Software Design and Development

Languages and Parallelism

Nick Maclaren

nmm1@cam.ac.uk

October 2018

Summary

Many issues are language– or parallelism–specific
This includes a rough overview of the main ones
Mainly information that is not commonly provided

Some are not taught in MPhil – for background
You may need to use them in your later career

But there is one critical rule to follow:

- Agree choices together with your supervisor

Your chosen project may have constraints
There may also be restrictions imposed by examiners

My References (1)

Courses designed for use independently
Lectures, practicals, worked examples, and more

https://www-internal.lsc.phy.cam.ac.uk/nmm1/Fortran/
https://www-internal.lsc.phy.cam.ac.uk/nmm1/C++/

https://www-internal.lsc.phy.cam.ac.uk/nmm1/MPI/

https://www-internal.lsc.phy.cam.ac.uk/nmm1/OpenMP/

C++ and MPI

You are being taught C++ and MPI
The comments do NOT refer to those courses
They are based on other experiences

I used to give the MPI course, but no longer do

We all agree (roughly) on how to use those
Please tell me of any discrepancies :–)

My References (2)

Lecture–only courses on background and principles
Including information that is very rarely described
First one is this course, and includes much of second

https://www-internal.lsc.phy.cam.ac.uk/nmm1/...
.../Development/
.../Arithmetic/
.../Parallel/
.../MultiApplics/
.../OldFortran/
.../MixedLang/

And some more, of less relevance

Choice of Language

The following are the two main relevant languages:

C++: very flexible, but very poor checking
Errors are easy to make and foul to locate
Also compilers can't optimise it very much

- However, it is dominant in many areas

Fortran: advised to use modern language
Much more powerful than Fortran 77
Fully upwards compatible, so old code still works
Much better checking and optimisability

Language Versions

Follow a standard: probably C++11 (2011 version)
Most portability, and compilers should be tested
Even with that, fancy features may be unreliable
    E.g. advanced templates and exceptions
- And don't use its threading –see later for why

Fortran 2008 (actually 2011, too) probably best
It includes coarrays (a PGAS parallel model)
Both gfortran and Intel support them
Last heard, needed special versions of both
Ask me offline if you want to know more

Auxiliary Languages (1)

C: a high–level assembler – treat it as such
Use it for interfaces, including system calls

Matlab/octave: use for quick test codes
Can also use them to write prototype programs
Very often used to prototype Fortran codes

Python/numpy is often used similarly
Probably fits better with C++ than Fortran

For a comparison of most of the above, see:
https://www–internal.lsc.phy.cam.ac.uk/nmm1/WhyFortran/

Auxiliary Languages (2)

Mathematica/Perl: only if you know them well
Harder to use equivalents of Matlab and Python

Some projects will have their own language variants
Often using preprocessors or C++ templates

And hundreds of others exist!

- And Python is an excellent scripting language!

Use it for data munging, process control etc.
You are strongly advised to learn at least one

C++ Problems

Main problems:

- C++ is a huge and complicated language
- Books etc. rarely cover scientific computing needs
  And some things (like N−D arrays) are very tricky
- C (and hence C++) has lots of evil gotchas
  Usually glossed over, but often cause trouble

I used c. 100 programming languages before C++
I was astounded at its complications and gotchas
They don’t make it correspondingly powerful

C++ References

Stroustrup, Bjarne (2008). Programming: principles and practice using C++. (1100 pages)

Very relevant and thorough, but hard
From scratch, 14 weeks at 15 hours per week!
I taught a course using it as a basis

Programming in Modern C++
https://www-internal.lsc.phy.cam.ac.uk/nmm1/C++/

Most especially 21a_Lib_issues.odp
and 24a_more_numerics.odp
Important, hard-to-obtain, information for scientists

C++ and Parallelism

Above all don't try to be clever – KISS

Other problem is compiler generating implicit calls to
copy constructors and assignment
Just like Fortran, but more pervasive

Most (simple) uses of MPI are no problem; see
https://www–internal.lsc.phy.cam.ac.uk/nmm1/MPI/

Especially lectures More on Point–to–Point
Miscellaneous Guidelines and
(if used) One–sided Communication

C++ and Threading (1)

Mere mortals should not try to use C++ threading
Gotchas abound for even encapsulated methods

Worst issue: container library not well–defined
Applies to OpenMP, and all forms of threading
and all forms of asynchronism

Some safe but restrictive empirical rules
For some guidelines, see Critical Guidelines in:
https://www–internal.lsc.phy.cam.ac.uk/nmm1/OpenMP/
Same rules apply to all forms of threading

C++ and Threading (2)

Beyond that, it is safer to write your own classes

- But even that is definitely not easy

Unless you can find a suitable class library

Much easier for OpenMP than any other threading

Ask me offline if you want to know why

Other recommendations are covered later

Fortran References

Look at the course:

Introduction to Modern Fortran
https://www-internal.lsc.phy.cam.ac.uk/nmm1/Fortran/

The first lecture gives several recommended books

Can learn it for 20% of effort as C++

Lots of books on Fortran 77 – which is not advised

- And many of them are VERY bad indeed

Fortran and Parallelism

Generally, not a problem, except for one aspect
Optimises best for OpenMP, SSE, VMX, Altivec etc.
Fortran is the language of choice for SIMD

But Fortran allows/requires implicit data copying
Essentially like C++ copy constructors etc.
Fortran 2003 (and MPI 3) handles that right

- Unfortunately, it's not yet generally available

See lectures 08 and 09
https://www–internal.lsc.phy.cam.ac.uk/nmm1/MPI/

Parallel Languages

Several designs extend languages for parallelism
GPU interfaces – CUDA, OpenCL and OpenAcc
OpenMP (shared–memory) – C, C++ and Fortran

Dozens of specialist parallel languages around
Few have any impact outside computer science

C++ and Fortran already mentioned

- You are recommended NOT to use UPC

Mixing Languages

There are only two relatively easy cases:

- Calling C and simple C libraries

Pretty well anything can do that, in some way

- Calling Fortran 77 from C/C++

E.g. LAPACK – can still use a Fortran 95 compiler

C++ and Fortran 95 in one program can be tricky

If you need to do that, use separate processes
https://www–internal.lsc.phy.cam.ac.uk/nmm1/MultiApplics/

Better Approach

If you need to do that, use separate processes
Can still build them into a single application

Beyond the scope of this course, but

https://www-internal.lsc.phy.cam.ac.uk/nmm1/MultiApplics/

Processes can still share memory on SMP
Use POSIX mmap or some form of shmem
Remember that explicit synchronisation is needed

Relevant Libraries

MPI interfaces – OpenMPI and MPICH
Intel and most HPC vendors have their own

NAG is best general, portable numerical library
LAPACK is open source linear algebra
FFTW is open source fast Fourier transforms
MKL and ACML are Intel's and AMD's math. libs
And lots and lots more, proprietary and open source

- Do NOT trust Numerical Recipes or the Web

www.netlib.org is often reliable, but not always

Unsuitable Libraries

A few libraries should not be included
More detail in my MPI and OpenMP courses

Mainly ones with fancy use of system facilities

- May be incompatible with MPI, OpenMP at least

- Avoid anything using the X Windowing System

The event handling may well interfere badly

If you need to, use separate processes
Just as when mixing C++ and Fortran 95

Algorithm References

Data management well covered in computer science
Cormen, T.H. et al. Introduction to Algorithms
Knuth,D.E. The Art Of Computer Programming
Also Sedgewick, Ralston, Aho et al. etc.

Most good, general numerical ones are very old
Best approach is to use NAG as reference
http://www.nag.co.uk/numeric/FL/...
.../FLdocumentation.asp
For specialist algorithms, seek expert in that field

(Not-)Moore's Law

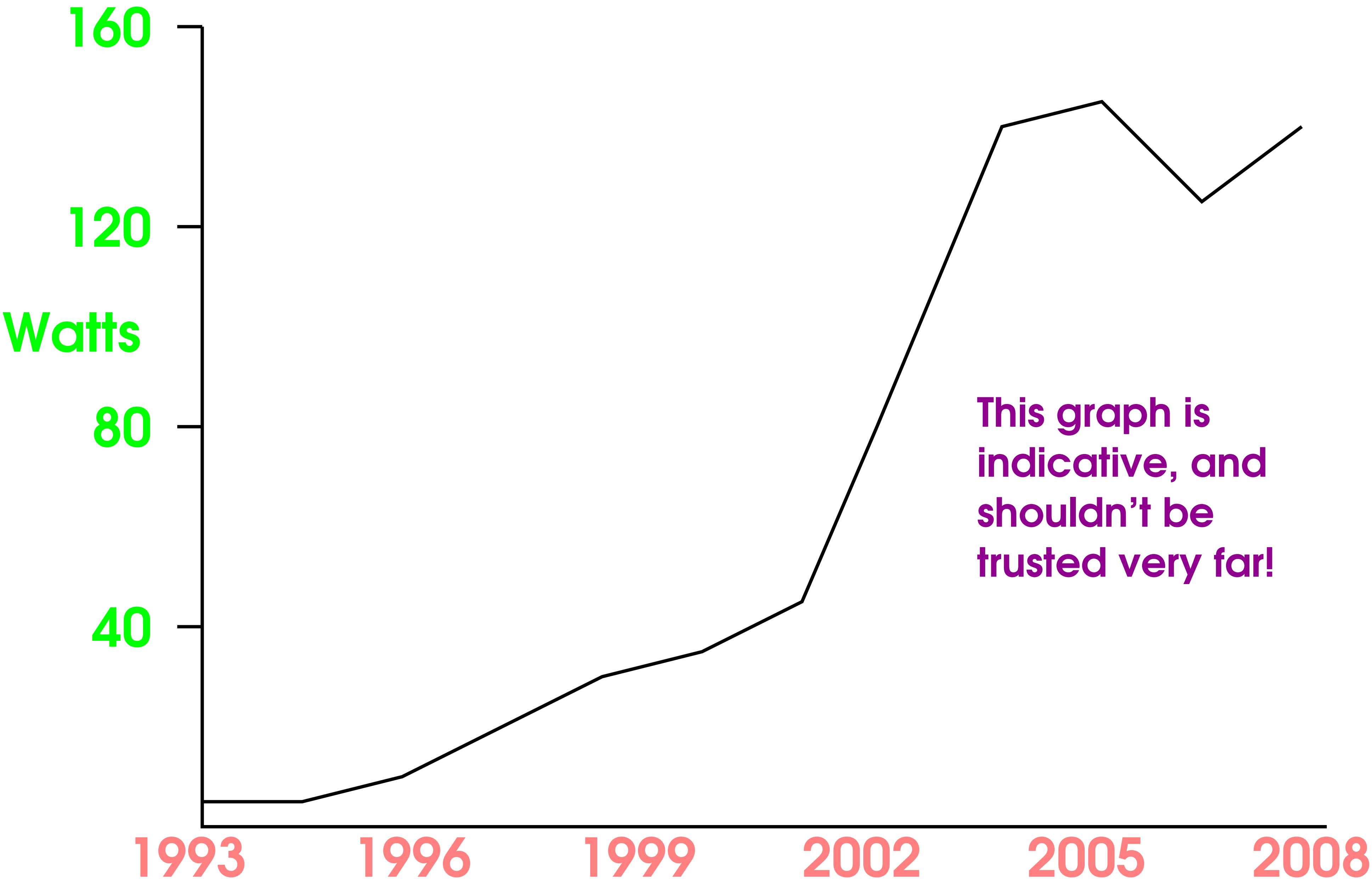
Moore's Law is chip size goes up at 40% per annum
Not–Moore's Law is that clock rates do, too

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

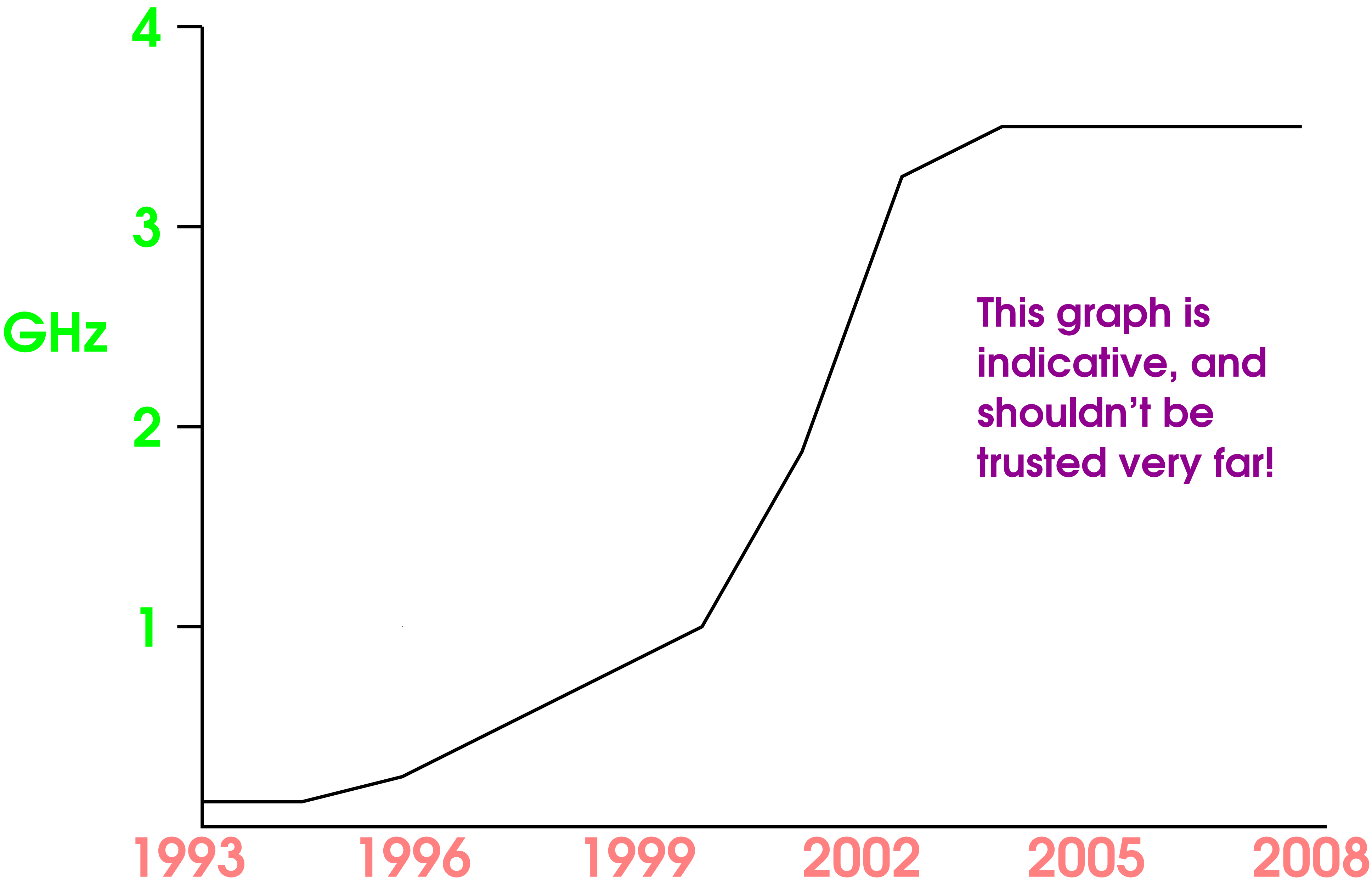
Not–Moore's held until ≈2003, then broke down
Clock rates are the same speed now as then

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

Toolkits

Usually libraries but sometimes preprocessors
Almost all are field– or model–specific
Vary from good to utterly ghastly, as usual

Most are shared–memory but some based on MPI

If a good one matches your requirement, use it
Not investigated and not covered in this course

Parallelism Books

There are a lot of fairly good books around
Including a few of the computer science textbooks
Most describe a few approaches as the only ones

There are nine and sixty ways of constructing
tribal lays,
And every single one of them is right!

From "In the Neolithic Age" by Rudyard Kipling

Note that it is frequently misquoted on the Web

- Don't trust the Web on parallelism, either

More Information (1)

This part is taken from a much longer course
It is still relevant, and goes into much more detail
https://www–internal.lsc.phy.cam.ac.uk/nmm1/
Parallel

Shared–memory people (not just Java) should look at

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

And, mainly for OpenMP but more general
https://www–internal.lsc.phy.cam.ac.uk/nmm1/...
...OpenMP/paper_7.pdf

More Information (2)

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

Multi-Process Parallelism

Applications are often made up of multiple processes
Can be run in parallel without programming

https://www–internal.lsc.phy.cam.ac.uk/nmm1/MultiApplics/
And lecture 1 of
https://www–internal.lsc.phy.cam.ac.uk/nmm1/Parallel/

Not covered further in this course

Types of Parallelism (1)

Hundreds of these, some purely theoretical
Only a few are relevant to this MPhil

Message passing (currently mainly MPI)
Main form for distributed memory (i.e. clusters)
But also works well on multi–core systems

Small vector units (currently mainly SSE)
Pure vector supercomputers are essentially dead

Attached SIMD units (currently mainly GPUs)
That is Single Instruction, Multiple Data

Types of Parallelism (2)

Shared memory threading (currently mainly OpenMP)
Latest C++ standard has some, but very low–level
Includes POSIX/Microsoft/Java threads
CilkPlus also belongs here, as do others
Only for multi–core systems

PGAS (Partitioned Global Array Storage)
Intermediate between MPI and OpenMP
Latest Fortran standard has coarrays
UPC (Unified Parallel C) is very trendy (and bad)

Key Factors

- More than a single node needs MPI or PGAS
MPI can be used between nodes, other ways inside

Shared memory easy to program, but hard to debug
But can add to serial program, incrementally
Many people try it, fail and use MPI instead

MPI, PGAS and GPUs need data distribution
Must start by designing that – can't easily add it later

Using and Debugging

Will start with important special cases

- This is not the only way to use them

This is the simplest way to design and debug
The way that most people will code their programs
- But there are many other approaches

Will then go onto more general parallel models

Small Vector Units

You use these as part of serial optimisation

- Overlap with MPI or GPUs can be inefficient

Need a suitable compiler and high optimisation

Typically Intel's and –O3 for SSE

Need to make your inner loops vectorisable

- Check that using the compiler messages

And that's more–or–less all you need to know
For advanced tuning, check the actual times

MPI, GPUs and VMX etc. (1)

See the course Scientific Programming with GPUs
This course describes only how to mix with MPI

- Encapsulate each type of use in algorithms
Design and test their interfaces in usual way
Don't need to worry about interactions just yet

- Can use MPI as a controller of the program
Can alternate MPI transfers, GPU and VMX use
And, under some circumstances, OpenMP

MPI, GPUs and VMX etc. (2)

- Easiest not to overlap MPI calls and GPU use
Consider separating by calls to MPI_Barrier
- Don't share GPUs between processes

Could also use OpenMP/threading, on single system
- But critical to use it only as controller
Again, don't share GPUs between threads
- And don't mix OpenMP/threading and MPI
Except using MPI between nodes and OpenMP within

Reasons are too complicated and messy for course
Include arcane details of MPI and system scheduling

Easiest Design

start: Use MPI to initialise
[Consider calling MPI_Barrier]
loop: Use GPUs to do calculation
[Consider calling MPI_Barrier]
Use MPI to synchronise data
[Consider calling MPI_Barrier]
Repeat from loop
stop: Use MPI to finalise

There is a little more on asynchronous use later

Using SMP Libraries

- Only one simple use: a threaded library
Libraries include NAG SMP, Intel MKL, AMD ACML

- Time is dominated by a few calculations
And some library already has SMP solver for it
Can then just call it, and problem is solved!

- In this case, one MPI process per system
Leave the multi–core use to the SMP library

- Can alternate this and using GPUs
Use the design above, replacing SSE by SMP

Shared Memory Parallelism (1)

- Many people use one MPI process per core
Same code runs on multi–core systems and clusters

- Currently, almost the only alternative is OpenMP
Sometimes, using OpenMP is easy and efficient
At others, it is evil to debug and tune

MPI + OpenMP is possible, but is more advanced
⇒ Use only one MPI process per node

Also, don't use a GPU in more than one MPI process

Shared Memory Parallelism (2)

- I investigated CilkPlus for this
Like the subset of OpenMP that I teach
It looks as if it is easier and safer to use
But it's now doubtful it will take off

- POSIX/Microsoft threads are NOT advised
Reasons are considerably outside this course
- C++ 11 threads are NOT advised, either

Parallelism Models

How you structure your application for parallelism
It's semi–independent of the parallel technology
E.g. can do anything in either MPI or OpenMP

- Changes how you approach problem

Especially as regards design and debugging

This lecture only summarises the main issues
Intended to point you in the right direction

Farmable Problems

Will describe these first, to get them out of the way

- Requirement divided into independent tasks

Fairly common, and easy to solve – examples:

Parameter space searching – finding best choice
Includes many forms of global optimisation
Anything where brute force is only solution

Monte–Carlo simulation – a bigger sample, faster
Remember to change random number sequence!

Simplest Approach

Code a task as a simple, serial program
Debug and test it, using an ordinary debugger

- Then wrap it up in a a parallel harness

Remember to keep the original serial form

Sometimes, you need make no changes whatsoever
Usually need very few, localised changes

- Parallelise using processes and not threads

Except when using GPUs, which are different

More details in the handout and even code in
https://www-internal.lsc.phy.cam.ac.uk/nmm1/

Why Use Processes?

- It looks more complicated, but is actually easier
The problems are far better understood

- Use pipes or files for input and output
Most program changes will be to do this

Controller creates input and merges output
All code to handle parallelism is in controller

Basic Master-Worker Design

- Parent application runs as controller

Manages several jobs in parallel
Each task gets a CPU from a pool (when free)

- It creates suitable job and its input
- Runs the jobs, and waits until they finish
- Collects their output and stores/analyses it

May run further jobs, perhaps indefinitely

Many ways of implementing this, often trivially

Easy Implementations

- A batch scheduler and serial jobs
Best to script the submission and collation
Generally most flexible and easiest solution

- Write an MPI controller – covered in its course
This is probably the easiest use of MPI

- Write a simple Python controller
This is a little harder, but not very much

Common Bad Solutions

- Perl, C etc. are significantly harder
There are some details in the extra information

- Writing a shell script is not advised
Almost impossible to do any error handling

- Using OpenMP or threads is not advised
One thread can compromise others too easily
Far too much changeable state is per process
There is no clean way to kill a stuck thread

Obtaining Parallelism

In general, you have to introduce parallelism
And that needs communication between the tasks

- The first rule is to use the most natural design
And secondly the one with least communication
Maximises debuggability and helps tunability

- Do NOT rush towards the coding!
Careful design is essential for success
Prototype to get timing and communication data?

Amdahl's Law

Assume program takes time T on one core
Proportion P of time in parallelisable code

Theoretical minimum time on N cores is

$T*(1-P*(N-1)/N)$

- Cannot ever reduce the time below $T*(1-P)$

Gain drops off fast above $1/(1-P)$ cores

Use this to decide how many cores are worth using
And whether to use SMP or clusters

- And whether the project is worthwhile at all

Practical Warning

The difference between theory and practice
Is less in theory than it is in practice

- Amdahl's Law is a theoretical limit

In practice, parallelism introduces inefficiency
Especially if the parallelism is fine–grained
Or frequent communication between threads

- Allow at least a factor of 2 for overheads

Need a potential gain of 4 to be worth effort
At least 8–16 if redesign is needed

Parallelism For Performance

- Most HPC uses a SPMD model
That is Single Program, Multiple Data

I.e. exactly the same program runs on all cores
But programs are allowed data–dependent logic
So each thread may execute different code

- In practice, HPC implies gang scheduling
All cores operating together, semi–synchronised
No theoretical reason for this, but it is so (today)

- Don't try to use dynamic core counts
That is best called an open research problem

SPMD Models

Simplest is master–worker – already covered

- But lock–free SPMD is reasonably easy to debug

A very ill–defined term, but here is what it means

- Workers communicate only with the master

Or by atomic access to global variables

This includes using reductions in MPI etc.

- Key is to avoid execution–order dependencies

Including any worker waiting on another

Especially, workers never lock access to any data

Asynchronism (1)

Can overlap communication and computation

- More in theory than in practice, unfortunately

Because synchronism at any level 'poisons' it

MPI progress issues are too complicated to cover
Covered in extra information for my MPI course

- Network operates independently of CPU

But TCP/IP is synchronous and needs CPU
Ethernet itself is similar, but becoming less so
InfiniBand is better, but drivers often aren't

- The memory subsystem is usually the bottleneck

Can be bandwidth, latency or conflict

Asynchronism (2)

Modern CPUs are almost all multi–core

- So can reserve some cores for communication

- Also GPUs can execute independently of CPU

If using only their own memory, no problem

- The memory subsystem is usually the bottleneck

Most CPU–bound codes are actually memory–bound
Can be bandwidth, latency or conflict

Many books and Web pages get this one wrong
Some of them describe what used to be the situation

Asynchronism (1)

Can overlap communication and computation

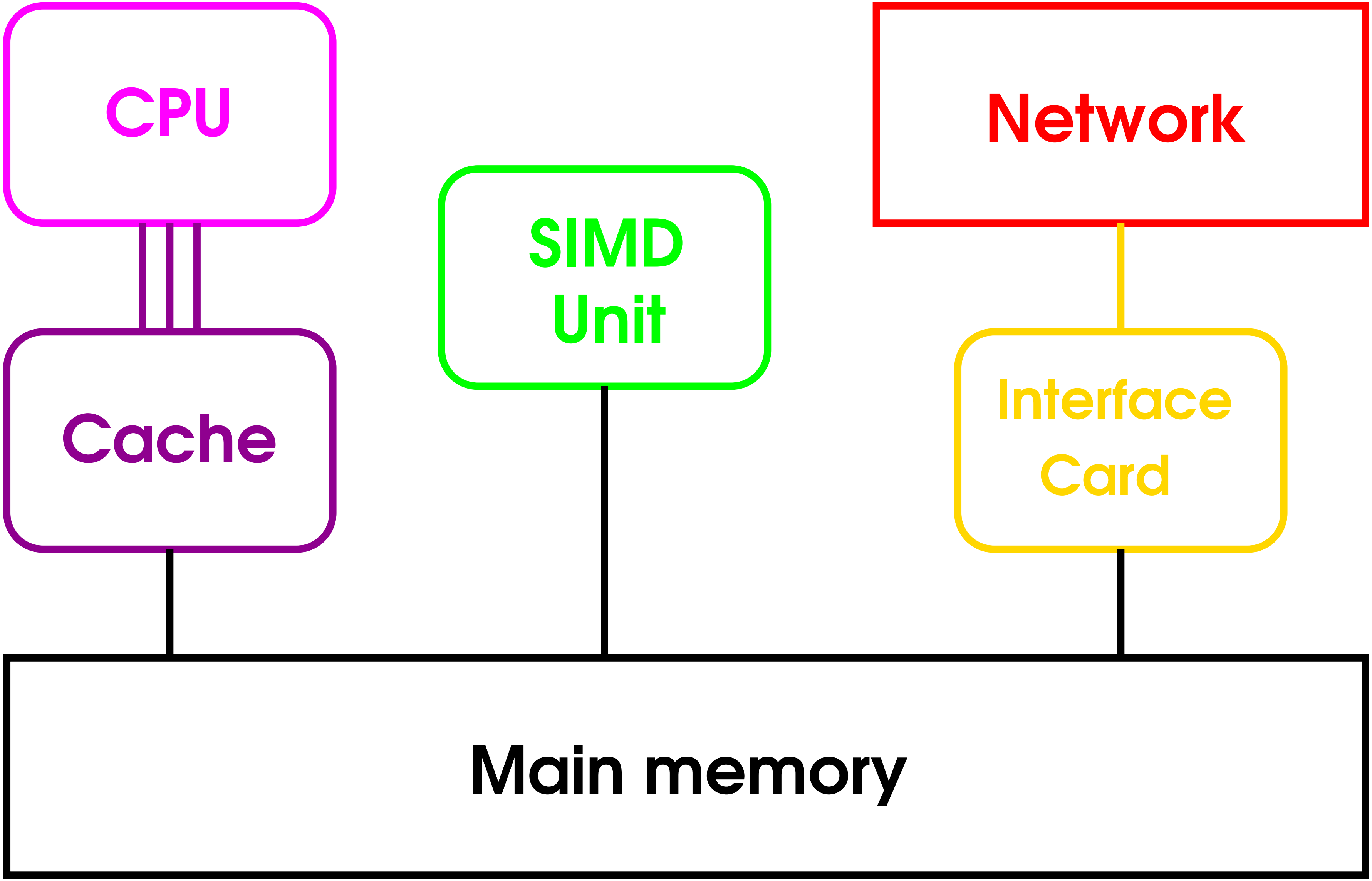
- More in theory than in practice, unfortunately

Because synchronism at any level 'poisons' it

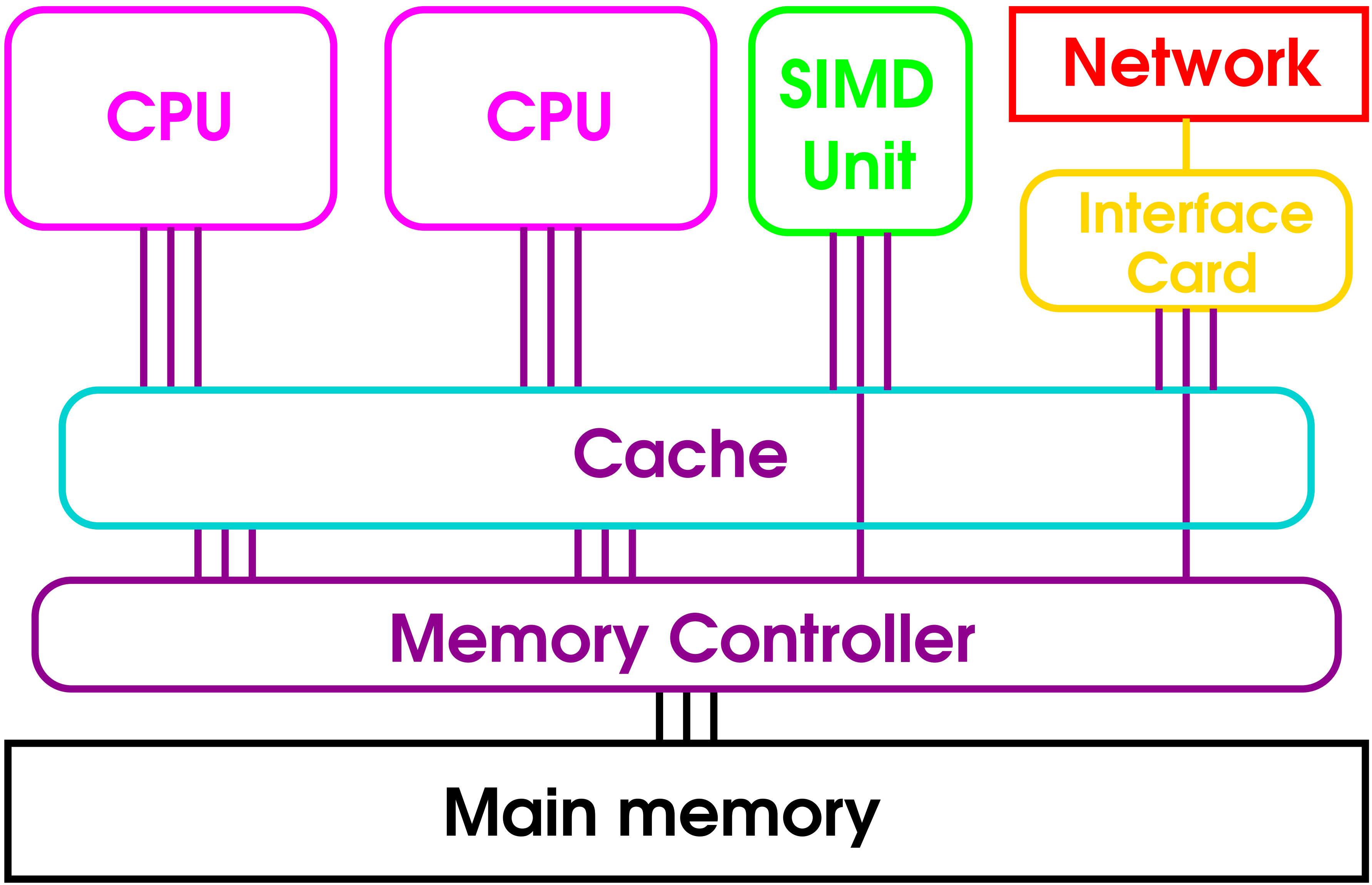
- The memory subsystem is usually the bottleneck

Most CPU–bound codes are actually memory–bound
Can be bandwidth, latency or conflict

Older Systems



Current Systems



Recommendations

- Do not rush into coding asynchronous programs
They can be a great deal harder to debug
Careful design is the key to success, as usual

- GPUs are best bet for making this work
Especially GPUs and MPI communication
But watch out, as the situation is complicated

- Remember the memory controller is a bottleneck
All of the GPUs, CPU and network need it
Overlapping memory access often causes conflict

HPC Models

Sometimes the problem has a natural model
If a suitable implementation provides it, use it
If not, must map the problem model to another

- Too complicated an area for this course

Will describe three of most important HPC models
Only ones I have seen used in production code

- Remember, careful design is critical

Some more details on this in my MPI course

Vector/Matrix/SIMD Model (1)

- The basis of Matlab, Fortran 90 etc.

Operations like mat1 = mat2 + mat3*mat4

Assumes vectors and matrices are very large

Very close to the mathematics of many areas

Often highly parallelisable – I have seen 99.5%

- Main problem arises with access to memory

Vector hardware had massive bandwidth

- All locations were equally accessible

Not the case with modern cache–based, SMP CPUs

Vector/Matrix/SIMD Model (2)

- Memory has affinity to a particular CPU

Only local accesses are fast, and conflict is bad

- Why LAPACK etc. use blocking algorithms

Some vector codes run like drains even if blocked

- Regard tuning as ALL about memory access

Same applies to using MPI and (somewhat) GPUs

Main cost is for the non–local accesses

- Hardest part of design is minimising those

Problem Partioning (1)

More a class of model, not a specific one

- Divide problem up into sections
Assign each section to a thread

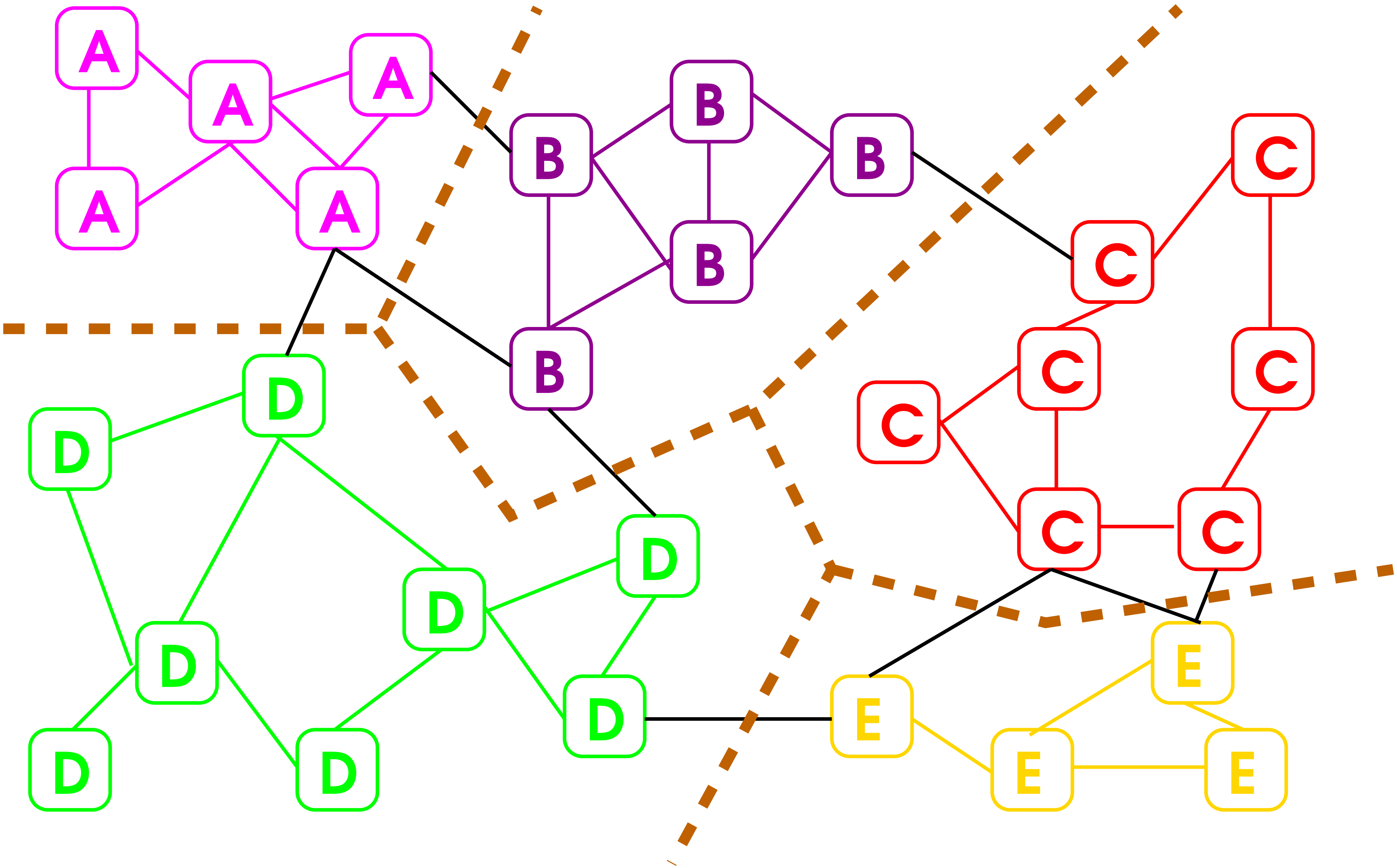
- Objective 1 is to keep it simple
- Objective 2 is to equalise CPU requirements
- Objective 3 is to minimise communication
  Especially threads waiting for others

Problem Partioning (2)

- Sometimes, partioning is natural and easy
E.g. in a motor, separate by component
Or by compound in a composite material
Or by species in a ecological simulation

- May need to group tasks together for threads
Use the objectives described above when doing that

Graph Partitioning



Problem Partioning (3)

Often done using spatial dimensions
Simplest use is a rectangular grid
Can assign indices by blocks or cyclicly

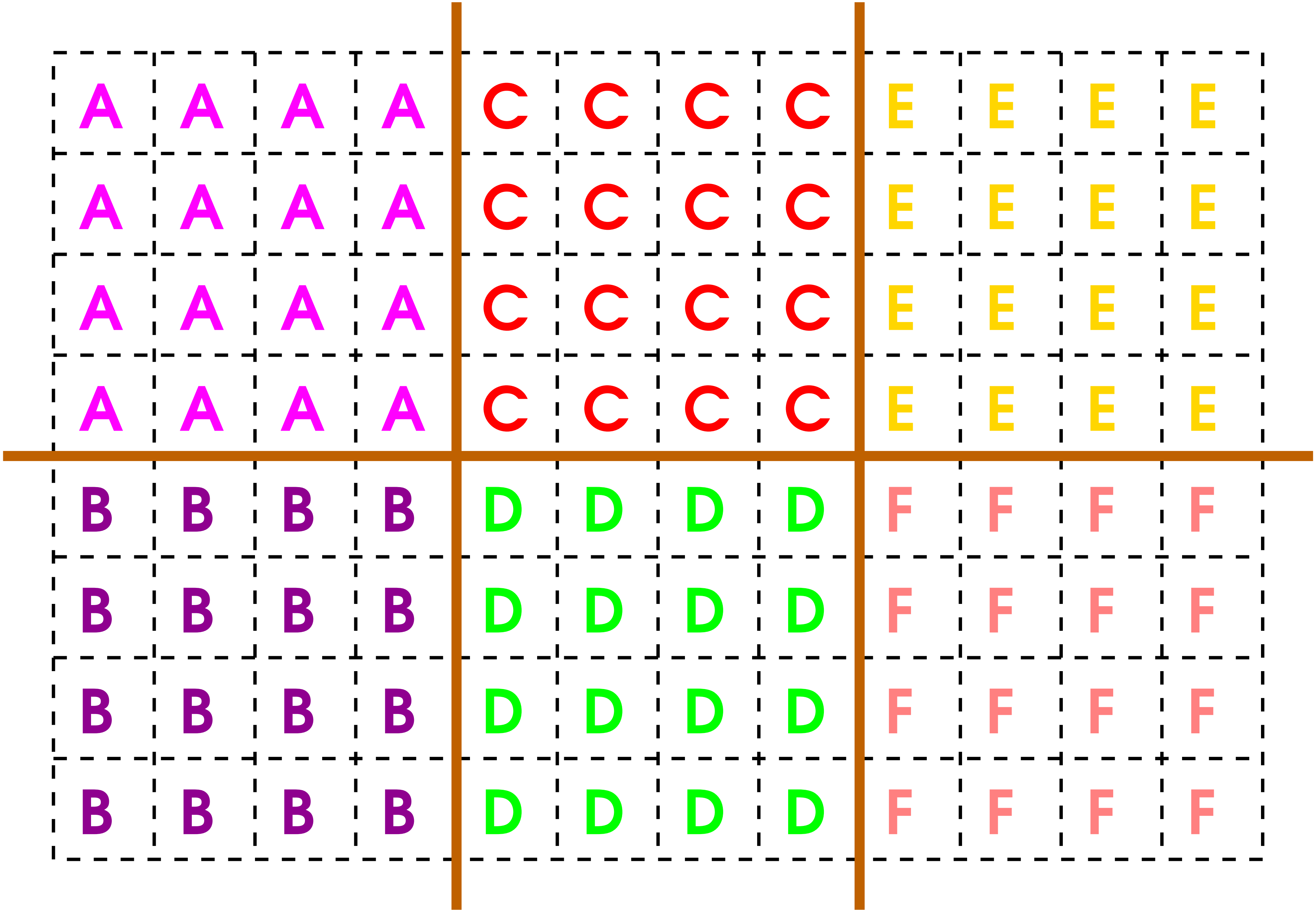
- Often some areas take longer than others
- And the communication often isn't uniform

So irregular divisions are often more efficient

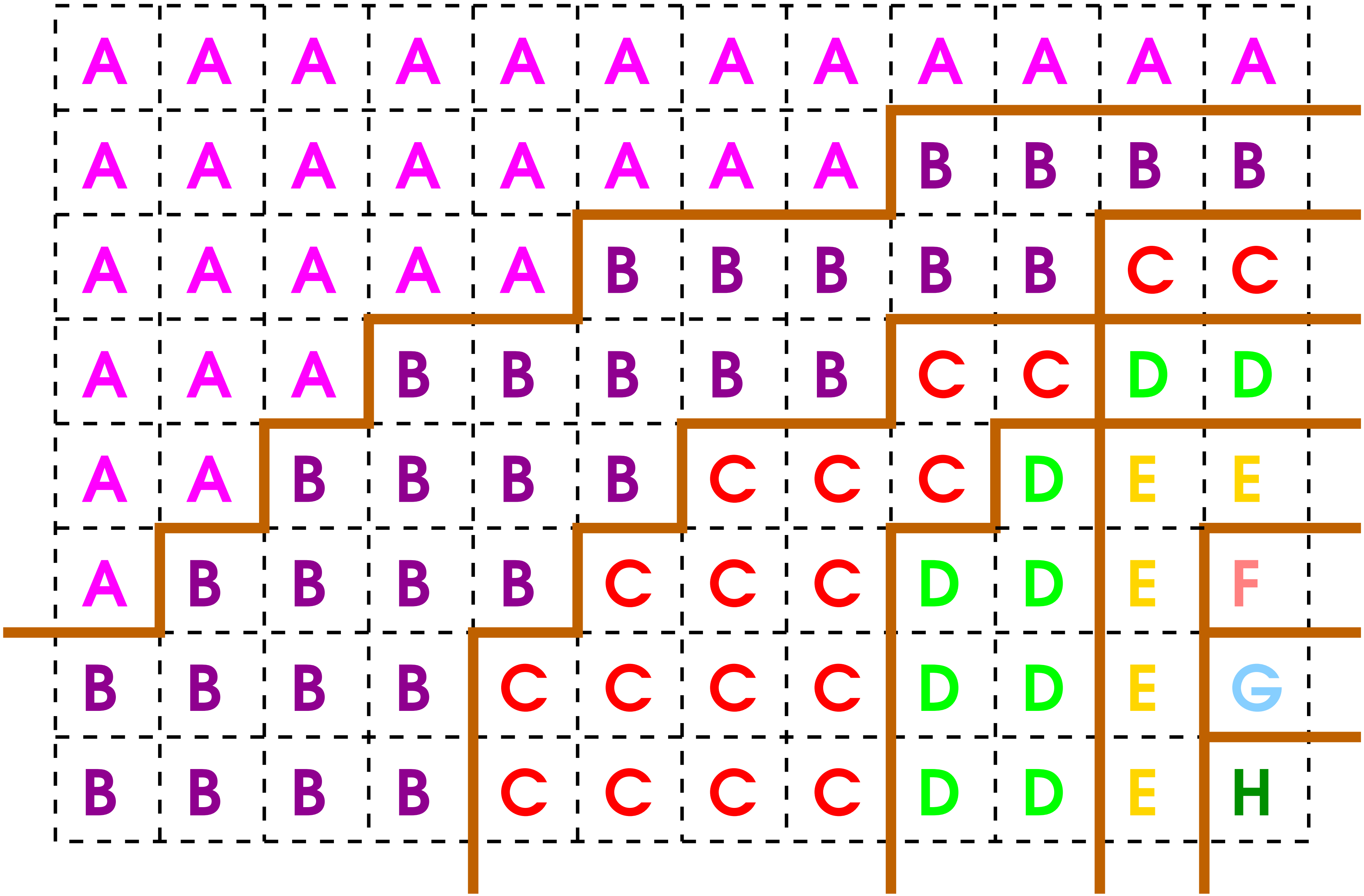
- More tedious and error–prone to program

E.g. mesh refinement, coordinate transformation, ...

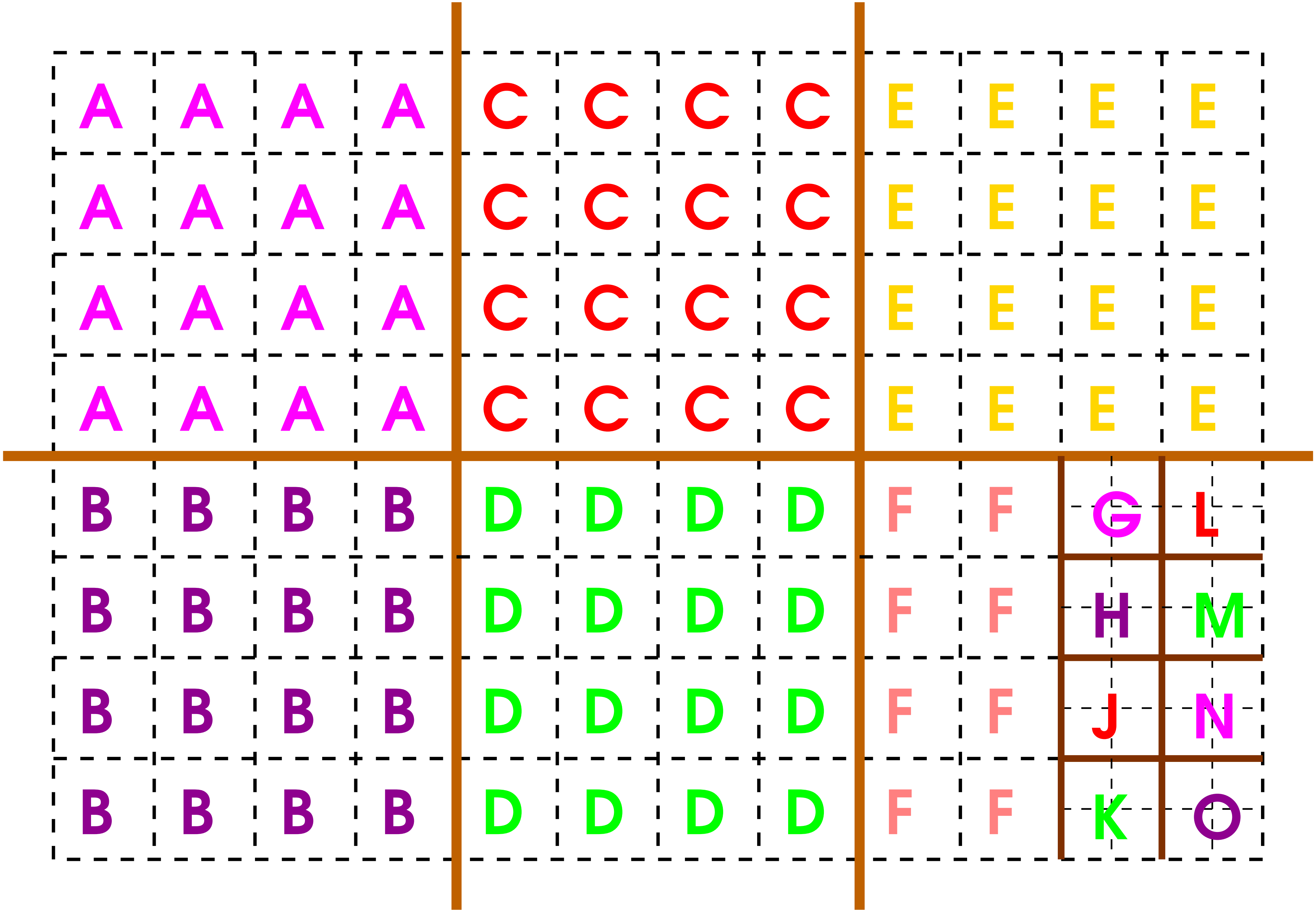
Block Partitioning



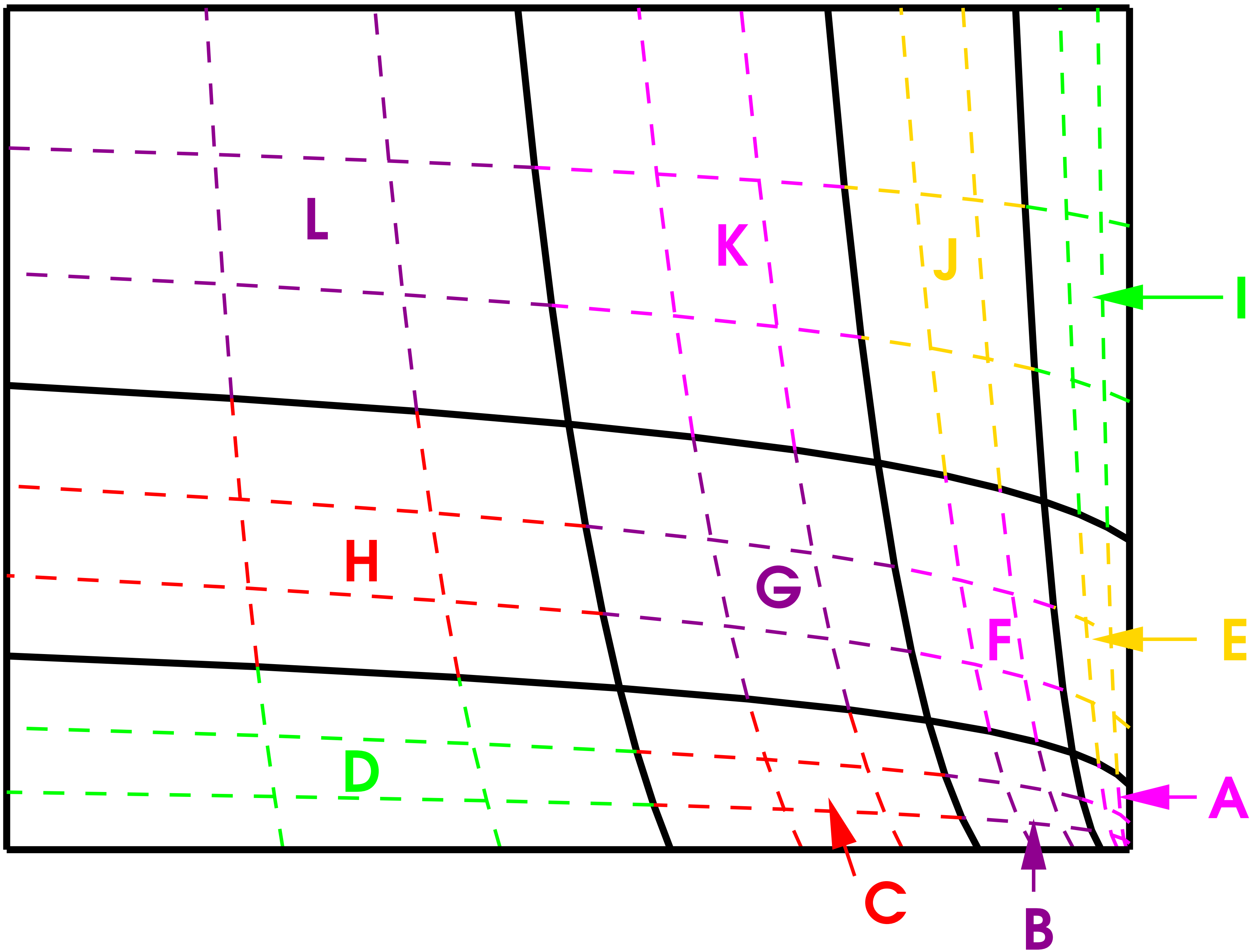
Irregular Partitioning



Mesh Refinement



Transformed Mesh



Other Possibilities

Several forms of cyclic partitioning
Triangles or tetrahedra can be used

And forms can be nested or otherwise combined

Also Voronoi/Dirichlet partitioning
Often used for irregular problems

Dataflow Models (1)

Can be useful for irregular problems

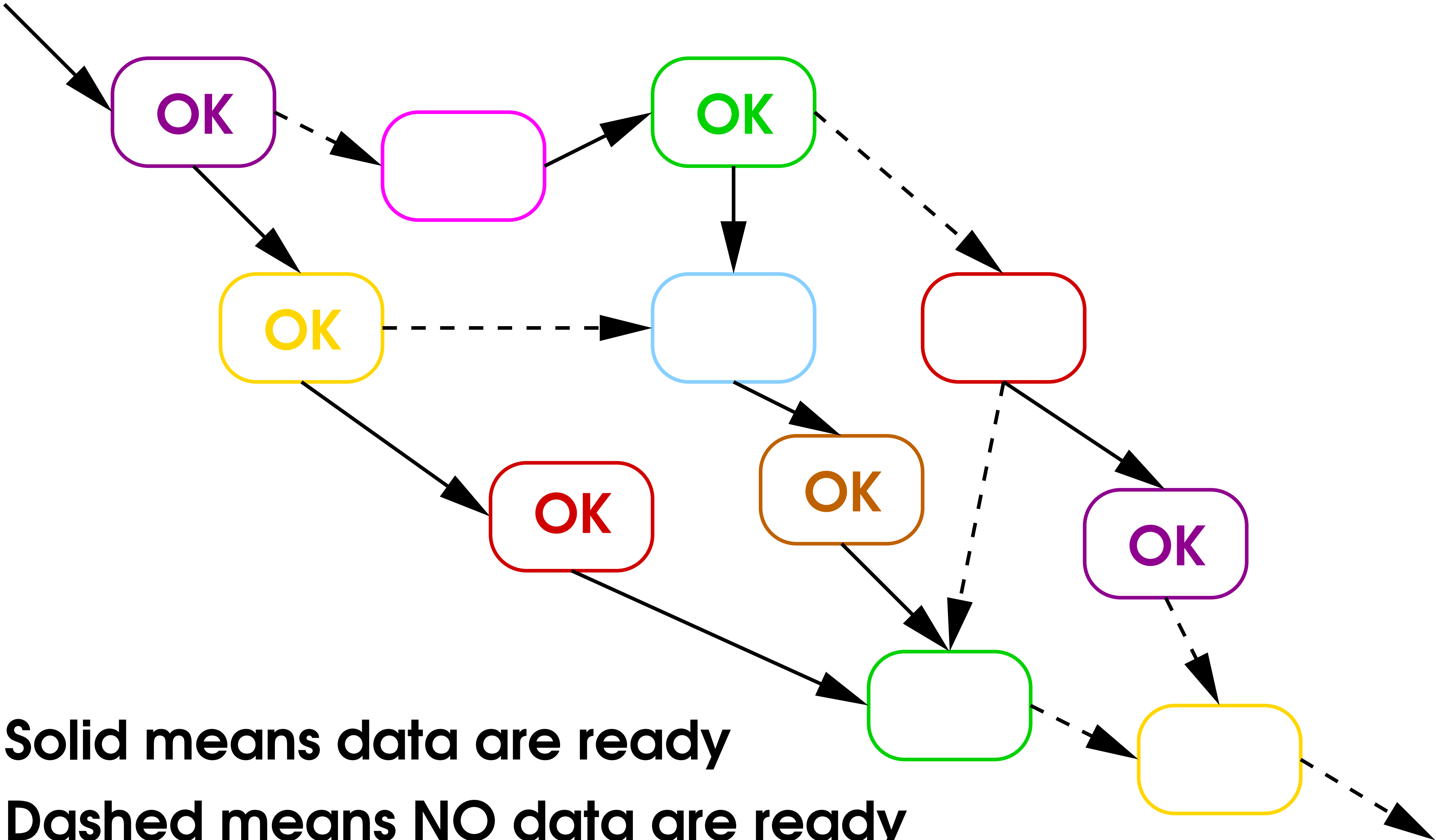
- If you don't find it natural, don't use it

Structure made up of actions on units of data
It defines how these depend on each other
The data are filtered through the actions
Actions run when all their input is ready

Input can be stacked up several deep
It may also be tagged if all input must match

See notes for more detail

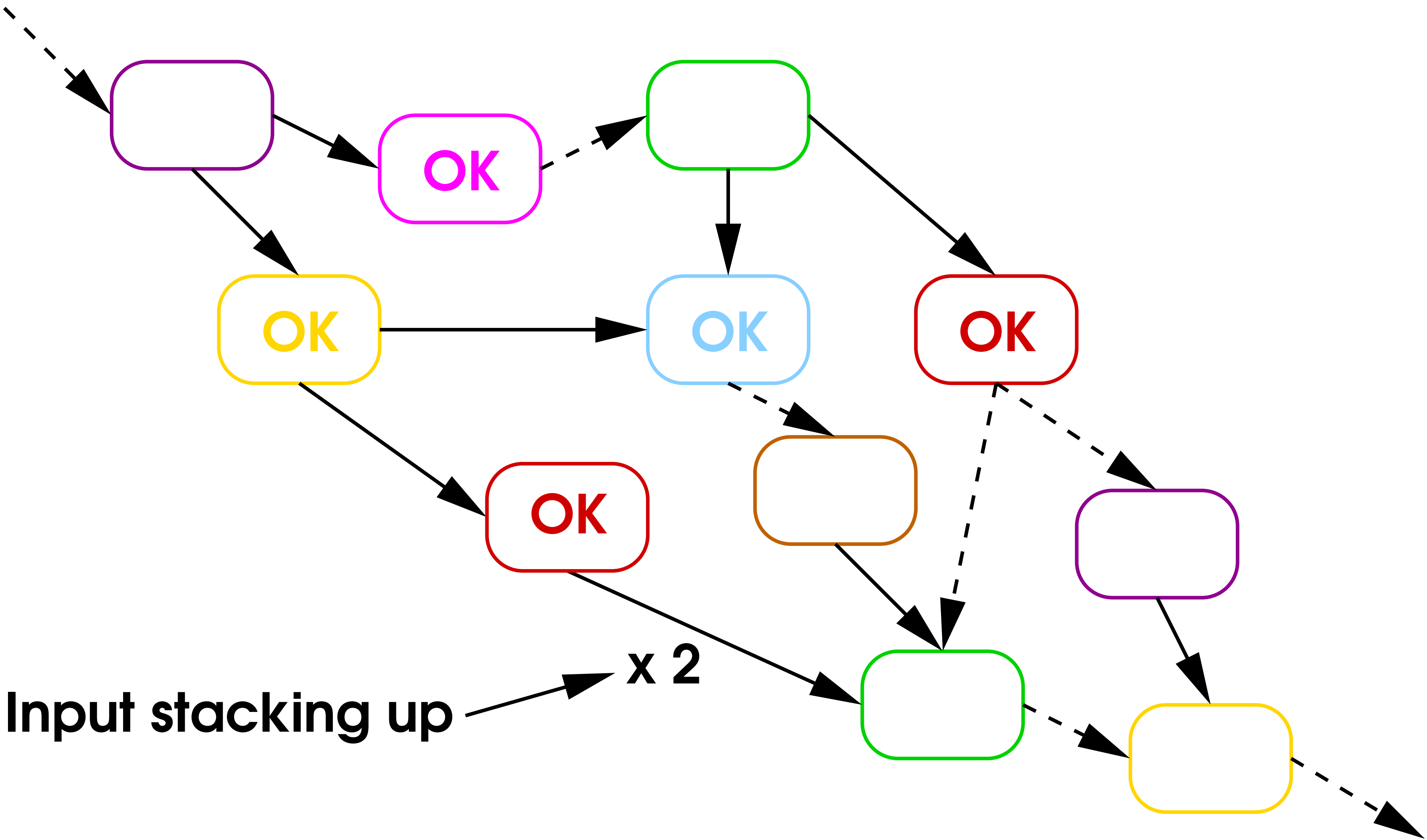
Dataflow (Step N)



Solid means data are ready

Dashed means NO data are ready

Dataflow (Step N+1)



Dataflow Models (2)

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

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

- The approach can make design a lot simpler
With a much higher chance of successful debugging

Designing for Distribution (1)

A good rule of thumb is the following:

- Design for SIMD if it makes sense
- Design for lock–free SPMD if possible
- Design as independent processes otherwise

For correctness – order of increasing difficulty
Not about performance – that is different
Not about shared versus distributed memory

- Performance may be the converse

There Ain't No Such Thing As A Free Lunch

Designing for Distribution (2)

- Next stage is to design the data distribution
SIMD is usually easy – just chop into sections

- Then work out need for communication
Which threads need which data and when
Do a back of the envelope efficiency estimate

- If too slow, need to redesign distribution
Often the stage where SIMD models rejected

Designing for Distribution (3)

- Don't skimp on this design process
Data distribution is the key to success

- You may need to use new data structures
And, of course, different algorithms

- Above all, KISS – Keep It Simple and Stupid
Not doing that is the main failure of ScaLAPACK
Most people find it very hard to use and debug

Using C++

Notes have some recommendations for using C++
Things that you will not find elsewhere
Very few references understand scientific computing
Bjarne Stroustrup's books are about the best