Multi-Core CPUs

Who Cares, Anyway?

Nick Maclaren

nmm1@cam.ac.uk

March 2009

Overview

This seminar is going to be grossly over–simplified

For end–users and administrators, not programmers
It is NOT about High Performance Computing

Will start with the present and next 1–2 years

- Why is multi–core the future?
- What is multi–core today?
- Using up to 8–way systems
- When to use and how to choose them

Why Multi-Core?

Parallelism is the next big thing in computing

Yawn!

Have heard that once a decade since mid–1970s
So why should we believe it this time?

To do with Moore's Law and Not–Moore's Law
Yes, but this time it's for real . . .

(Not-)Moore's Law

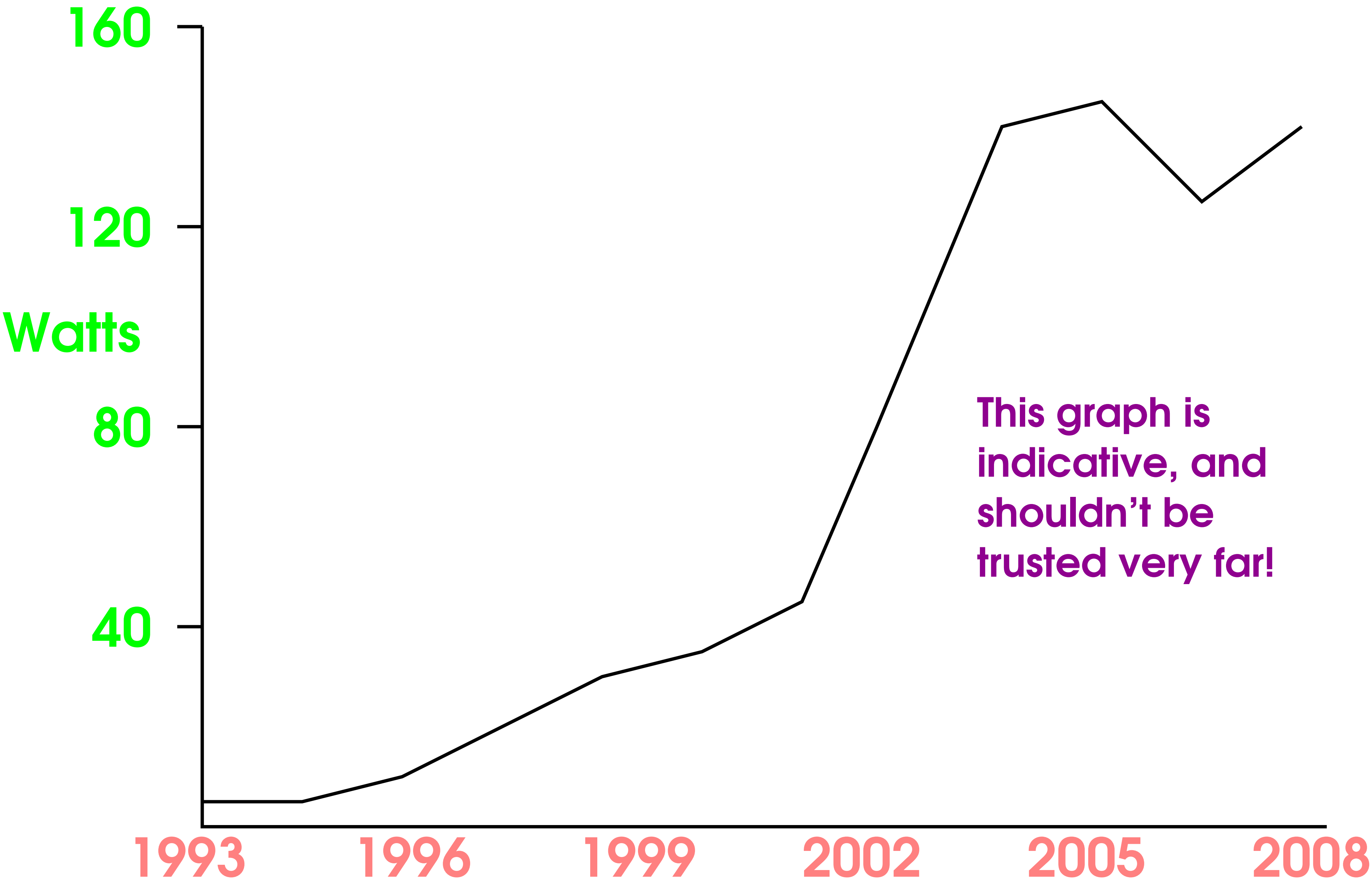
Moore's Law is chip size goes up at 40% per annum
Not–Moore's Law is that clock rates do, too

Moore's Law holds (and will for a decade or so)

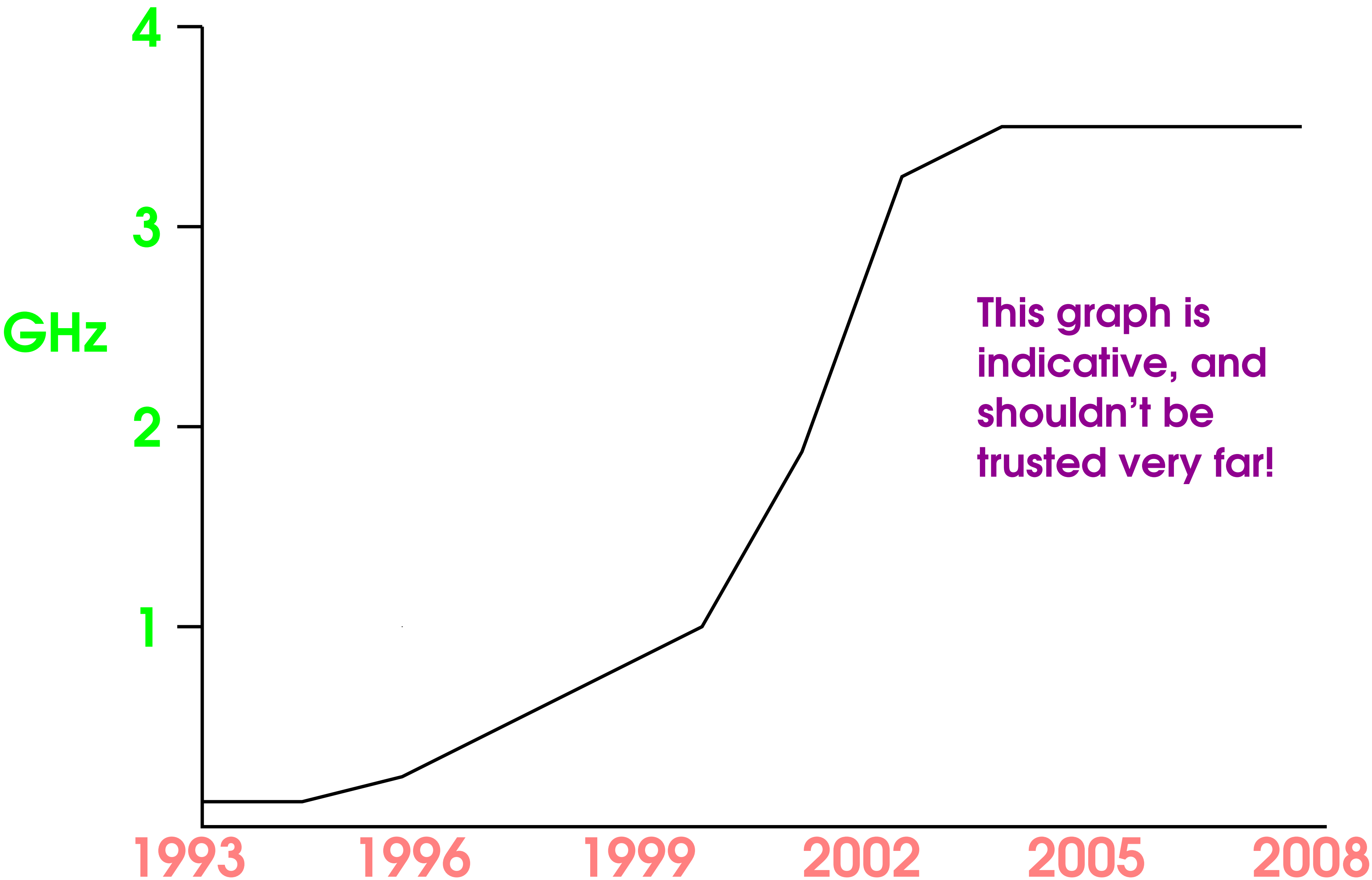
Not–Moore's held until ≈2003, then broke down
Clock rates are the same speed now as then

Reason is power (watts) – due to leakage
See http://www.spectrum.ieee.org/apr08/6106

Power Consumption of CPUs



Clock Rate of CPUs



Manufacturers' Solution

Use Moore's Law to increase number of cores
So total performance still increases at 40%

2009	–	typically 4 cores
2014	–	typically 16–32 cores
2019	–	typically 128 cores

Specialist CPUs already have lots of cores
Used in areas like HPC, video, telecomms etc.
Currently irrelevant to "general" computing

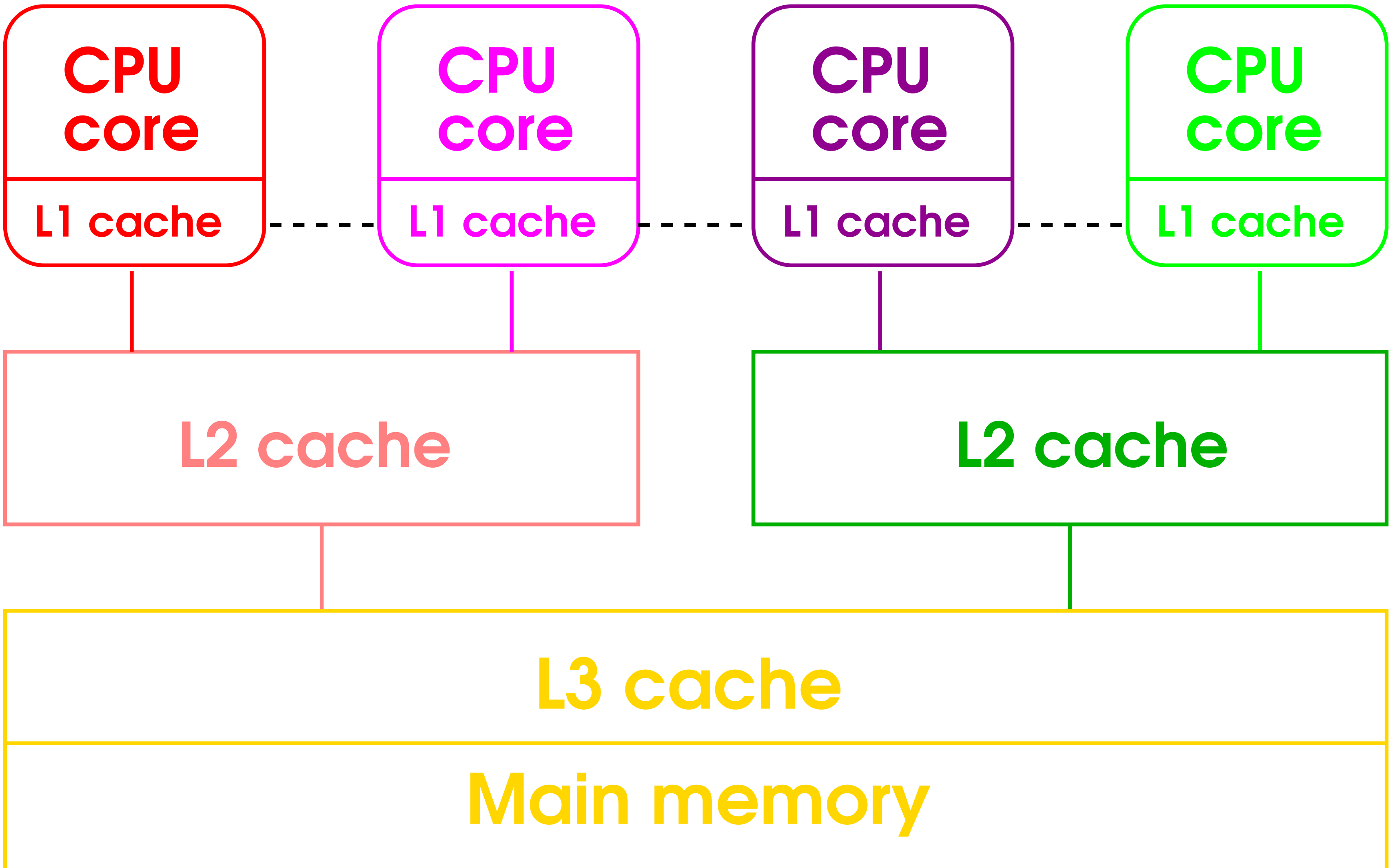
What Is Multi-Core Today?

Exactly the same as multi–socket yesterday
Even down to the system programming level

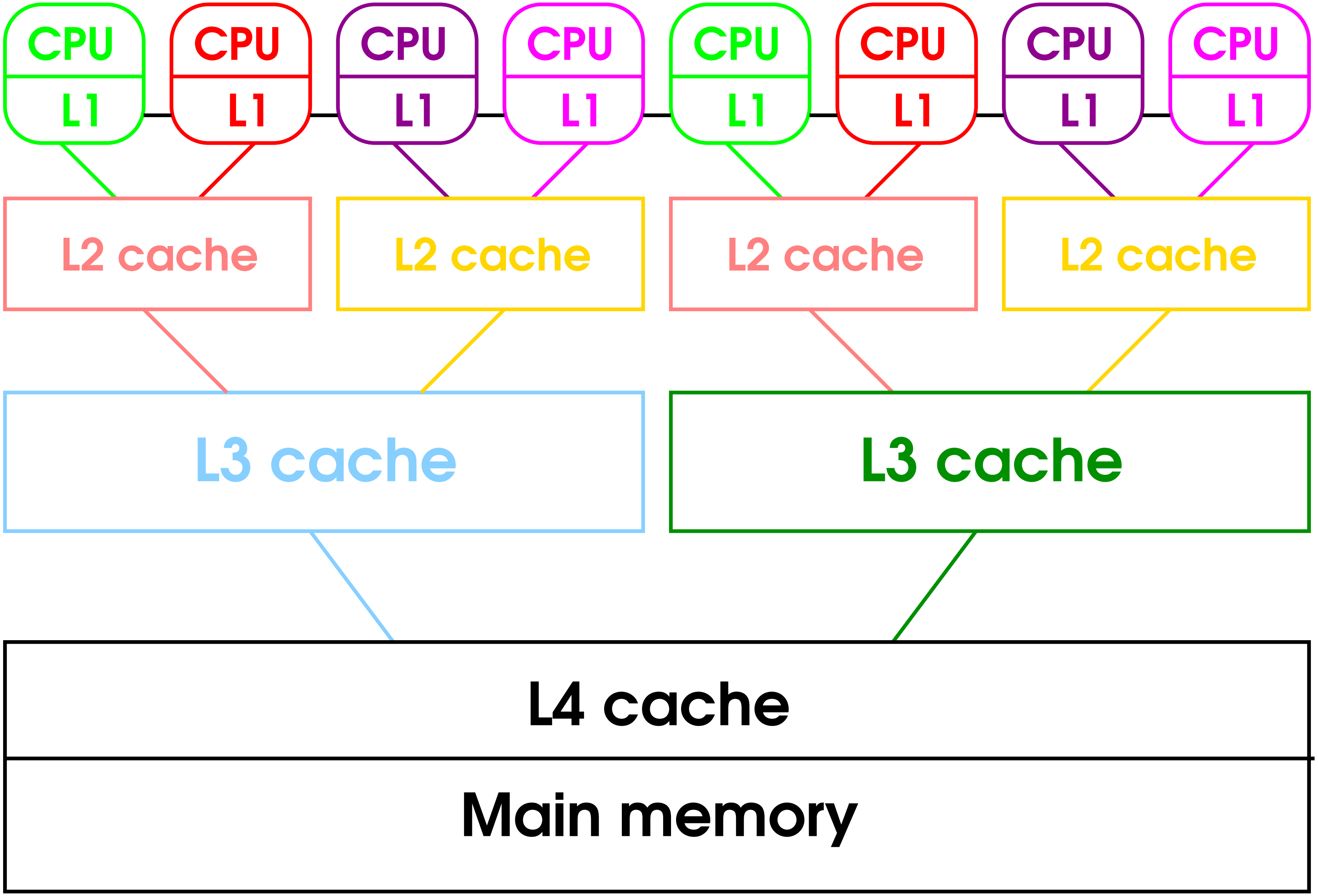
Same is true for multi–socket, multi–core
Don't need to worry about the details

Only important subtlety is NUMA design
Stands for Non–Uniform Memory Architecture
May apply to more than memory, e.g. I/O ports

NUMA Design


CPU core
L1 cache
CPU core
L1 cache
CPU core
L1 cache
CPU core
L1 cache
L2 cache
L2 cache
L3 cache
Main memory

Future NUMA


CPU
L1
CPU
L1
CPU
L1
CPU
L1
CPU
L1
CPU
L1
CPU
L1
CPU
L1
L2 cache
L2 cache
L2 cache
L2 cache
L3 cache
L3 cache
L4 cache
Main memory

How Do They Work?

Each core is essentially an independent CPU
A thread runs on exactly one core
The system schedules threads onto cores

Don't need to worry about thread interactions
Only kernel developers need consider them

Interactions can affect performance
HPC people can have major problems with that
Anyone else can usually avoid most of them

Cache Coherence

CPU ensures that all cores see the same data
Transfers data between caches if needed

So why do you need to know about NUMA?

Almost entirely performance (will come to later)
Hence some implications for system administration

We can be talking about a factor of ten or more

Servers Timeline

2009–10 – few problems for administrators

2011–12 – parallelism issues increasing
Memory bandwidth will be a major bottleneck
Configuration increasingly about parallelism
Most daemons will be parallel, in some way

2013–14 – parallelism is critical
All administrators need to understand the issues
Serial daemons may be critical bottlenecks

2015 onwards – parallelism rules, OK?

Using Multi-Core Today

Let's assume you are running a server
There are lots of independent, active tasks

Just Do It

Modern systems are designed for such use
You will often see near–optimal throughput

You may need to do a little tuning, of course

Current Daemons

Most start one serial thread per task
Think of a Web server as a typical example
Rarely have reliability problems (see later)

Modern schedulers are designed for such uses

It's Nick being optimistic – one for the record!

Nearly As Simple

Let's assume you are running a workstation
Spending most of your time using GUIs etc.

GUIs typically have half-a-dozen active tasks
1 core good, 2 cores better, 4 cores best

But you won't get much benefit from 8+ cores
Unless your applications are themselves parallel

The Problem Requirement

One application or daemon is the bottleneck

If it's serial, adding more cores won't help
Why not just run your old computer until it breaks?

This is why applications are being made parallel
That needs a redesign, not just tuning
Not easy to do, and most developers get it wrong

Will come back to this one

When To Use Them

Fairly obvious from the above – but only in theory!

Most “CPU–bound” programs are memory–bound
CPUs are 100–1,000 times faster than memory
$2\times$ cores need a $2\times$ better memory subsystem

Problem made worse by arcane scalability problems
E.g. 8–socket Opteron systems run like drains

Doubling core count can even reduce throughput

Choosing Systems

Avoid the leading edge, and will rarely have trouble
E.g. not maximum core count supported on board

Not yet much of a problem except for HPC people

Currently little experience with quad–core systems
Especially in their multi–socket configurations

A few people around the University have them

Benchmarking

Never trust results from lower core−counts!

Don't trust the Web or vendor benchmarks, either
Very few of them stress the memory subsystem

No substitute for benchmarking realistic workloads
Or wait until someone reliable has tried the system!

I have some artificial benchmarks, which can help
Expose problems with the memory subsystem

Configuration Issues

Can be slow because of system configuration
Not often a problem for 2–4 cores, anyway

A few applications run an insane number of threads

Solution:

Constrain number of active threads, somehow
Typically ALL you need to do!

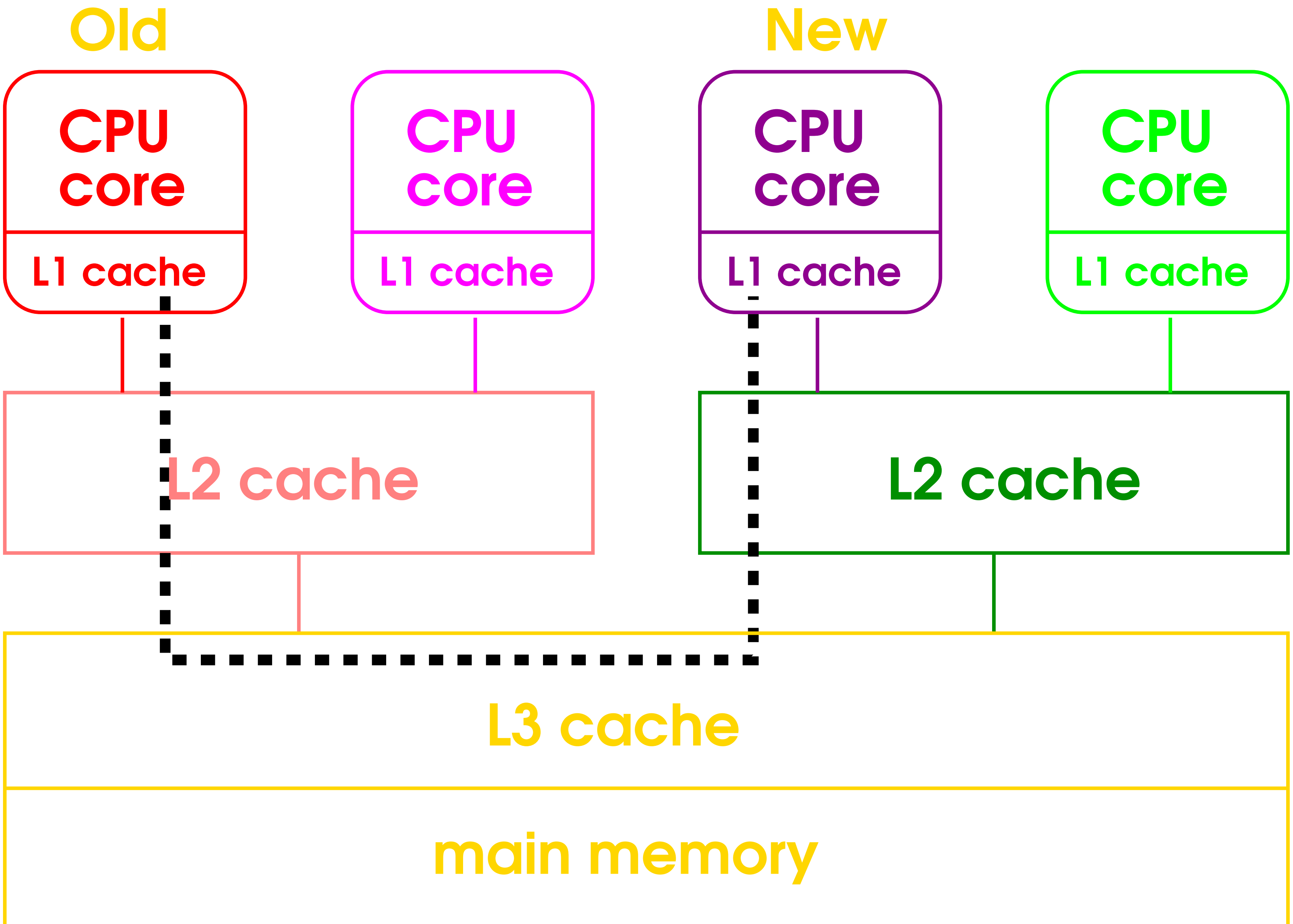
Why is This?

Not usually scheduler bugs, despite appearance
More often too many, interacting threads

Multiple applications may start too many threads
Hence too much context switching or waiting
Applications often assume they are the only one

The scheduler may migrate them between cores
Obviously has to copy their data between caches
Too much of that is obviously very inefficient

Migration


Old
New
CPU core
L1 cache
CPU core
L1 cache
CPU core
L1 cache
CPU core
L1 cache
L2 cache
L2 cache
L3 cache
main memory

So Far, So Good

That's all I am going to say about where we are
There really aren't a lot of major problems

Except for the HPC people, of course, ...

Any comments on what I have said?

The Next 5 Years Or So

Mostly about what will probably happen
I.e. if current directions continue unchanged

But we might see some revolutionary changes
I don't expect them until after 2020, though
And have NO feeling for the form they will take!

Will cover:

- Hardware developments (fairly clear)
- Software developments (much less so)

Hardware Developments

Core count doubling every 2 years – reminder:

2009	–	roughly 4
2011	–	roughly 8
2013	–	roughly 16
2015	–	roughly 32
2017	–	roughly 64
2019	–	roughly 128

Memory and I/O Bandwidth

Won't keep up with core count, unfortunately
May cause server owners a lot of headaches

bandwidth $\approx$ clock rate $\times$ pin count

Clock rate is not going to increase much
And nor are pin counts, on physical grounds
Sockets are already 50 sq.cm and 1500 pins!

- Don't waste money on cores that you can't use!

Intel's Good News

Intel's Nehalem (Core i7) is much better
It is implemented much like AMD's Opteron

Roughly double the bandwidth of the current CPUs
So (obviously) can support double the core count

This improvement is a one–off – can't be repeated
And delivering it needs 25% of the CPU power
But it keeps the pressure off for 1–2 years

AMD needs to catch up with Intel (badly) . . .

Niagara, Larrabee etc.

Sun and Intel 32–core CPUs
Low power, because of low clock rate

First is aimed for "transaction processing"
Second is for graphics, to displace GPUs
Also IBM/Sony Cell, NVIDIA GeForce/Tesla

Will they take off? Place your bets now . . .
Might make 128+ cores mainstream by 2013

Important for HPC and games, but not much else
The question is whether that will change

Application Timeline

Crystal ball failure

Almost all will claim parallelism by 2013
But we all want to know what they will deliver

30+ years of almost no progress is hard to explain

It is very unclear what is going to happen now

Software Developments

Nobody here's a programmer, right?

Only going to cover effects on applications
As seen by ordinary users and administrators

No change for existing application designs
I.e. the one thread per task model
This is often called "natural" parallelism

Many others will need parallelisation, too
Needs code reorganisation at a much smaller scale
A lot harder – and very unreliable

The Easy Cases (1)

Consider a Web server – that's almost trivial
Can support clients pro rata to cores

Each request runs no faster, of course
They had better not update shared data much
And they should interlock properly when they do

A lot of "Internet servers" are like that

The Easy Cases (2)

Vendors' standard libraries are already parallel
Unfortunately, only for HPC–style uses
Very useful for scientists, but not for others

Video rendering is also highly parallelisable
That's why GPUs deliver the performance they do

System configuration

The basic rule has already been mentioned
But, as core counts get to 8 and above:

Scheduler/applications configurations must match
Not doing so may cause applications to hang

May need to bind kernel threads to specific cores
Perhaps the ones that have the I/O ports
The number of pitfalls in that area is legion

Problems – What Problems?

Will arise with poorly–parallelised applications

Some will run like drains or even hang
Often depending on what other ones are running

Some will misbehave rarely and unpredictably
Even worse, most such failures are unrepeatable

That is the problem I regard as most serious
How on earth do you get those fixed?

Doom, Gloom And Boom?

It won't be catastrophic – or probably not
But it's going to be rough for a decade or more

An increasing rate of erratic misbehaviour
And a lower rate of vendors fixing such problems

There has been a lot of research over 40 years
How to make parallel coding easy and reliable
But effectively no actual progress . . .

That's All

That's all that most people need to know
Probably a lot more than most people want to
Not going to give the other slides, unless asked

They explain my statements, but are a bit geekish

Race Conditions

More–or–less all programs use explicit threading
Threads are purely serial, and share memory
Programmers must manage all synchronisation

Doing it is easy – getting it right is a foul job
Much harder than serial, and few people can

Almost all such failures are due to race conditions
Where the program assumes actions are in order
Does not explicitly force it by synchronisation

Why Is This Rare?

Probability proportional to square of action rate
Only actions that affect thread interactions

Web servers etc. may see a few failures a year
Only hard–core HPC people see them at all often

General codes are being parallelised like HPC
Will take many years to learn how to do that

They fail only on heavy loads of complex tasks
HPC people are familiar with that scenario . . .

Debugging Those Problems

This is one of the hardest things in programming
Current tools provide little or no help
But they look (and are sold) as if they do!

Failures occur perhaps one time in a hundred
Adding checking/tracing will move or stop them
Unrelated changes will often do the same

That's beyond most programmers and vendors

The Real Gotchas

There are actually some far nastier problems, too

Ill–defined language standards is the main one
Problems with memory consistency is the other
But it's not feasible to describe either, simply

- There is a very important consequence

Unreliability will be a problem for a long time

Will mention one hardware/software problem

Cache Coherence

I.e. all threads see same view of memory

VERY hard to deliver, reliably, for complex reasons
Not needed by many parallel programs, today

All current CPUs cut corners, significantly
But very, very few programmers know that ...

Also problems with firmware/software interrupts
A single−bit ECC error could cause program failure
Possibly also TLB misses and floating−point fixups

Sequential Consistency

I need to delve into some geekish topics
To explain why cache coherence is hard

CPU 1 writes to location A
CPU 2 writes to location B
CPU 3 reads locations A and B
CPU 4 reads locations B and A

Can CPU 3 see A updated before B?
And CPU 4 see B updated before A?

Main Consistency Problem

Thread 1	Thread 2
A = 1	B = 1
print B	print A

Now did A get set first or did B ?

0 – i.e. A did   0 – i.e. B did

Intel x86 allows that – yes, really

So do Sparc and POWER

Another Consistency Problem

Thread 1

A = 1

Thread 2

B = 1

Thread 3

X = A

Y = B

print X, Y

Now, did A get set first or did B ?

Thread 4

Y = B

X = A

print X, Y

1 0 – i.e. A did

0 1 – i.e. B did

Consistency Issues

But that's just due to too much optimisation, isn't it?

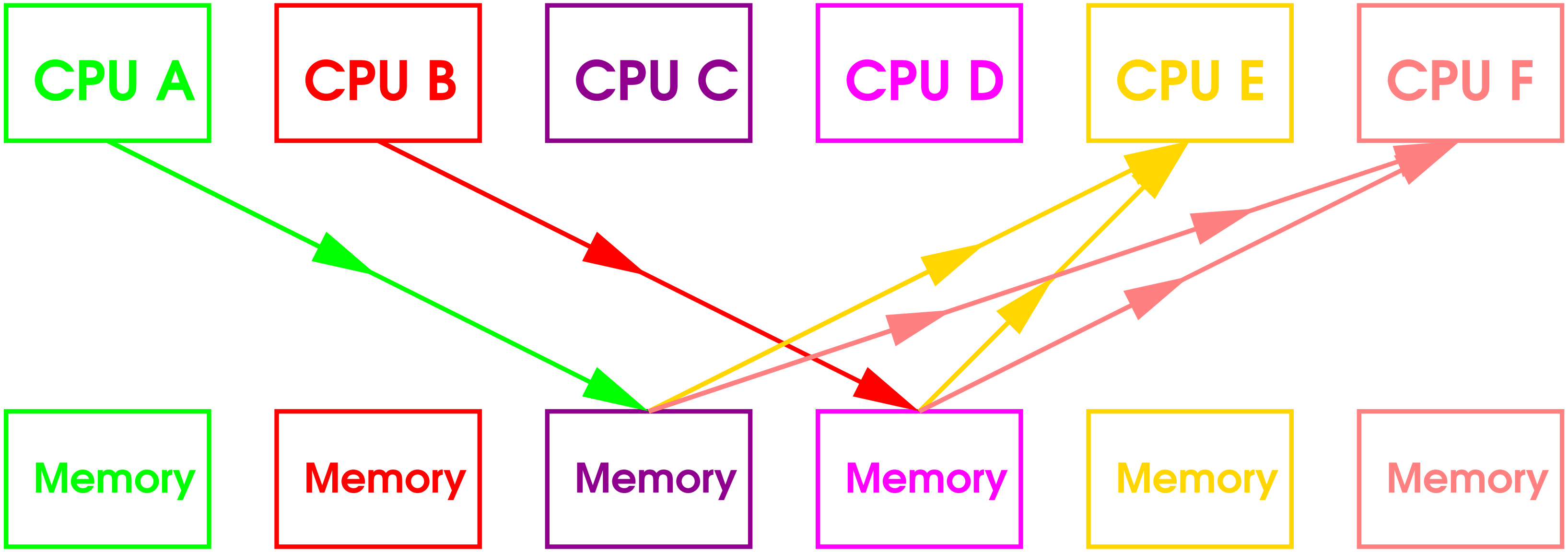
NO!!!

It is allowed by all of C, current C++ and Fortran
AND it is one of the common hardware optimisations
⇒ It can happen even in unoptimised code

- Parallel time is very like special relativity

Different observers may see different global orderings

Cache Coherence



The hardware handles each transfer independently (i.e. 'in parallel')

What Happens

No current CPUs synchronise the updates
So “time” isn’t consistent across cores
But almost all programmers assume that it is

Both cause very rare, unrepeatable wrong results
Probability proportional to square of core count
[Actually quadratic in total event rate]

Only hard–core HPC people see them at all often

Language Standards

Even most “experts” don’t realise how bad this is
Causes trouble porting many threaded programs
It’s not that system/compiler is broken

POSIX, C and C++ concepts subtly incompatible
Can’t even guess what an implementation will do

Rarely specify anything about non–memory actions
Modes, locales, signalling and even I/O
Unclear how much synchronisation is needed

Consequences

Simple code, examples etc. rarely show the issues
Complicated code, and system/library calls do

Problems worse with high levels of optimisation
Why HPC people see it and most others don't

Signal–handling problems are a real nightmare
Many programs use those for communication

Daemons may crash/hang/etc. when prodded

Performance And Tuning

Often hard to separate debugging and tuning
Speed changes can expose hidden race conditions

Many codes will fail if delays are too long
Just as many networking applications do today

Vendors' configuration demands often incompatible

Tuning parallel code is as difficult as debugging
Details omitted because of lack of time