Parallel Programming

Options and Design (Part I)

Nick Maclaren

nmm1@cam.ac.uk, ext. 34761

March 2010

Summary

Topic has been (slightly artificially) split into two
Simple solutions are mentioned as they arise

- Reasons for parallelism, and basic design
  Strengths and weaknesses of each approach
  Thread pools, client–server, CPU farms etc.
  Most important models of HPC parallelism

- Available parallel implementations
  OpenMP and other shared–memory models
  PGAS (Co–Array Fortran, UPC etc.)
  MPI and other message passing models

Beyond the Course

Email scientific–computing@ucs for advice

http://www–users.york.ac.uk/∼mijp1/teaching/...
.../4th_year_HPC/notes.shtml

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/

Summary of This Part

Why do we want to code for parallelism?

Thread pools, master/worker, CPU farms, etc.

Using those, effectively and efficiently

HPC parallelism – where serial is too slow

Very high level HPC design and modelling

Before Starting

Coding is something a programmer does
System configuration is something a sysadmin does

- For parallelism, they need to work together

Course is for programmers (and sysadmins)
Will mention some of the general points later
May be a bit confusing to people who are only one

- You needn't be both programmer and sysadmin
- You do need to collaborate with the other

The Word Scheduling

Unfortunately, cannot avoid using it ambiguously

- First meaning is job scheduling
  Assigns jobs to systems (perhaps CPUs)
  A high–level task, done by an application
  Condor, GridEngine, LSF, PBS etc.

- Second meaning is thread scheduling
  Assigns threads (kernel and user) to cores
  Suspends threads to take interrupts
  A low–level task, done by the kernel core

Reasons and Design

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

Why Use Parallelism?

- Most common use is doing many tasks at once

Dominates in commerce – common in academia
Some scientific calculations are also like this

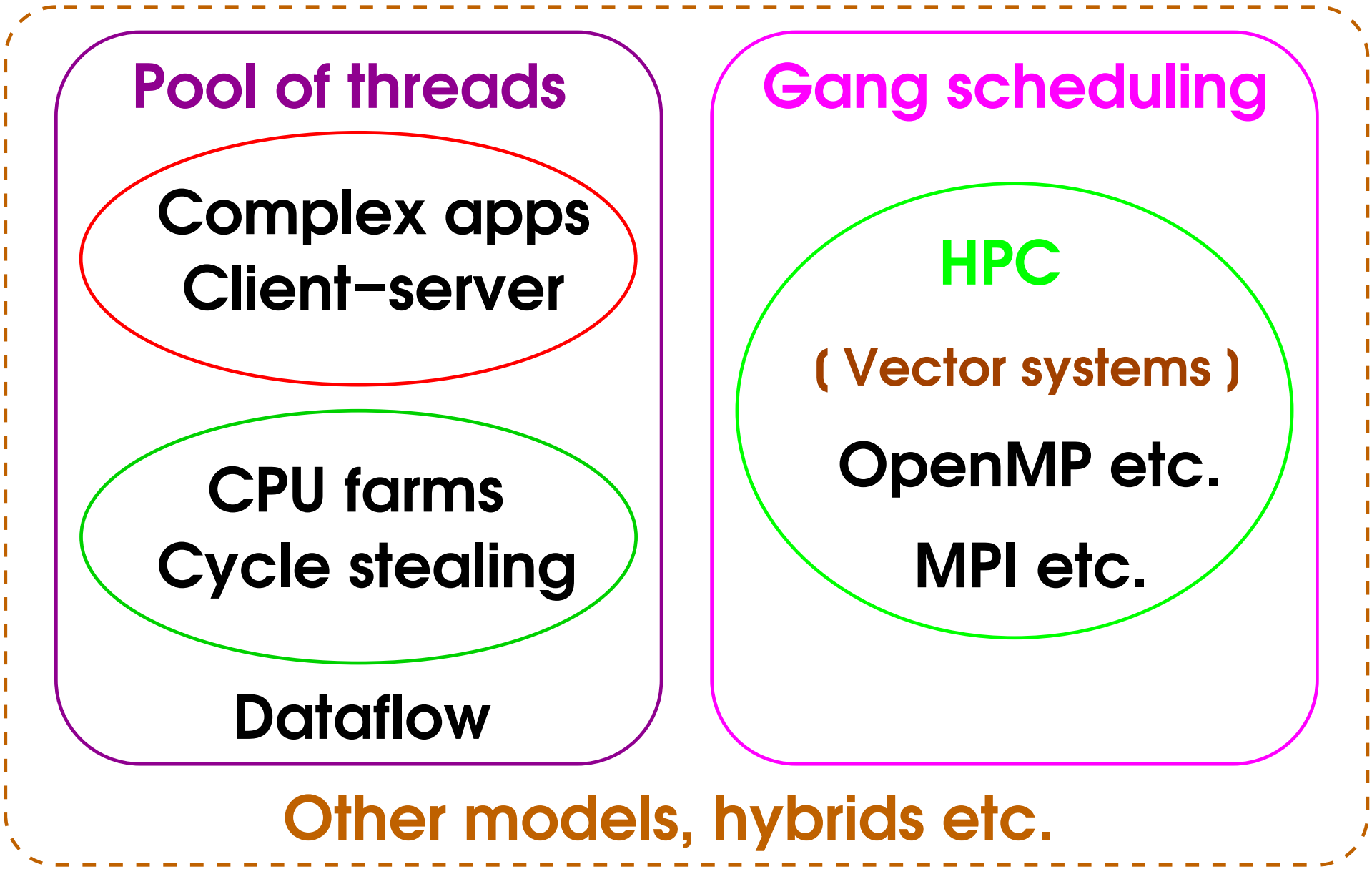
- Main other use is for more performance

As in HPC – High Performance Computing
This is MPI territory, where it dominates

- Difference between the two is critical

But it is only two sides of the same coin
Need to step back and think about objectives

Parallelism Landscape



(Not-)Moore's Law

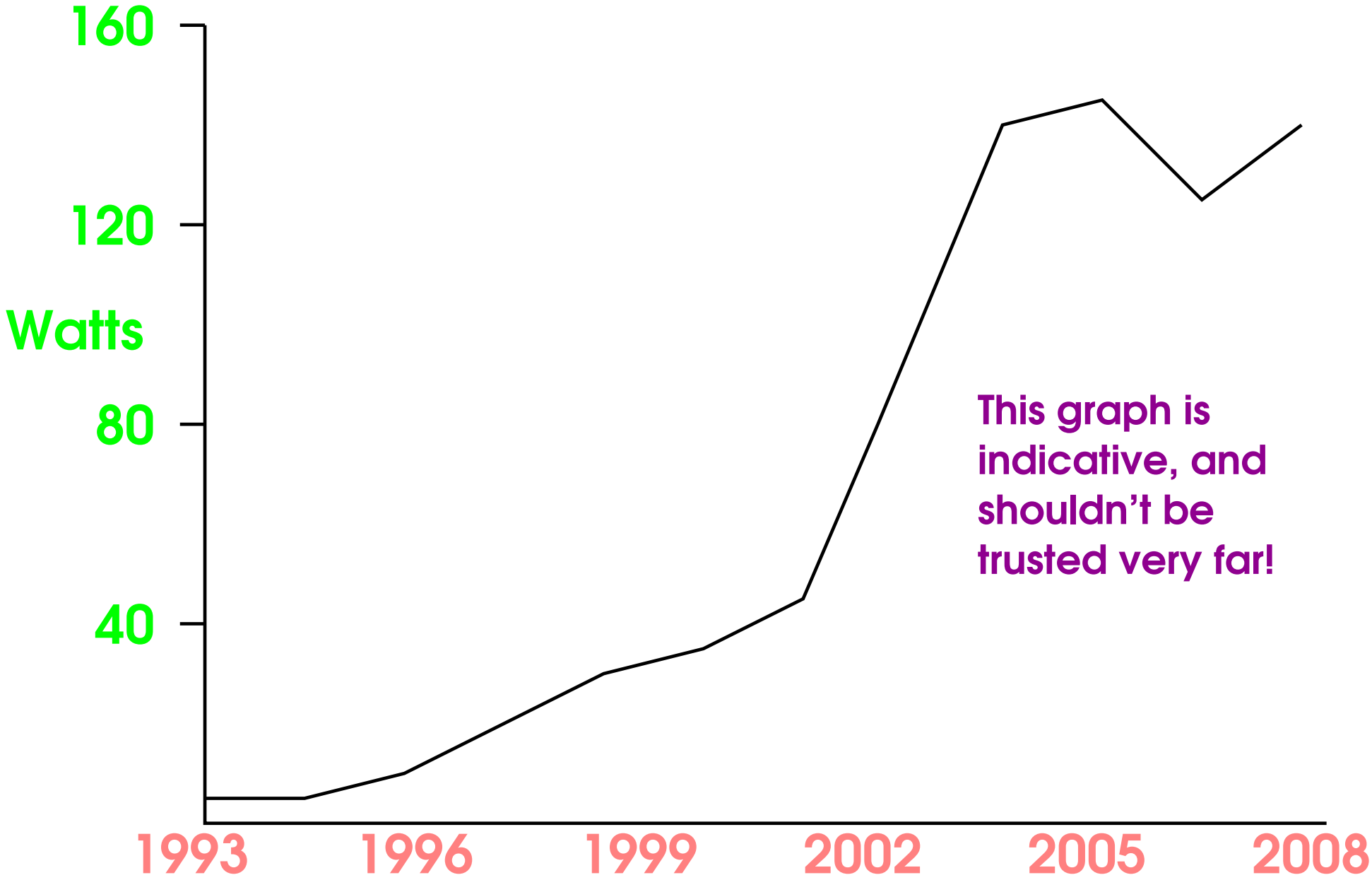
Moore's Law is chip size goes up at 40% per annum
Not–Moore's Law is that clock rates do, too

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

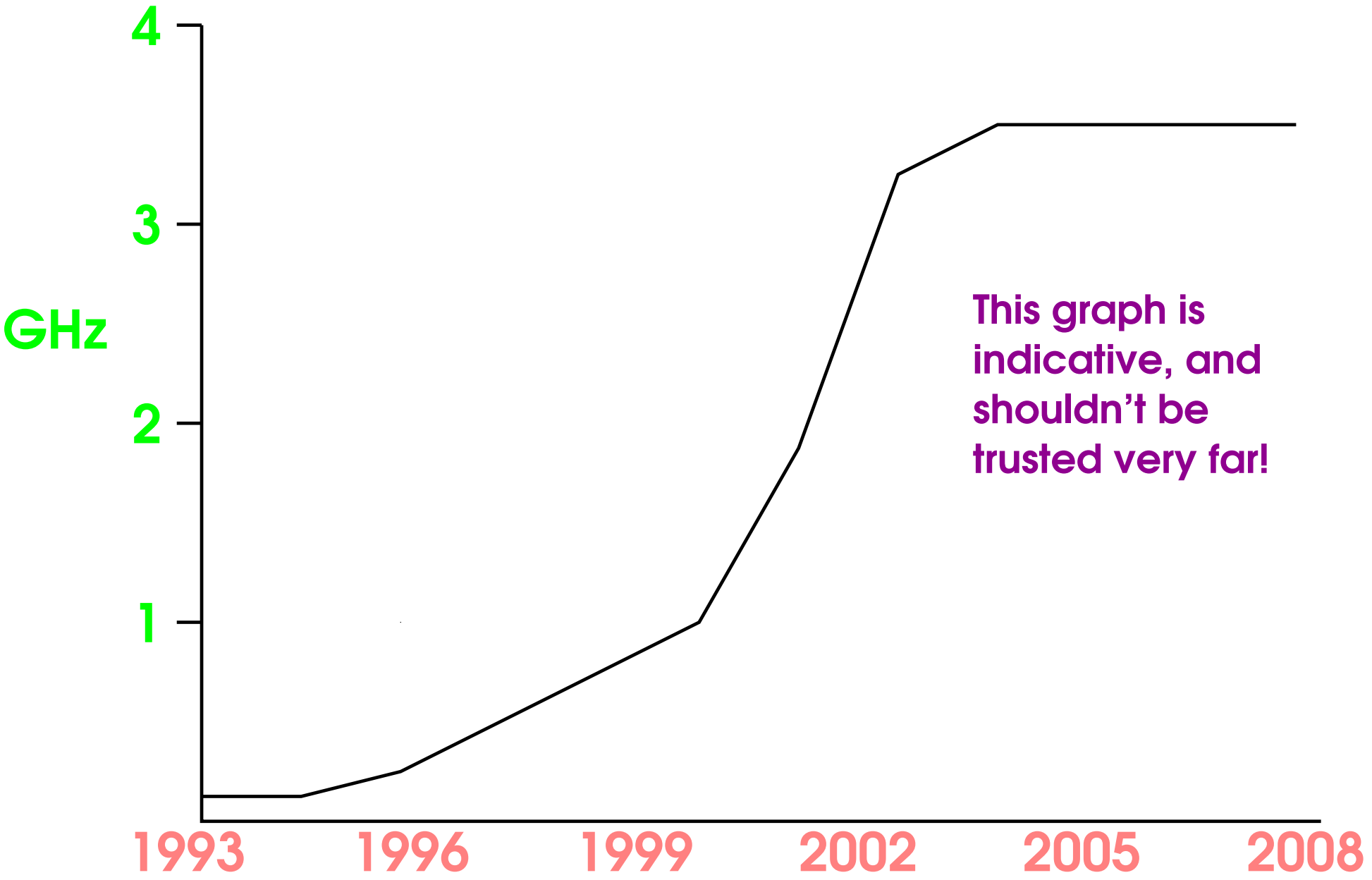
Not–Moore's held until ≈2003, then broke down
Clock rates are the same speed now as then

Reason is power (watts) – due to leakage
See http://www.spectrum.ieee.org/apr08/6106
A recent Intel presentation said the same

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.
Mostly irrelevant to "general" computing

Many Tasks at Once

- Objective is genuine parallel execution

Several disparate tasks to do, semi–independently

Equivalent of a manager delegating tasks
The extra performance comes as a result of that

- In practice, uses natural parallelism only

The tasks are large scale components
Consider them as complete sub–applications

- Most fine–grained models are purely theoretical

Pool of Threads Model

Requirement divided into semi–independent tasks
Each task gets a CPU from a pool (when free)
Low–level schedulers/dispatchers tuned for this
No current HPC language uses this – why not?

Dominates in client–server work (Web servers etc.)
Many complex, big applications – even compilers

CPU farms and cycle stealing also use this model
Not true HPC, but very common in scientific use
Will come back to this later

Basic Master-Worker Design

- Parent application runs as controller

Manages several jobs in parallel

- 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

May start a job upon external request
May start a job any time a CPU is free

Classic Client-Server

- Parent application runs continuously

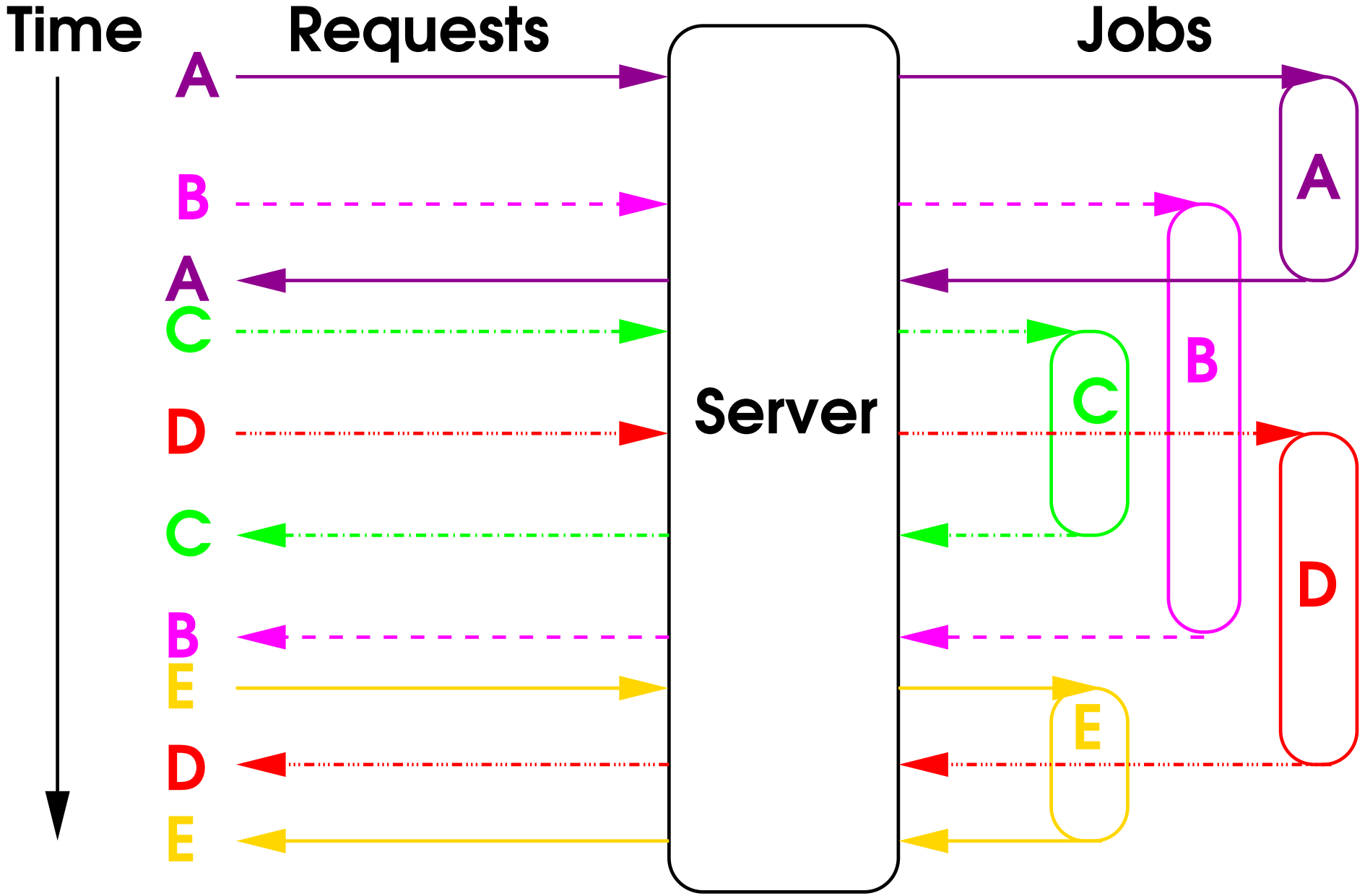
Clients make a series of requests
Application spawns a 'job' for each
Running jobs is the job scheduler's business

- Jobs run essentially independently

Talk to parent mainly at start and finish
No direct communication with other jobs

Most Internet servers are classic client–server
Web (e.g. apache), Mail (e.g. Exim), FTP, etc.

Client–Server



Implementation Approaches

- Client–server is only conceptually simple

Will mention just shared/distributed memory

- Shared is most common for Internet services

POSIX/Microsoft/Java threads are most common

OpenMP/MPI–2 can be used, but very rarely are

- But I don't advise most people to do that

Spawning processes is almost always better

Including on a single SMP system

POSIX/Microsoft/Java Threads

- Few languages support it (Fortran, C++, C don't)

And the POSIX standard is a complete mess
The Microsoft specification isn't much better

- One thread can compromise others too easily

Obviously, pointer/bounds errors can corrupt data
But too much changeable state is per process
There is no clean way to kill a stuck thread

One extreme example is signal handling
Dozens of other areas, in all relevant languages

Spawning Processes (1)

- It looks more complicated, but isn't, actually

The problems are far better understood

- See later under CPU farms and job scheduler

This is generally the CPU simplest solution

- Or use Python etc. for the controller/harness

Writing shell scripts is common but not advised

C, C++, Perl work, too, but are painful

http://www.ucs.cam.ac.uk/docs/course-notes/...
.../unix-courses/multiapplics/

Spawning Processes (2)

- Use pipes or files for input and output

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

- Processes can be in any language (e.g. Fortran)

Run each process ($\equiv$ job) serially
No modification needed to run, in most cases

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

Complex Applications

This is where the topology is more complex

- All processes communicate directly

- The communication isn't generally too hard
- The synchronisation can be a nightmare

Needs to match the kernel scheduler's design
Some slides on why at very end of course

This course won't go into this in any detail
Complex applications are, er, complex

Typical Examples

Start with operating systems and desktops!
More relevantly, many large scientific applications
Also decomposable activities like make –j

May separate input decoder from analysis
More commonly, separate both from GUI
Or run separate tasks in parallel, like make –j

With any complex application: stop; think; design

- Code in haste; repent at leisure (and you will)

CPU Farms

These are a sort of master–worker usage
A pool of threads used for performance

- The job scheduler is the parent application

PWF/MCS Condor is (was?) a (fairly) typical example

- Each job gets a dedicated CPU

It runs until it stops, and then the next runs
Jobs may be run on request or queued

- Each job is non–interactive and independent

No direct communication with other jobs

Overall Design

- Possibly interactive program creates jobs

Sets up their input and submits them

- Job scheduler runs the queued jobs

A pre–built controller for spawning processes

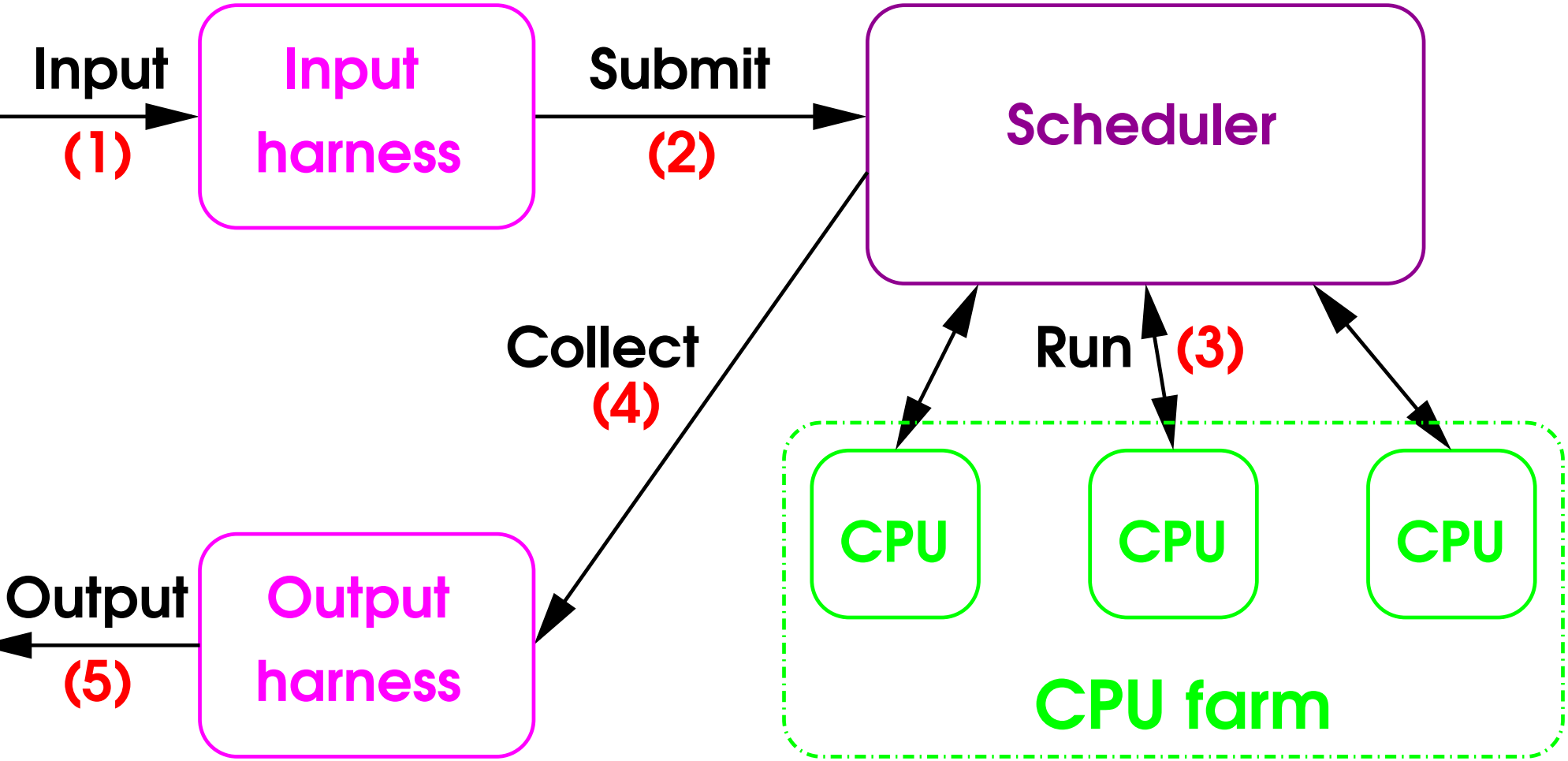
- The jobs are serial and batch

I.e. using one CPU, non–interactively

- Possibly interactive program checks completion

Reads their output and creates the results

Manual (Interactive?) Harness



Typical 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!

Often used as one phase of more complex analysis
Common in bioinformatics and many other areas

- That is all that many scientists need

CPU Farms vs Background Tasks

- The jobs are just serial applications

Precisely how you develop and debug them!

- A dedicated CPU gives predictable times

With all of the usual caveats ...

- A dedicated system gives better RAS

In desperation, rebooting recovers the resources

You can use background processes for this
Don't get either of last two advantages

Cycle Stealing

- Using idle cycles on people's workstations

Touted for 30+ years by the over–optimistic
Needs cooperation between all people involved

Some sites/people have got it to work

- But generally, it's close to a disaster

CPU time isn't the problem – memory and swap are
As is scheduling and any synchronisation

When will a 24 CPU–hour background job finish?
This week, next week, sometime, never ...

Job Schedulers

- Much the simplest way of running CPU farms

GridEngine, Condor, LSF, PBS etc.

- That is a pre–debugged controlling application

Writing non–trivial ones is a major pain in the neck

Much harder than just controlling background tasks

- Don't mix interactive use and batch scheduling

Nobody has ever got them to work well together

- Ask for help configuring systems / job schedulers

Rolling Your Own

This is a summary of several previous slides

- Do that only on a single SMP system
  Just possibly on a private, fixed cluster
- Don't mix it with interactive work
  If you must, don't use all cores for it
- Write it in Python, Perl if you must, etc.
- KISS – Keep It Simple and Stupid

Beyond that, it is easier to install a job scheduler

Automating CPU Farms

As usual, the harness can be automated

- Many problems are inherently iterative
  Most searching and global optimisation
- Very useful for long–running problems
  Controls automatic checkpoint/restart
- Also for CPU farms used as back ends
  Some HPC client–server applications

Long-Running Problems

- Most systems have a fairly small job time limit

For RAS, maintenance etc. – often 24 hours

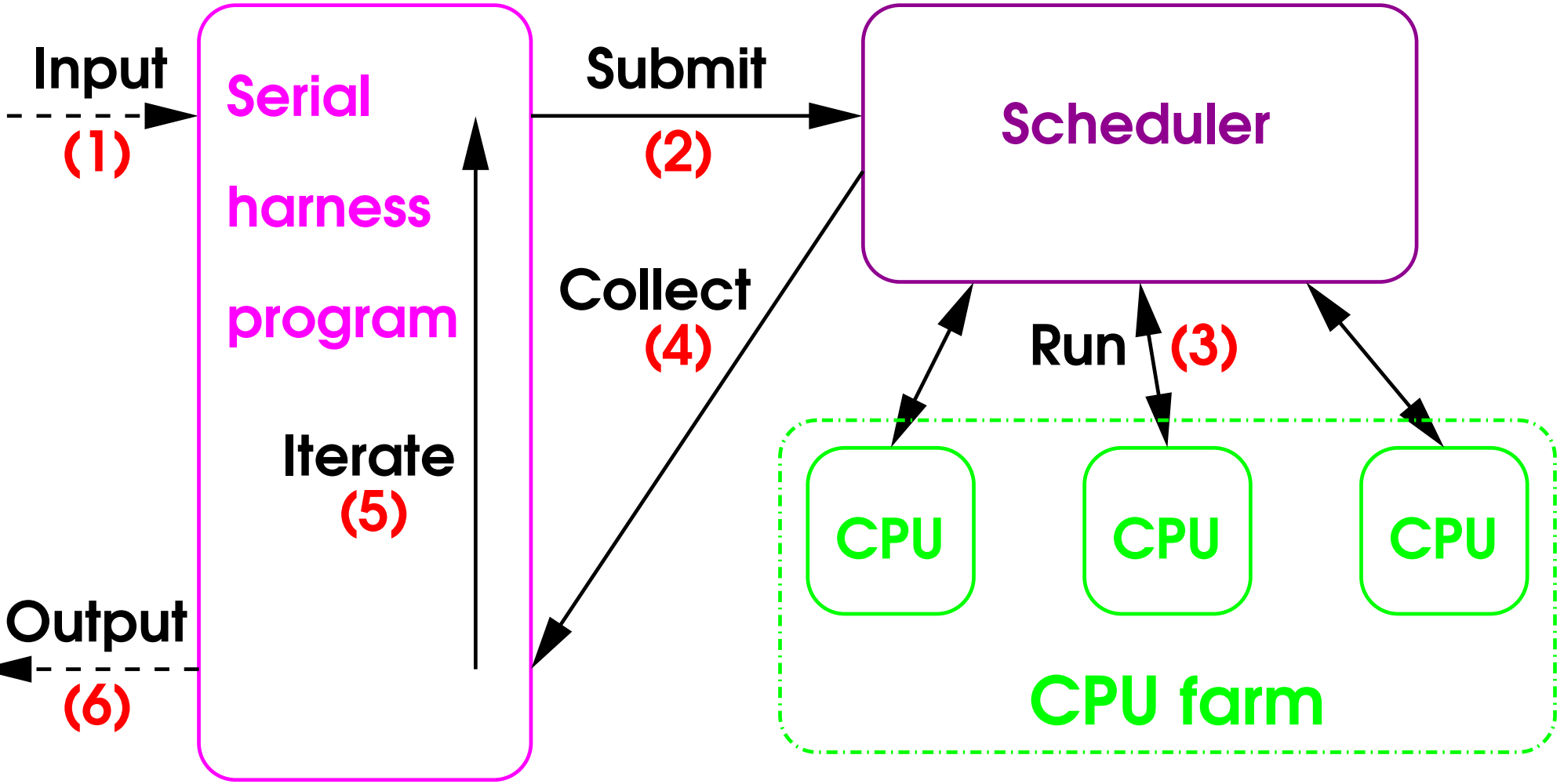
- A program may write its current state to a file
  [This is often called checkpointing]
- The job may resubmit another as it finishes

It starts by restoring from the checkpoint

- Best to use alternate checkpoint files

In case of a crash while it is being written
Ask for help if you need it on this one

Iterative Harness



Harness Design

The harness does not need to run continuously

- A common mistake to write one that does

Schedulers are designed to recover from stoppages
Power cuts, system failure, system upgrades etc.
Doing that for the harness is truly painful

- Best design is to keep its state in files

It collects finished jobs, analyses their results
Submits new jobs, Emails a progress report
Run it (manually or automatically) at intervals

Beyond That?

You can automate many forms of failure handling

- As always, be careful to write fail–safe code

See your job scheduler for relevant features

- Harness + CPU farms is very flexible

Don't assume you need a monolithic application

Can use different harnesses for different purposes
Changes to the jobs are typically small

Dynamic CPU Pools

This is where CPUs enter and leave the pool
Important for maintenance in large clusters
Failures need rebooting or replacement

- Don't even try to handle this yourself

Even the best job schedulers have some difficulty
It's not your problem – don't make it so

- Your harness does need to handle jobs failing

Often half–way through updating their state files!

Dataflow

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

Structure made up of actions on units of data
Rather than defining the order of execution

- Useful when designing your program structure

Very useful for handling irregular problems

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

Will return to and describe it later

More Performance (1)

- Vector systems etc. are more–or–less defunct

Simple SSE etc. handled entirely by the compiler
$\Rightarrow$ As a serial optimisation in code generation

- But what about GPUs?

Very like an attached vector processor
Look up FPS on the early PCs (1980s)

- Biggest problem is getting data in and out

Fairly easy to program (in C/C++), but hard to tune
We will come back to these later

More Performance (2)

- Most HPC uses a SPMD model
  (Single Program, Multiple Data)

Either for distributed memory or shared memory

- In practice, HPC implies gang scheduling

All cores operating together, semi–synchronised
No theoretical reason for this, but it is so (today)

Beyond that is an interesting research problem
Which isn't good news if you just want to do it!

- It can be and is done in practice, successfully

HPC Parallelism

- This is a single, very large calculation

Not always CPU limited, may be memory limited
May be I/O limited, but not covered in this course

- Need to extract parallelism from the application
- Then need to map it to a suitable parallel model
- Then need to implement that design

- Do NOT rush onto the last phase!

Careful design is essential for success

Gang Scheduling (Kernel)

HPC almost always uses gang scheduling

- All cores run the code, or none of them do

It makes analysis and tuning much easier

There is a major problem with kernel schedulers
Modern ones often handle this very badly indeed
Gang and time–sharing scheduling are incompatible

- It is best to use whole systems for HPC

Different scheduling options and even configuration

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

Practical lower bound more like 2*T*(1–P)

If That Isn't Enough?

Need to parallelise serial parts of code

- No point in proceeding otherwise

- Often needs complete redesign of program

Removing serial dependencies from structure
Using slower, more parallelisable algorithms
Yes, doing that can be truly painful

But it's better than completely wasting your time

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

Trivial Case

- Time is dominated by a few calculations

E.g. SVD, n–D optimisation, PDEs
Some library already has suitable parallel solvers
Can then just call it, and problem is solved!

- Several suitable libraries for SMP systems

Main portable library is NAG SMP (+ FFTW, sort–of)
Vendor libraries – ACML, MKL, Sun etc.
Very little in the public domain – see later for why

- Very little for clusters – but check ScaLAPACK

Intel MKL does something, but have not tried it

Dynamic Core Counts

Some SMP libraries will adapt dynamically
If they actually work, then it's not your problem
If not, you can specify the number of cores

- You are not advised to go beyond that

HPC with dynamic core counts is a open problem
I.e. too hard for most researchers in the HPC field!

- Running more threads than cores is Bad News

Some systems seem to crawl into a hole and die!

Embarrassingly Parallel (1)

Some applications are naturally almost farmable
Several obvious, semi–independent, large tasks
Or they can easily be rewritten to become like that

One classic example is video rendering
- Separate scenes are fully independent
- Each frame is almost independent
- And a frame can be divided into sections

Need to fix up the boundaries afterwards

- Last requirement means not fully farmable

Embarrassingly Parallel (2)

- If conveniently farmable, why not do so?

Can run on almost any system, including PWF/MCS/DS

E.g. Monte–Carlo or parameter space searching

If not, have to decide between following:

- Separate processes with message passing
- Separate processes with shared memory
- Some form of threading in one process

I.e. general HPC, but easy to make efficient

Most people use MPI, but any method is feasible

HPC Models (1)

Let's assume that the problem isn't so easy

- First key question is which HPC model to use

The closer to the problem specification, the easier

Only some models have current implementations
And some are much more scalable than others

For this, the only solution is top–down design

- Choose the concepts first, then the structure
- Only after that, start designing the program itself
- Programming is the last and least of the tasks

HPC Models (2)

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 a lecture course

- If in ANY doubt, ask for help

Will describe three of most important HPC models
Only ones I have seen used in production code
If you come across another, please tell me

Vector/Matrix 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
But vector hardware is essentially defunct

- A good basis for SMP autoparallelisation

I.e. where the compiler does it for you
Usually needs quite a lot of manual tuning
Including explicit calls to SMP libraries

Vector/Matrix Model (2)

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

- Memory has affinity to a particular CPU

Only local accesses are fast, and conflict is bad

- Some vector codes run fast, some like drains

Vector/Matrix Model (3)

Normal solution is OpenMP for vector codes

- Regard tuning as ALL about memory access
- Only experts should try this on clusters

You can often get very large speedups quite easily
E.g. by keeping both matrix and matrix^T
Using the one that is better for memory access

- Problem is tricky, but well understood

Please ask for help if you hit this one

Problem Partioning (1)

More a class of model, not a specific one

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

Remember the video rendering example?

- 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

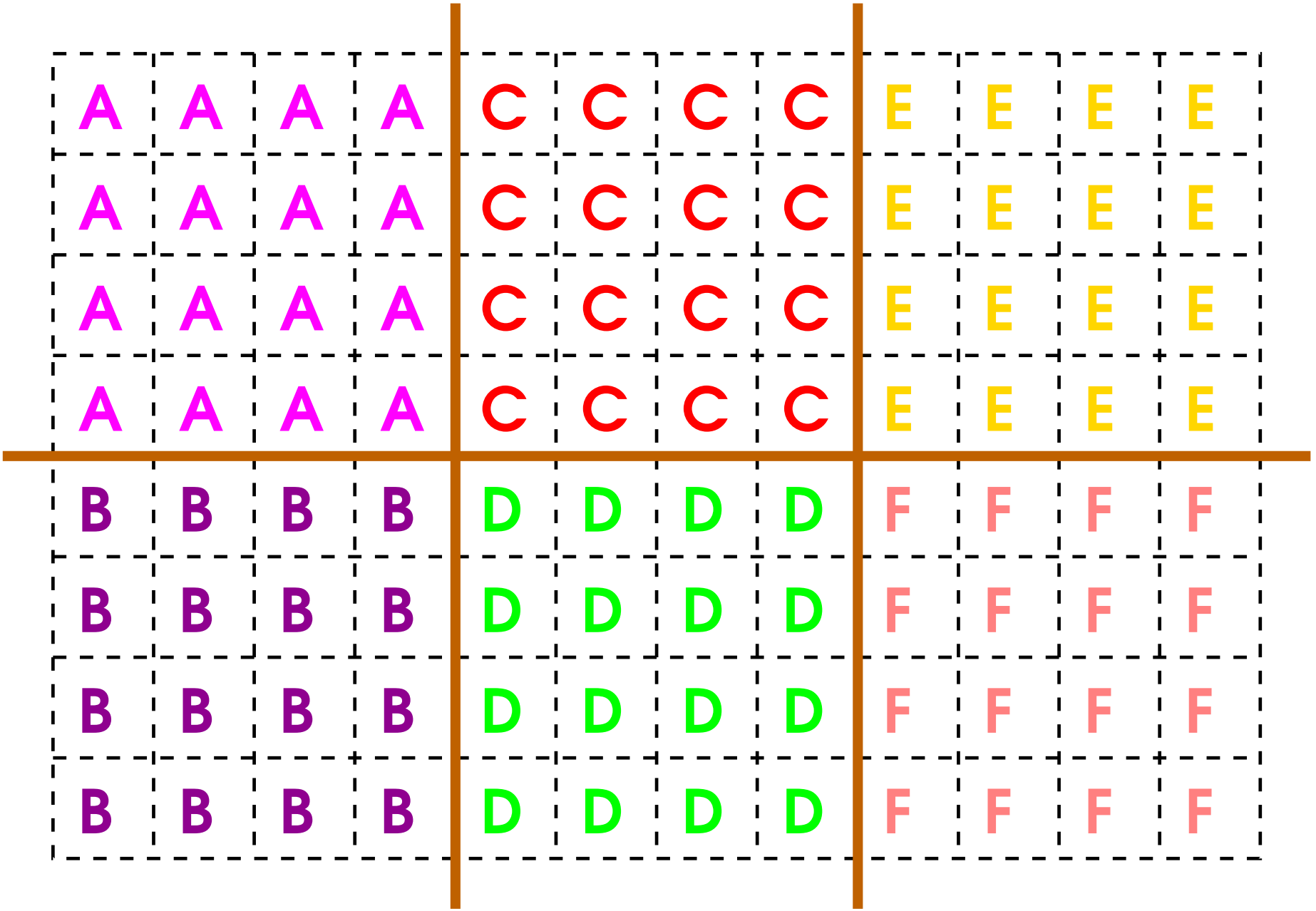
Problem Partioning (2)

Sometimes, partioning is natural and easy
More often it is artificial and confusing
As an example of that, look at ScaLAPACK

- Careful thought is never wasted in this

Often done using spatial dimensions
Simplest use is a rectangular grid
Usually simple blocking but can be cyclic

Block Partitioning



2–D Cyclic Partitioning (1)

A	C	E	A	C	E	A	C	E	A	C	E
B	D	F	B	D	F	B	D	F	B	D	F
A	C	E	A	C	E	A	C	E	A	C	E
B	D	F	B	D	F	B	D	F	B	D	F
A	C	E	A	C	E	A	C	E	A	C	E
B	D	F	B	D	F	B	D	F	B	D	F
A	C	E	A	C	E	A	C	E	A	C	E
B	D	F	B	D	F	B	D	F	B	D	F

2–D Cyclic Partitioning (2)

A	B	C	D	E	F	A	B	C	D	E	F
F	A	B	C	D	E	F	A	B	C	D	E
E	F	A	B	C	D	E	F	A	B	C	D
D	E	F	A	B	C	D	E	F	A	B	C
C	D	E	F	A	B	C	D	E	F	A	B
B	C	D	E	F	A	B	C	D	E	F	A
A	B	C	D	E	F	A	B	C	D	E	F
F	A	B	C	D	E	F	A	B	C	D	E

Problem Partioning (3)

- It may be simpler to use a non–spatial criterion

E.g. in a motor, separate by component
Or by compound in a composite material
Or by species in a ecological simulation

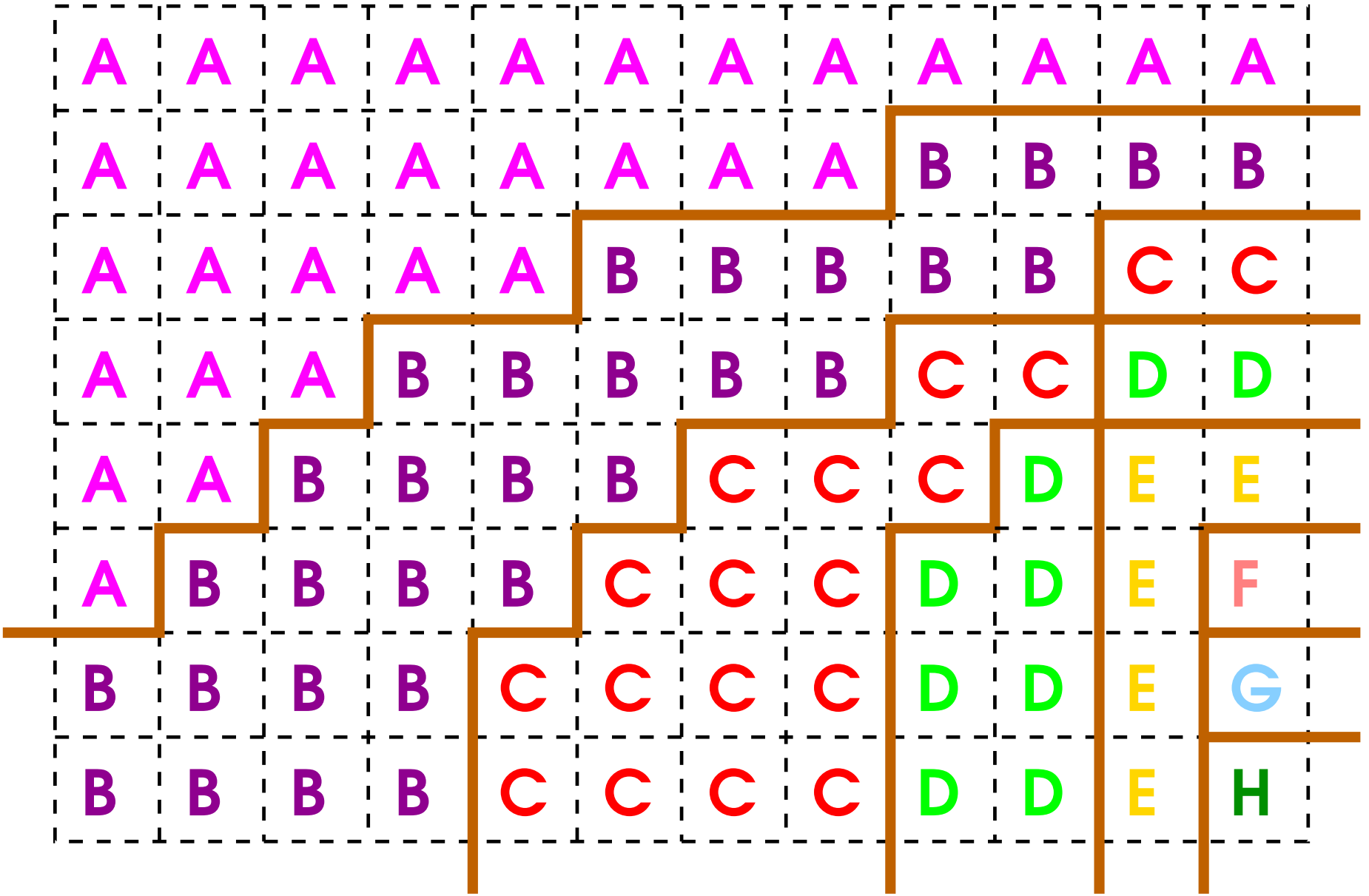
- Often some threads take longer than others
- And the communication often isn't uniform

So irregular divisions are often more efficient

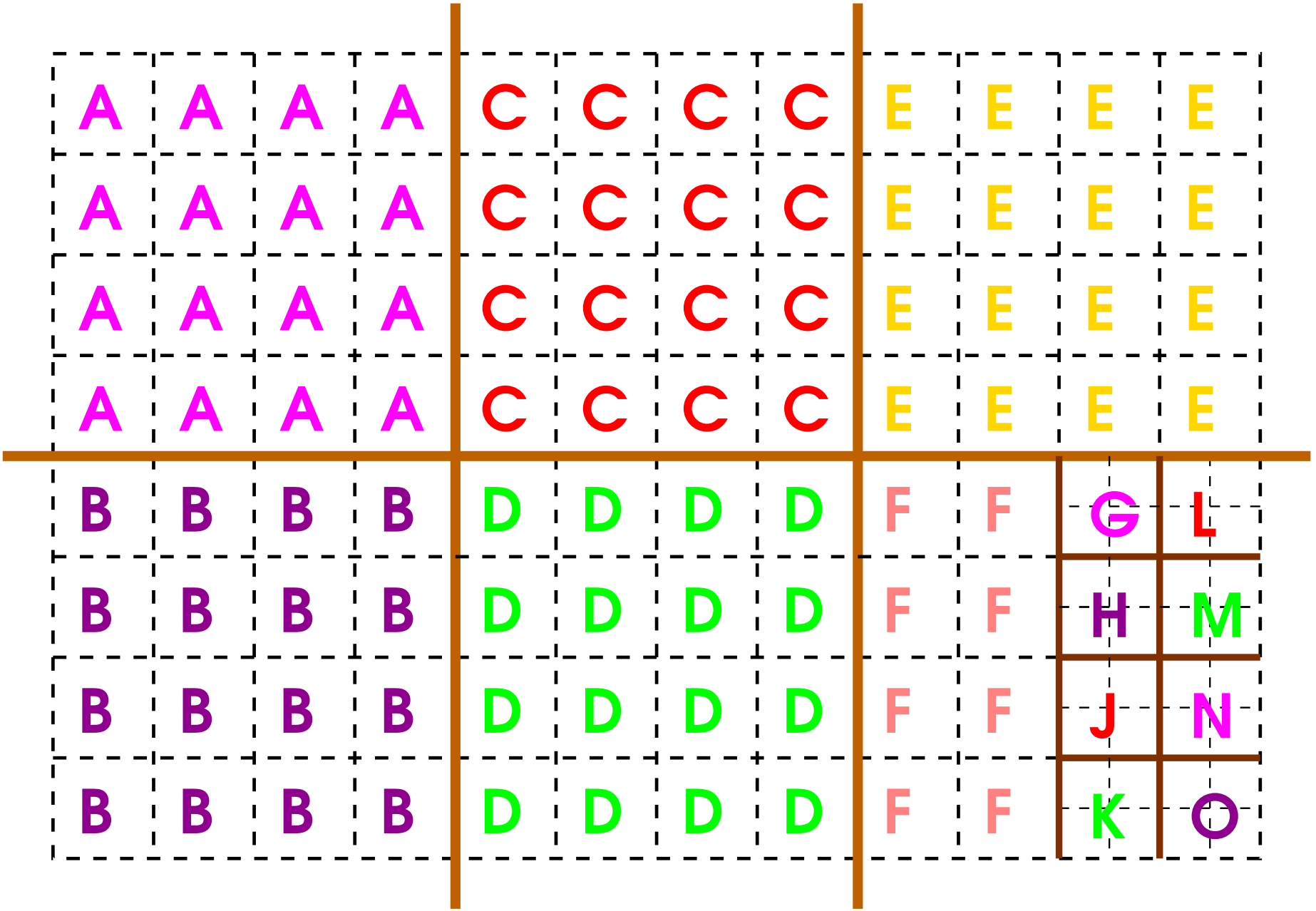
- More tedious and error–prone to program

  E.g. multi–grid, mesh refinement, coordinate transformation, ...

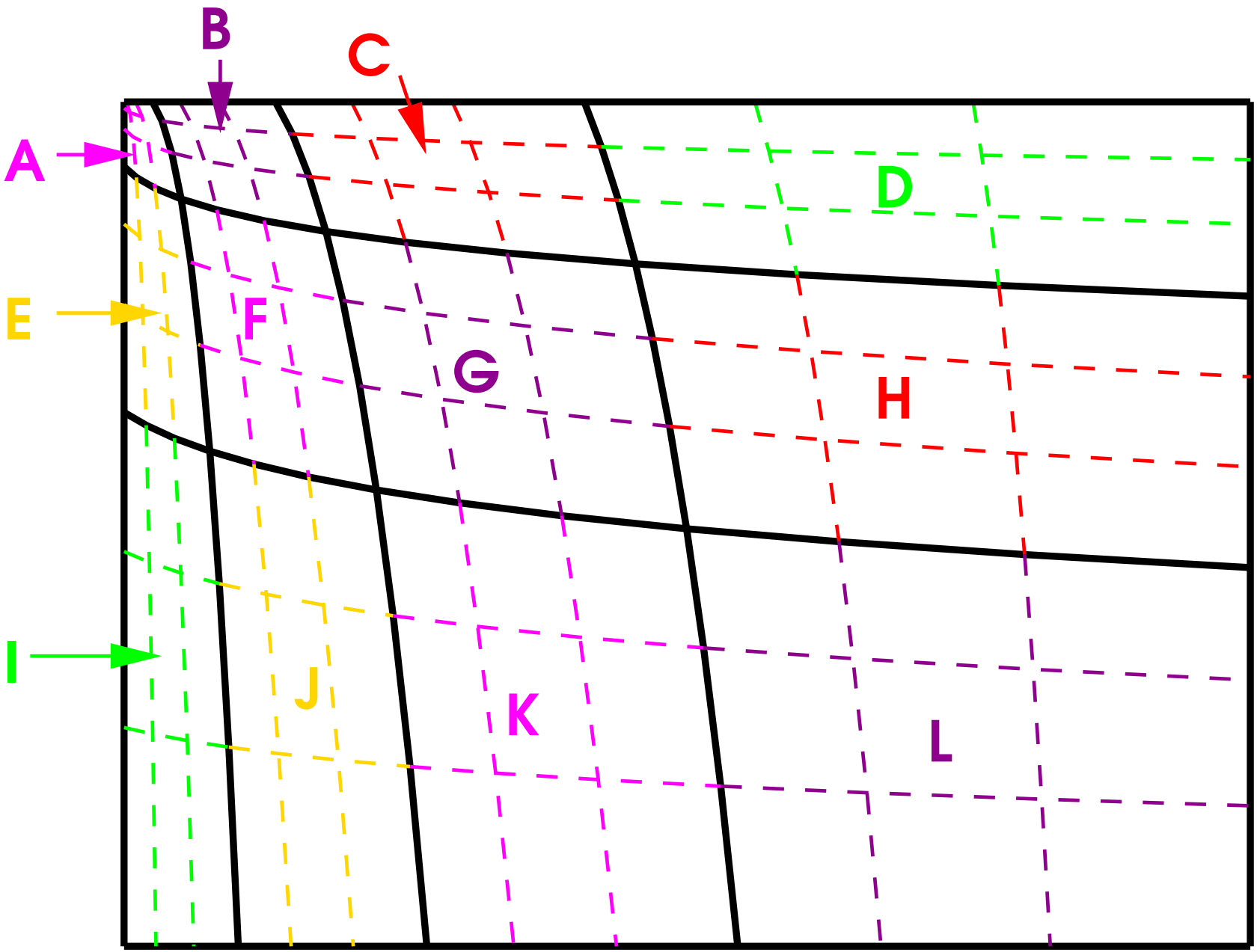
Irregular Partitioning



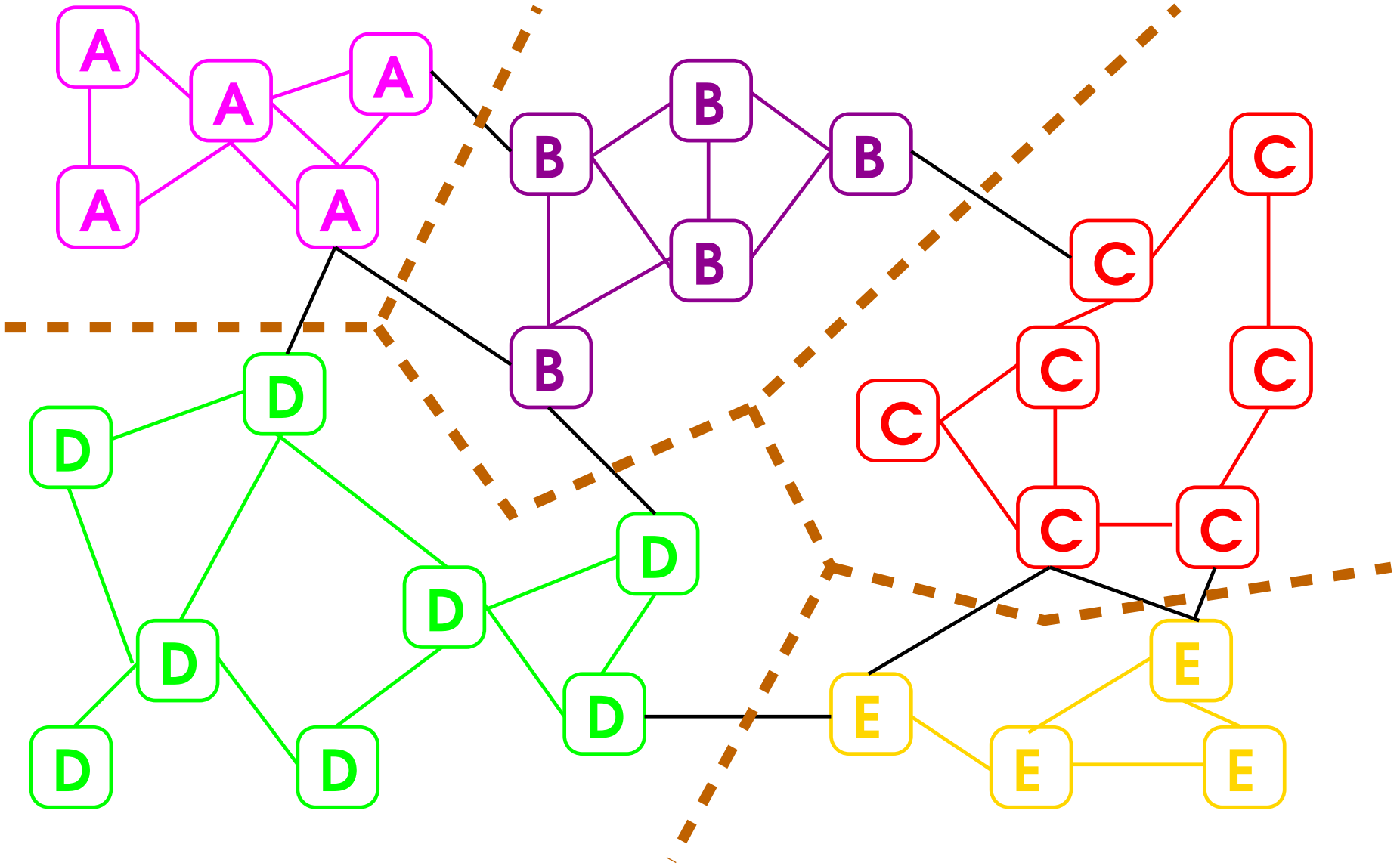
Mesh Refinement



Transformed Mesh



Graph Partitioning



Problem Partioning (4)

- As always, start with the simplest approach

Time each thread and count communication
Estimate the possible improvement

- Then estimate the extra complexity involved

Allowing more time for debugging than you expect

- Complicate the program only if worthwhile

Dataflow Models (1)

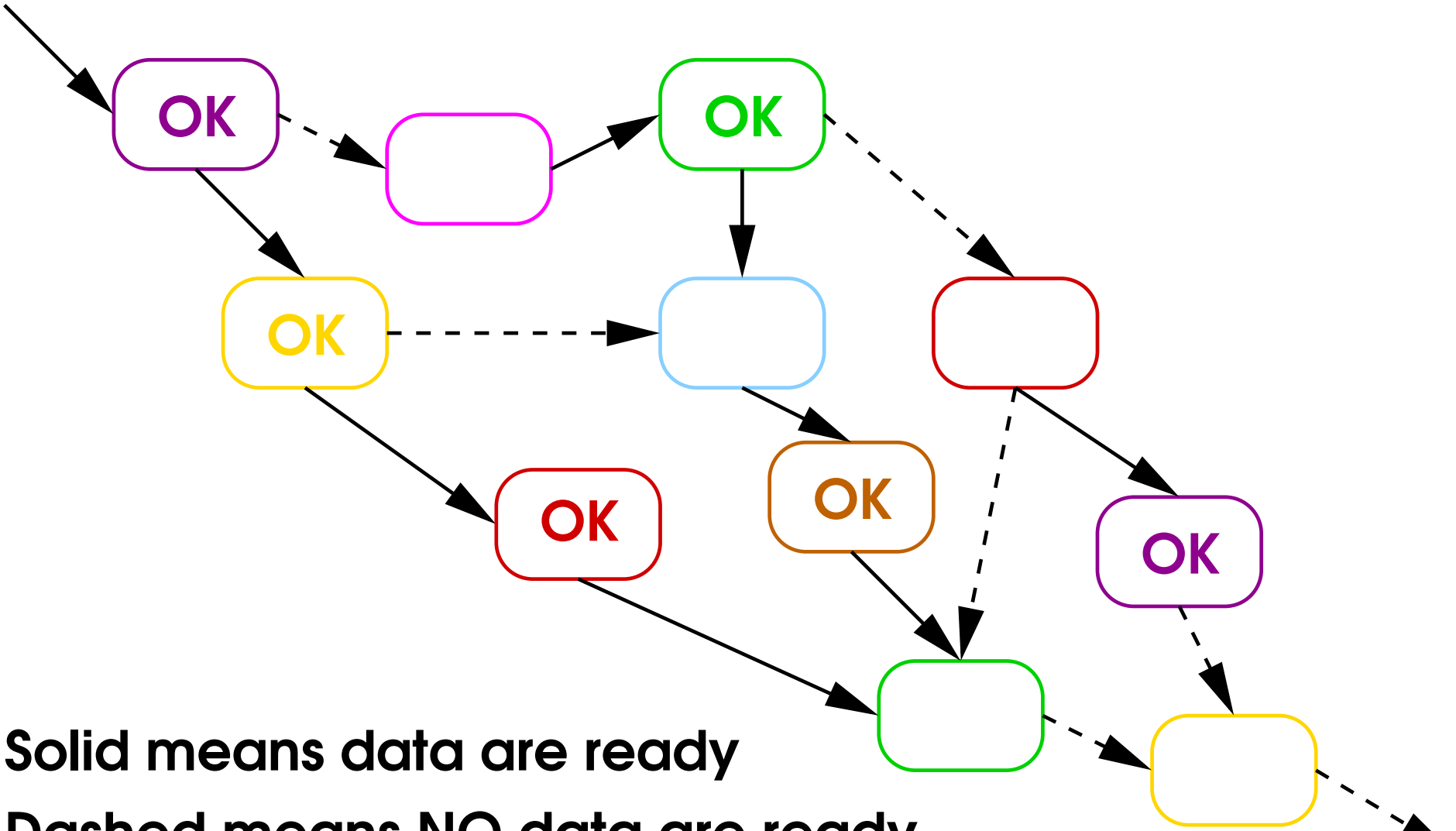
Reminder: useful for irregular problems

- 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

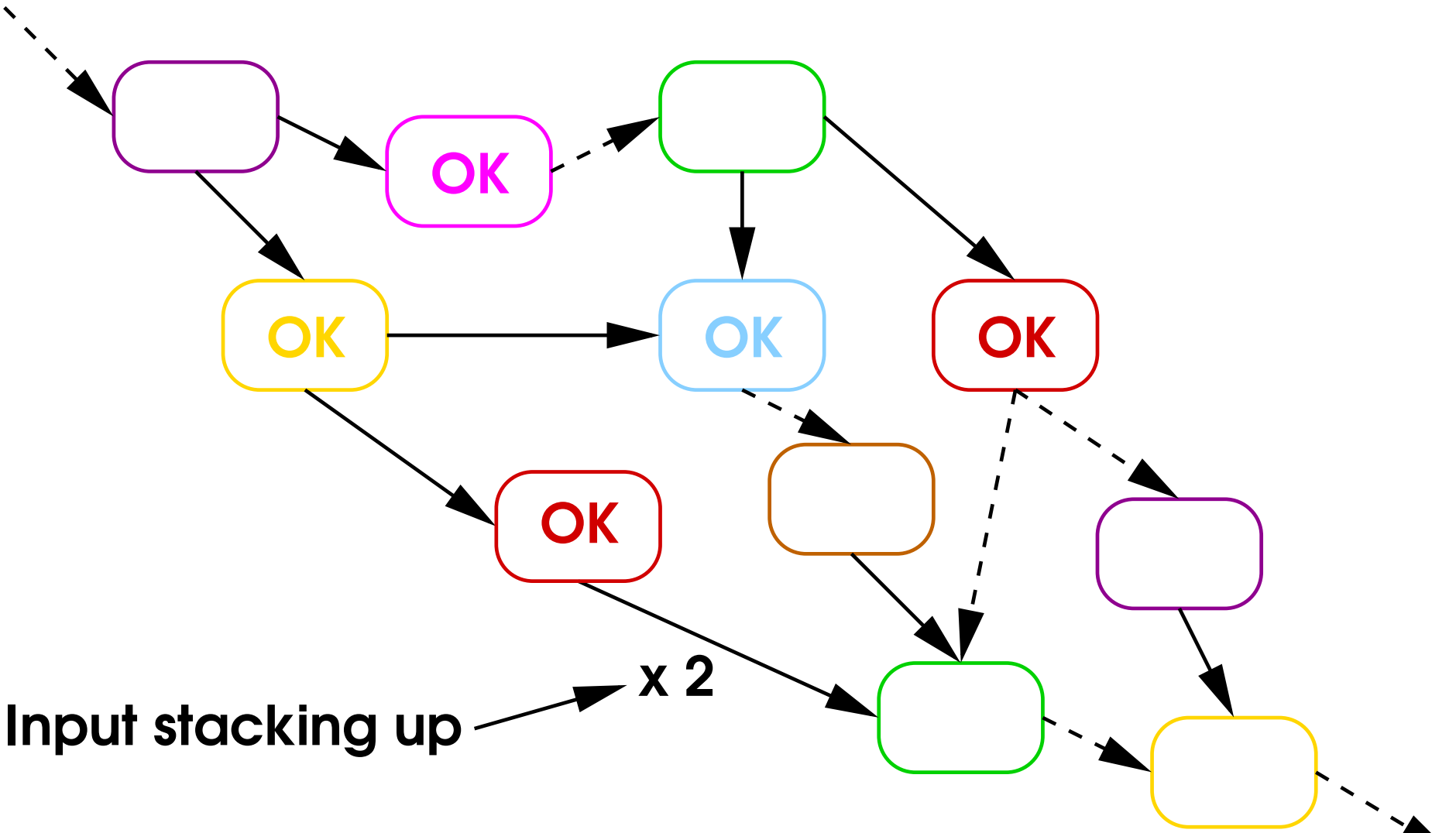
Dataflow (Step N)



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

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

Dataflow Models (3)

- The approach can make design a lot simpler

With a much higher chance of successful debugging
Only the network and data flow affects correctness

- The scheduling affects the efficiency

I.e. the average parallelism actually delivered
It separates correctness and efficiency

And it maps irregular problems to gang scheduling
I.e. to run thread pool problems on HPC systems

Lock-Step vs Asynchronism (1)

- A semi–orthogonal aspect of the HPC model

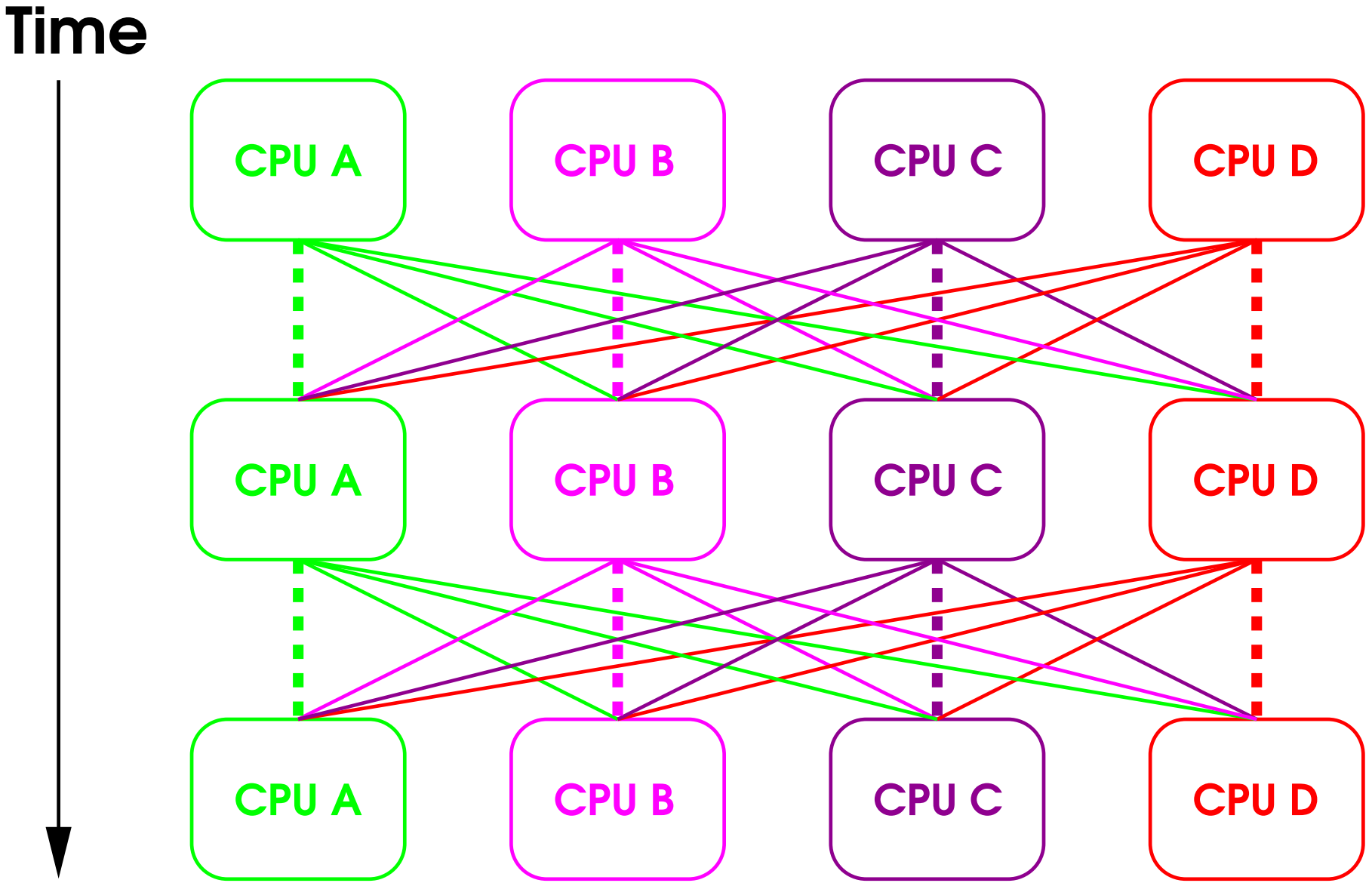
Makes essentially no sense with the vector model
That is almost always in lock–step mode

And dataflow is more natural with asynchronism
But is very important for problem partitioning

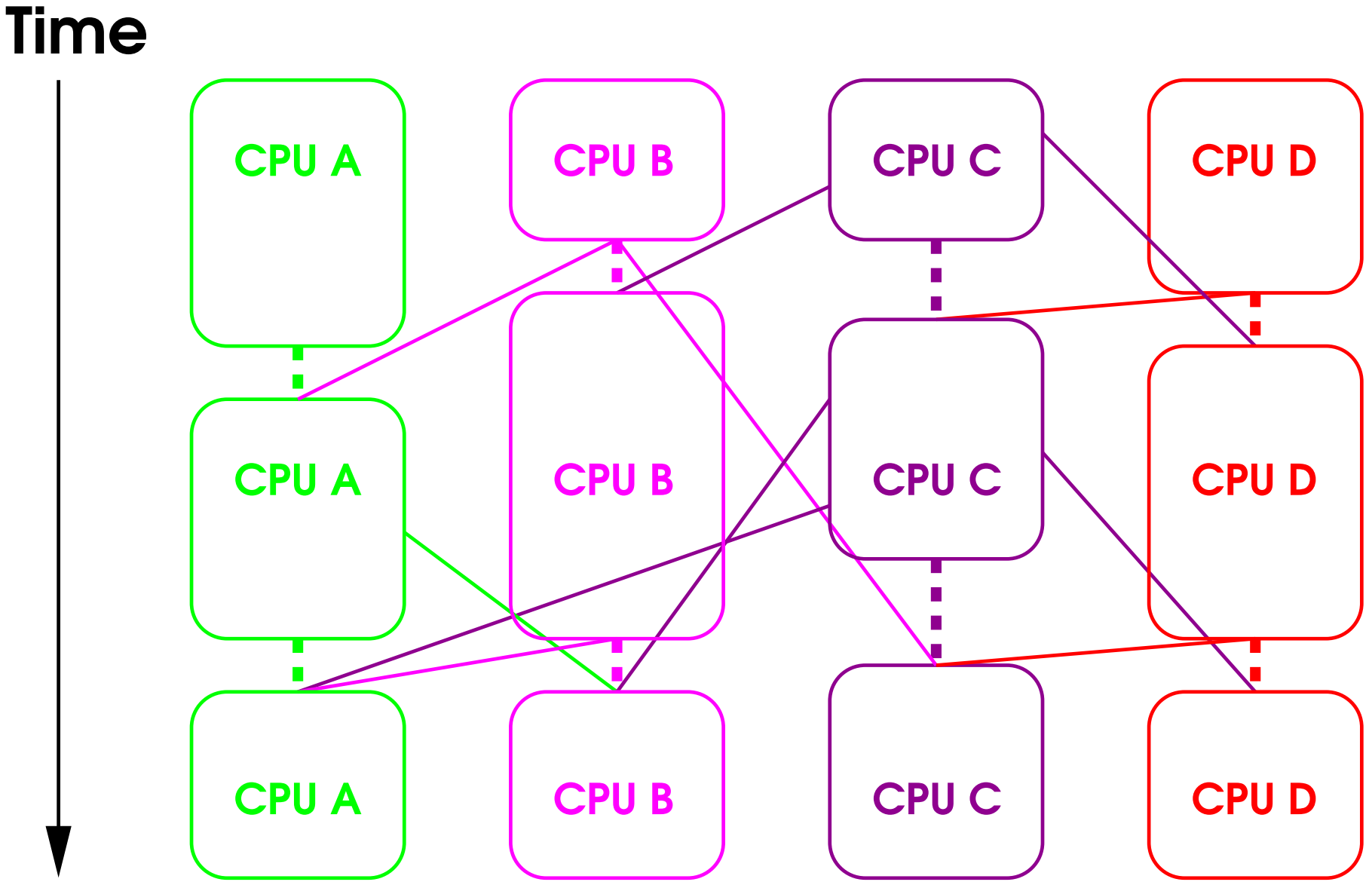
This is should all threads keep in lock–step?
I.e. alternate computation and communication

- Or 'run ahead' until they block waiting for data?

Lock–Step Execution



Asynchronous Execution



Lock-Step vs Asynchronism (2)

Asynchronism looks as if it is more efficient

- Very deceptive and may be completely wrong

Lock–step is easier to debug and tune
Often implemented better – perhaps much better
Explanation of why is well beyond this course

- Just note that you have two options here

Use the most natural for your problem

- Don't mix the two, unless you like difficulties

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 the 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

Asynchronism (2)

Modern CPUs are almost all multi–core

- So can reserve some cores for communication

- Also GPUs can execute independently of CPU

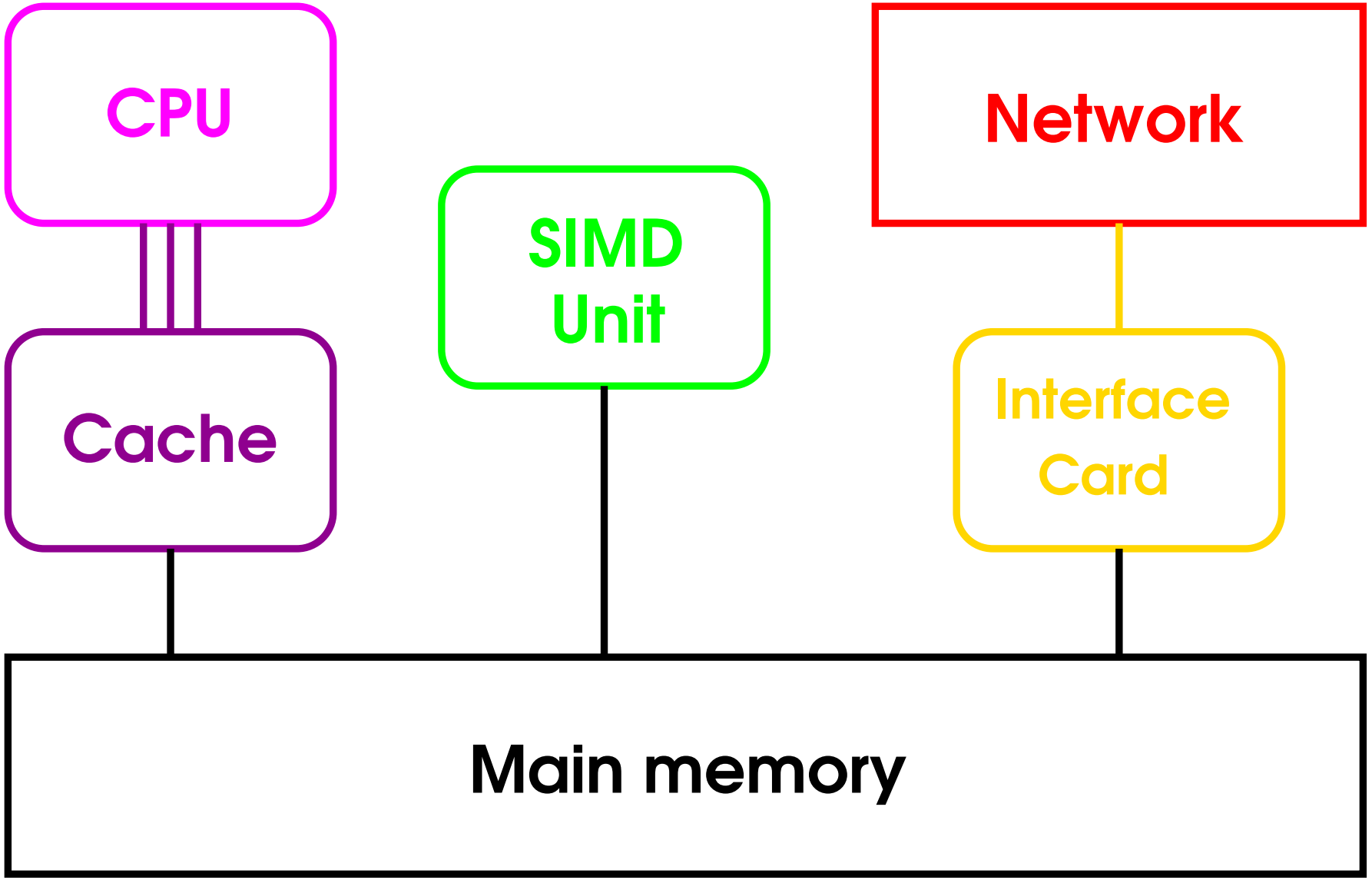
If using only their own memory, no problem

- The memory controller is usually a bottleneck

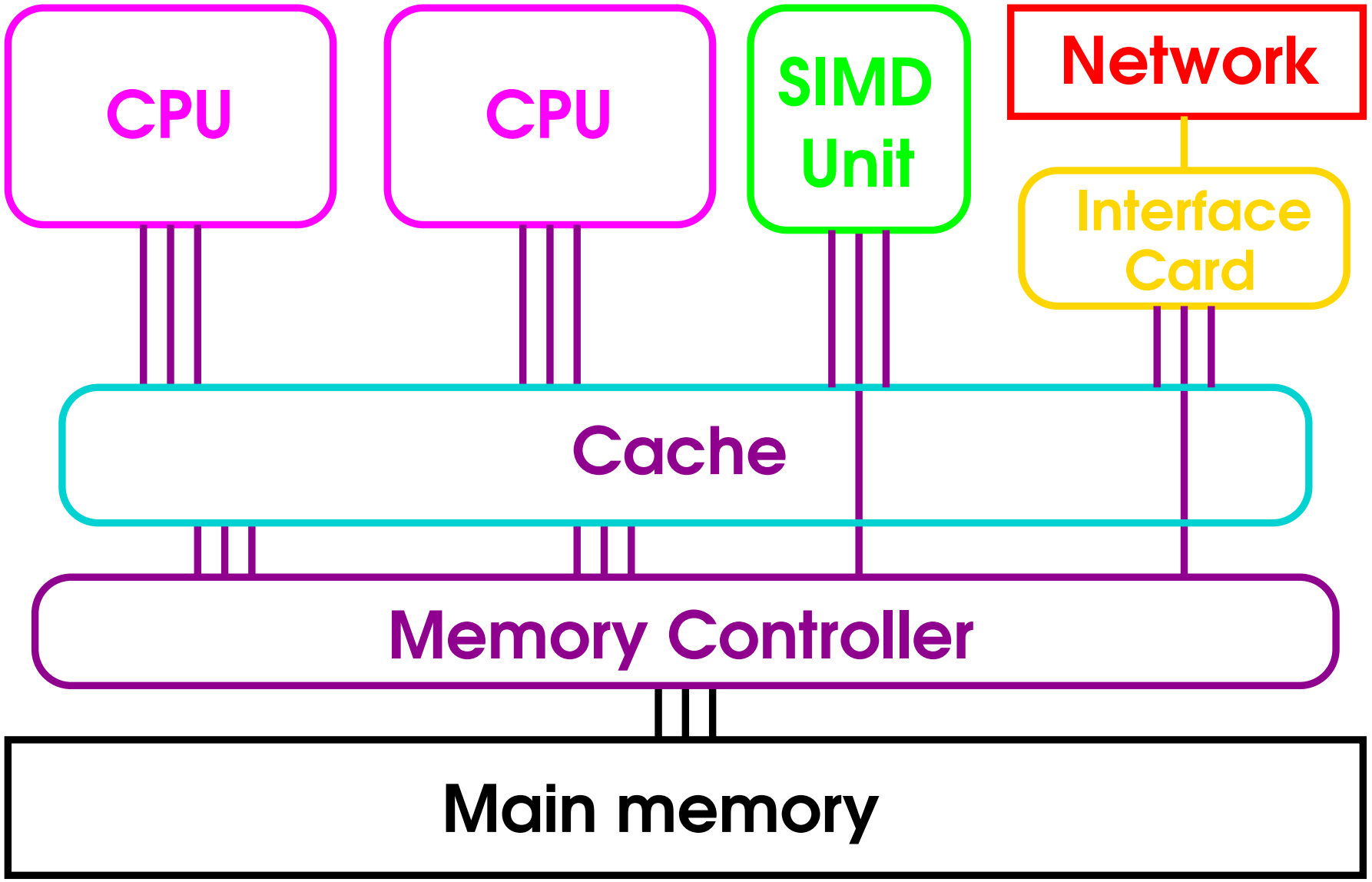
Most CPU–bound codes are actually memory–bound
Can be bandwidth, latency or conflict

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

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

Fat Nodes

Consider a cluster of 4–socket Opterons
Or 2–socket, quad–core Intel, for that matter
On–board communication is fast; off–board is slower

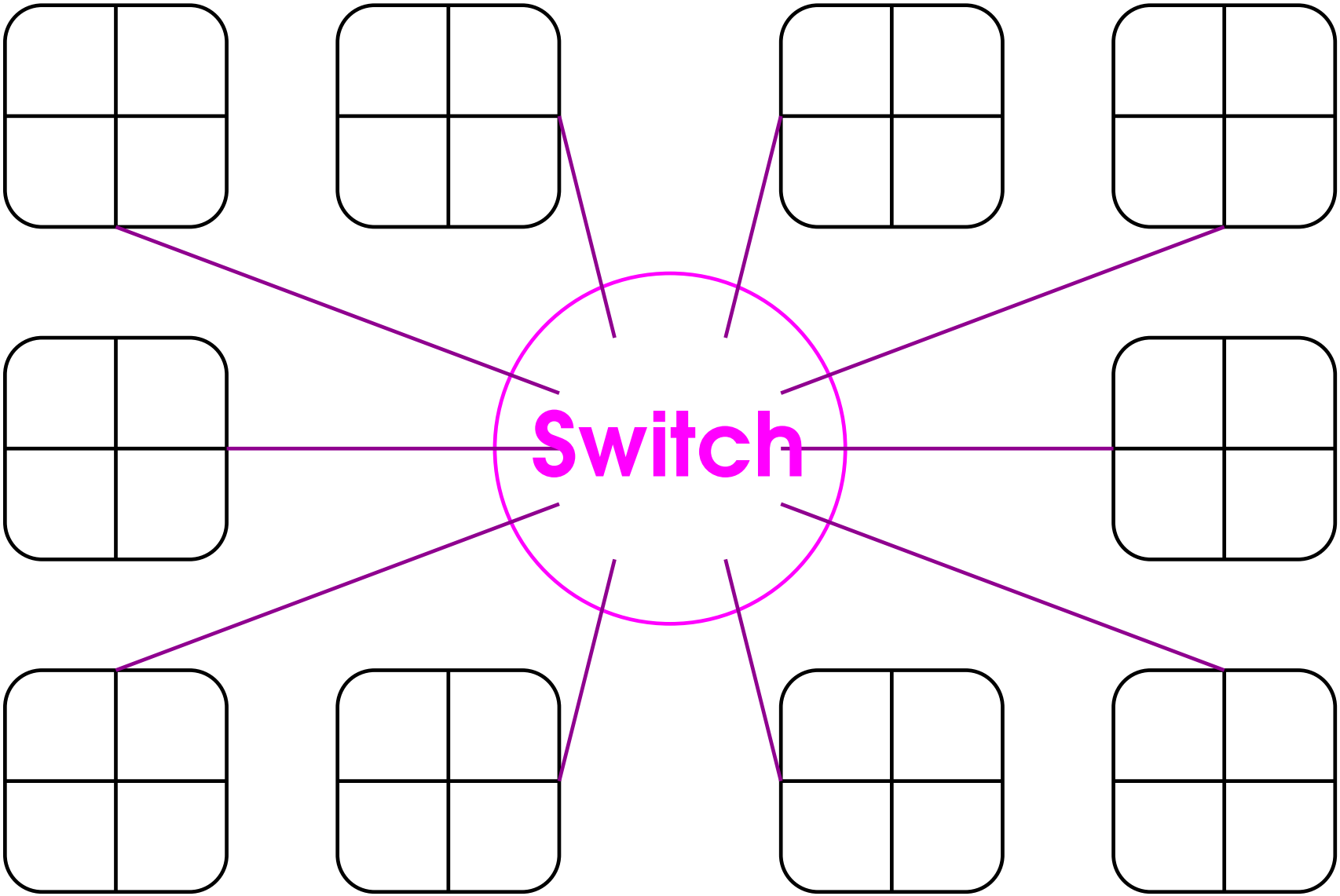
You may find it worthwhile optimising for this case

- It's tricky to do, so don't rush in

Most people won't find that it is worthwhile

Just a special case of a non–uniform topology
There is a lot about topologies in the literature

Fat Nodes



Topologies (1)

Optimising for them is an utterly foul task
Switches and most networks are fairly efficient
$\Rightarrow$ Most people don't bother with topologies

Efficient:
    Single switch, fat tree, hypercube, 3–D torus
Tolerable:
    2–D torus, 3–D grid, Krautz graphs (perhaps)
Problematic:
    2–D grid (mesh, lattice), twisted ladder
Dire:
    1–D torus (i.e. ring), chain (1–D grid)

Topologies (2)

Papers / Web often talk about the diameter
i.e. maximum number of hops between nodes

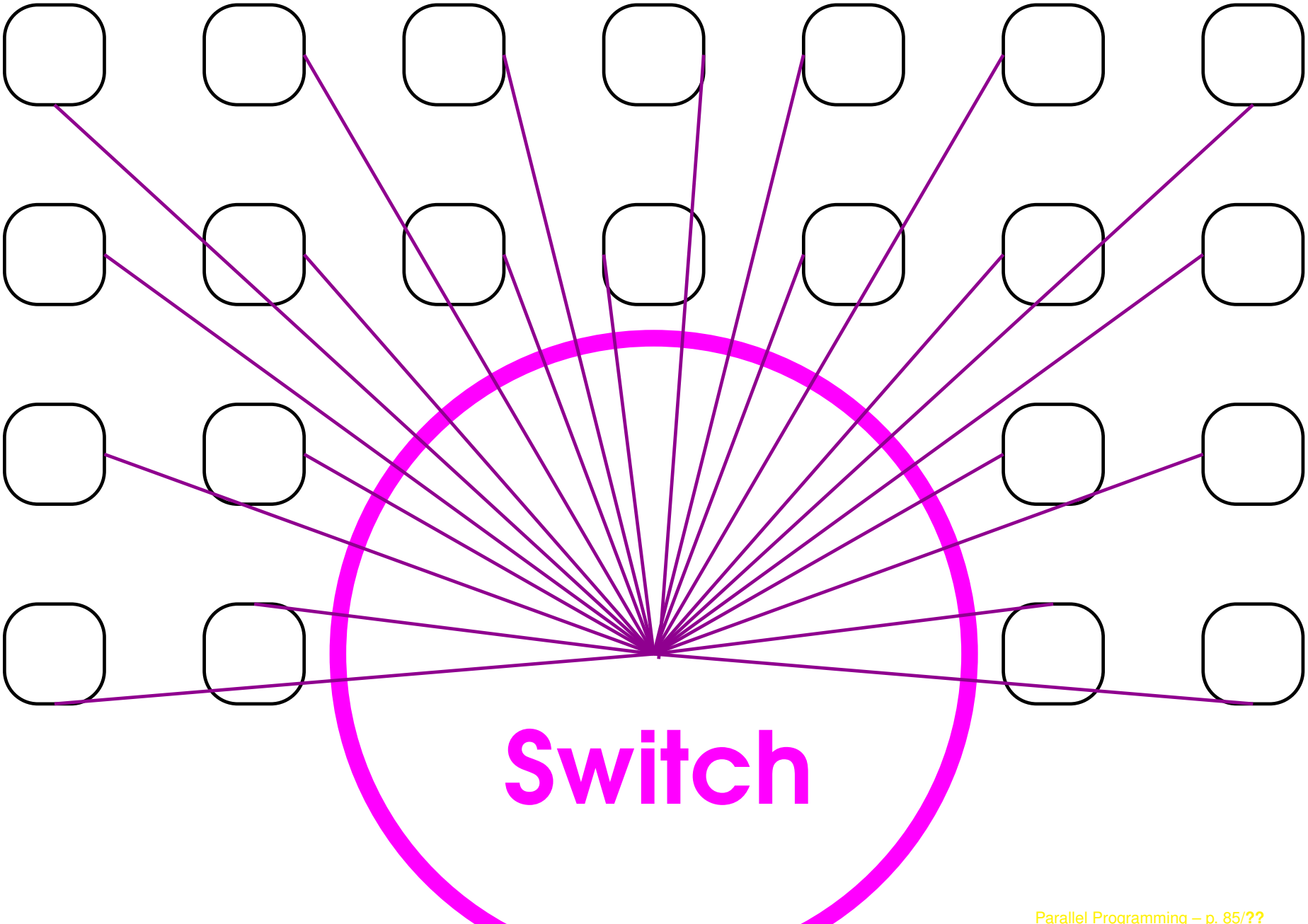
- More often, path congestion is more important

Problems of 8–socket Opteron and 4–socket Intel

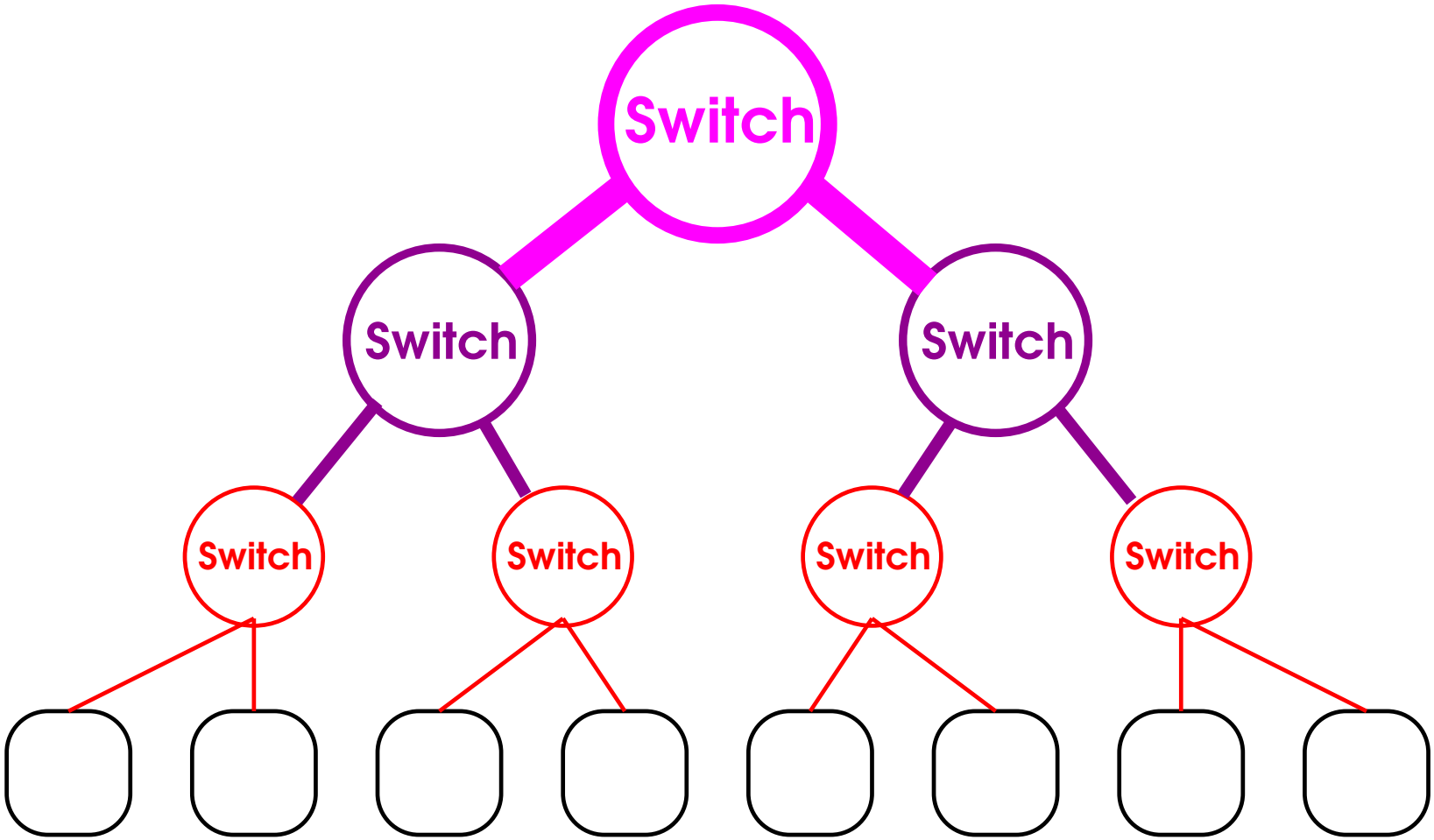
Drawing routing diagrams is left as an exercise
Packets pass singly through links and nodes
and a whole path must be free for a transfer

- Ask for help if you have problems in this area

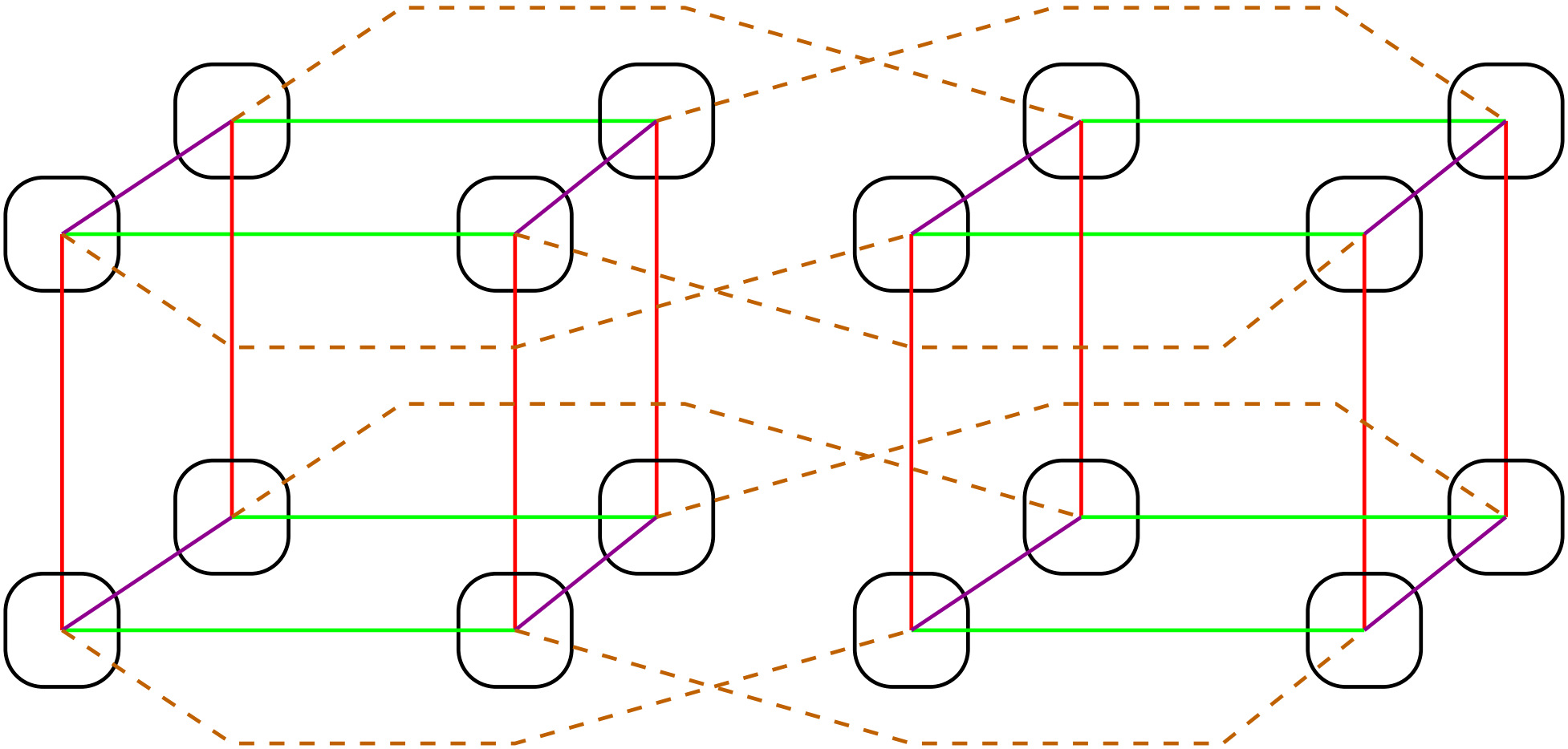
Central Switch



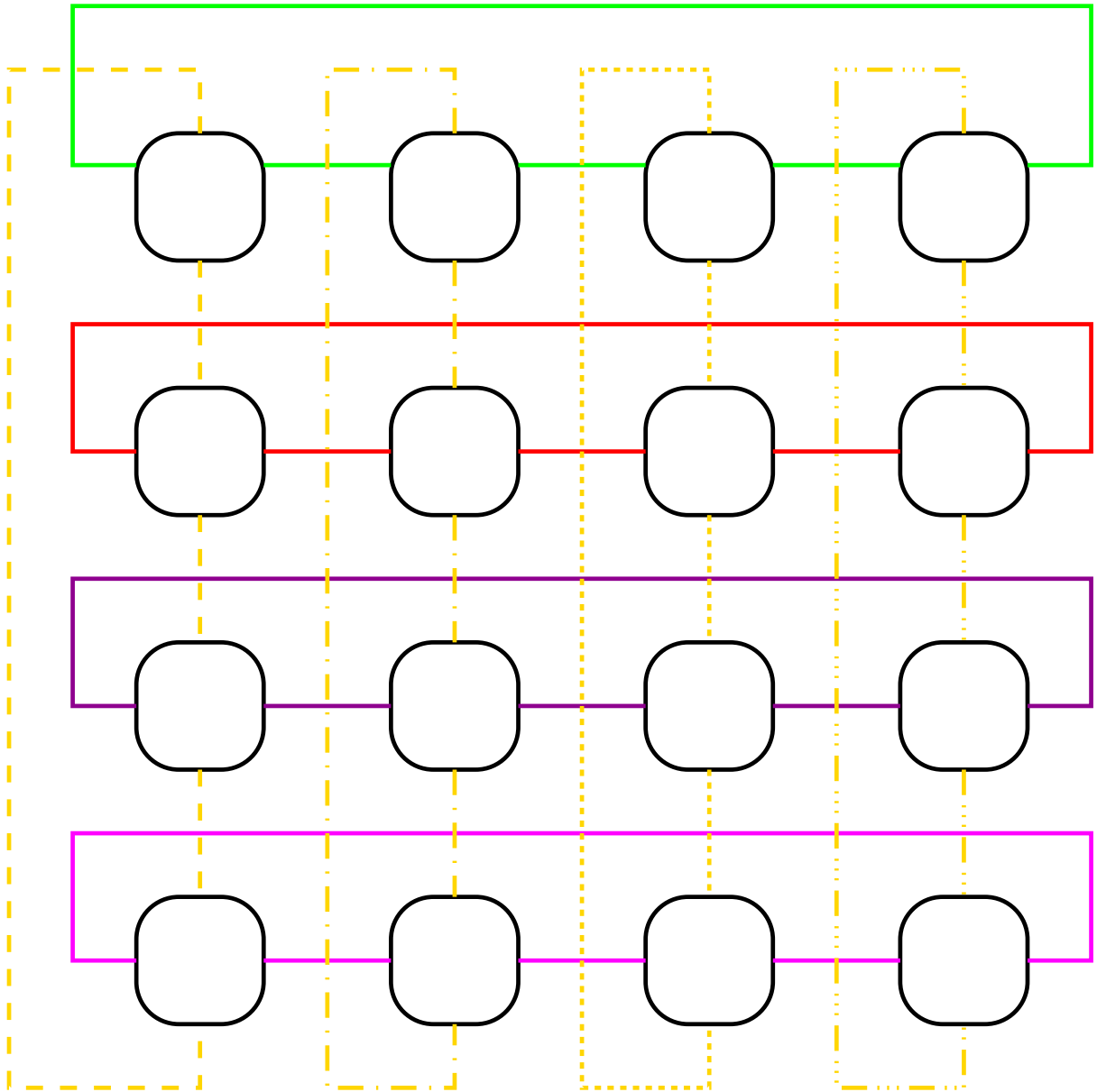
Fat Tree



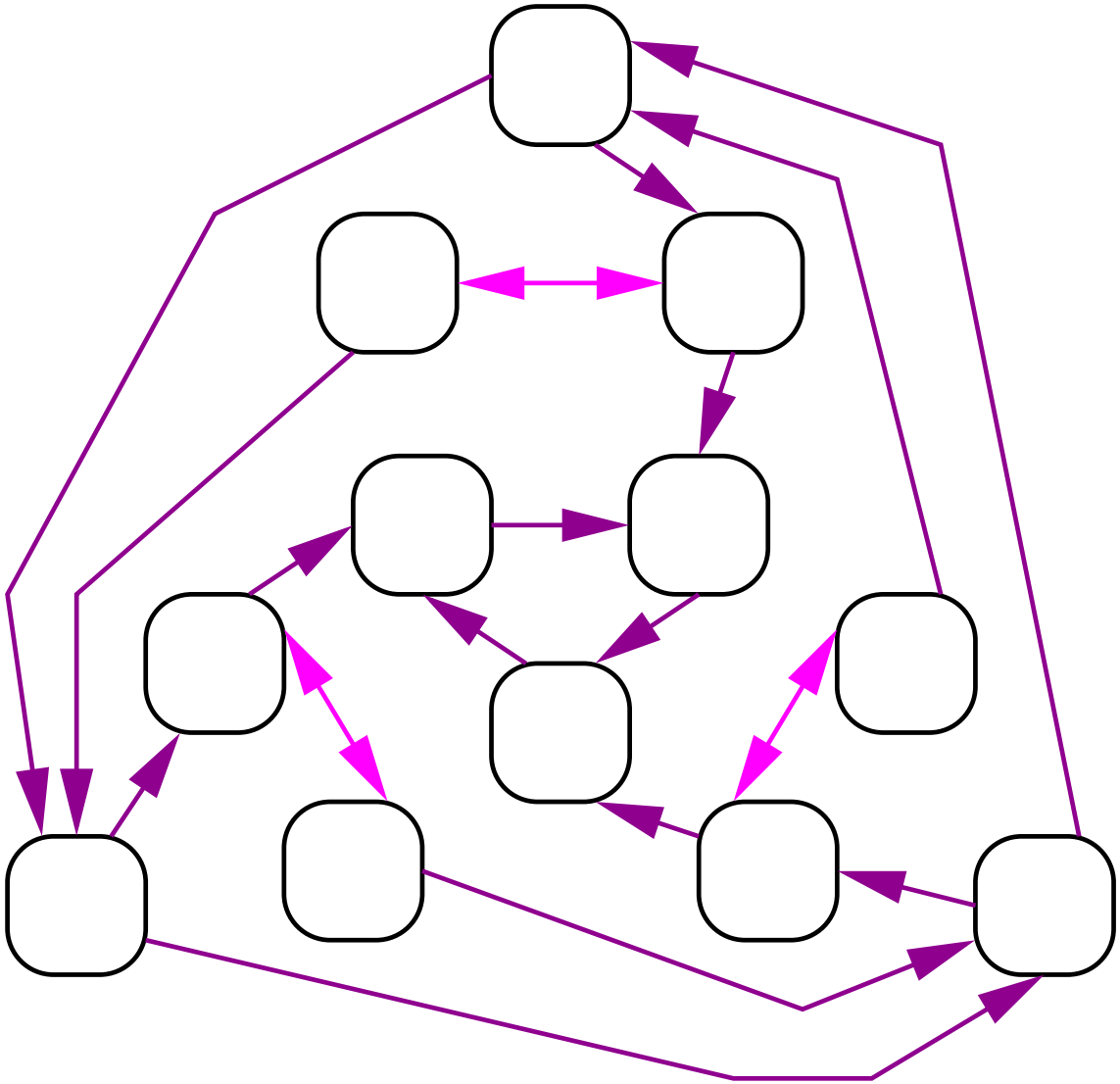
4–D Hypercube



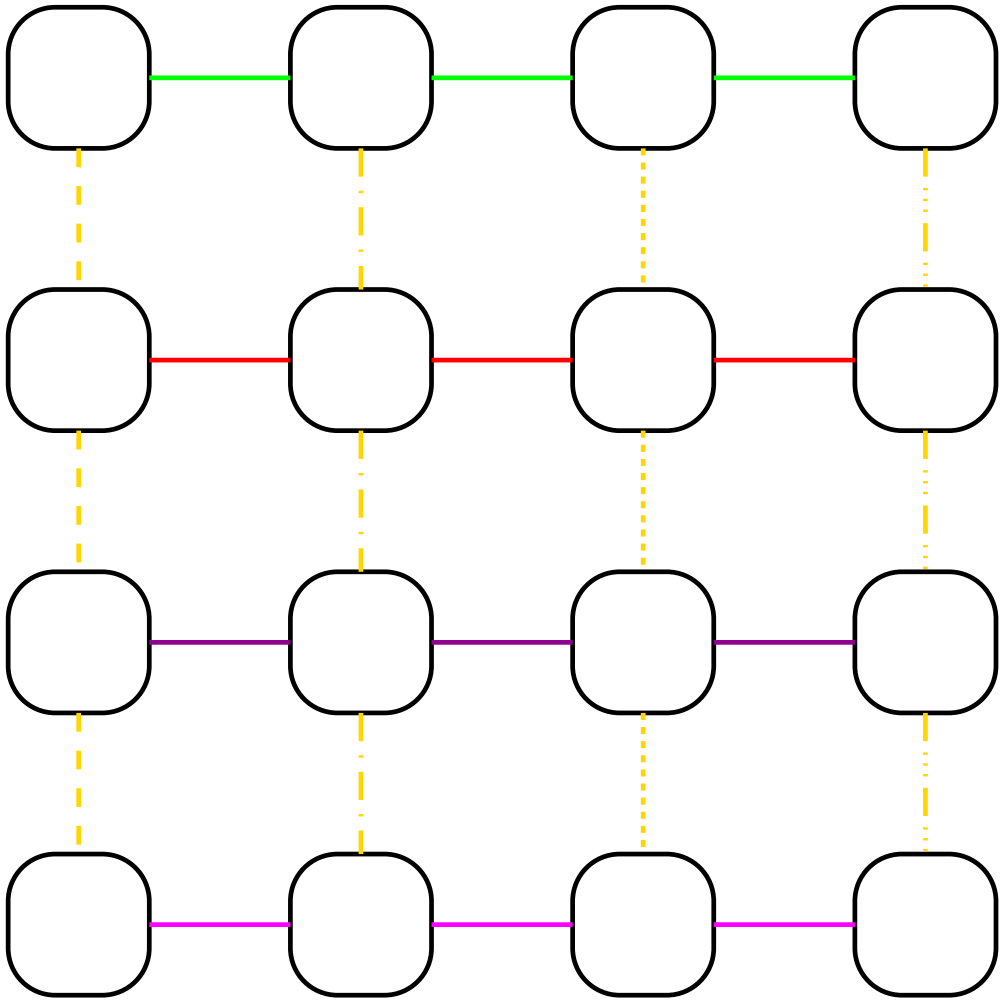
2–D Torus



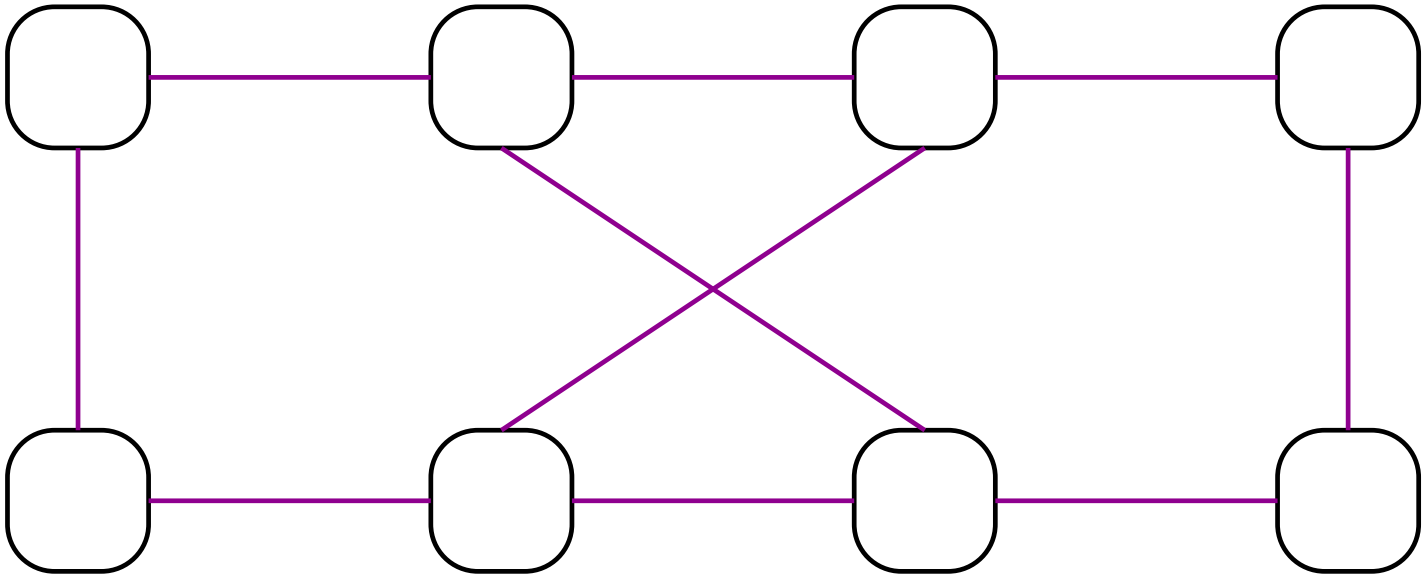
Krautz Graph



2–D Grid/Mesh



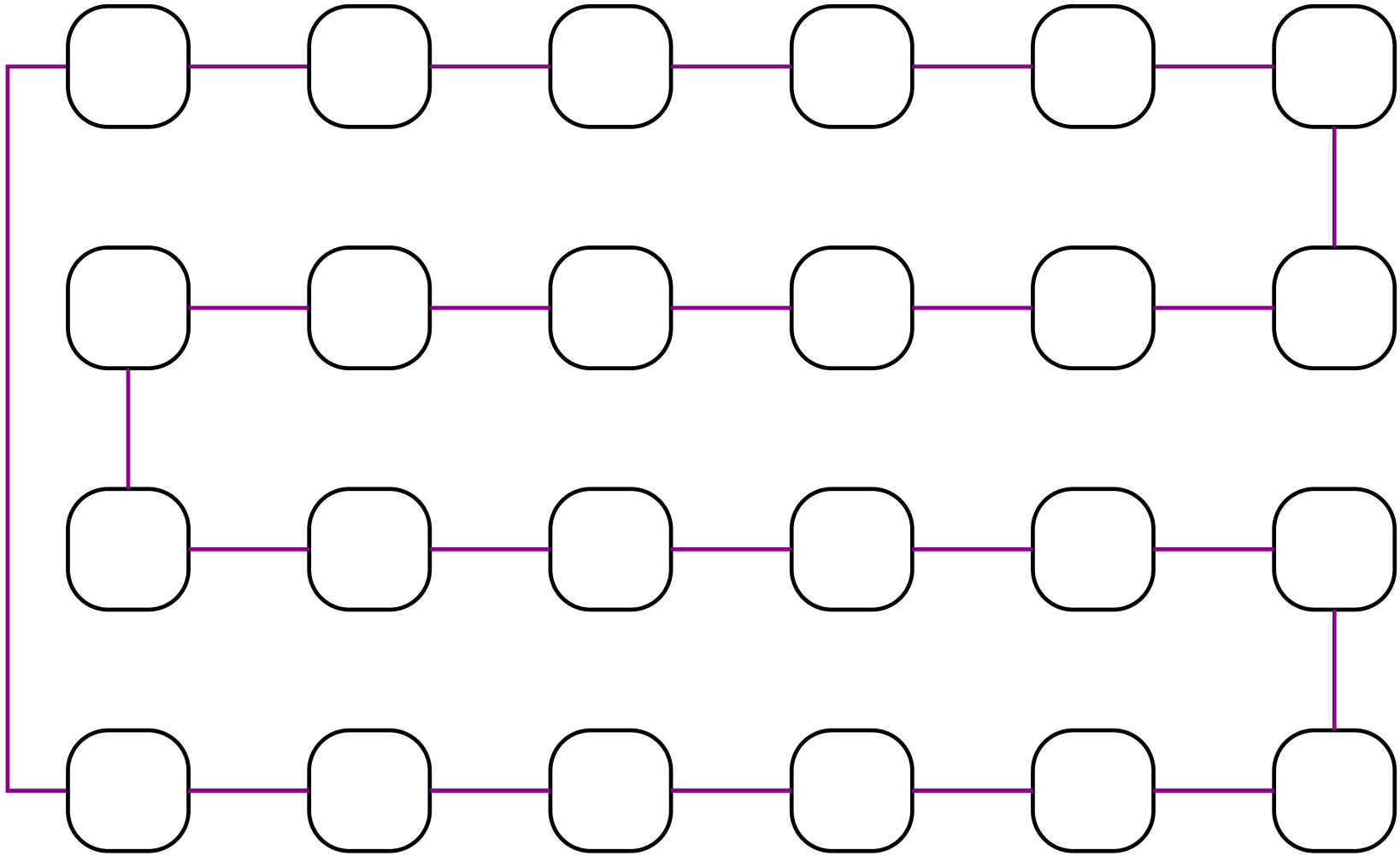
Twisted Ladder



Most 8–socket Opterons use this

Some use a grid (i.e. without the twist)

1–D Torus (Ring)



1–D Grid (Chain)