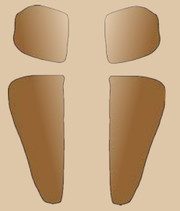
Example Code



- Fortran, C++ – threads, MPI, OpenMP. TBB. C++0x Futures.
- Java – threads, executors, java.util.concurrent. Futures.

Are futures the future?

Data parallelism is different from what most people who like threads think of as parallelism.

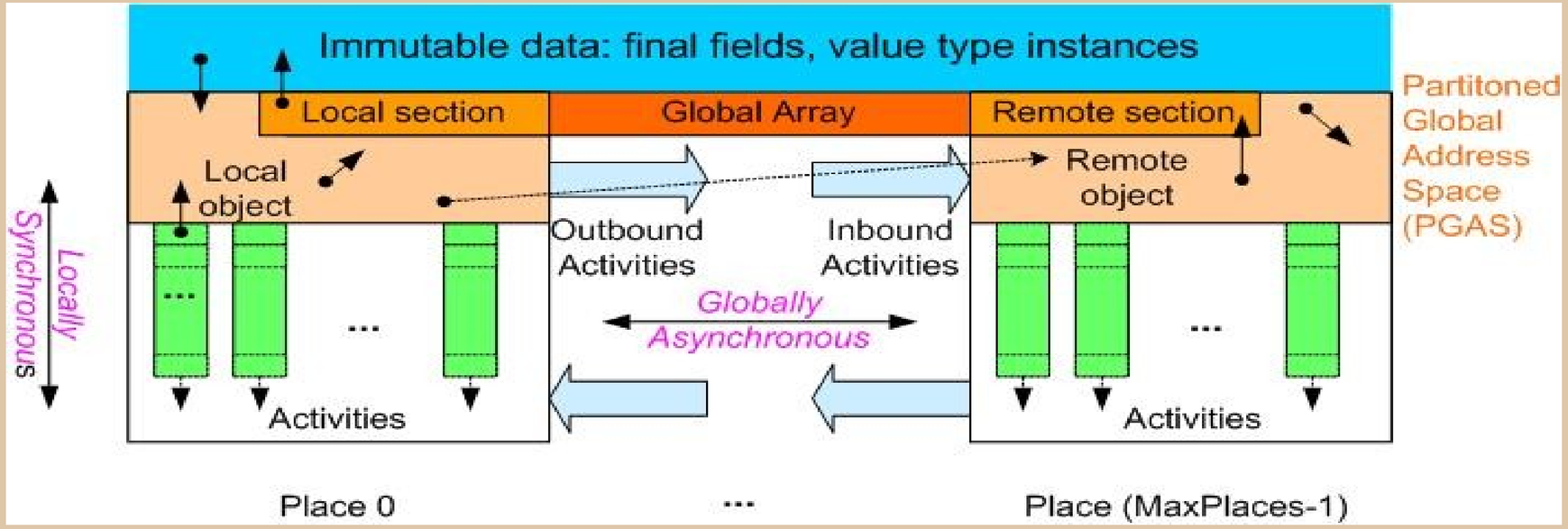


What else to do?

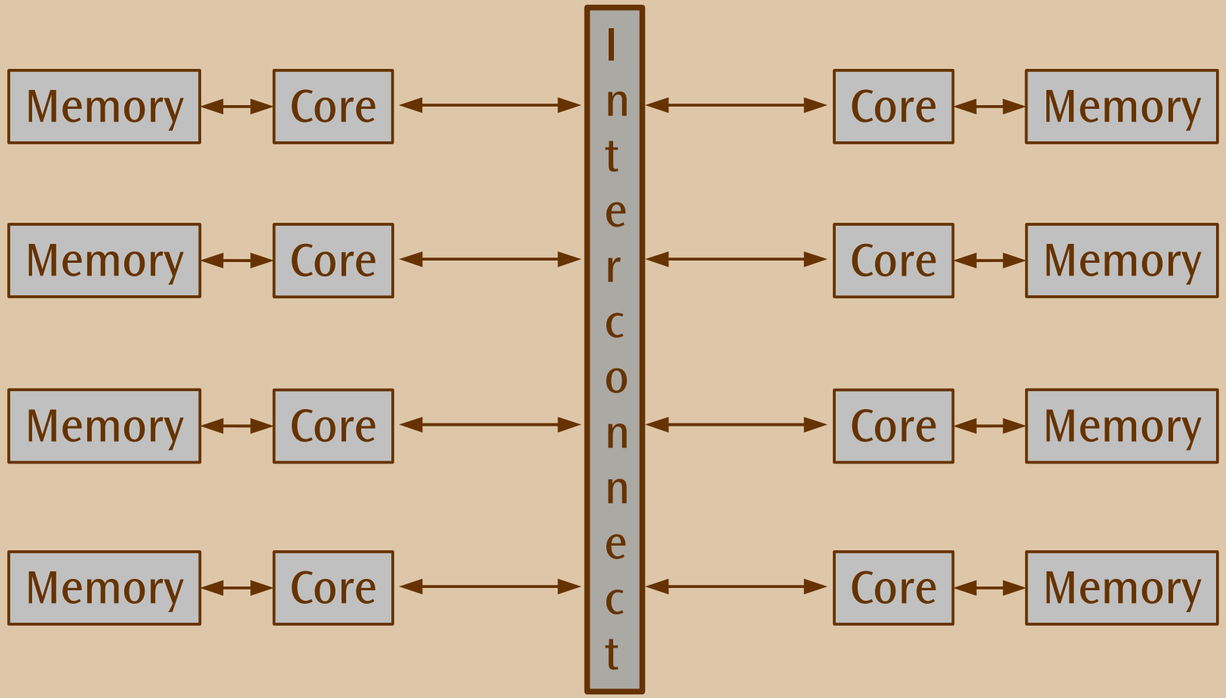
- Invent PGAS (partitioned global address space) languages:
  - Chapel
  - X10
  - Fortress

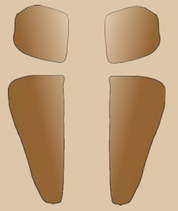
PGAS

- Virtual address space is partitioned and mapped to the distributed memory of the underlying machine:
  - Chapel: domains
  - X10: places



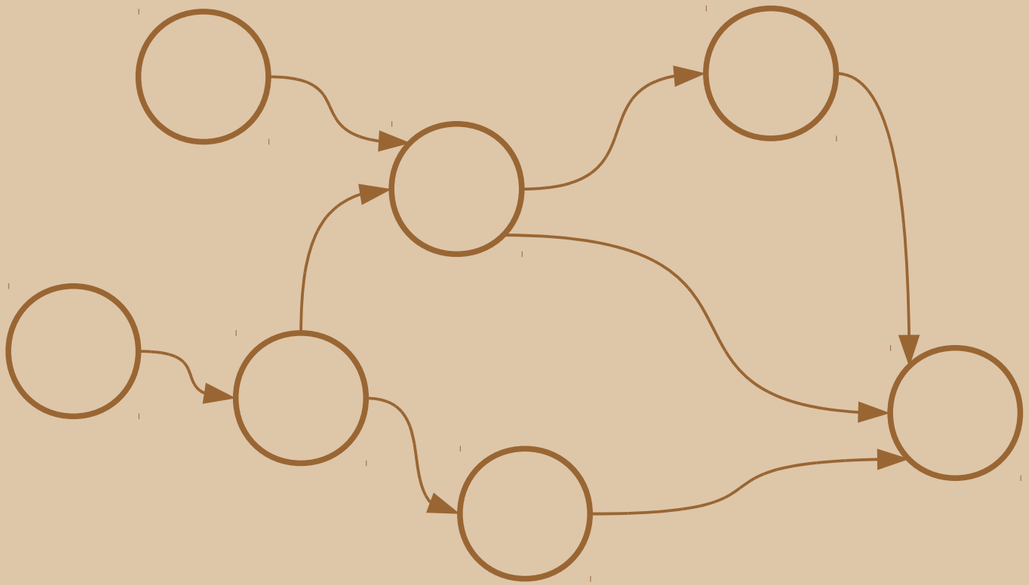
http://dist.codehaus.org/x10/tutorials/x10-2.0/SuperComputing2009/SC09_X10_Tutorial.ppt

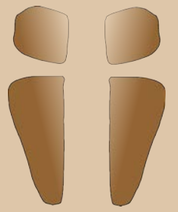




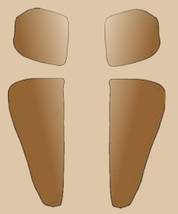
What better to do?

- (Re-)Invent process-based models of computation:
  - Actor Model: Erlang, Scala, D, . . .
  - CSP: C++CSP2, JCSP, Groovy CSP, Python-CSP, . . .
  - CSP-like: Go, . . .
  - Dataflow: GPars, DataRush, . . .

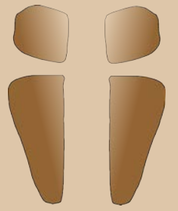




It’s all about small processes passing messages to each other.

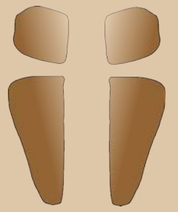


A different way of structuring software.



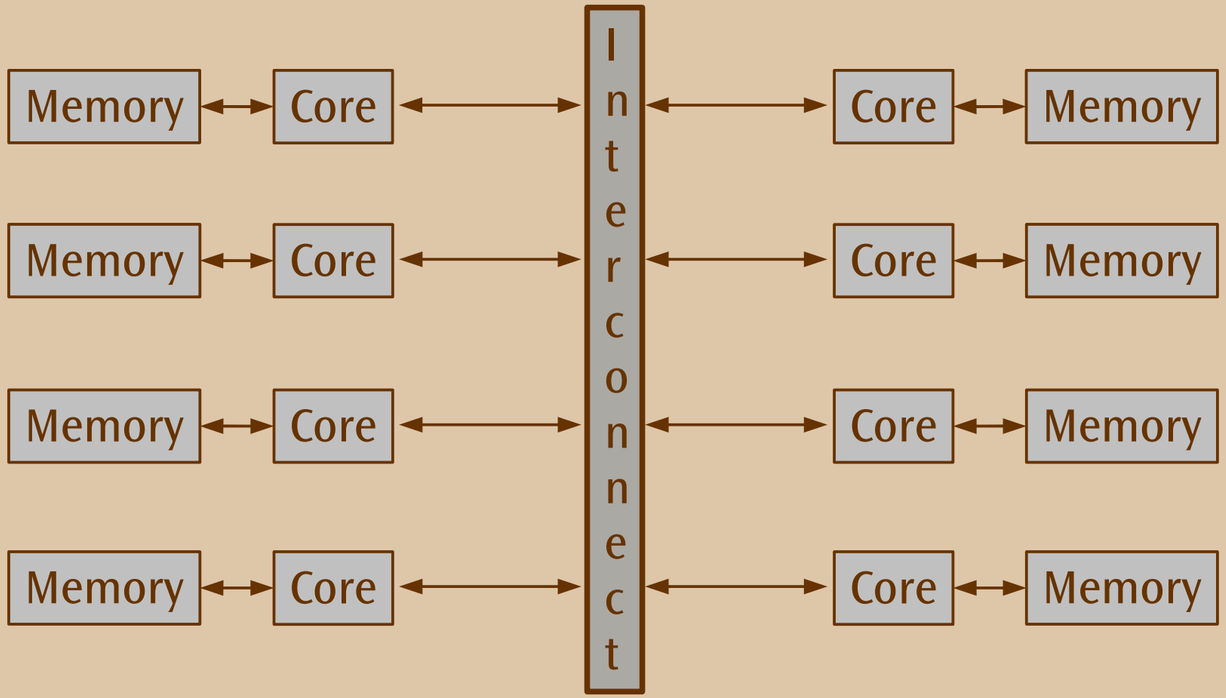
A return to the old ways of processes and message passing.

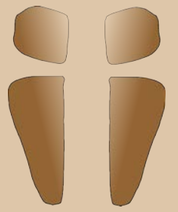
Threads are there as implementation infrastructure.



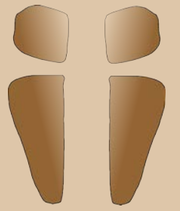
Who's Doing It?

- Scala – actors, (especially using Akka).
- GPars – using Groovy to bring things to Java programmers.
- Python-CSP, PyCSP.
- Go – goroutines are a form of CSP.
- D – has what amounts to an Actor Model.
- C++ – Just Threads Pro will provide actors, data parallelism, etc. over C++0x threads; C++CSP2.

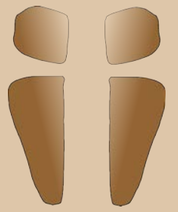




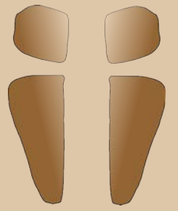
This is all fine for computation and HPC, but what about applications people actually use?



What is the problem?

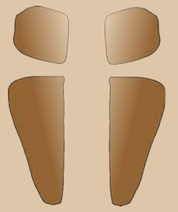


An operating system requires all resources under management in a single address space.

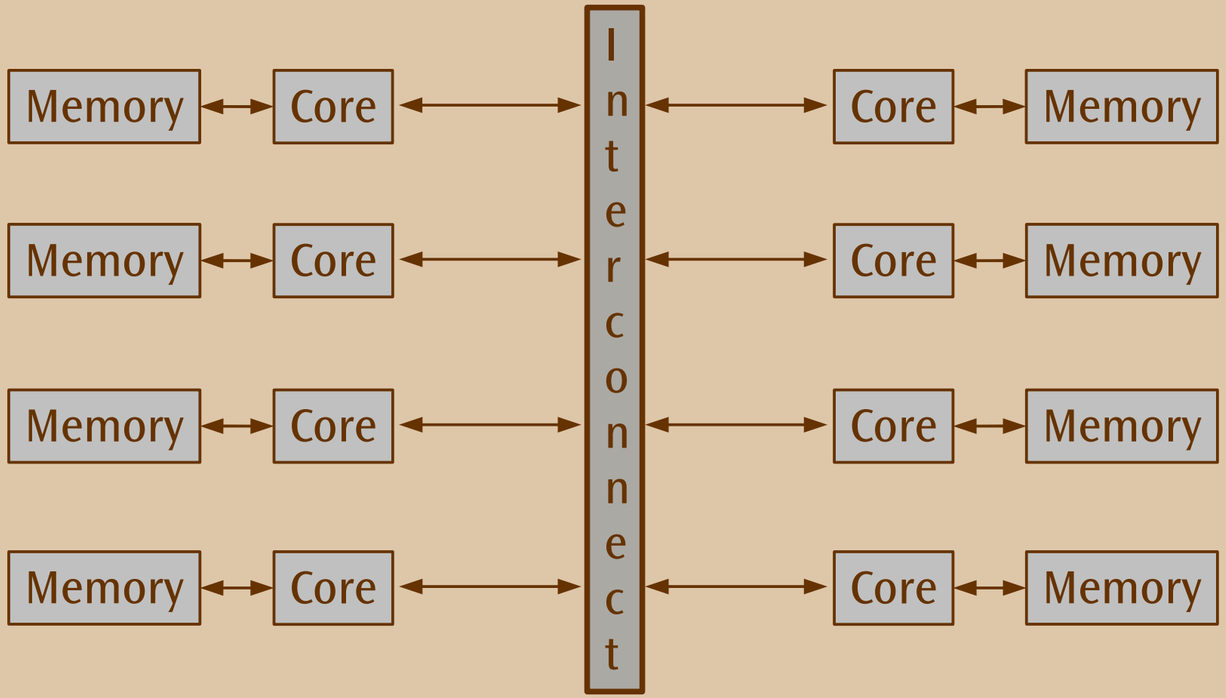


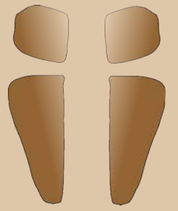
Current Operating Systems

- Have no NUMA (non-uniform memory architecture) concept.
- Windows and Linux assume a single, flat address space.
- Multiple address spaces mean multiple instances of an operating system communicating by message passing.
- Current distributed system technologies do not work well at chip speeds, they work at network speeds.



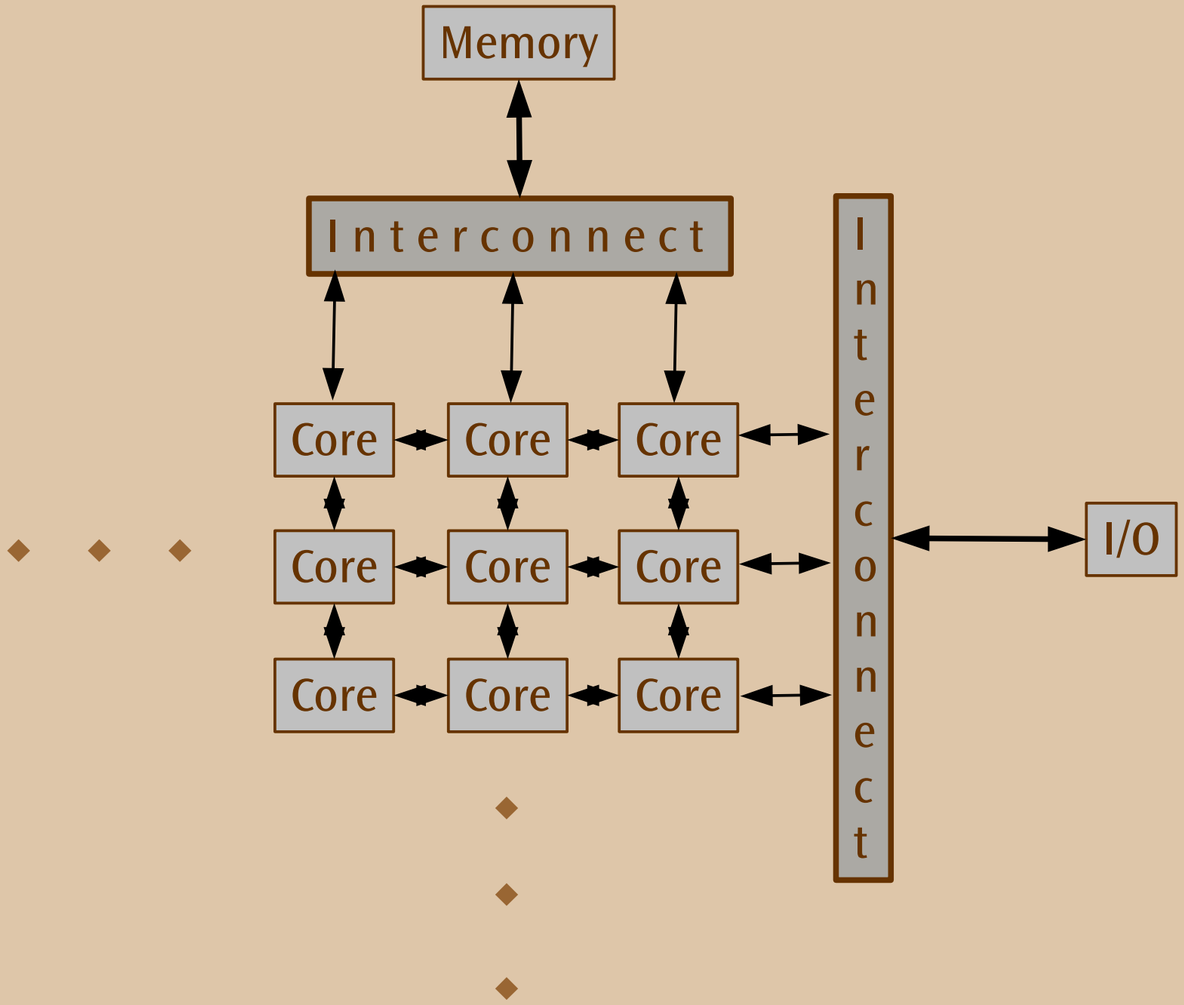
But most of the Top 500 use Linux and what's good for HPC must be good for the new architectures.

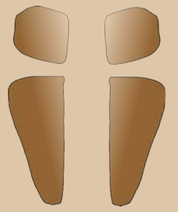




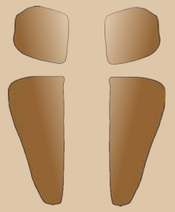
Think Intel.
Think AMD.

Think Tilera.

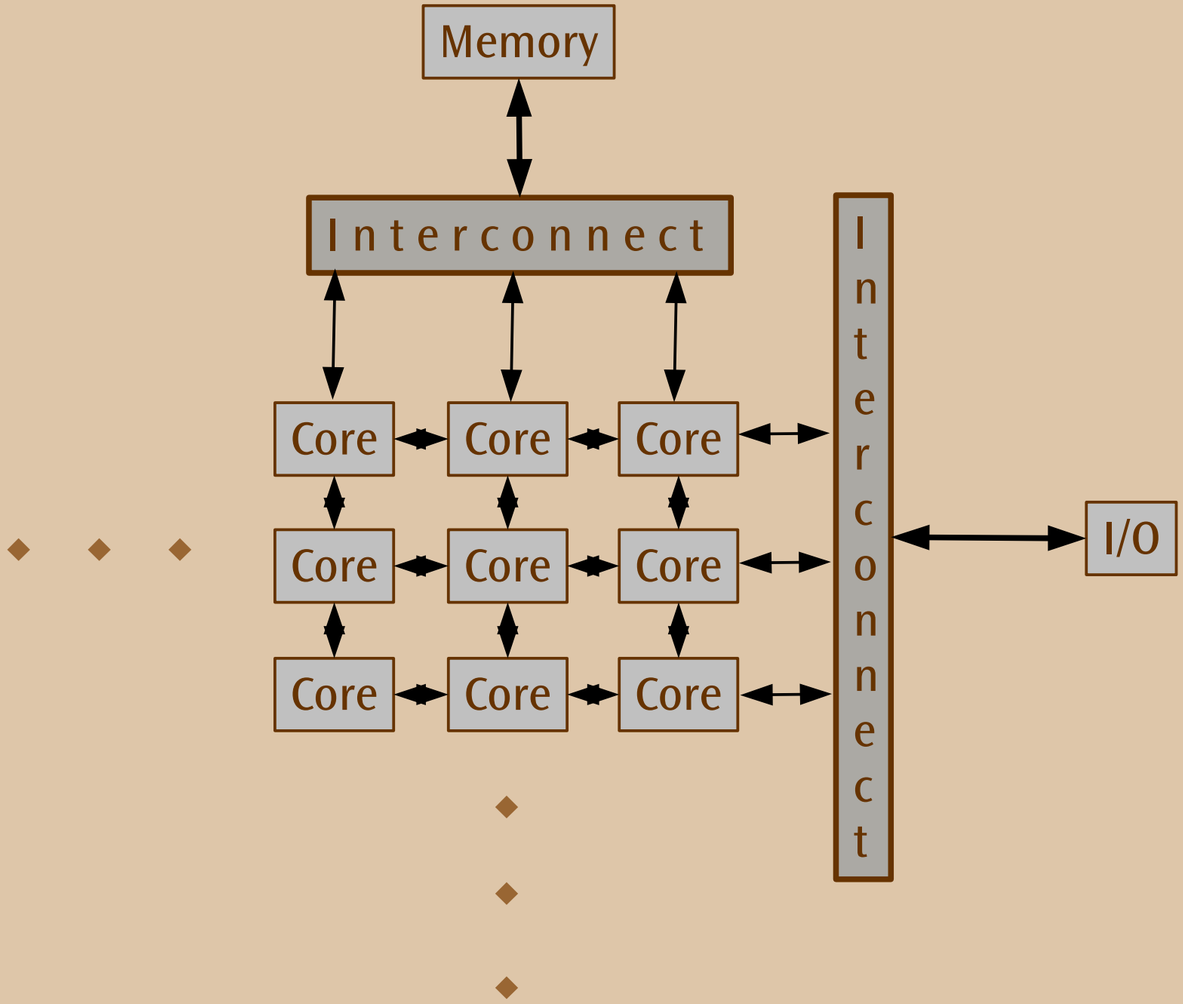


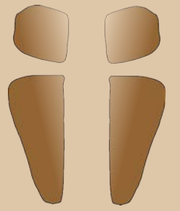


Think Hypervisors.

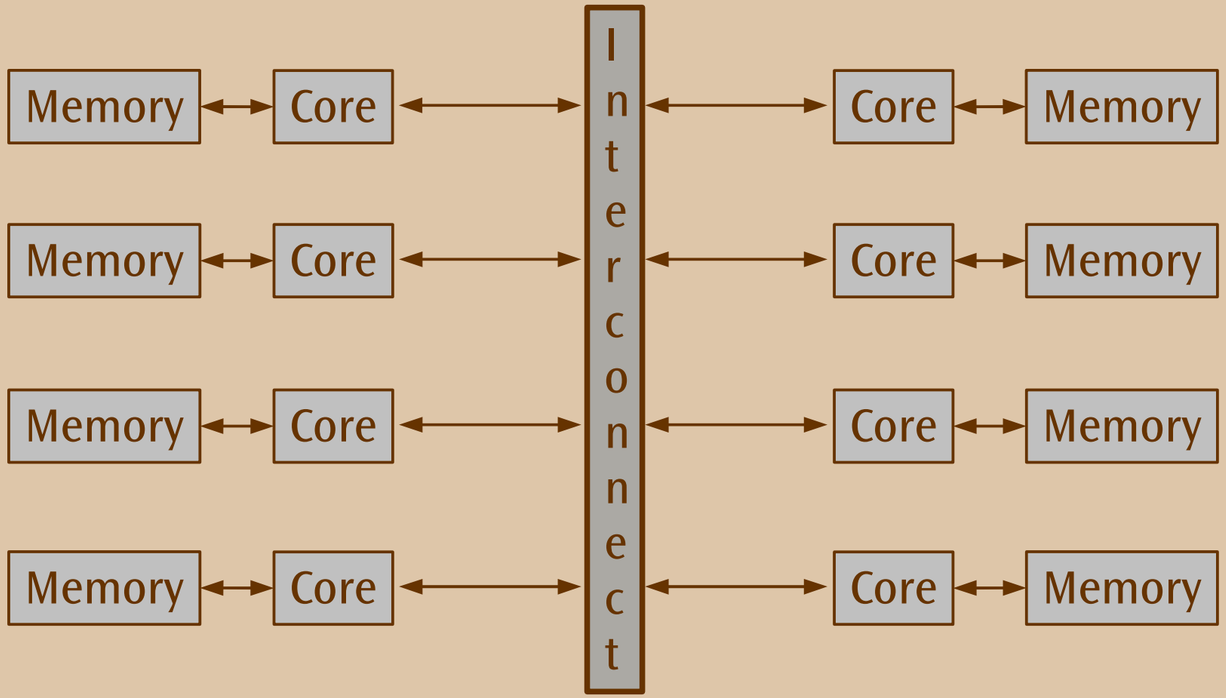


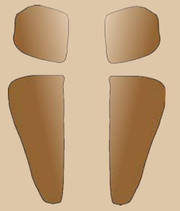
Partition processors into groups, run different operating systems on different groups.



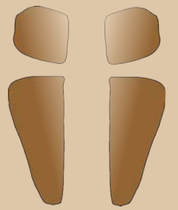


Still has a single memory bus!

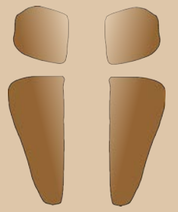




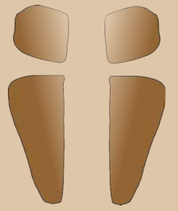
Heterogenity is the future and is coming.



Intel has said so.



Cores on a chip will not all be the same architecture and some of them will be GPGPUs.



Operating System Lottery

- Can any of:

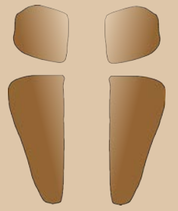
  Linux

  Windows

  Mac OS X

  Solaris

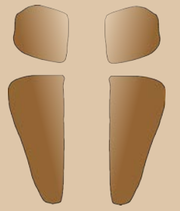
  cope with a heterogeneous collection of processors?



A Way Forward

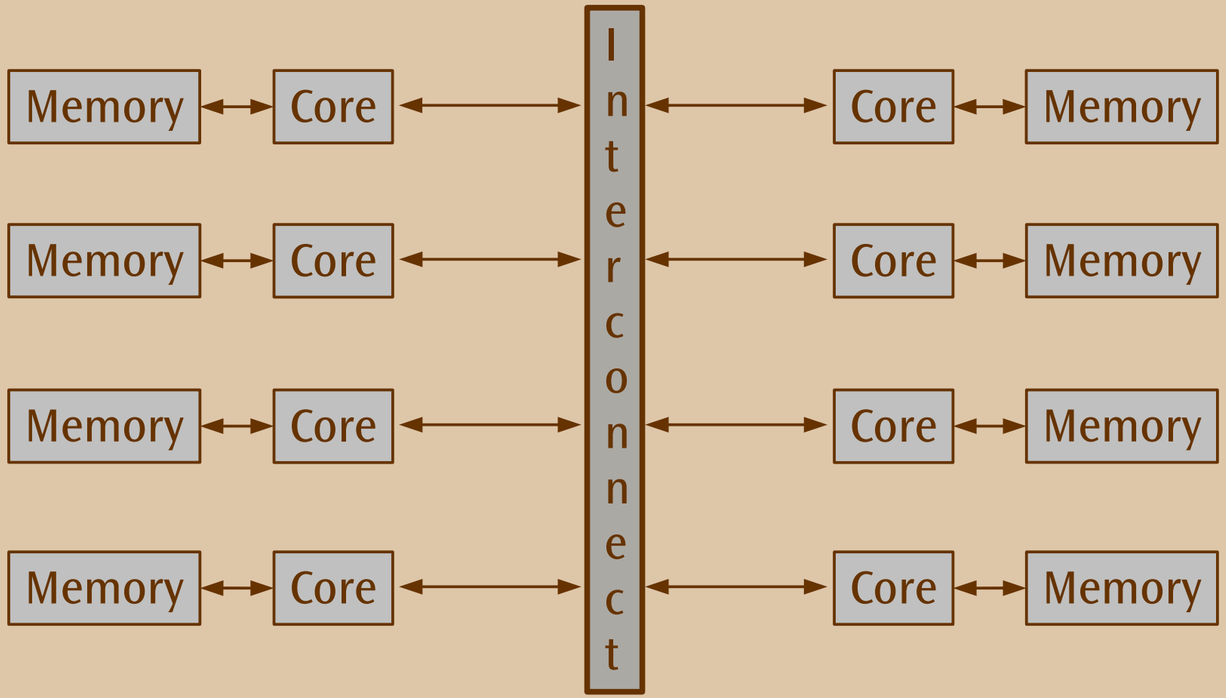
- New operating system using a PGAS programming model:
  - Chapel
  - X10

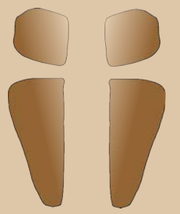
  as operating system implementation languages?



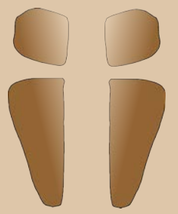
More likely?

- Use a microkernel on each core.
- Load up additional modules, as needed, on a core by core basis.
- Have Linux, Windows and Mac OS X API emulation modules.

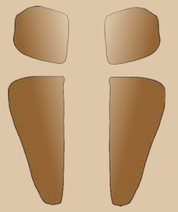




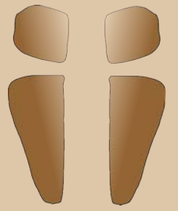
Is L4 the future architecture of operating systems?



Can it be arranged for an instance of Word / OpenOffice.org to have its' own processor with appropriate OS emulation?

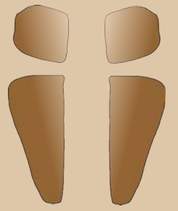


It all depends on how much memory each core has.

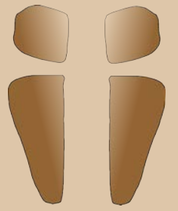


Options

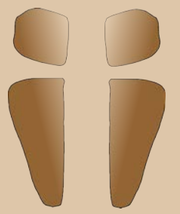
- If cores can only have a few tens of MB then it remains cache and the single memory bus architecture is here to stay.
- If cores can have hundreds of megabytes, then each can be independent and the current single memory can become secondary storage.



It's all just questions and speculation.



But change is coming.

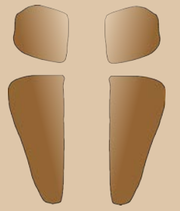


Multicore hardware

Programming models

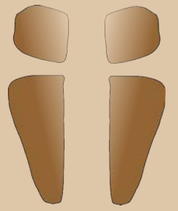
Programming languages

Dynamic vs. static



Summary – 1

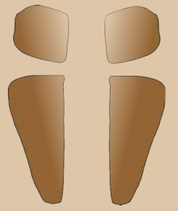
- Uniprocessor thinking is now insufficient to the task of software development.
- Shared memory multi-threading is an operating system technique not an applications programming one.



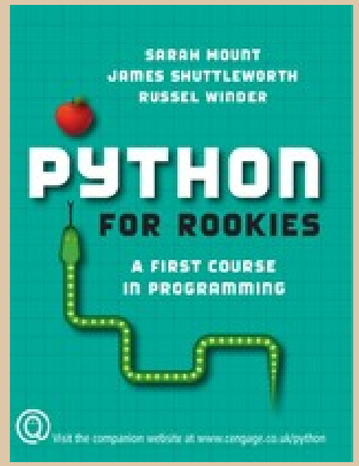
Summary – 2

- Models such as:
  - Actor
  - Dataflow
  - CSP
  - Data parallelism

  are the ones that applications should be structured with.
- Dynamic languages have expressibility but poor performance: applications can mostly be written in dynamic languages for ease of development and maintainability with performance critical sections written in static languages.



Blatant Advertising

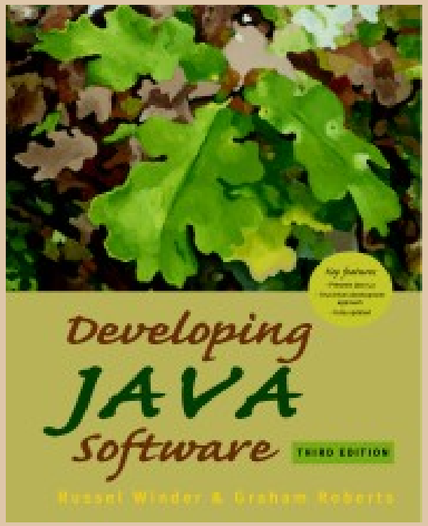


Python for Rookies

Sarah Mount, James Shuttleworth and Russel Winder

Thomson Learning

Now called Cengage Learning.

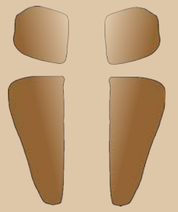


Developing Java Software *Third Edition*

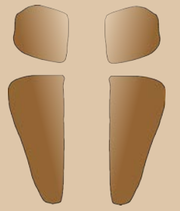
Russel Winder and Graham Roberts

Wiley

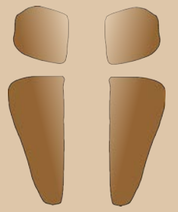
Buy these books!



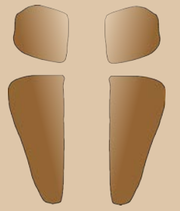
Processor hardware is rushing off into the multicore/many-core future.



Software is lagging behind, dramatically.



Again.



Parallelism can be Groovy

Dr Russel Winder

It'z Interactive Ltd

russel@itzinteractive.com