Parallel Programming (1)

Introduction and Scripting

Nick Maclaren

nmm1@cam.ac.uk

February 2014

Introduction (1)

This is a single three–session course
Each session follows on from the earlier ones

- Some people will drop out – not a problem

Often because they need only some of the course
It is designed to be useful even when people do that

⇒ But please fill in a green form

Do that even if you aren't certain

Introduction (2)

- This part starts with an introduction
Running complete serial programs in parallel
And a brief overview of parallel programming

- Second part is parallel programming proper
Including currently used parallel environments

- Third part is on shared memory models
Currently popular, but hardest form to use correctly

Summary of This Part

- A very brief introduction to the background

All jargon used will be explained – but please ask

- Using multiple copies of serial programs

- More complicated structures of programs

- Overview of parallel programming

- Choice of parallel environment

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

The Web Of A Million Lies is a serious underestimate
And, unfortunately, a great many books are no better

Beyond the Course

Parallel/

Courses on MPI, OpenMP and others

There are some more references in the second half

Choosing Options

- Check what other people in your field do

Not always the best, but often the safest approach

- Use a reliable book or course as a guide

Reading list for computer science courses can help
But remember that many push a particular dogma

- Be very cautious of designing from scratch

It is extremely hard even for the best experts

(Not-)Moore's Law

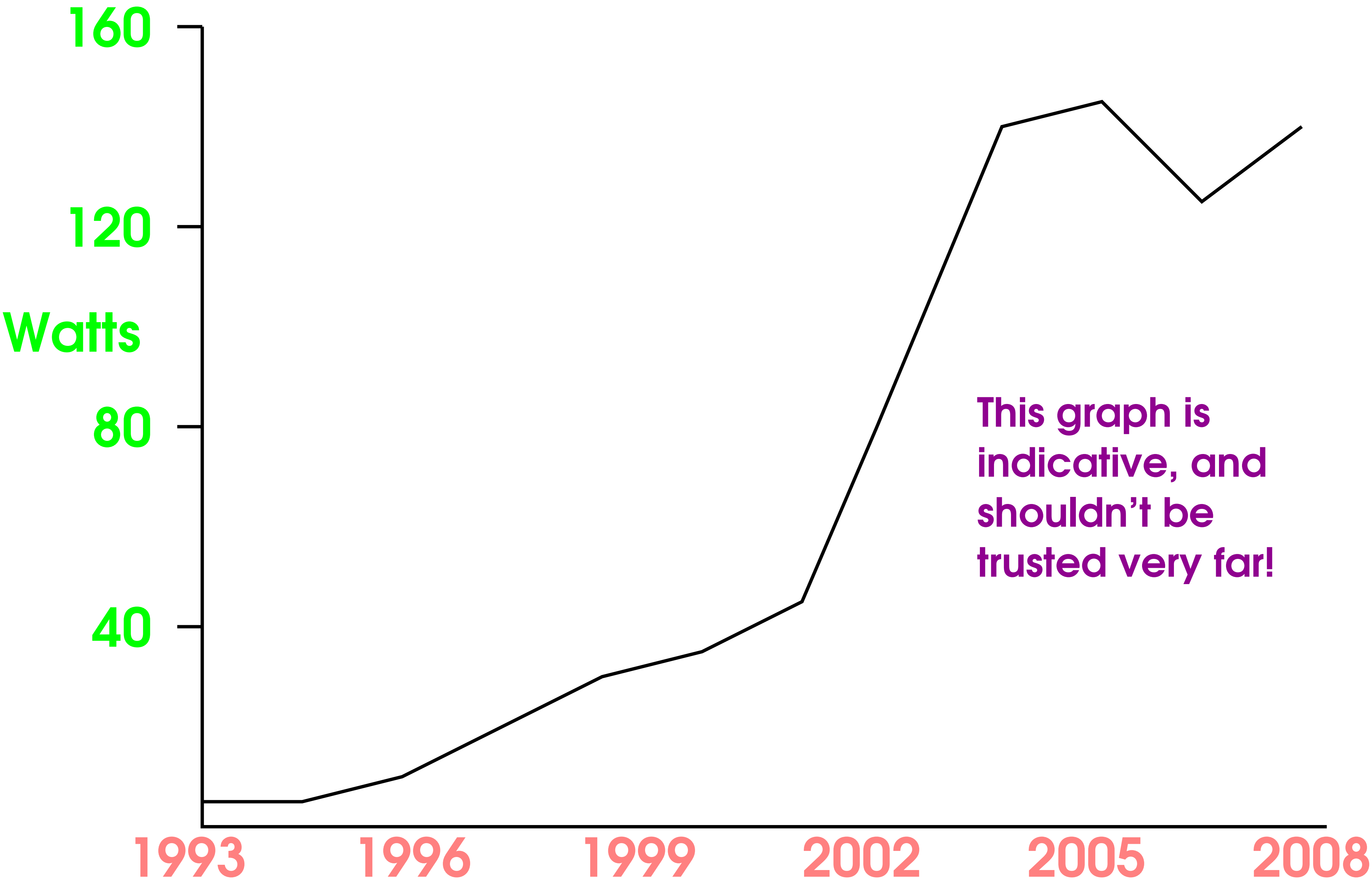
Moore's Law is chip size goes up at 40% per annum
Not–Moore's Law is that clock rates do, too

Moore's Law holds (and will for some years yet)

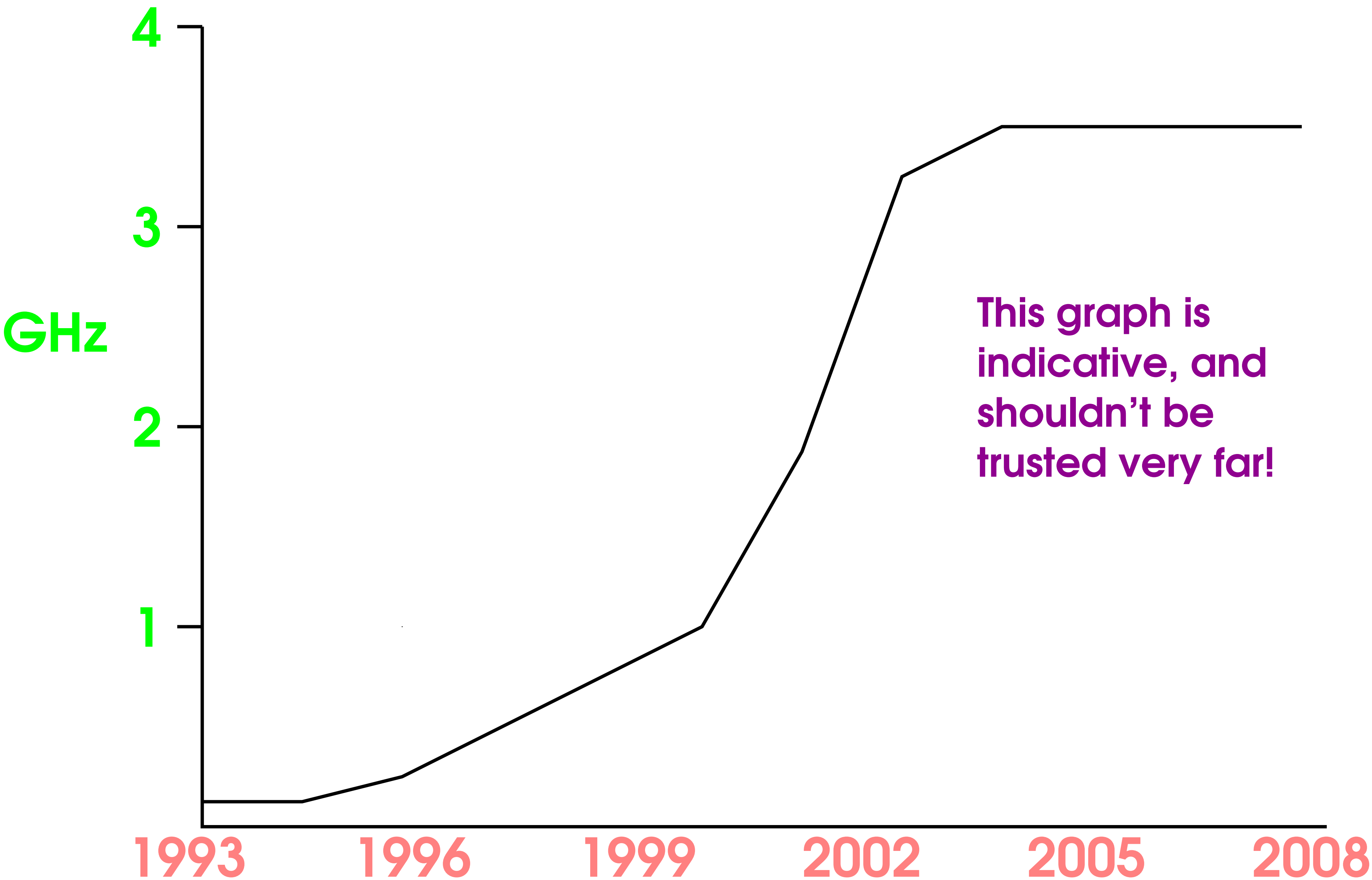
Not–Moore's held until ≈2003, then broke down
Clock rates are the same speed now as then

Reason is power (watts) – due to leakage
See http://www.spectrum.ieee.org/apr08/6106
Figures from (2013) show the graphs remain flat

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%

2010	–	typically 4–8 cores
2015	–	probably ~32 cores
2020	–	perhaps 128+ cores
2025	–	heaven alone knows!

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

- except for GPUs used for HPC codes

Before Starting

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

- For parallelism, they need to work together

This course is mainly for programmers
Will mention some of the general points later

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

- You do need to understand configuration issues
  You don't need to understand the details

Programming Environments

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

- There are dozens of possible combinations

Some are easier than others, some make little sense
The details matter mainly to implementors

- Course is in terms of programming model

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

The Word Scheduling (1)

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 logical thread scheduling

Assigns logical threads to system threads
A mid–level task, done by the compiler/ library
OpenMP, CilkPlus, C++11 etc.

The Word Scheduling (2)

- Third meaning is system thread scheduling

Assigns threads (kernel and user) to cores
Suspends threads to take interrupts
A low–level task, done by the kernel core

First two (partially) controlled by the programmer

Third by the sysadmin – needs privilege
Often as part of system configuration – i.e. fixed
That is a very nasty area, and not covered further
Some remarks later, on affinity control

Status Report

Done a very brief introduction to the background

Now using multiple copies of serial programs

Using Many Serial Programs (1)

You do not always need to rewrite serial programs
Often you can run many copies of them in parallel
Two main methods of doing this:

- Farmable problems – independent, serial tasks

E.g. Monte–Carlo or parameter space searching

- Multi–component problems – interacting programs

E.g. process pipelines, Web browsers
Many commercial scientific applications are like this

Using Many Serial Programs (2)

You need to write a controlling program or script
Also called a harness or controller
Best language for this is usually Python

There is a separate course on this:
MultiApplics/

The following is only a summary of it
With some extra information for parallelism

Basic Master-Worker

- Parent application runs as controller

Manages several jobs in parallel

- It creates suitable jobs 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

- Think of Web or FTP servers, job schedulers etc.

Farmable Problems

- A large number of independent, serial tasks

Think Monte–Carlo or parameter space searching
Can use almost any system, including PWF/MCS/DS

- Matches about half of scientific computing needs

- The jobs are just ordinary, serial programs

You may need to tweak their input and output a little

Use a master/worker design, with a simple master
also known as controller or harness
It just runs separate jobs, that do the actual work

A Very Common Design

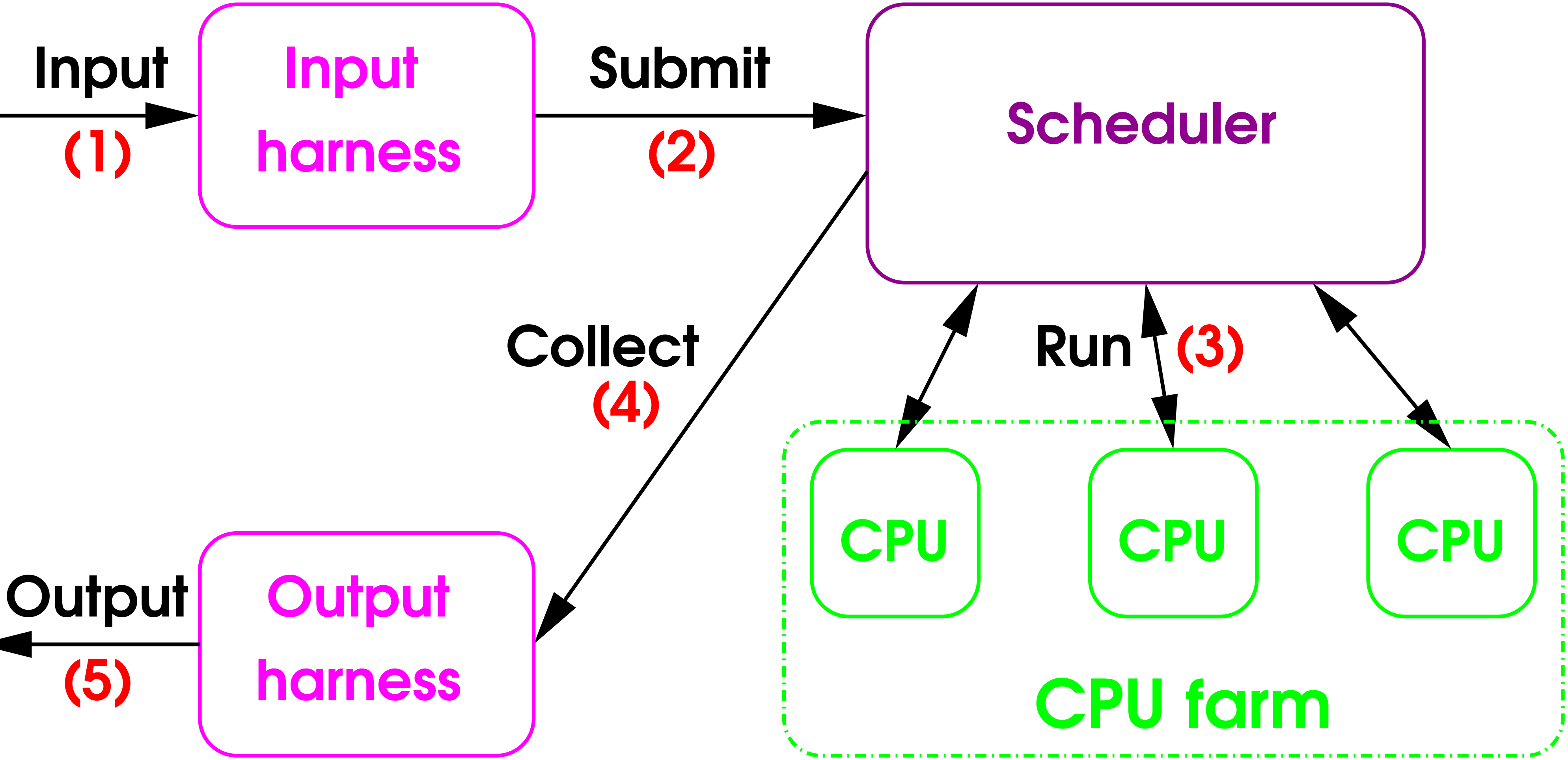
- Possibly interactive program creates jobs
Sets up their input and submits them

- ‘Job scheduler’ runs the queued jobs
It is a controller used for spawning processes

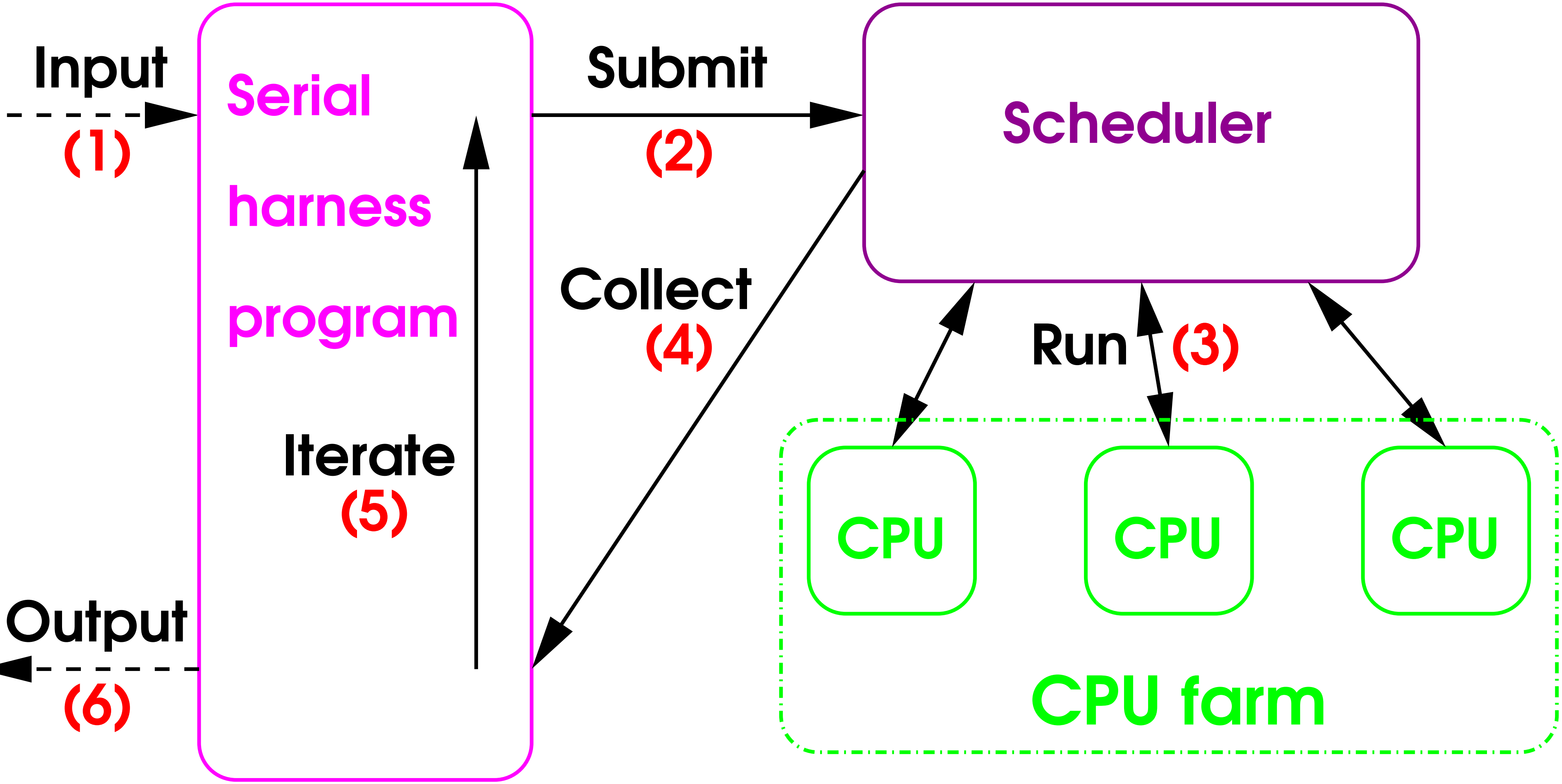
- The jobs are serial and batch
I.e. using one CPU, and are non–interactive

- Possibly interactive program checks completion
Reads their output and creates the results

Manual (Interactive?) Harness



Iterative Harness



How Many Jobs?

No point in exceeding number of CPU cores

- Very often use fewer, even less than half

Many reason is limited memory performance
Limits on bandwidth, accesses per second etc.
Rarely enough to keep all cores running at full speed

- Also watch out for running out of resources

E.g. memory, swap space, disk space, I/O capacity
Can cause other programs to misbehave or fail

⇒ Don't try to do too much at once

Shared Systems

Anything used by other people or for running services
Includes desktops used for your own interactive work

Too many jobs can slow down other processes
Also, problems can occur with kernel scheduling

- Most commonly seen with memory intensive ones

Especially with GUIs and some fancy networking
they may hang, misbehave or fail

- Always allow enough resources for other work

nice will NOT usually help – despite common claims!

Choosing a Master + Scheduler

- Best solutions are a job scheduler, Python or MPI

Not advised to use a shell, C/C++ or even Perl
It is much harder and needs much more skill

- Strongly advised not to use threading

Spawning processes looks harder but is simpler

MultiApplics/

Job Schedulers (1)

- Much the best way of running CPU farms
GridEngine, Condor, LSF, PBS etc.

That is a pre–debugged controlling application

- A sysadmin must install and configure them
Doing those needs privilege (i.e. root access)

- Configuring them is tedious and can be tricky
For just farmable applications is usually easy
Ask for help configuring systems / job schedulers

Job Schedulers (2)

- Using schedulers is typically much easier

Farmable jobs need only one CPU core

Usually create a script file, and submit by a command
It may start with the description of the job
Followed by the commands to execute in the job

If the job description is by command parameters
Just create another script file to use the command

Check if jobs have finished, and then look at output

Using Python (1)

Use the subprocess module and class Popen
Can be done, very easily, in a couple of dozen lines
That includes fairly thorough checking of success

Description of how to do it in:
MultiApplics/

- Note that killing the master is a Bad Idea

At the very least, will lose output and miss checking
Job schedulers can handle that, but are complicated

Using Python (2)

If master on one system and workers on another
Popen should use the ssh command

.ssh on the worker systems should avoid passwords
I.e. set up known_hosts and authorized_keys
I.e. put your username on the master into latter

- Can be problems if master system crashes

Will usually lose output, miss checking and more

- And you must avoid running too many jobs

Using MPI (1)

Generally advised only if you already know MPI
But may well need it for your non–farmable problems
It is the way to use clusters for large problems

Installing/configuring easier than for job schedulers
It is essentially trivial on a multi–core system
And you need only a hostfile for a cluster

Installing MPI rarely needs any privilege
E.g. the usual open–source versions don't

Using MPI (2)

Actually using it is no problem for an MPI programmer

Exercise 2.2 is trivial; 9.1 is much better:
MPI/

If you start jobs using the C system function call
(EXECUTE_COMMAND_LINE in Fortran)

- You must add the checking and error handling

Using it has similar constraints to using Python

Using a Shell

- This is commonly done, but is not advised

It is very hard to code reliable error handling
If you must, use bash, NOT csh/tcsh

Can spawn background processes on local system
in Unix, also erroneously called jobs
Or can use ssh etc. to run on other systems

Handle I/O using files, pipes or named FIFOs

- And remember to watch out for failures

If you don't notice, you will lose data

Status Report

Done using multiple copies of serial programs

Now more complicated structures of programs

Many Tasks at Once (1)

Approach is more general than just farmable
Each job may be different, with dependencies

Simple example is a streaming pipeline as in Unix
Computing use of Henry Ford's production pipeline
Data ⇒ Job A ⇒ Job B ⇒ Job C ⇒ Job D ⇒ output
Yes, Unix pipelines can run in parallel, automatically!

- Get parallelism only if all jobs are streaming

Mustn't read all input first, process and then write

Many Tasks at Once (2)

Many other control structures are possible
MultiApplics/

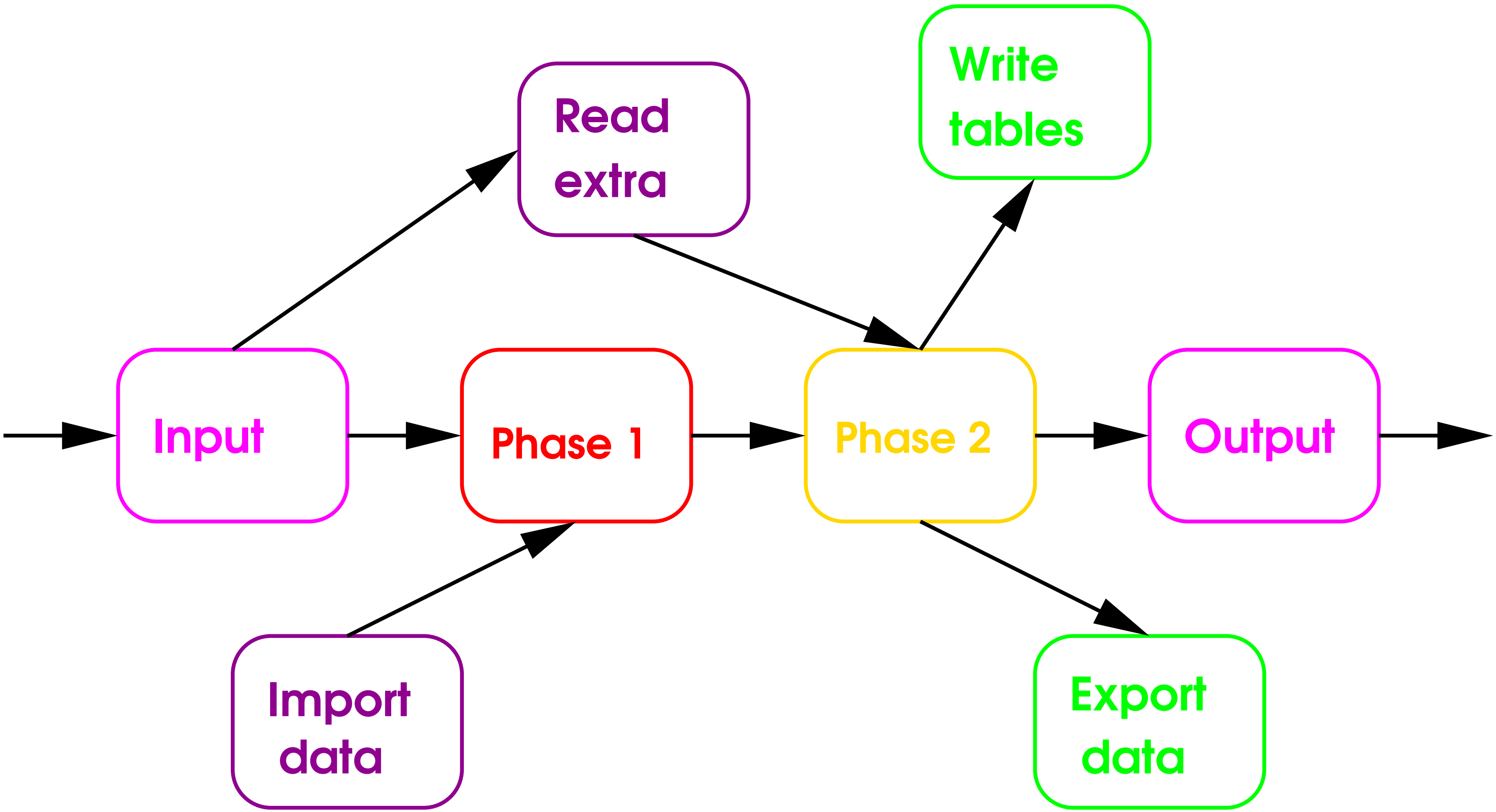
Anywhere a complex analysis has several phases
Common in bioinformatics and many other areas

Note that you don't have to run them in parallel

- Still a good way to design large applications

Many commercial scientific applications do that

Example of Structure



Running in Parallel

- Objective is genuine parallel execution
Equivalent of a manager delegating tasks
The extra performance comes as a result of that

Think of how you can split into separate tasks

- Key factor is must be semi–independent
But note that pipelines are not fully independent

- In practice, uses natural parallelism only
The tasks are large scale components
Consider them as complete sub–applications

Control Structure

- This should not have any cycles in its control flow

I.e. it should be a Directed Acyclic Graph (DAG)

- Use only streaming I/O between processes

Avoid two–way communication if you can

It is possible to write correct cyclic structures
But easy to cause deadlock and livelock

- Treat such problems as parallel programming

We shall discuss that in a little while

Duplex Communication (1)

E.g. process A asks process B for some data
It looks simple, but there are some foul gotchas
Also includes any use of duplex pipes, which exist

- Processes A and B may block each other

So communicate only using atomic transactions

Will describe only the simplest form of these
Works using almost any communication mechanism

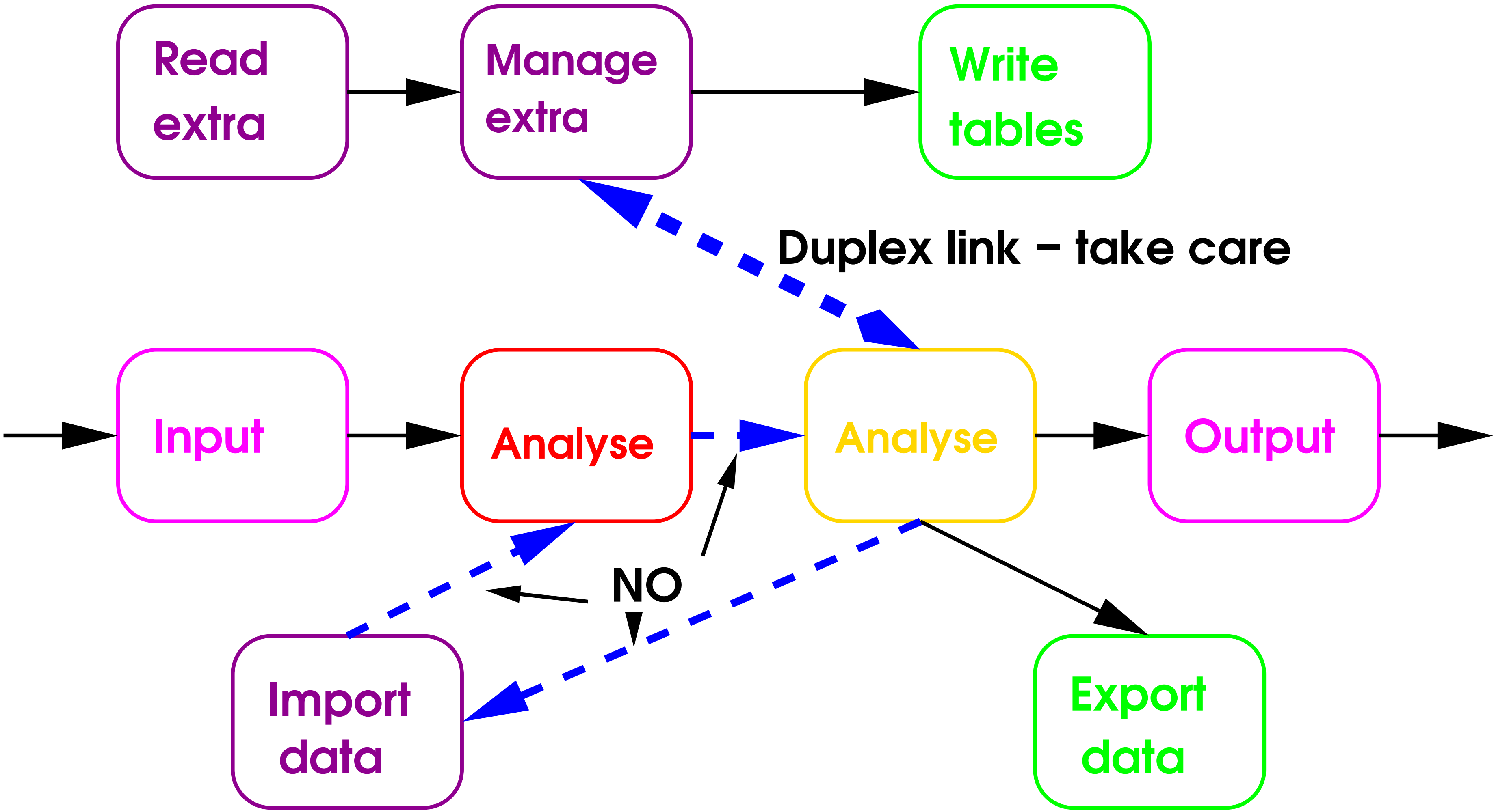
Duplex Communication (2)

1: Process A sends a message to B
2: Process B reads the complete message
3: Process B sends a response to A
4: Process A reads the complete response

⇒ Don't overlap with any other communication
I.e. not between 1 and 4 for process A
And not between 2 and 3 for process B

⇒ Don't interleave reading and writing
Can get deadlock in TCP/IP if you do
MPI will not deadlock, even in that case

Avoid Cyclic Structures



Beyond That?

Can automate many forms of recovery from failure

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

See your job scheduler for relevant features
Can do it in Python or MPI, but take care

- Harness + serial processes 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

Potential Problem

Job schedulers have limited job dependencies
You can implement only some control structures
E.g. unlikely to support even streaming pipelines
Will usually need Python or MPI

- Which is a problem if anything crashes

You don't want to restart from the beginning!
This may not be a problem – so it is left to later

- Frequent backups save a lot of wasted time!

Applies generally, so not covered in this course

Status Report

Done more complicated structures of programs

Now overview of parallel programming

Writing Parallel Programs

Will now cover how to write parallel programs
Often by parallelising an existing serial one

Be warned: parallelism is always tricky
$\Rightarrow$ 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

Why Use Parallelism?

- Most common use is doing many tasks at once

Dominates in commerce – common in academia
Often master/worker, but with communication

- Main other use is for more performance

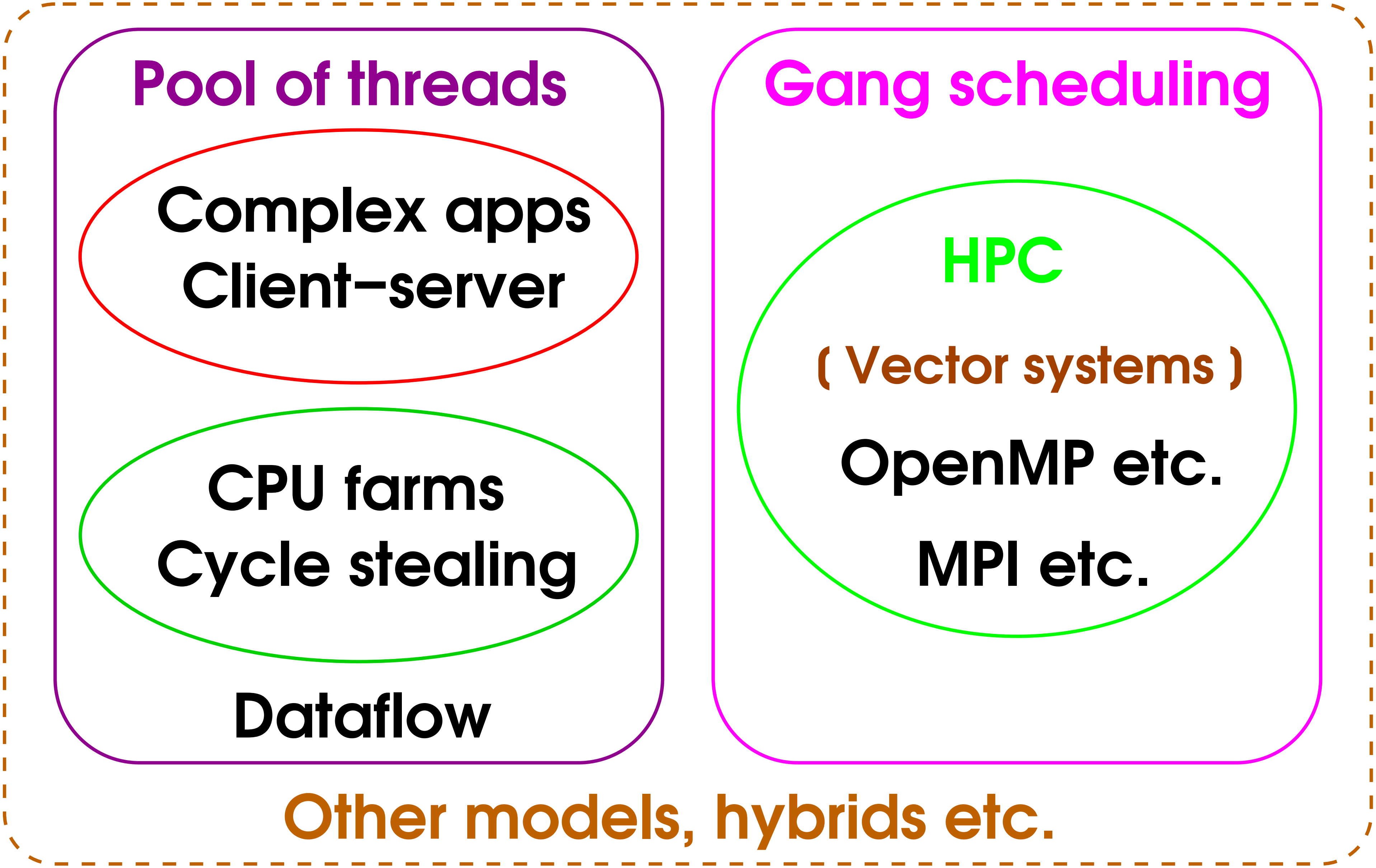
As in HPC – High Performance Computing
Probably more common in research communities

- A variant is to handle larger problems

Whether limits are time or memory

Also intermediate uses – and several others, too

Parallelism Landscape



Many Tasks vs HPC

- Difference between the two is critical

But it is only two sides of the same coin

- Also, the distinction is weakening rapidly

Thread pool models being used for more performance
Currently, a minor use, but could be major by 2020

And other models proposed by computer scientists

Need to step back and think about objectives

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
  except for embarrassingly parallel problems
- At least 8–16 if redesign is needed

Embarrassingly Parallel (1)

Some applications are naturally almost farmable
Usually, 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

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

Embarrassingly Parallel (2)

- Easy to tune, but see later warnings

Otherwise covered only incidentally in this course

- Data consistency warnings are critical

People get careless if things seem to be simple

One reason most Web servers are so unreliable
Any that update data – e.g. sales sites, registrations
Especially data corruption and incorrect behaviour

- Parallel database design is fiendishly hard

Programming and hoping simply does not work

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

- Time spent on design is never wasted

Often need many times more than for serial code

- Don't assume better performance is automatic

Can easily get 95% parallel and $2\times$ slower

Status Report

Done overview of parallel programming

Now choice of parallel environment

More Performance (1)

Many forms of parallel hardware and programming
But think in terms of programming model – e.g.:

- Single Instruction Multiple Data (SIMD)

E.g. a serial program operating on whole matrices
Old vector systems, GPUs, SSE/AVX etc.
Some simple OpenMP and other threading use

- Distributed memory with message passing

For performance, this essentially means MPI
Currently, only solution for very large problems

More Performance (2)

- Partitioned Global Address Space models

Fortran coarrays, UPC etc.
A bit like an intermediate form between previous two

- Separate threads with shared memory

OpenMP, CilkPlus, POSIX/Java/C++ threads etc.
OpenMP and CilkPlus also have SIMD aspects

By FAR the hardest model to use correctly
Unfortunately, look as if they are the simplest

- These are the ones being touted all over the Web

More Performance (3)

- Dataflow models (as in Prolog language)

Common in hardware and embedded systems
Mainly useful as program design for some problems

Other models are possible, and may become relevant
This is a very active research and development area

⇒ Expect significant changes in the next decade
Heaven alone knows what or exactly when!
Especially true for shared–memory models

Matlab etc.

Using multiple cores is automatic – for some functions
Will be useful only if your arrays are large

- May be a problem if multiple Matlab executions
Probably solutions to that, so check up if needed

A parallel toolbox for using MPI or GPUs
Costs extra, and MPI looks tricky to use

No significant local experience that I know of

Python

- Python threading runs entirely serially
This is due to a restriction of its design

- The multiprocessing module does run in parallel
It is most unusual and potentially system–dependent
No experience with it, so can't say much more

- There are also interfaces to MPI
Should be easy, but be careful about data transfer

Java

Its own shared–memory threading since 1998
Not good for performance, so not really covered

⇒ But following lectures are relevant

The problems and design apply to all threading

Choice of Environment

- Often constrained by existing practice in your area
Don't change environment without good reason
But don't use an inappropriate environment, either

- Firstly, can you just run multiple serial codes?
If so, why not? Techniques were described above

Matlab, Python and Java also covered earlier

- Usually need to use Fortran, C or C++
C++ usually via C and tricky – see later lecture

Choosing MPI (1)

- Leader is MPI – Message Passing Interface

Library callable from Fortran and C

Choose this if any of following:

Need to use clusters or similar HPC systems
Need more memory than one system
In 2014, means 500+ GB
Need highly portable or stable in long term

Choosing MPI (2)

Simplest environment to learn, not always to use

A 3–day course, covering all you need

MPI/

Choosing OpenMP (1)

- Next is OpenMP – shared–memory threading

Language extension for Fortran, C and C++

Consider this if any of following:

Want to use parallel libraries like LAPACK
Need to parallelise only small part of code
Need to parallelise existing serial code
Requirement easily mapped to SIMD model
Requirement easily mapped to tasking model

Choosing OpenMP (2)

Easy to code, but not to debug or tune
The gotchas are subtle and very nasty
Not too hard, if use in strictly disciplined manner

- Be warned – OpenMP and C++ do not mix well

Parallelising C–style C++ code is OK – see course

A 2–day course, covering SIMD (perhaps tasking)

OpenMP/

Choosing GPUs

- GPUs means specific high–end graphics cards
Generally using CUDA, sometimes others

Consider this if all of following:

Need to parallelise only small part of code
That part easily mapped to SIMD model

Beyond that is possible, but really for experts only

A course is part of MPhil in Scientific Computing

Other Choices

Two look half–sane and useful, but are new

- Fortran coarrays are an alternative to MPI
- CilkPlus are an alternative to OpenMP for C++

No other threading is recommended
The possibilities and reasons are in the next lecture

Dead Environments

These were once used, but are now effectively dead
You sometimes see these in old programs
Don't try to run unchanged – probably won't work

PVM – Parallel Virtual Machine
Originally to use spare cycles on people's desktops
Rewrite to use MPI

HPF – High performance Fortran
An earlier attempt to add a parallel interface
Rewrite to use OpenMP (or Fortran coarrays)

Combining Environments

Can use MPI across a cluster or HPC system
And OpenMP or GPUs within a node

Generally, it's more effort than it justifies
But some codes can run quite a lot faster

- Beyond that "Here be dragons"

Investigate Techniques

- Don't reinvent the wheel

Designing parallel algorithms is seriously hard
Many common problems have been solved, fairly well

There are some good, efficient parallel libraries
Especially linear algebra and matrix operations
Mainly for shared–memory based on OpenMP

- Often, only the data access pattern matters

Then can adapt an algorithm with a similar pattern

Hiatus

There is too much to cover in one afternoon
We have covered parallel use of serial programs
And overview of current parallel environments

Next lecture is on parallel programming as such
Includes describing above choices in more detail
If you are likely to use them, advised to attend

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