Parallel Programming (2)

Parallel Programming As Such

Nick Maclaren

nmm1@cam.ac.uk

February 2014

Summary

- Extra information and 'patterns'
- A description of SIMD patterns
- A description of tasking patterns
- Current parallel environments
- Introduction to shared memory environments

Reminder

Be warned: parallelism is always tricky

⇒ Become a competent serial programmer first

Do NOT underestimate the challenge of this

You may need to redesign some or all of the code
Usually data structure and often algorithms

Beyond the Course (1)

You are strongly recommended to look at this link:
http://parlang.pbworks.com/f/programmability.pdf

- Ignore the details – note its summaries

Its book has quite a good overview of options
Goes into details I don't (except for dataflow)

Patterns for Parallel Programming
    Mattson, Sanders and Massingill
    Addison–Wesley ISBN 0–321–22811–1

Beyond the Course (2)

http://www.hector.ac.uk/support/documentation/...
...//userguide/hectoruser/hectoruser.html

See "References and Further Reading"

http://www.epcc.ed.ac.uk/library/documentation/...
.../training/

http://www-users.york.ac.uk/~mijp1/teaching/...
.../4th_year_HPC/notes.shtml

http://docs.oracle.com/javase/tutorial/...
.../essential/concurrency/

SPMD

That is Single Program Multiple Data
One program runs multiple threads or processes

Almost universal for most parallel programming
Obviously needed for threading (on modern systems)

- Replicate the same executable on distributed ones

In theory, not essential – but, in practice ...

- Implies that all systems must be near–identical

Versions + configuration of system + environment

Programming Environments

Reminder:

These are a combination of hardware and software
E.g. a cluster with MPI, a multi–core CPU with
OpenMP, an NVIDIA GPU with CUDA

- Course is in terms of programming model

How you design and program your parallelism
Will cover most of those used in scientific applications

Programming Patterns (1)

Related to this course's term programming model
Increasingly common in books and Web pages

- Mainly a conventional design for parallel coding

Traditionally called models or methodologies

- General parallelism is too complicated to use

Debugging is very hard and tuning worse
So the solution is to use a constrained subset

E.g. my OpenMP course teaches a SIMD model
Simplest way to parallelise large matrix operations

Programming Patterns (2)

- Use an established pattern – don't just code
Success rate of people doing that is very low indeed

- Patterns don't solve the most serious problems
But reduce opportunities for many common mistakes

- Watch out for pundits pushing dogmas to excess
Remember the Kipling quotation?

- Try to match your actual problem
But your available systems may constrain you

Embarassingly Parallel (1)

Can use a very simple ‘pattern’ for this

- Often, threads are used like processes

Used in threaded services like Web servers
In Java, POSIX and Microsoft etc.

- Most data are read–only or thread–local

Shared updatable data are treated specially

- All data sharing is explicitly synchronised

Don’t rely on ‘happens before’ or similar
Typically using locks or critical sections

Embarassingly Parallel (2)

- No assumptions made about data consistency
If in doubt, enforced using explicit mechanisms
Full fences or even barriers (see glossary)

- For threading, library use is synchronised
Often done only in serial mode, and barriered
Critical for signals and such nasties

- Beyond that, is task for real experts only
Morass of conflicting, misleading specifications
With more gotchas than you believe possible

Status Report

Have covered extra information and ‘patterns’

Now onto a description of SIMD patterns

SIMD Designs (1)

SIMD means Single Instruction, Multiple Data
I.e. a serial program runs with parallel data

Think of a vector system when you say this
E.g. A = B + exp(C), where A, B and C are vectors

- Oldest parallelism model and about the simplest
Probably most heavily used in scientific computing

⇒ This course uses the term loosely
Includes things like FFTs and sorting

SIMD Designs (2)

Can often code and debug just like serial
Optimisation well–understood and may be automatic
Can often compare results in serial and parallel

Includes MMX / SSE / AVX / VMX / Altivec

- But regard them as part of serial optimisation

Not covered further in this course

Data are usually distributed across CPU cores
Think of each core as owning a subset of the data
Problems when one core needs another's data

SIMD Designs (3)

NVIDIA GPUs also use this model
And you can use clusters of systems this way, too

Correctness needs remote accesses synchronised
Too much remote access harms performance

- This aspect is what you need to concentrate on

Details of both very dependent on interface used

- Applies on shared–memory as much as on others

Don’t believe Web pages that say that it doesn’t

Vector/Matrix Model (1)

Best studied and understood of SIMD models
Very close to the mathematics of many areas

- The basis of Matlab, Fortran 90 etc.

Operations like mat1 = mat2 + mat3*mat4

Assumes vectors and matrices are very large

- A good basis for SMP autoparallelisation

I.e. where the compiler does it for you

Often highly parallelisable – I have seen 99.5%

- Main problem arises with access to memory

Vector/Matrix Model (2)

Vector hardware had massive bandwidth

- All locations were equally accessible

Not the case with modern cache–based, SMP CPUs

- Memory has affinity to a particular CPU

Only local accesses are fast, and conflict is bad

- Many good algorithms or even tuned software

E.g. for matrix multiply or transpose
Complete pivoting and similar are the problems

SIMD Tuning

- Regard tuning as ALL about memory access

Aim to minimise access to data on other CPU cores
Problem is tricky, but well understood

Note that minimising access has several aspects:
amount transferred, number of transfers and
waiting for data and conflict

You can often get very large speedups quite easily
E.g. by keeping both matrix and $\text{matrix}^{\text{Transpose}}$
Using the one that is better for memory access

SIMD Systems – OpenMP (1)

Probably the easiest SIMD environment to use
All it needs is a multi–core CPU or SMP system
Most desktops and servers, plus Intel Xeon Phi

OpenMP is extended language for Fortran and C/C++
Available in most compilers, including gfortran/gcc
Several similar environments – e.g. CilkPlus

Shared memory means data ownership is dynamic
Don't need to bind data to cores, but still important

SIMD Systems – OpenMP (2)

Debugging and tuning are not easy
Don't believe Web pages and books that say that it is

- Most shared–memory bugs don't show up in tests
Usually only on real data, after hours of running

- Usually get wrong answers, not crashes etc.
Intel has some tools that may help

- Tuning is about memory conflict, which is tricky
Shared–memory tuning is hard even for experts

SIMD Systems – OpenMP (3)

- Design program to be correct and efficient

SIMD is usually simple and regular, which helps

- Develop, test and tune components separately

Can usually do this in otherwise serial programs

- With discipline, it's often not too hard

There is a course on doing just that for OpenMP

OpenMP/

SIMD Systems – CilkPlus

CilkPlus is a C++ extension, by Intel
It has a Fortran–like array subset notation
There is a gcc extension for most of it

- Unfortunately, works only on fixed–size arrays

And (at present), it generates only AVX/SSE code

Cleaner/simpler than OpenMP, and has potential
This aspect not quite ready for use, unfortunately

Main functionality is tasking (see later)

SIMD – Using Raw Threads

PLEASE DON'T

That is like writing your whole program in assembler
It is much harder even for the best experts
Applies even when using toolkits like Intel TBB

- For SIMD designs, use a SIMD environment
- For others, use another high–level environment

SIMD – Using MPI etc. (1)

- Can use clusters and very large problems
Main reason people do it instead of using OpenMP
Now always MPI, but perhaps Fortran coarrays

Problem is need to transfer data between processes
Solution is to use regularity of SIMD designs
Design data transfer and program logic carefully

- A bit harder than threads – but debuggable!
If data transfers wrong, results are usually wrong

SIMD – Using MPI etc. (2)

- Don't just convert a shared–memory program

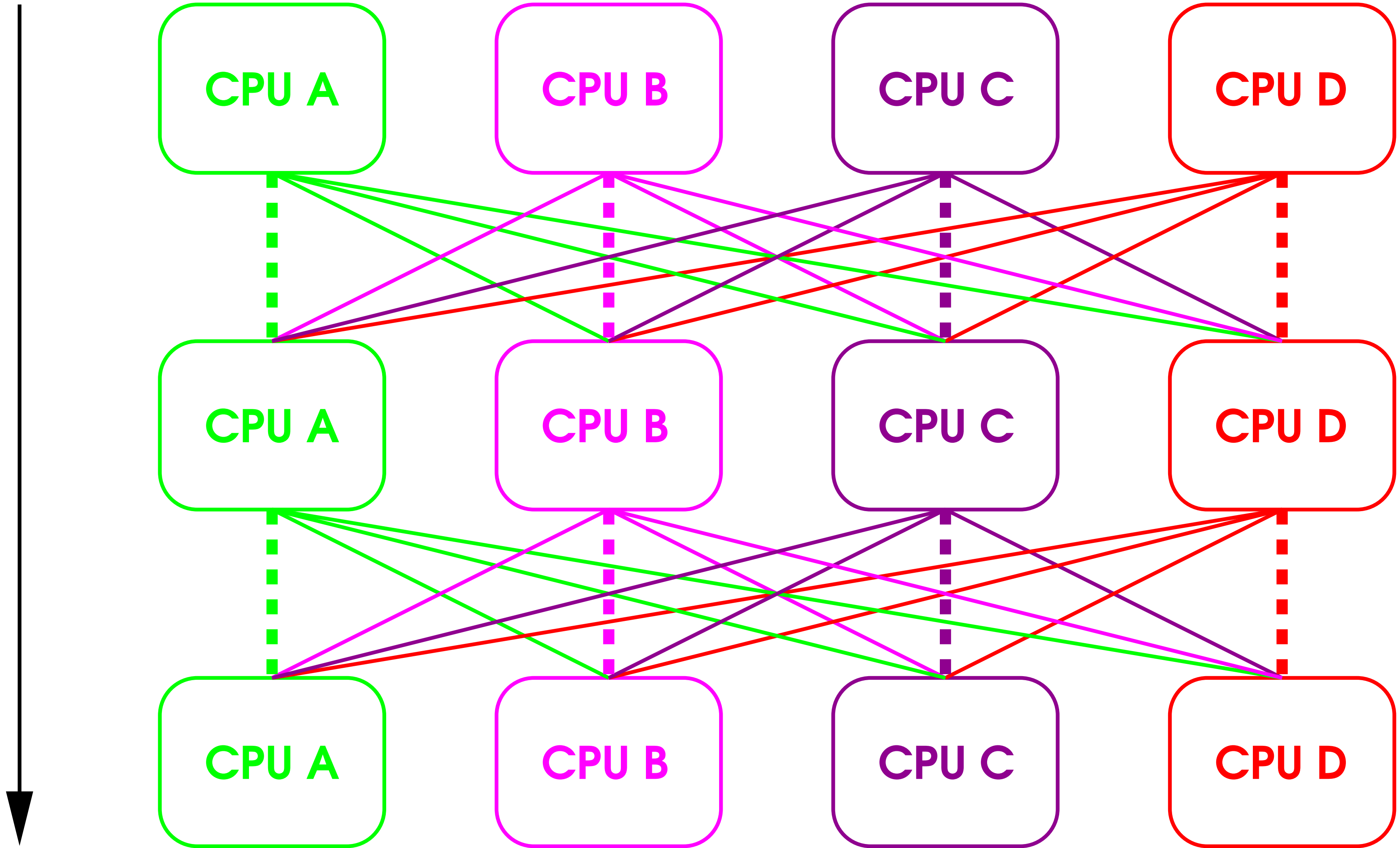
That usually gives poor performance, at best

- Find how other programs solve similar problems

MPI is the most common form of HPC coding
Not trivial but has been solved many times

E.g. many approaches use time step designs
Alternate computation and communication phases
Not the only approach, but often an efficient one

Time–Step Design



SIMD – Using MPI etc. (3)

The MPI course doesn't teach using SIMD, as such
But it covers all of the MPI features you need

MPI/

SIMD Systems – GPUs

Extended GPUs to use for HPC
Will describe current leader (NVIDIA Tesla)

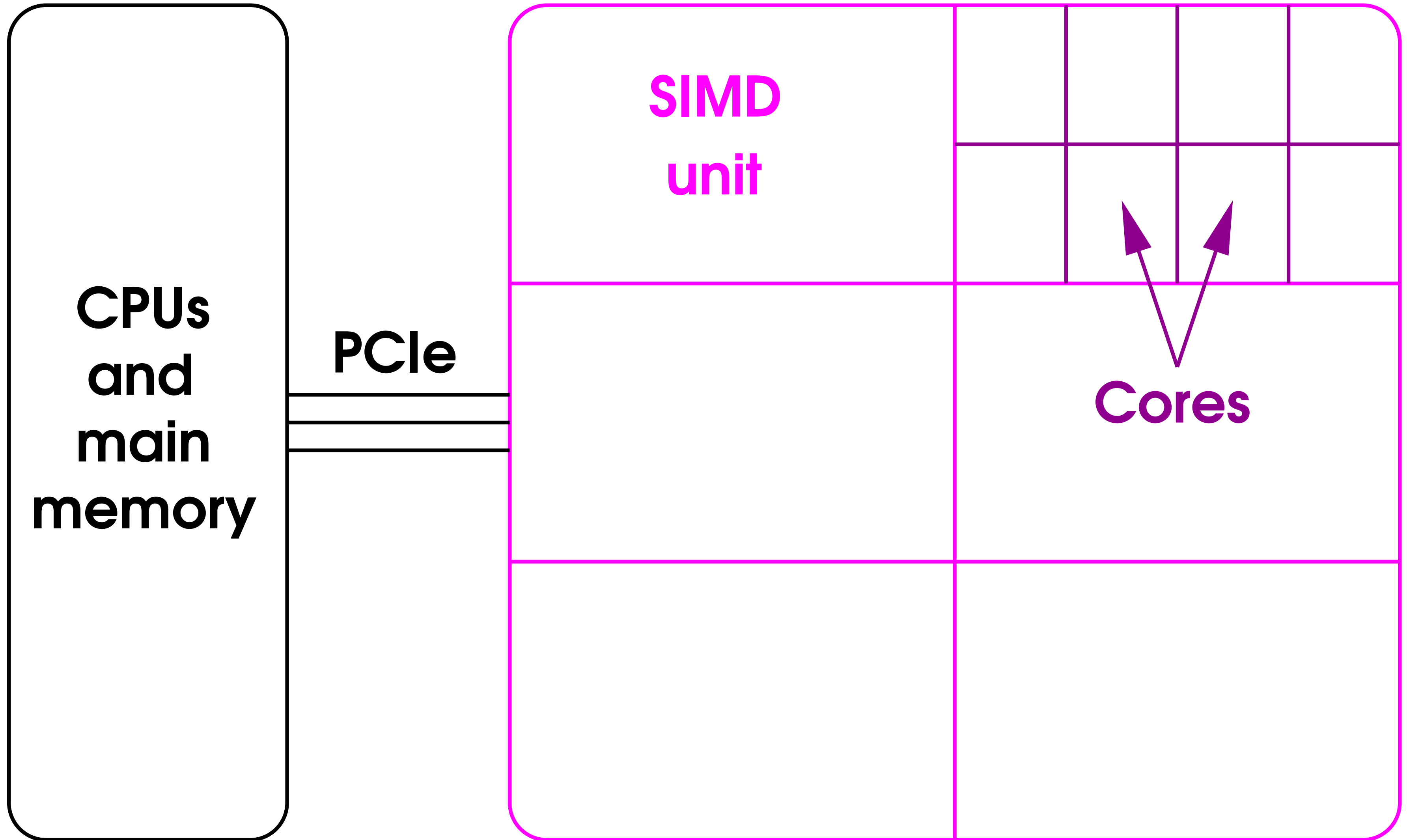
Hundreds of cores, usable in SPMD fashion
Cores are grouped into SIMD sections
Expensive to synchronise and share data

Can be 50–100 times as fast as CPUs

- Only for some applications, after tuning

And double precision is often 2–20$\times$ slower

NVIDIA GPU Design



CUDA, OpenCL, OpenAcc

CUDA, an extended C99/C++, is NVIDIA only
OpenCL more portable, less commonly used

OpenAcc is now in OpenMP 4.0, just out
Not yet in most compilers, and unreliable when it is
Not investigated for usability and implementability

- Almost all Cambridge GPU programs use CUDA

Programming said to be fairly tricky

GPU Use (1)

- Rules for sharing memory are trickiest part
Complicated and absolutely critical to get right
NVIDA cuda−memcheck may help here

Problem is fitting program into restrictive GPU model
Anywhere from easy to effectively impossible

- Tuning is where the worst problems arise
Critically dependent on details of application

GPU Use (2)

- Don't forget CPU⇔GPU transfer time

Can often run some serial code on the GPU
Runs very slowly, but may eliminate transfers

A course is part of MPhil in Scientific Computing

GPU Precision Issues

Graphics is numerically very undemanding
Double precision is often very much slower
- But most scientific codes critically need it!

Watch out!

Some precision–extension techniques can help
- Dating from the 1950s to 1970s, some newer

Most now used by the GPU–using community

It's tricky but many problems are solved
- But don't just program and hope!

Status Report

Have covered a description of SIMD patterns

Now onto a description of tasking patterns

Tasking (1)

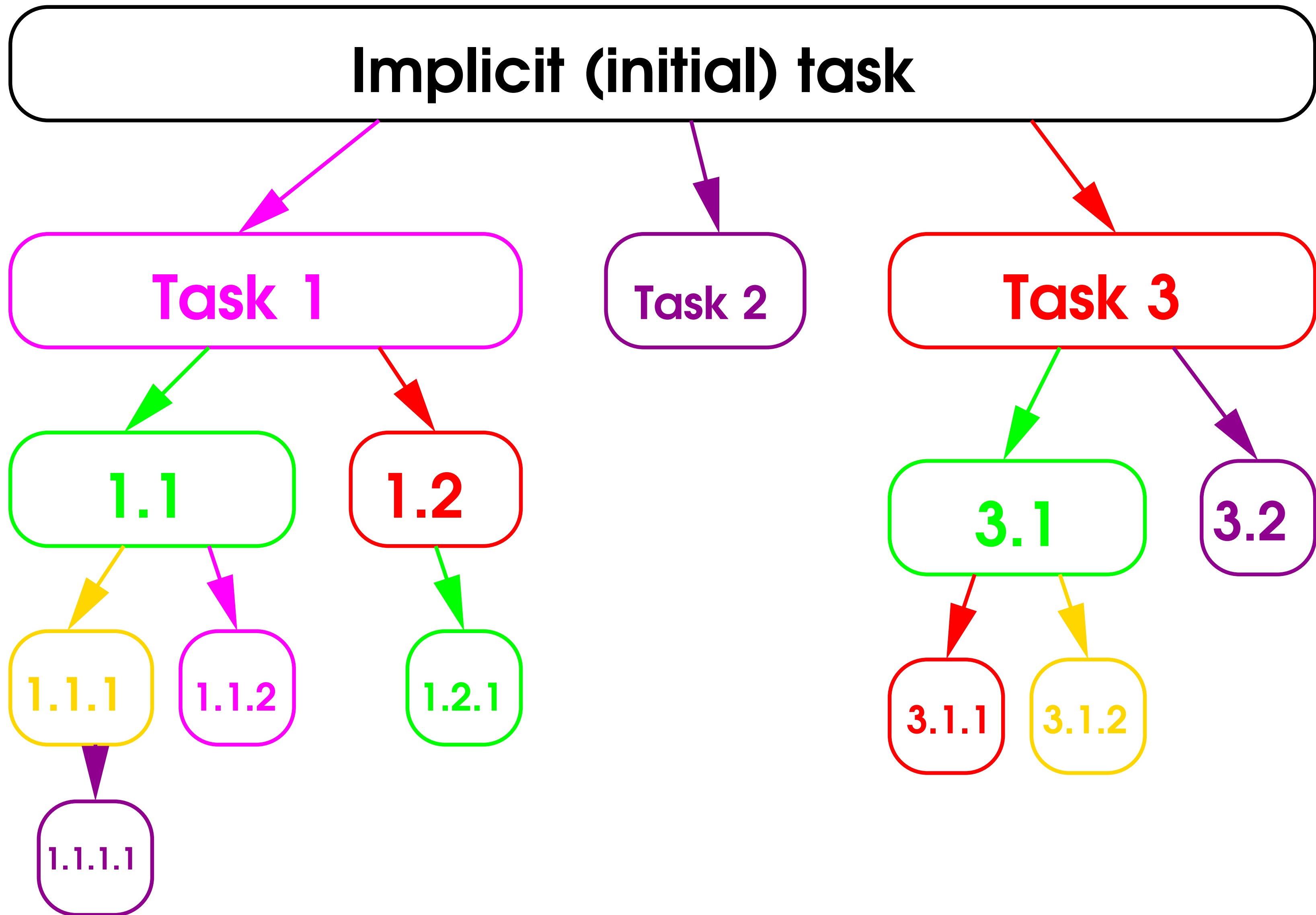
- Next cleanest pattern is probably tasking
Procedure calls run asynchronously on own threads
Useful for irregular problems; tuning can be tricky

- Can still have subtasks, creating a hierarchy
Also tasks fit well with dataflow designs (see later)

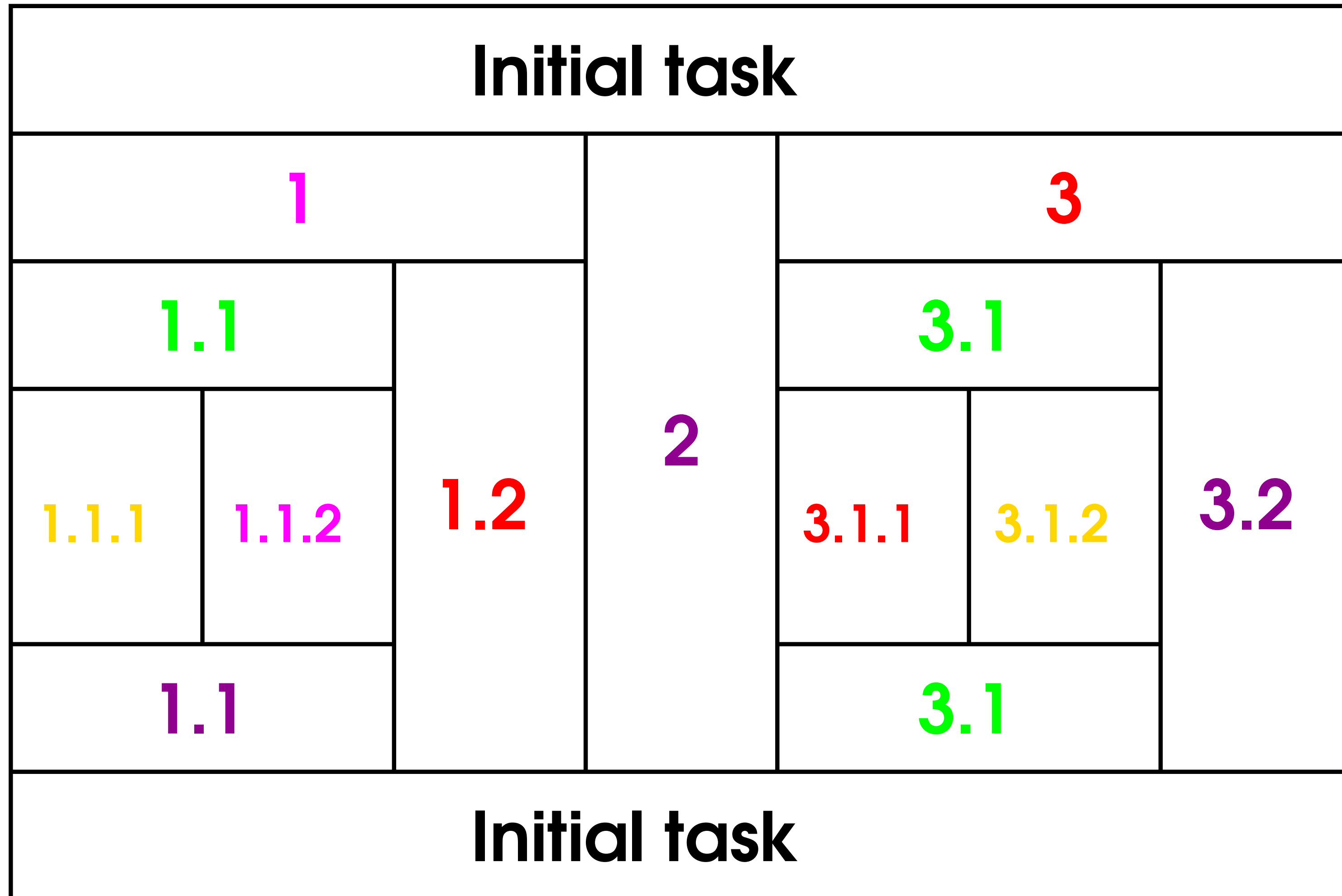
Most common with shared memory mechanisms
But can be used with distributed memory, too

A bit like programming with background processes

Hierarchical Trees



Task Execution



Tasking (2)

- Fairly easy to use if tasks entirely separate

Design program to keep them as separate as possible

- Easiest to achieve if task procedures are pure

I.e. only updates outside task are via arguments
Updatable arguments must not be aliased, of course

- Any global data used does not change during task

Such data should be read–only by all tasks

A way of viewing this is that tasks should be atomic
Often called transactions – see later

Tasking (3)

- Synchronisation between tasks is not advised

Very easy to cause deadlock or livelock
FAR better to split them into separate tasks

Need to design the task structure very carefully
⇒ Designing by using a dataflow model may help

There is one lecture on using tasks in:

OpenMP/

Tasking (4)

Can use tasking with any threading environment
$\Rightarrow$ But the real problems are data races etc.

Usual ones are OpenMP (and perhaps CilkPlus)

MPI is fine if tasks justify data transfer

Tricky to use on GPUs (for complicated reasons)

Dataflow (1)

- Useful when designing your program structure

Very useful for irregular problems and tasking

I failed with some complicated threading designs
I then designed using dataflow, and got them going

- If you don't find it natural, don't use it

Fits best with tasks (using any environment)
Not suitable for GPUs (for complicated reasons)

Dataflow (2)

- Currently no mainstream dataflow systems

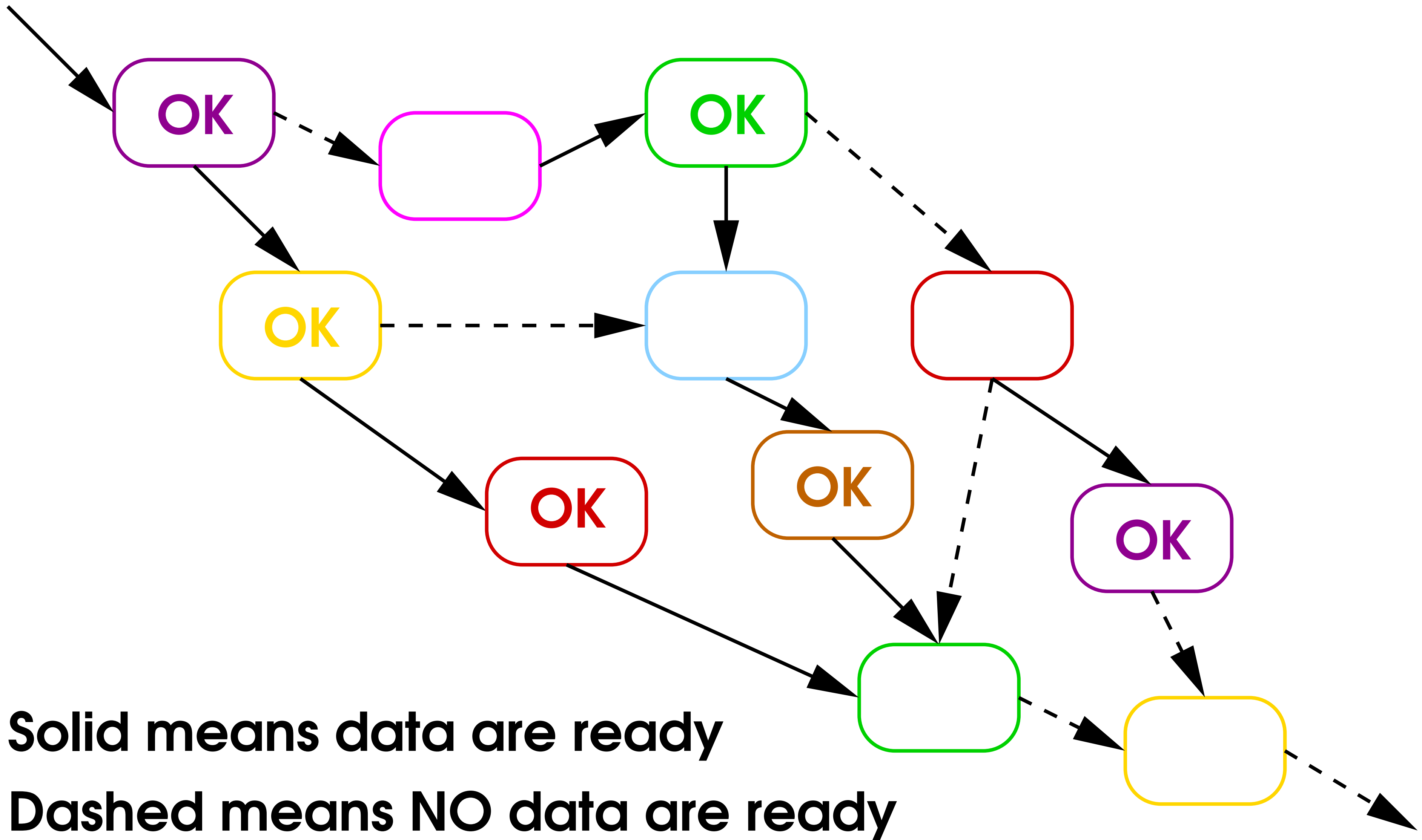
Sadly neglected, in programming languages
Only recent language of importance is Prolog

Structure made up of actions on units of data
Design how actions transfer their data packets
Usually use a DAG, but cyclic structures are possible

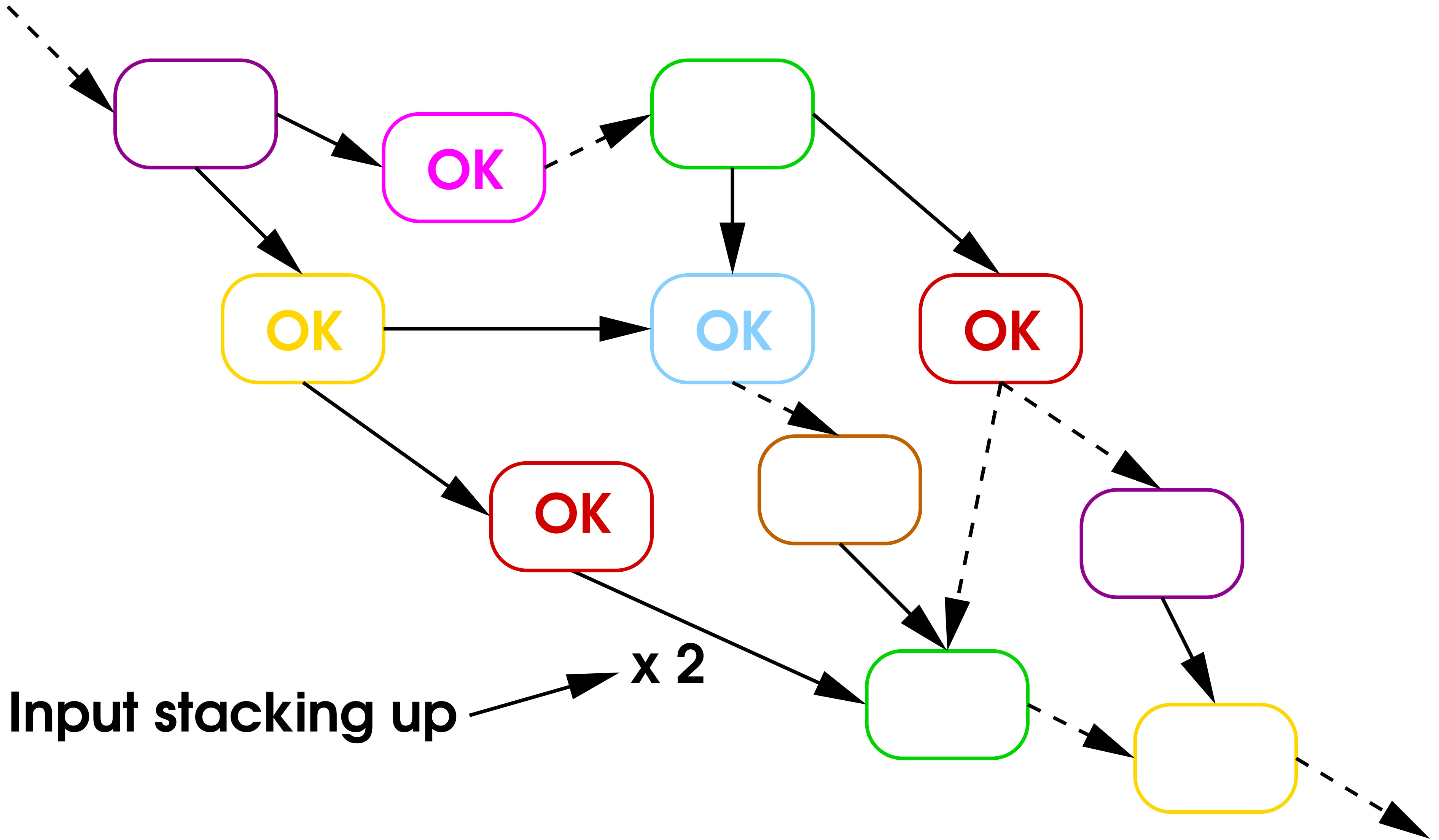
- Correctness of program depends only on structure

Order of execution affects only performance

Dataflow (Step N)



Dataflow (Step N+1)



Dataflow (3)

Each ‘data packet’ is stored in some queue
And is associated with the action it is for

Queues usually held in files for MPI
Queues usually held in memory for OpenMP

The program chooses the next action to run
The priority does matter for efficiency
But it is separate from correct operation

This is a gross over–simplification, of course

Transactions

Not a model, but a very important technique
Just a compound action packaged to be 'atomic'

- Makes it much easier to avoid data races

Described the form for duplex communication earlier

But technique is equally useful for tasking
I.e. task procedure written to be atomic

Transactions for Tasking

- A transaction must include all data accesses

Only exception is for globally read–only data
Yes, a read–only transaction can often be needed

- Generally use some form of locking

By far the easiest to get working correctly
Watch out if using multiple locks!

Can be implemented in other ways, not covered here
Retry on conflict is common, but very tricky

Other Patterns

Those are the two most common approaches

Each has more variations than I have described

And there are lots of less common patterns

Status Report

Have covered a description of tasking patterns

Now onto current parallel environments

GPUs already covered under SIMD patterns

Parallel Environments

Will now describe classes of parallel environments
Not patterns, but the underlying mechanisms

Start with distributed memory – currently MPI
Plus a zillion such environments in commerce

Then onto shared memory – very trendy
This is where most of the complexity is

Distributed Memory

Often incorrectly called MIMD (see glossary)
Each process runs like a separate serial program

- The communication is always by message passing

But, in theory, I/O and other methods can be used

- Think of serial programs communicating by Email

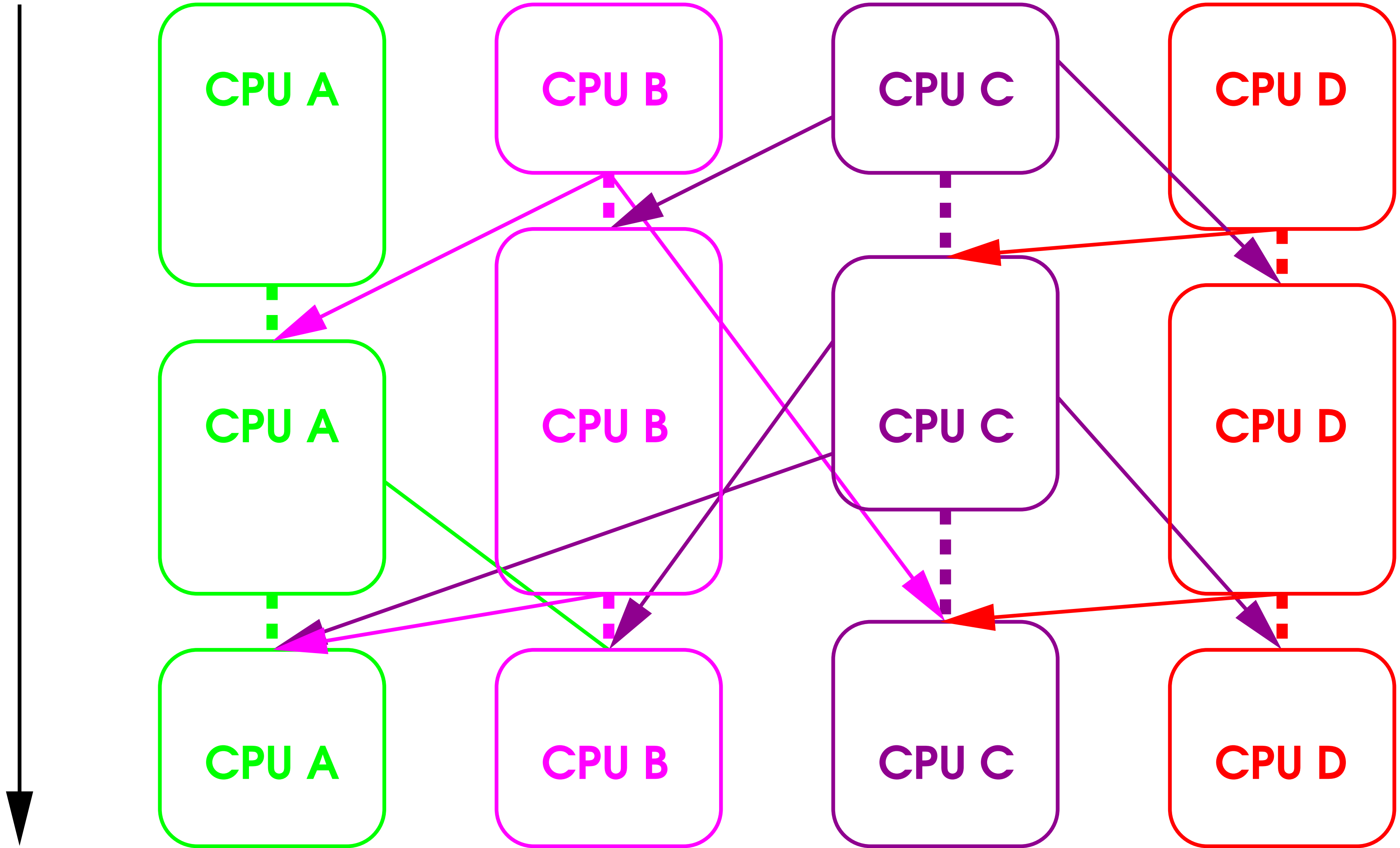
Or, if you prefer, process–to–process I/O transfers

Many interfaces used in commercial applications

- But MPI dominates in scientific computing

Stands for Message Passing Interface

Message Passing



MPI (1)

MPI is a library callable from Fortran and C
Hence C++ and anything with a C interface

- Same program runs on multi–core CPUs, clusters and supercomputers, without any changes

- With care, it is reasonably efficient on all of them
And it scales to many thousands of parallel processes

- Always worth considering, even for desktops

MPI (2)

MPI is 20 years old now, still going strong
The basis of all distributed memory libraries

- Only guaranteed safe approach for the long–term

MPI is portable over both systems and time

It may be superseded for applications programming
But will be used to implement any such replacement

- Don’t underestimate the importance of stability

MPI (3)

- Biggest difficulty is managing distributed data
Must handle all inter–process transfers yourself

- Advantage is that even that problem is explicit
Means that debugging and tuning are much easier

Not covered in detail here – see the course:

MPI/

Look at Vampir for tuning – feedback is positive

PGAS (1)

Stands for Partitioned Global Address Space
or Partitioned Global Array Storage

Each thread/process has mostly separate data
- Special arrays are partitioned across those

Access remote data with special syntax
- Strict rules for synchronisation of updated data

Being pushed by USA DoD – Department of Defense
ASCI – Accelerated Supercomputer Initiative
Little evidence of much use in real scientific research

PGAS (2)

Said to be easier to use than MPI by its fans
$\Rightarrow$ No evidence and it is a dubious claim

- Fortran 2008 has standardised coarrays

This is described in more detail shortly

- UPC – Unified Parallel C – NOT recommended

Specification is very bad, and usability dubious
Almost all use by USA CS depts with UPC grants
No mainstream compilers after 14 years

- Several experimental languages (e.g. IBM X10)

Using PGAS

- Much easier to access remote data than in MPI
You still have to use special syntax to do it

- But implicit transfers introduce data races
Covered later under shared–memory threading

In much of MPI, data races simply cannot occur
You can still get the transfer logic wrong, of course
Mistakes will always show up as wrong answers

⇒ PGAS may take off, but don't hold your breath

Fortran Coarrays (1)

Reasonably well–designed and unambiguous

Cray, IBM and Intel compilers support them
gfortran will, but probably not very soon
Intel and gfortran implemented using MPI

Main scientific programming interest is in Germany
E.g. HLRS, Stuttgart – a Cray site
But there is some on the UK (e.g. EPCC, Edinburgh)

http://www.hector.ac.uk/cse/training/coarray/

Fortran Coarrays (2)

- Feedback on the Intel compilers welcomed

Also anything on definite interest in Cambridge

I am closely involved in their design, but cynical

But they are a very plausible environment

Status Report

Have covered current parallel environments

Now introduction to shared memory environments

Shared Memory (1)

All threads/processes have access to all memory

- Unfortunately, that isn't exactly right ...

There are three common classes of shared memory

- Shared memory segments, POSIX mmap etc.

Shared between processes on same system

Can be useful, especially when memory is limit
E.g. read a large file, and keep only one copy of it
Just a useful technique – not covered here

Shared Memory (2)

- Virtual shared memory, of various forms

The environment provides a shared memory interface
But runs on a distributed memory system

PGAS can be regarded as a restricted form of this

- Avoid its general form like the plague

Algorithms often need to match the memory model
$\Rightarrow$ The efficiency can be dire – often hopelessly so

Shared Memory (3)

- Separate threads with almost all memory shared

By FAR the hardest model to use correctly
Unfortunately, look as if it is the simplest
Touted as being easy all over the Web and more

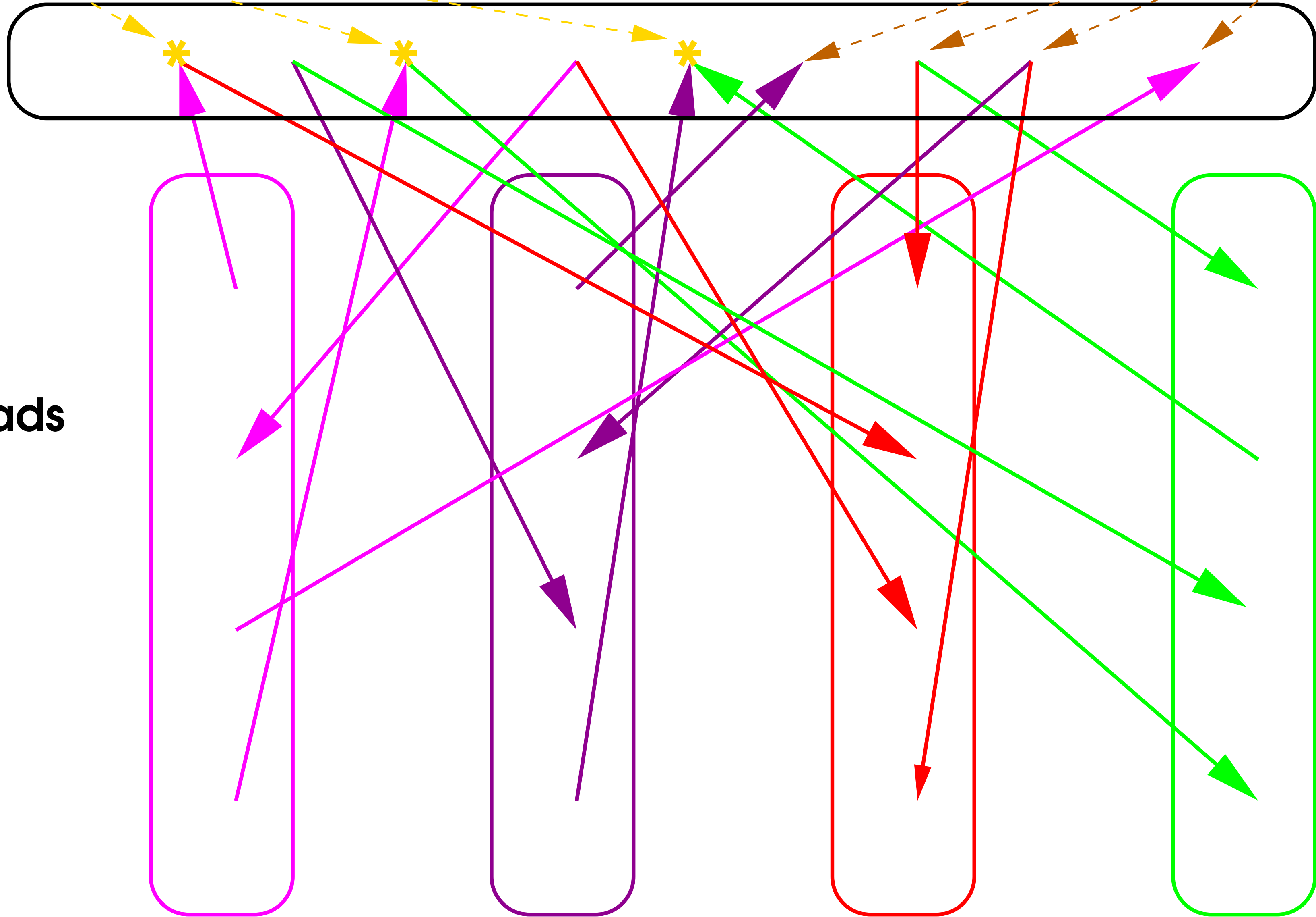
- Correctness needs all accesses synchronised

Key to success is strict discipline of data access

Errors are called data races or race conditions

- Cause of almost all hard–to–find problems

Shared–Memory Threading



Benefits of Threading

- When 90% of time is spent in 10% of code
Often need to parallelise only the 10%
Can also add often add parallelisation incrementally

Not needing to distribute data is minor gain
Data separation is key to performance
Updating interleaved data can run like a drain

A Simple Data Race

Thread A	Thread B
X = 1.23	X = 4.56

Synchronise

print X

Quite likely to print any of 1.23, 4.56,
1.23000015259 or 4.55999938965
Same would apply if it printed in any other thread

- And subtly wrong answers are NOT nice!

Another Data Race

Thread A		Thread B
		X = 1.23
	Synchronise	
print X		X = 4.56

- Even that can do exactly the same!

Any of 1.23, 4.56, 1.23000015259 or 4.55999938965

Data Races

Such corruption is common for compound types
E.g. complex numbers, classes, array assignment
A data race in I/O to a file is likely to corrupt it

Multiple unsynchronised accesses must be

- To completely separate locations
- Without exception, for reading only
- Without exception, only from one thread

- Object of discipline is to ensure that is so

Easiest with SIMD, but other models are used, too

Debugging (1)

Frequency of failure is $O(N^K)$ for $K \geq 2$ (often 3 or 4)
N is the rate of cross–thread accesses
The number of threads is also relevant, non–linearly

- And the word probability is critical here

Almost all data races are not reliably repeatable

- Finding them by debugging is very hard

Diagnostics add delay, and bugs often hide
Often same with debugging options or a debugger

Debugging (2)

Many programs 'work' because N is small
The MTBF is often measured in weeks or years
Who notices if a Web service fails 0.1% of the time?

In HPC, there are many more data accesses
The MTBF is often measured in hours or days
Complete analyses often take days or weeks

- Solution is to avoid data races while coding

Not easy, but not impossible with discipline

Specialisations

- Almost everybody uses some specialisation

Simplifies thread creation and scheduling
Only rarely helps to avoid data races

- OpenMP and CilkPlus have SIMD and tasking

Easiest to use – covered previously, not repeated
Provides a little help avoiding data races

- Most common is perhaps a fixed set of threads

Very like a PGAS model for shared memory
Similar data race problems and solutions for both

Epilog

Have now covered a more–or–less complete overview
Except for the shared memory area

Next lecture is on shared memory programming
More details on options, including ones not mentioned

Includes issues that you really need to know
If you are likely to use it, advised to attend

- If you might not attend, please:
  Fill in and hand in the green form, today