Parallel Computing
CM30225

Russell Bradford

2023

CM30225

Parallel computing as a topic has been around for as long as computers have been around

But recently it has come back into fashion. . . for reasons to be explored in this Unit

You have PCs, laptops and phones that are multicore: multiple processors are in the mainstream

This Unit will look at hardware and software in the context of parallel computing

Unit Outline

Structure of this unit: starting with 3 hours lectures per week

- Wednesday 10:15
- Thursday 10:15
- Friday 14:15

The aim is to cover the necessary material early in the semester which will leave the last few weeks free for revision and problems classes; and to lay the groundwork for the assignments

Unit Outline

Assessment

Usual combination of assessed coursework and exam: two pieces of coursework plus exam

1. Shared memory programming (15%)
2. Distributed memory programming (10%)
3. End of unit exam (75%)

Unit Outline

Assessment

Coursework timelines (subject to change):

1. set Thu 19 Oct
   due Wed 15 Nov
2. set Thu 16 Nov
   due Mon 8 Jan 2024

Feedback on coursework will be provided via Moodle. There will be general feedback that applies to many people and some individual feedback

Note that marking parallel programs is *very* time intensive (for reasons you will learn in thus unit!), so please don't expect a speedy turnaround

C

The coursework will be writing some parallel programs in C on a supercomputer

Though you **must** already be familiar with writing C, you may wish to brush up on your C in preparation

There is a “Remind Yourself About C” document on the Unit Web page

Unit Outline

Week 6 (starting 6th Nov) will be a “consolidation week”

No lectures for the whole of Computer Science (CM Units)

Presumably other Departments will carry on as usual

Unit Outline

Aims To give students the ability to recognise and understand the problems and opportunities afforded by parallel systems; to recognise the differing types of parallelism available and make advised choices between them; and to take advantage of progress in technology as modern computers become ever more parallel.

Unit Outline

Learning Outcomes Students will be able to:

1. write and debug simple parallel programs;
2. recognise the issues surrounding concurrent access to data;
3. describe the various kinds of parallel hardware, parallel programming methodologies and the relationship between them

Unit Outline

Skills required:

1. Comfortable writing C
2. Ability to think through complicated situations

Unit Outline

1. Basics: supercomputers and the consequences of Moore's Law; bandwidth vs latency; speedup, efficiency, scalability; Amdahl's & Gustafson's Laws; Flynn's taxonomy, SPMD; distributed, shared, NUMA and other memory architectures.
2. Shared memory computing: multicore systems (cache coherence and bottlenecks); mutual exclusion and critical regions; low level constructs including POSIX threads and synchronisation methods such as barriers, locks, semaphores, etc.; language-level support including monitors, OpenMP; vector and array (SIMD), HPF, Cn.

Unit Outline

3. Distributed computing: clusters, message passing, MPI. Programming using MPI (and SLURM).
4. Parallel algorithms and data structures.
5. Topics in Parallel Computing: examples might include HPC; MapReduce; distributed file systems; the Grid; GPGPU and OpenCL; instruction level parallelism (SWAR, VLIW).

Here Be Dragons

Note that this is a Final Year Unit, so is a lot more stretching than previous years. It contains a lot of material as parallelism is a big subject

Also it is *very important* that you are a confident programmer with good experience in C. Otherwise you will be spending a disproportionate amount of time on the coursework. Do think very carefully about this

Many in the past have assumed “it will be ok, I can wing it”, and subsequently had great difficulty in the coursework

The coursework is trivial as a sequential program, but very testing as a parallel program

Unit Outline

Resources

The subject of Parallel Computing is nearly as old as that of computers and so there are *lots* of books

None of them really suitable for this course, as we will try to take a broad overview of the subject

Part of the problem of parallel computing is that there is no simple unified model (like von Neumann for sequential computing), and everybody has their own idea on how things should be done

Leading to loads of books saying “this is the one true path to parallel computing”

Take them with a pinch of salt!

Unit Outline

Resources

Some books I found on my shelf:

Hardware

- “Highly Parallel Programming”, Almasi & Gottlieb, Benjamin Cummings
- “Computer Architecture and Parallel Processing”, Hwang & Briggs, McGraw-Hill

Unit Outline

Resources

Software

- “Concurrent Programming Principles and Practice”, Andrews, Benjamin Cummings
- “Introduction to Parallel Computing”, Kumar, Grama, Gupta, Karypis, Benjamin Cummings
- “Concurrent Programming”, Burns & Davies, Addison-Wesley
- “Designing and Building Parallel Programs”, Foster, Addison Wesley
- “Distributed Algorithms”, Lynch, Morgan Kaufmann

Unit Outline

Resources

Theory

- “Principles of Concurrent and Distributed Programming”, Ben-Ari, Prentice Hall
- “Communicating Sequential Processes”, Hoare, Prentice Hall

Unit Outline

Resources

N.B. Some of these were given to me by the publishers so I'm not saying they are the best books out there

The thing to do it look at several and find one that suits you: they contain roughly the same material

Unit Outline

Resources

You don't need me to tell you that there is a large amount of material out there on the Web?

Wikipedia is fairly accurate in this area: but, as usual with Wikipedia, you should check with other sources

There is a Unit Moodle page, but as Moodle is so horrible I tend to use my own Web page:

`http://people.bath.ac.uk/masrjb/CourseNotes/cm30225.html`

Standard Introductory Slides

Remember:

You are expected to do some work outside of lectures

Lectures are the *start* of the learning process, not the end!

These slides are reminders to me on what to say in lectures

They are often abbreviated in style, and so are not the whole story and would not be suitable to be quoted verbatim in an exam

Standard Introductory Slides

Don’t try to copy everything down from the slides in lectures—the slides will be available after each lecture

Instead, make a note of what is important and use that later—in conjunction with the slides—to guide your further reading and study

Standard Introductory Slides

Do not rely purely on my notes for your revision

People who do this live to regret it

Like every Unit, you are expected to read around the subject for yourself

You need to take your own notes, read, and *participate*

You don't expect to get fit simply by paying to joining a gym. . .

"If you have college courses in CS, buy the books and spend day and night the few days before class going through the books and taking notes and answering questions and programming examples before the first class even starts. If you really want to do this in your life, that's what you should do, not just wait for the education to be handed you. Those who finish at the top will always be in high demand. You can learn outside of school too but you have to put a lot of time into it. It doesn't come easily. Small steps, each improving on the other, is what to expect, not instant understanding and expertise."

Steve Wozniak, co-founder of Apple

Standard Introductory Slides

Computer Science is not a spectator sport

Anon

Background

You have a problem you wish to solve faster. What do you do?

1. You think hard and devise a better solution

Clearly this is a stupid thing to do. Programmers are much too lazy to do this

2. You get a faster processor

Better. This used to work, but not any more: processors have pretty much levelled off at around the 3-5GHz mark and don't seem to be getting faster

Background

3. You get a multicore machine and run the problem in parallel

This *must* be the solution!

Isn't it?

One purpose of this Unit is to make you realise this is actually the *hardest* way of doing it!

In reality, No. 1 is best, then No. 2, lastly No. 3

Background

Consider the following:

- it takes one person ten months to build one house
- it takes ten people one month to build one house
- it takes 100 people one-tenth of a month to build one house

Why is the last so implausible?

Background

When there are 100 people running about they will get in each others' way; fight over limited resources like bricks; some will have to sit and twiddle their thumbs while they wait for others to finish: you can't plumb a bathroom until the bathroom has been built

And so on

And when there are more workers, you will need more managers — not building themselves but making sure workers are doing the right things

Simply adding more people won't necessarily get it done faster

Sometimes adding more people will make it go *slower* as they get in each others' way

Background

But we can scale in a different way:

- it takes one person ten months to build one house
- it takes ten people ten months to build ten houses
- it takes one person 100 months to build ten houses
- it takes ten people 100 months to build 100 houses

This is much more believable: adding more people we can build more houses simultaneously

In reality, we won't get a perfect speedup like this, due to resource contention issues, but we can get pretty close

Background

Most people think parallel computing is about making things go *faster*

Up to a point, but they will soon be disappointed

Much more likely to succeed is to make things *larger*

This scales much better

Background

The first is *process* parallelism, also called *task* parallelism

The second is *data parallelism*

Two very different ways of getting more in a given amount of time

Background

You all have had the situation where someone tries to help you do something and it's ended up taking *longer*

There is the basic time it takes to solve the problem: then there are substantial overheads in the coordination of the parts of the solution

The overheads can easily be larger than the problem itself

This is the reality of parallel computing

Often a parallel version of a small problem will be *slower* than the sequential version

Only when the problem is made large enough to overcome the overheads will it become faster than doing it sequentially

Background

So cost (the number of cpu cycles) of a parallel computation $=$ cost of computation $+$ cost of management of parallelism

Ideally, we want the cost of management of parallelism to be minimal

But, if you are not careful, or the problem is such that this is inevitable, we can find that the cost of management of parallelism can dominate

Background

Another huge issue is that people have enough difficulties with programming sequential machines

Some would say that sequential programming is not yet a “solved” problem

Parallel programming is *much* harder

If you think you understand parallel programing, you definitely don’t

Background

You have all the issues of sequential programs **plus** lots more

And they are issues that many programmers have difficulty even understanding

Particularly as they have been trained to program for sequential machines and have habits and assumptions that are simply invalid for parallel machines

Background

Have I convinced you that parallel programming is difficult yet?

Well, it's worse than you can imagine!

Background

You will see the terms *parallel* and *concurrent*, with some people using them interchangeably

But it is sometimes important to make a distinction between the two

concurrent means your computation has separately executable parts

parallel means those parts are being executed *at the same time*

Concurrency is about structure, parallelism is about execution

Background

So, “concurrent” means in parts, and those parts may or may not be running simultaneously

For example, they might be scheduled one at a time on a single core CPU)

And “parallel” when we are explicitly talking about stuff running at the same time on multiple pieces of hardware

> *Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.*
> Rob Pike

Background

Asynchronous programming is an example of non-parallel concurrency.

This has been around for a long time in many disguises: *futures*, *promises*, *coroutines*, *generators* and others

The idea here is that when some code would block, e.g., waiting for some I/O, rather than the processor sitting and waiting doing nothing, the code should direct the processor to execute some other task

Later, when the I/O is ready, the processor can come back to where it was and continue from there

Background

The code makes its own decision on what to do: moving between different parts of code, ensuring the processor is always actively working

This is scheduling *within* the code, without involvement of the Operating System

As we know, any call to the OS entails a large amount of CPU overhead, which we avoid here

These are major points of async programming: avoid OS overheads and keep the processor busy

Background

So async code is concurrent (structural), but not parallel (execution)

Programming async code is very complicated and shares many features with programming parallel code

Modern programming languages are starting to support async programming natively, e.g., JavaScript, Swift, C++, Rust, Python and more

Constructs in the languages hide varying amounts of the gory details of choosing and switching between tasks

Background

Async programming is good in cases where we have lots of tasks that mostly wait, e.g., I/O

Parallel programming is good in cases where we have lots of tasks that mostly compute

Async is cooperative while parallel is preemptive

Async is for *waiting* in parallel

Background

In this unit we shall be concentrating on parallelism (though lots of what we say also applies to async programming, too)

Exercise Reflect on how you might use **both** async and parallel programming in one program

Background

In contrast to concurrent and parallel, you might hear of *serial* and *sequential* both being used to describe non-concurrent/non-parallel systems

Serial and sequential mean the same thing

Background

Moore's Law

Why is parallelism an important topic these days?

There is a famous "law" that describes how hardware has progressed over the years

It is an **observation** on how the components in integrated circuits were shrinking over time as engineering advances were made:

Moore's Law (1965):

> *the number of transistors in a chip doubles every two years*

Background

Moore's Law

There are a number of points to be made

- it's not a "law" in any real sense, but an *observation* on how chips progress
- Moore did not say *speed* doubles, as often mis-quoted
- some variants say "18 months" instead of "two years", but the history of this statement is complex
- it is somewhat self-fulfilling, as engineers tend to use it as a target for the development of each next generation of chips

Background

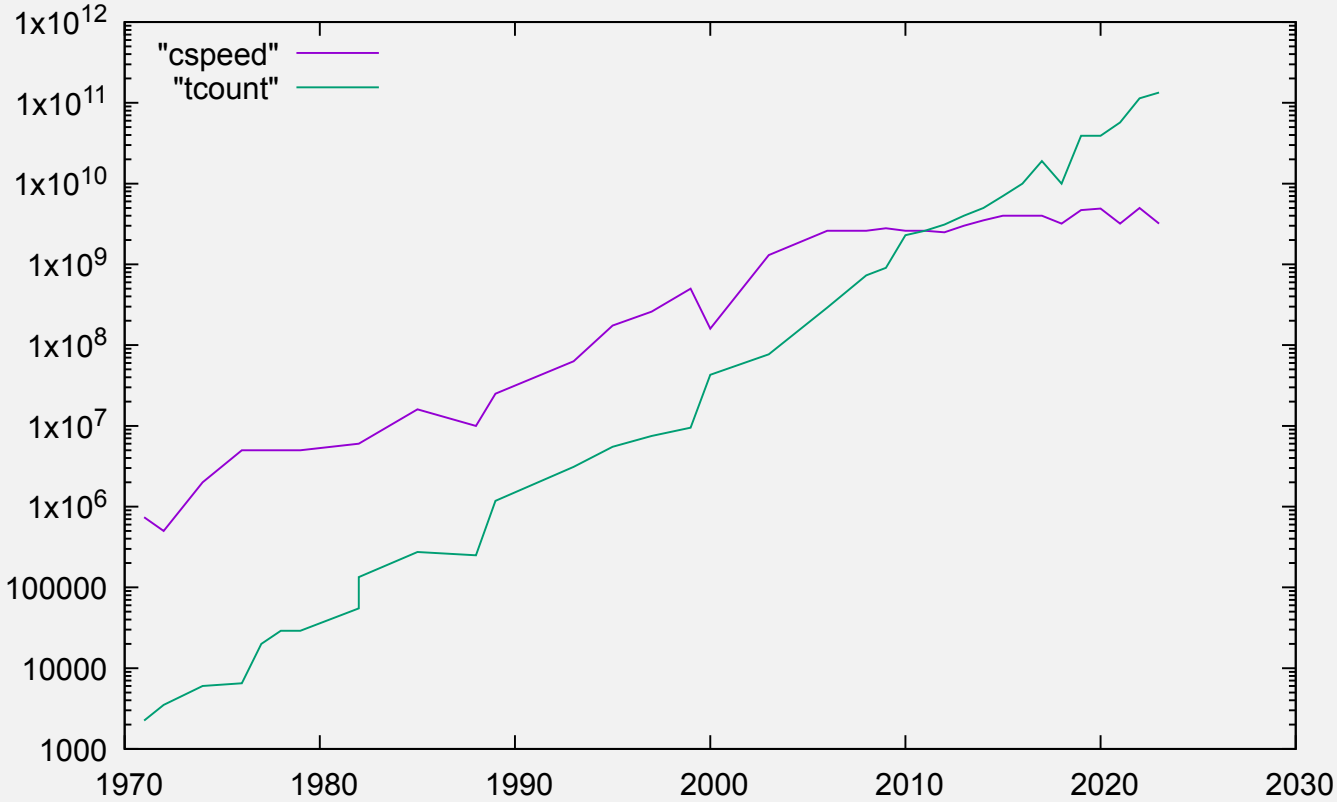
Moore's Law

There is some economics in there, too: the profit margins on silicon wafers mean that it is better to have fewer larger chips than lots of smaller chips

So CPUs tend to keep to the same area, meaning a CPU will have more and more transistors, not that we have more smaller CPUs

Background

Moore's Law



Log of speed and transistor count against date of Intel processors

Background

Moore's Law

We can see why people thought that Moore's Law was about speed: for a long time both transistor count and speed went up exponentially

In 2005 people would have said that CPUs would be running at 480GHz by 2020

However, over the last few years speed has stopped increasing

But, crucially, the transistor count continues to increase

CPUs stay the same physical size

Background

Moore's Law

Engineer:
What are we going to do with those extra transistors?

Marketer:
How are we going to convince people to buy the new CPUs?

Solution:
multicore processors

Chips with more than one CPU on them

Background

So now chips in new PCs are all multicore

Dual and quad core is everywhere; 64 core processors are around; 128 cores are arriving soon (PC-style architecture)

Many cores is great, but we are going to have to find out how to make best use of them

But simply having two CPUs generally won't make our program go twice as fast: overheads like interference and communication between parts of the computation is going to be a problem

Background

To repeat: all this hardware is all wonderful except for one point

> *This computational power is only useful if we can write the software to exploit it*

Your phone might have eight cores, but it is likely very little software it runs is capable of using all their power simultaneously

Software is far behind hardware and has a lot to do to catch up

We are still in the dark regarding parallel software

Background

A Brief Aside

Note that Moore's Law also applies to memory: memory chips have been doubling in capacity at a similar (perhaps faster?) rate

But the *speed* of delivery of data from memory to processor(s) has always lagged behind the speed of processors

Giving a problematic gap between speed of processors and speed of memory (both in bandwidth and latency)

The gap has decreased a little over the last few years, but on the other hand multiple processors need more memory bandwidth

We shall see memory is a big bottleneck in parallel systems

Background

Moore's Law

Moore's Law has been going for 58 years so far

It must come to an end at some point: the end has been predicted many times in the past, but so far technology has kept moving onwards

Chip designers think it will keep going for several years yet, some predict decades

Moore himself thinks perhaps it will last until 2025

And — looking at Intel's products the last few years — it might currently be taking 5 years to double transistor counts

Background

Moore's Law

Exercise Some current top end chips have over 100 billion transistors, and 7000 cores. If Moore's Law continued, how many transistors and cores would they have in 10 years? In 20 years?

Exercise Read about Moore's Second Law (aka Rock's Law)

Background

Moore's Law

Software is getting slower more rapidly than hardware is becoming faster
Wirth's Law

Software efficiency halves every 18 months, compensating Moore's law
David May

The speed of software halves every 18 months
Gates' Law

What Intel giveth, Microsoft taketh away
Anon

Background

There is nothing new in Computer Science and that includes parallelism. Back when large supercomputers were first popular they had been parallel for a long time

For example, a common kind of hardware was the *vector processor*

This is for data parallelism, namely scaling the data, not the speed (directly)

E.g., add together these 100 pairs of numbers to produce 100 results

Background

A vector processor is a collection of 10s, or 100s or 1000s of fairly simple CPUs (technically not proper full CPUs, just ALUs: see later)

However, in a vector processor, the CPUs are not independent of each other: at each point in time each processor is doing the *same operation*

But on *different data*

So it can add 100 pairs of numbers simultaneously: data parallel

This is called *single instruction multiple data* (SIMD) processing

Background

And there are other ways of making parallel machines: if you want to make a really big machine, for a long time the architecture of choice has been the *cluster*

This is “simply” large numbers of normal PCs connected together with a network, with your program spread across the nodes (the PCs)

We can get both process and data parallelism from this architecture

The hardware is commodity, so clusters with thousands of CPUs are common; clusters with millions of cores exist

Background

Some words: be aware different people use these terms in different ways

- core: a single processing element, can be just an ALU or can have its own instruction decoding unit
- cpu: sometimes just a synonym for core, sometimes a chip which contains one or more cores
- processor: similar to cpu
- node: a motherboard that can have one or more slots for multi-core cpus that share some local resource on the motherboard, particularly memory
- cluster: a collection of nodes connected by a network

Background

For example, the Azure machine you will be using for the coursework has four nodes, each consisting of two chips, each with 24 cores

Background

From `www.top500.org`, the fastest (publicly known) computer in the world is (June 2023):

> Frontier (USA), 8,699,904 cores, comprising AMD EPYC 64C cpus at 2GHz; plus Radeon Instinct GPUs; using 23MW power; with Slingshot-11 interconnect; running HPE Cray OS

This peaks at about 1.2 exaflops

1 exaflop is a quintillion (10^{18}) floating point operations per second

Background

This is the first machine to pass the “exaflop barrier”

HPE is Hewlett Packard Enterprise

Slingshot is a high performance network; a Cray technology, with (e.g.) hardware support for MPI

HPE Cray OS is a variant of SUSE Linux Enterprise Server

Background

But lots of cores is easy: just expensive

> *Anyone can build a fast CPU. The trick is to build a fast* **system**.
> Seymour Cray

Background

The main problem in a cluster is the slow communications between the CPUs

A typical network connection is millions of times slower than a memory bus: milliseconds rather than nanoseconds

To move data from one node in a cluster to another is (relatively) immensely slow

Programming a cluster is all about moving the data: we might be able to do a million machine instructions in the time it takes to fetch some data from another node

Background

On a machine with a million cores it can be faster to do a million adds on one core rather than ship out the adds to the CPUs; do a million adds in parallel; then collect the data back together

Just having an immensely parallel machine doesn't mean it's always better to use the parallelism

Background

In a large parallel machine (cluster or otherwise) processing power is cheap, but data are expensive

This means you have to think about your programs differently

It might be faster to recompute the same value 1000s of times across many cores than compute it once and communicate it everywhere

A very different mindset is needed!

Classifications

We need to classify the kinds of parallelism we shall be looking at

A simple classification was devised by Flynn (1966)

- Single Instruction, Single Data (SISD). Traditional, von Neumann, single core machines
- Single Instruction, Multiple Data (SIMD). As in a vector processor. Multiple cores all doing the same operation in *lockstep*, but on different datastreams
- Multiple Instruction, Multiple Data (MIMD). Multiple cores doing different things to different datastreams. What most people (wrongly) think parallel computing is all about

Classifications

- Multiple Instruction, Single Data (MISD). Something to fill in the last combination of letters. Sometimes interpreted as *redundancy*, e.g., airplane flight control where they have multiple (different!) computers all processing the same data

		Data	
		Single	Multiple
Instruc-	Single	SISD	SIMD
tion	Multiple	MISD	MIMD

Classifications

Flynn's classification is nice and simple, so people have extended it further, in particular sub-dividing MIMD

- Single Program, Multiple Data (SPMD). Recall SIMD runs the same program on multiple cores in *lockstep*, so every core is executing the same instruction. SPMD runs the same program (on different data) on a MIMD machine, with each core going their own way, particularly on loops and conditionals
- Multiple Program Multiple Data (MPMD). A MIMD machine not running SPMD. So each core running potentially different programs, e.g., producer-consumer models, or systolic pipelines (see later)

Classifications

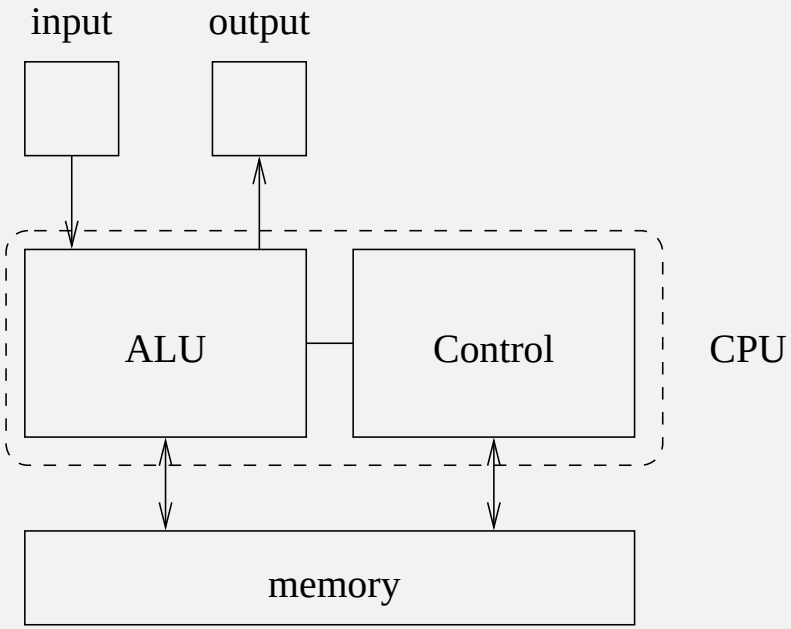
Of course, there are many more classifications we need to look at

We can think of how the parts of the architecture are connected

Classifications

Uniprocessor

A *uniprocessor* (*unicore*) or *sequential processor* is the traditional von Neumann architecture of a single CPU, memory, etc.



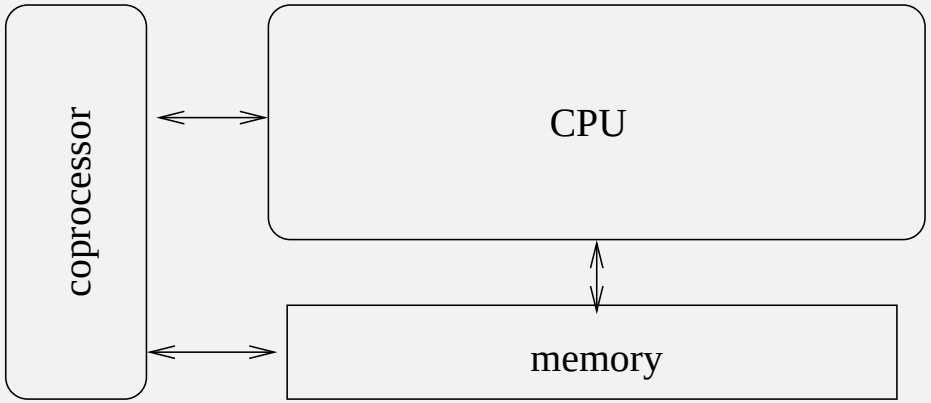
von Neumann Architecture

A hugely successful model that enabled the computer revolution to take place

Classifications

Coprocessor

A *coprocessor* is a non-general processor used as a worker by the processor



Coprocessor

Currently very popular in the form of graphics cards

Classifications

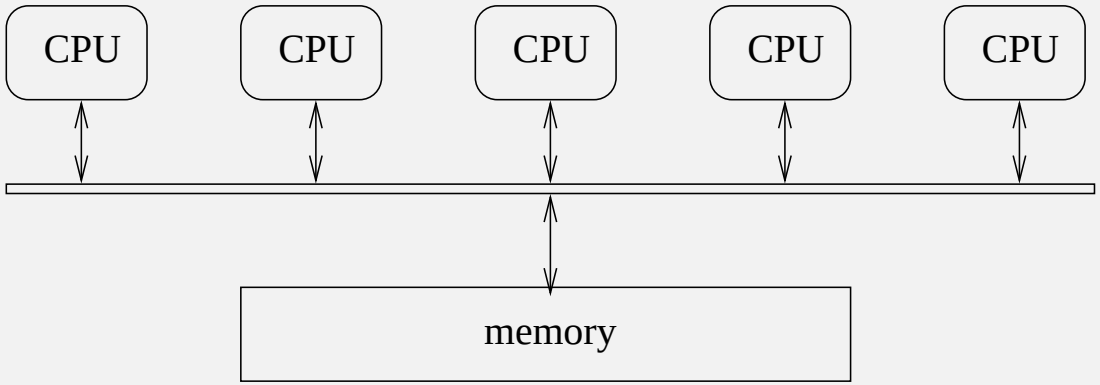
Multiprocessor

A *multiprocessor* is a loose term applying to most parallel architectures, except occasionally SIMD, which usually doesn't have multiple full cores

Classifications

Shared Memory

A multiprocessor has *shared memory* when the cores access memory on a shared bus



Shared Memory

Cores share each other's data: if one core modifies the value of a value in memory, the other cores see that change

Classifications

Shared Memory

In reality, the shared bus can be a lot more complicated, e.g., a tree or ring structure

In this example, we have *symmetric* shared memory: every CPU has the same equal access to the shared memory

Classifications

Shared Memory

This is possibly what most people think of as a typical parallel architecture

Unfortunately, it has a lot of problems as an architecture

In particular, the memory is a bottleneck

Memory and memory buses are slow relative to a processor anyway, and when you have several cores all trying to access memory simultaneously it gets much worse

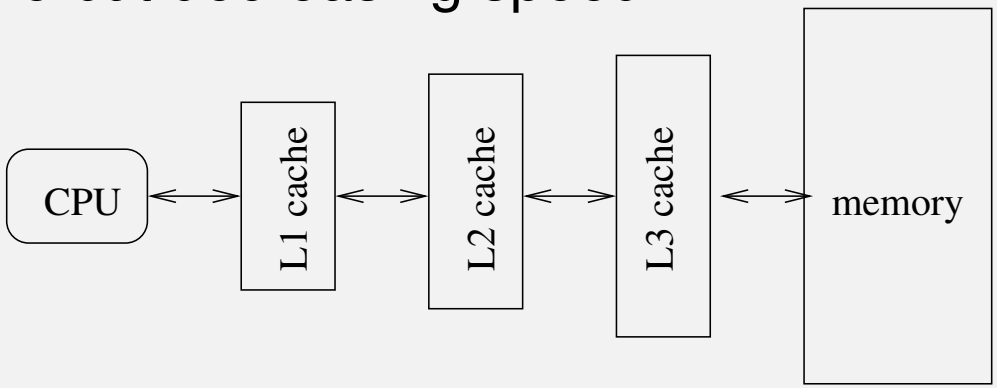
Classifications

Shared Memory

Even single core processors have a problem with the speed disparity, so they use fast (but small) intermediate cache memory

A small (because it's expensive) chunk of very fast memory where you store copies of a few of the values you are currently using from main memory

Sometime two or three (occasionally four) levels of cache of increasing size but decreasing speed

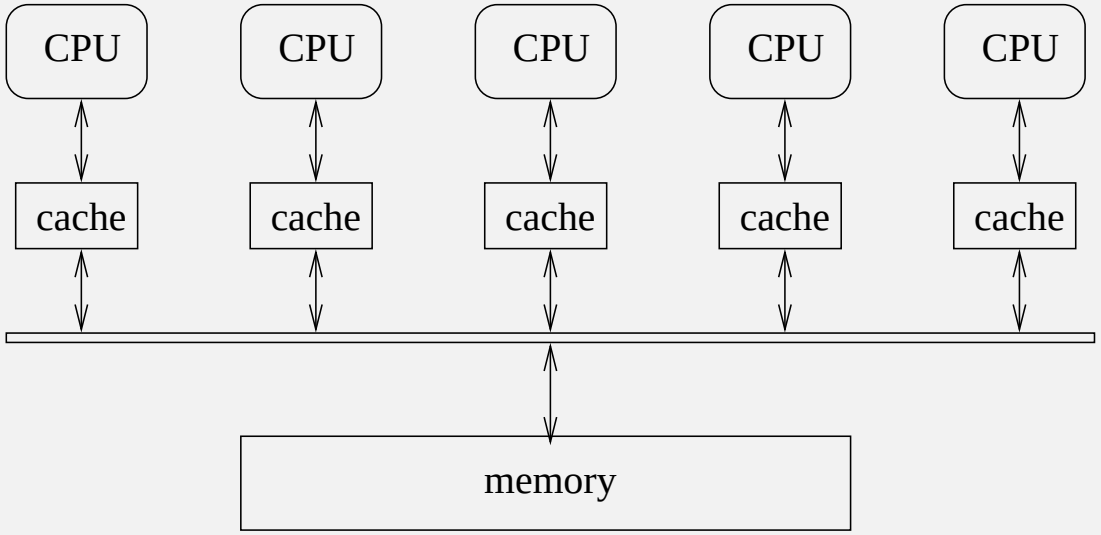


Levels of cache

Classifications

Shared Memory

So shared memory machines try to cut down the traffic on the bus by using caches



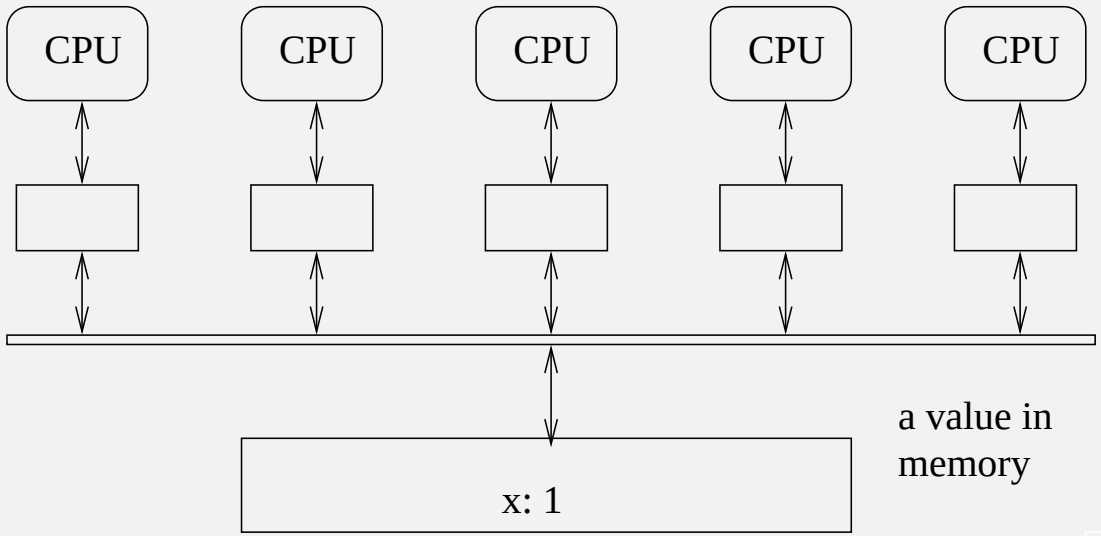
Memory caches

Each core has its own chunk of fast cache memory: this cuts down on use of the bus

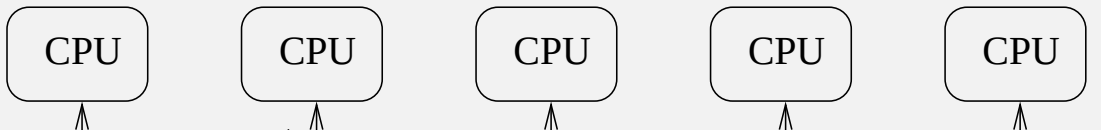
Classifications

Shared Memory

If a core is manipulating the value of a variable it will be loaded into the cache and operated on there, rather than over the bus in main memory



A value in memory



Classifications

Shared Memory

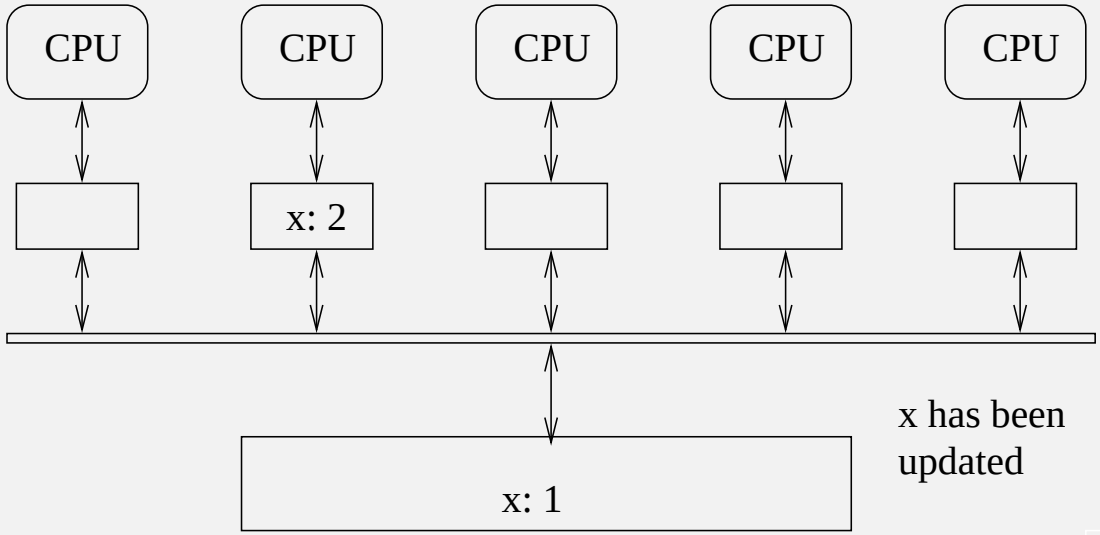
This reduces pressure on the shared bus: but now we have the problem of *cache coherence*

A CPU only updates its cached copy; the global copy remains at its old value for a while

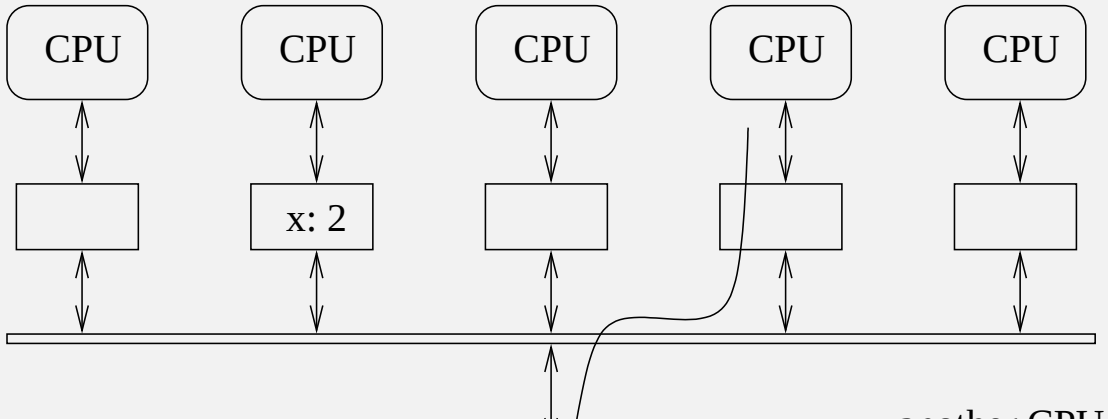
So if another core want to read the value before the updated version has been written back, it will get the old value

Classifications

Shared Memory



x has been updated in cache



Classifications

Shared Memory

Even worse, dependent on timing, you don't know if the first CPU has written the value back or not

Meaning different runs of the same program can produce different results, dependent on what else happens to be going on in the system

This is an example of a *race condition*: differing orders of execution of concurrent parts of a system produces varying outcomes

This particular example is a *data race*: a race condition that involves updating data

Classifications

Shared Memory

Not what we want, as we can't control the vagaries of hardware operation

You might get the right answer on hundreds of runs; it doesn't mean your program is correct!

And it might always happen to be right on your machine, but wrong when run on some other machine

Classifications

Shared Memory

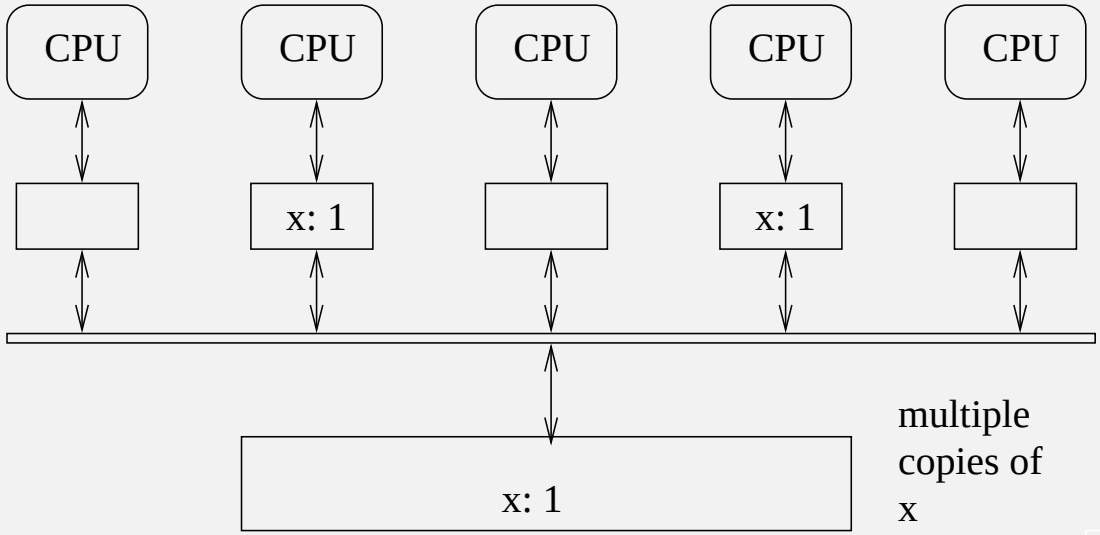
There are other ways to fail, too

Others cores might be doing the same: reading and updating the value. Thus there can be several conflicting copies of what is supposed to be the same variable in different caches

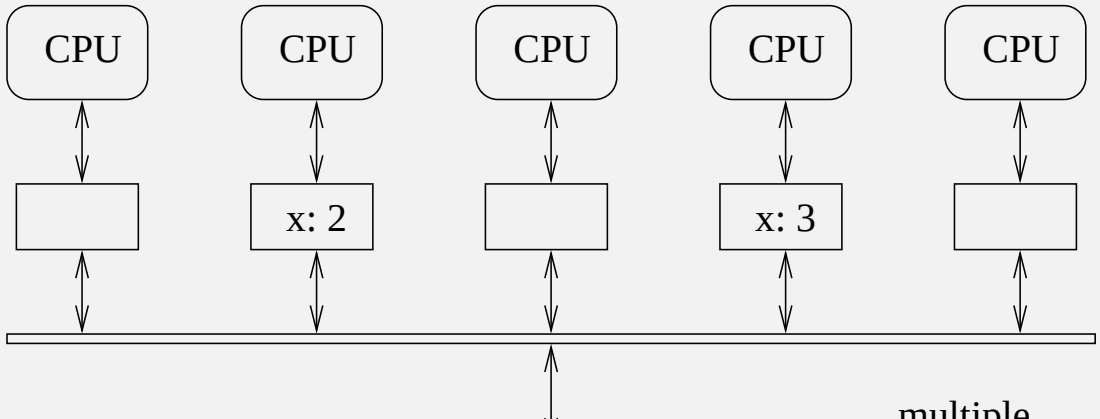
When one core updates the variable the other cores will still be using their own in their caches

Classifications

Shared Memory



Multiple copies of x



Classifications

Shared Memory

The *cache coherence* problem is the issue of trying to make sure all cached copies of a variable are kept in sync

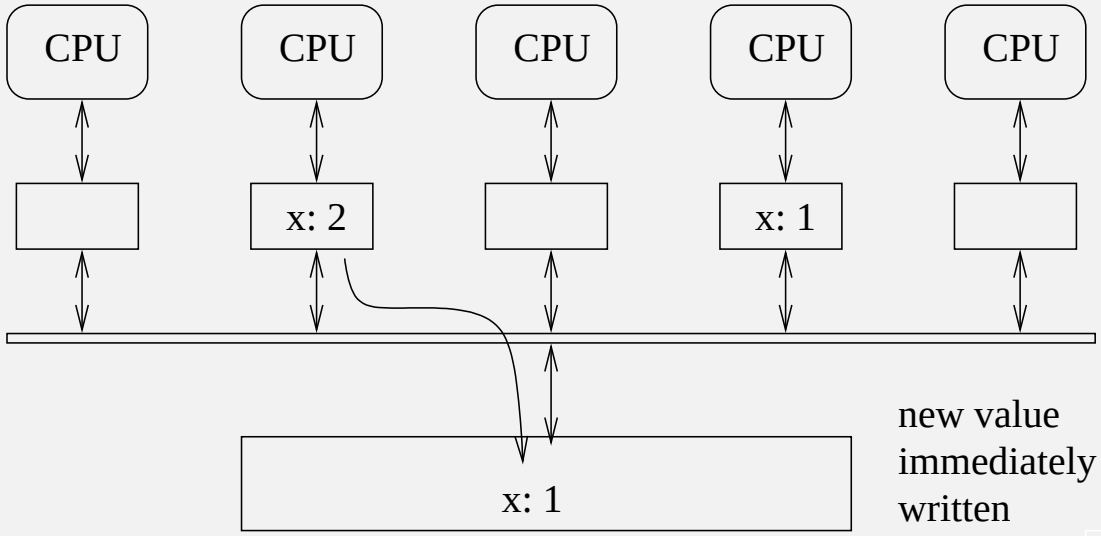
This might be done in several ways

E.g., in the *snarfing* protocol, whenever an update is made the value is immediately written through the bus (increasing traffic on the bus...) to main memory. The other caches are watching the bus and if they have a copy of the variable they update their copy with the value being written (they "snarf" the new value)

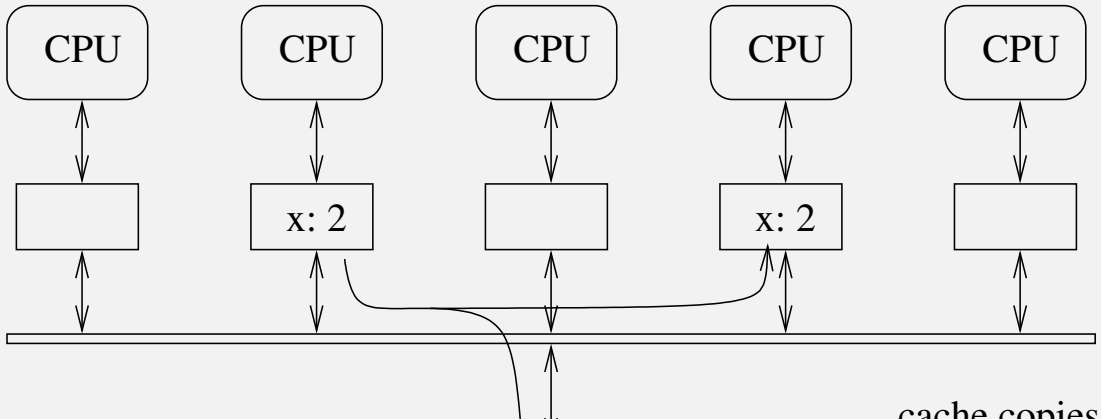
This is expensive in hardware and does not scale well to large number of cores as every write must go through the bus

Classifications

Shared Memory



New value immediately written to memory



Classifications

Shared Memory

But this is better than you might imagine as typical code reads values much more than it updates values

In `x = y + z` two values are read, one is written

So this kind of cache-watching is more effective than you might think

Secondly, well-written code will avoid using shared values in the first place. Sharing mutable state across threads is bad design (more on this later)

Classifications

Shared Memory

Other solutions might be to try to balance the memory/cpu speed disparity

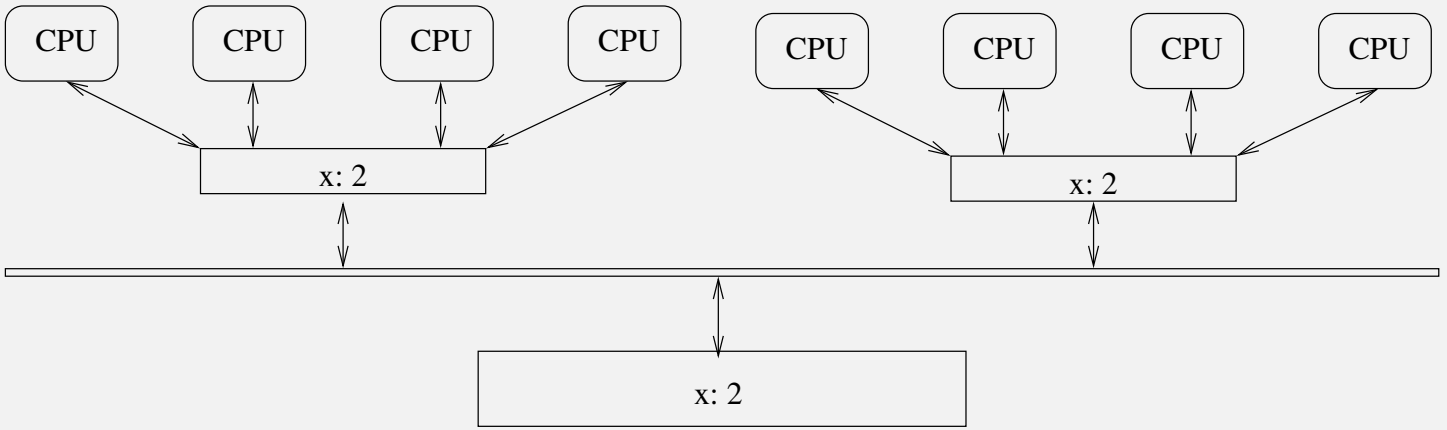
You could use very fast buses and main memory: not a solution due to cost

Or use slow processors: IBM tried this and it was surprisingly good!

Classifications

Shared Memory

Exercise Modern architectures are more like:



Modern memory architectures

Does this solve the problem?

Classifications

Shared Memory

Unfortunately, such *symmetric* shared memory does not scale well, perhaps a few 100s of cores, with complex hardware support in the caches

Ampere has a 128 core Arm architecture

Intel have just announced a 288 core x86 chip (Sept 2023)

Classifications

Shared Memory

Exercise Read about cache coherence mechanisms: snoopy caches; directory based; snarfing; MSI; MESI

Exercise Another complication to symmetric shared memory is when the *cores* are not identical: read about *performance* and *efficiency* cores (P-cores and E-cores) used by Intel, Apple and others

Classifications

Shared Memory

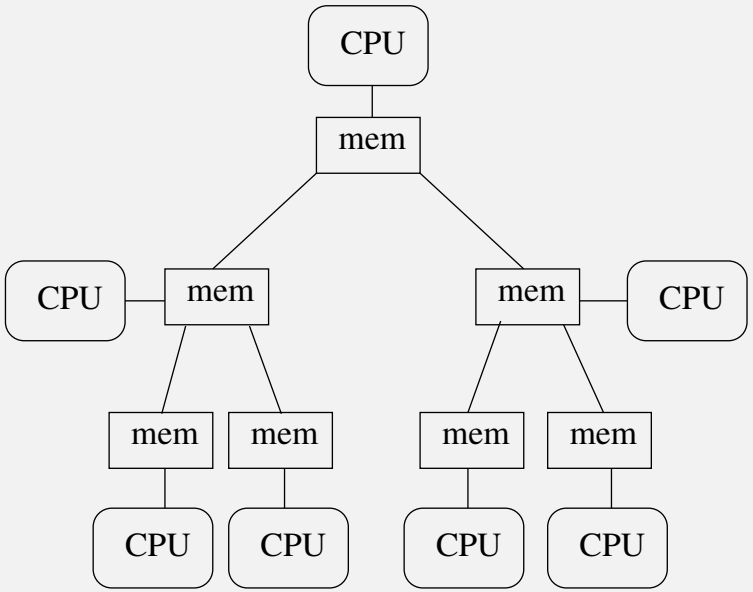
Symmetric shared memory is the model that current small machines (multicore PCs) use

It is well suited to MIMD, but note that SIMD also uses symmetric shared memory, but with a different access pattern

Classifications

NUMA

So if symmetric, i.e., uniform access, shared memory does not scale, we can try managing memory in other ways



Example NUMA

Each processor has a chunk of memory, but can also access memory of other processors, perhaps arranged in a tree

Classifications

NUMA

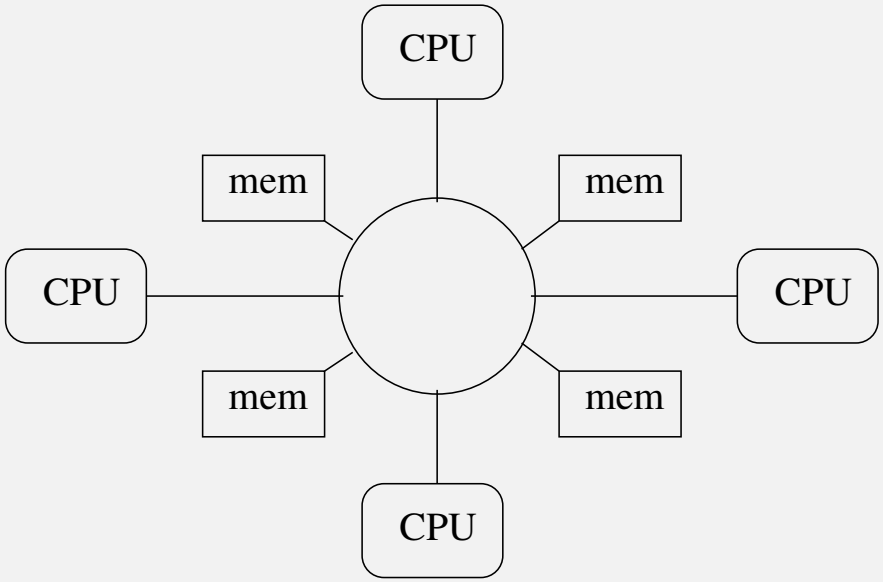
A processor will have fast access to its closest chunk of memory, but slower access to more remote memory

And different chunks of remote memory will have different access speeds

Classifications

NUMA

Of course many other topologies have been tried: star, ring, hypercube, full interconnect, and so on



Memory in a ring

This architecture evens out the access time to different chunks of memory a little

Classifications

NUMA

These are *non-uniform memory access*

NUMA shared memory scales much better than symmetric shared memory

By scaling here we mean you can build larger machines with more processors cost effectively

But there is a downside: now programs and programmers (and the OS) have to worry about *data locality*: data a processor needs should be kept close to that processor

It can make a huge difference to the speed of a program if the data is not where it should be

Classifications

NUMA

If data is close to the processor that is using it, it will go faster than if the data has to be fetched from further away

So you try to keep data near the relevant processor

Or the computation on a processor near to the data

Of course, if data needs to be used by several processors, this becomes a very difficult scheduling problem

Classifications

NUMA

NUMA implementations stratify the memory in terms of "distance"

For example:

- direct connection on the local memory bus
- on the same node
- one hop away
- two hops away
- and so on

Classifications

NUMA

Though this is often simplified to: local, remote, and “far away”

The OS or system libraries or the programmer will try their best to place data in appropriate memory to minimise latency, using their knowledge of the NUMA hierarchy and their knowledge of the program’s needs

The programmer ideally would have a good idea of the architecture of a machine before writing code for it

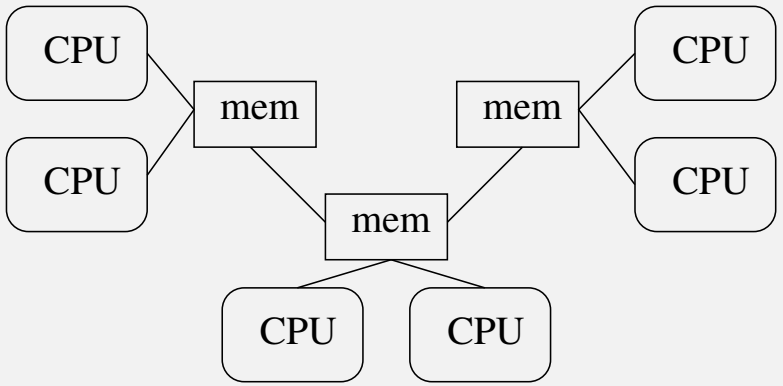
And so the portability of a program is in question

This is still a matter of great research and development!

Classifications

NUMA

And, of course, there are hybrids where CPUs share some memory symmetrically and some memory NUMA



Hybrid NUMA

Classifications

Distributed Memory

NUMA allows architectures to scale to greater numbers of processors, but it won't scale indefinitely, perhaps a few 1000s of cores

If the problem is the memory bus bottleneck which means you have to keep cached copies of a value, and then you have the problem of keeping coherence amongst the copies, why not simply *not* have shared memory?

Distributed memory says each processor's memory it its own and is entirely separate from every other processor's memory

Classifications

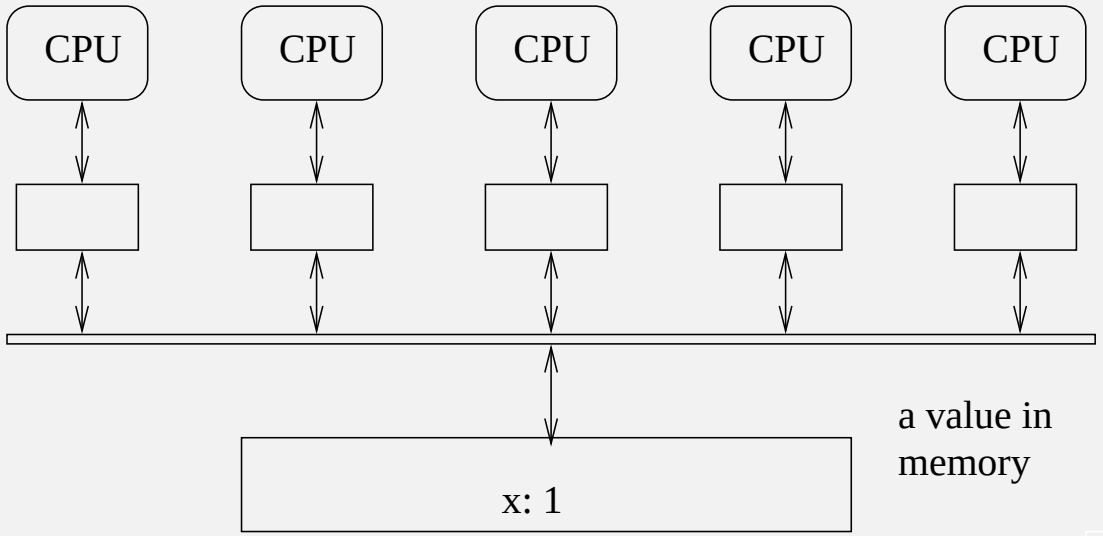
Distributed Memory

Shared memory processors share a *single memory address space*: within a single process memory location 42 on one processor refers to the same thing as memory location 42 on every other processor, as it's the same memory

The variable `x` on this processor is the same as the `x` on that processor (assuming SPMD)

Classifications

Distributed Memory



Shared address space

Classifications

Distributed Memory

Processors in a distributed memory architecture each have their own, separate, address space

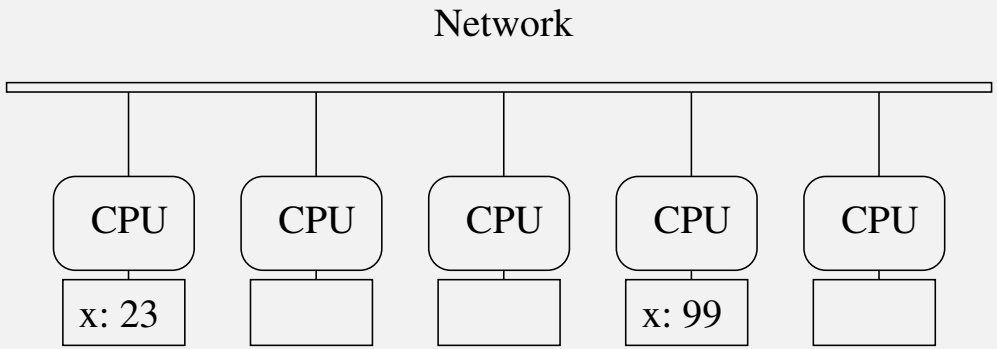
Memory location 42 on one processor is entirely separate from memory location 42 on every other processor

Each processor has their own version of variable `x`, nothing to do with any other `x` on other processors

Classifications

Distributed Memory

Each processor has its own memory



Distributed memory architecture

Typically connected by a network, rather than an expensive memory bus

Classifications

Distributed Memory

To get at data on another node a processor sends a *message* to that node, which will reply with the data

Clearly this *message passing* will be very much slower than simple shared memory accesses

Memory access across a network can be factors of thousands, perhaps millions times slower than local memory

The position of data is now *very* important

Your code has to change, too

Classifications

Distributed Memory

Think of a shared memory operation:

`x = y;`

`x` gets the value of `y`, “simply” read from memory

Compared with the overhead in distributed memory of creating a message, sending, waiting, reading the reply

See MPI (Message Passing Interface) later, but conceptually we have to write

```
x = FetchDouble(remotecpuname, "y");
```

Classifications

Distributed Memory

Some underlying message passing library does the hard work of the messaging

Your code become much more complex to write

Both in needing a lot more text, and in needing thought on where to put your data

Classifications

Message Passing

Note that you can also use message passing on a shared memory architecture

Doing so might be useful for coding or program structure reasons

The underlying messages are now probably implemented as simple accesses to shared memory

Some parallel programming systems (see later) *only* provide messaging across threads (often via mechanisms called *channels*), thus masking the underlying architecture and improving program portability across architectures

Classifications

Distributed Memory

When using distributed memory you try to keep the data a process needs on the processor it is running on, maybe even replicating data or replicating computations, and access remote data as little as you get away with

You have to balance the cost of the computations against the cost of the data movement

An ideal that is rarely achieved in real programs

Of course, if you replicate data that gets updated, you immediately have a coherence problem again, but now your own code has to deal with it

Classifications

Distributed Memory

Note that replicating read-only data (e.g., tables of values) will be fine: there is no coherence issue with multiple copies of data that never changes

But you do need to put a lot of thought into replicating read-write (mutable) data

Classifications

DMA

More sophisticated systems have extensive hardware support for messaging

They have specific *direct memory access* (DMA) hardware that accesses memory independently of the CPUs

Thus messaging proceeds independently of the CPU: communication is *asynchronous* with computation, freeing the CPU to do something else while the message is being processed by the DMA hardware

Thus allowing more computation; but at the cost of more complicated programming

Classifications

Computation vs. Communication

The call to `FetchDouble` above could return immediately and allow your code to continue computing on something else, rather than waiting for the value of `y` to appear: but you can't use `x` until the value has arrived some time later

Of course, you now need some mechanism to be notified when the value *has* arrived, and so you can now use `x`

Such asynchronous programming is very hard to get right

But this idea of overlapping computation and communication is important and will reappear many times

Classifications

Distributed Memory

In distributed systems the concept of single shared values has to go completely out of the window

The value of `x` here is nothing to do with the value of `x` there

Programs have to be written with this in mind: global shared mutable values are simply not a good idea, even in uniprocessor programs!

Classifications

Distributed Memory

Distributed memory is the architecture used by clusters: each node is effectively a PC

Very suitable for SPMD, not so suitable for SIMD

Even with the huge message passing overhead, clusters are very popular, particularly with very large problems where the overhead is small relative to the rest of the computation

The computations do have to be huge!

Classifications

Distributed Memory

Not suitable for small problems, or problems where data need to move a lot between processors

Scales *very* well as an architecture. Clusters of over a million cores exist: see the TOP500 list

Classifications

Scaling

Making big machines is easier with distributed systems, too

When we try to add CPUs to a shared memory system, we have to pay a great deal for the complicated memory architecture as it means redesigning the silicon and building new chips

This can quickly swamp all other costs, so making scaling a shared memory system impractical

In contrast, the cost of adding CPUs to a distributed memory system is “simply” the cost of the CPUs and the networking

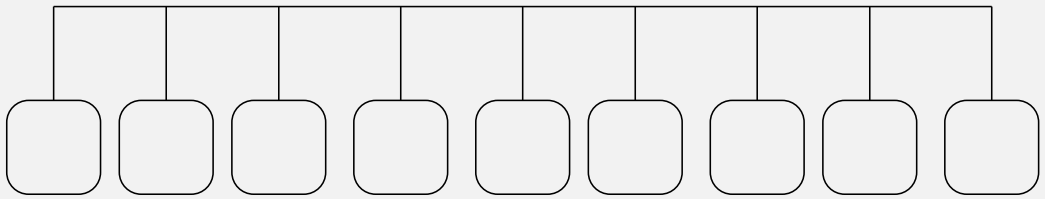
This is roughly linear (per CPU) price scaling

Classifications

Distributed Memory

However, when scaling a cluster we should take care to scale the network, too, otherwise we have exactly the same kinds of bottleneck issues that shared memory systems have

In a network like



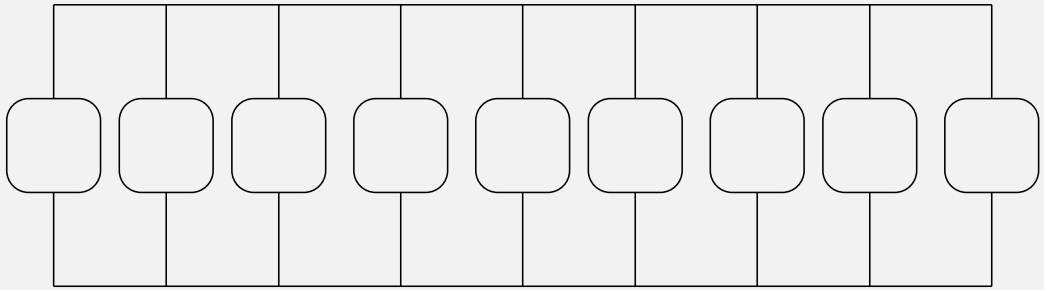
Simple shared network

the single shared network is clearly a bottleneck

Classifications

Distributed Memory

So we need to scale the network. There are many choices:



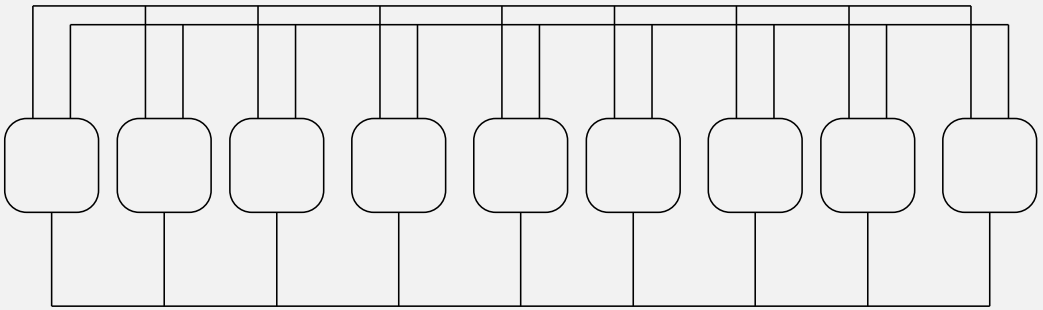
Network with two interfaces

Each processor would use one interface to communicate with processors 0, 2, 4, etc.., and the other interface to processors 1, 3, 5, etc., thus spreading the load

Classifications

Distributed Memory

Or three interfaces



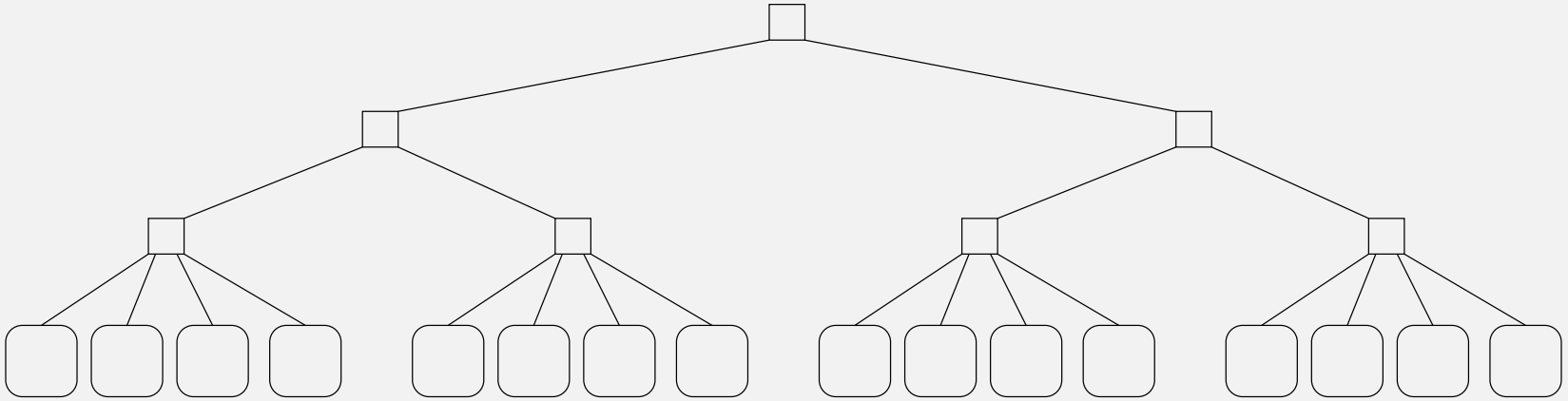
Network with three interfaces

But this gets expensive very quickly

Classifications

Distributed Memory

Trees are a good way of connecting things:

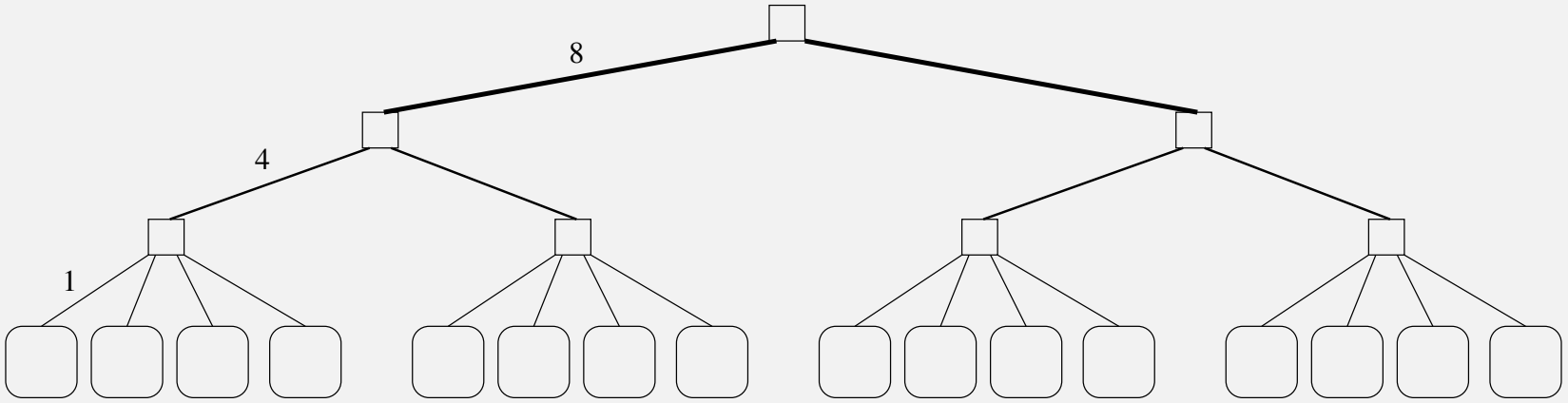


Tree network

Though the upper links now are a bottleneck, and we have introduced another non-uniformity

Classifications

Distributed Memory



Fat Tree

In a *fat tree* links up the tree have larger bandwidths, thus allowing full simultaneous bandwidth between each pair of nodes

Classifications

Distributed Memory

Though the latency between nodes will vary

In practice, a full fat tree is quite expensive, so real fat trees tend to skimp on the upper links a bit, e.g, 1, 2, 2 in the above diagram would be much cheaper to build (and a “2” would probably be a pair of “1”s)

Thus trading bandwidth for cost

Many other topologies exist, such as hypercube, torus, Banyan, etc.

Exercise Azure uses a *Clos network* within its datacentres. Read about this

Classifications

Distributed Memory

The point here is that this is relatively cheap to do with a distributed memory network. But adding bandwidth by doing this kind of connectivity in a shared memory system is extremely expensive as it likely needs new silicon

Adding bandwidth in a network is relatively cheap

But decreasing latency is very expensive whatever the system

Classifications

Virtual Shared/Distributed Virtual Memory

Some programmers don't like the fact that distributed memory machines require programming using message passing and prefer the shared address space model: shared memory is easier to write programs for (they claim)

They can use *virtual shared memory*

Just as virtual memory is a way of converting virtual memory addresses into physical memory addresses, virtual shared memory is a mechanism to have a single, virtual, address space that is converted into distributed physical addresses

Thus this is also called *distributed virtual memory* and *distributed shared memory*

Classifications

Virtual Shared/Distributed Virtual Memory

Reading and writing variables will be implemented by a message passing layer hidden from the programmer in the OS or systems libraries

So the programmer won't have to care about it and they can write programs as if the whole of memory was one big chunk

The programmer writes the simple "`x = y`" and the compiler/OS converts this into a shared memory access or a message call as appropriate

But it will be very NUMA to data

Classifications

Virtual Shared/Distributed Virtual Memory

Unfortunately, programmers *do* have to care as the speed of a program will be very hard to predict or control, depending on how data is distributed across memory and the particular NUMA architecture it is running on

How long does the assignment "`x = y`" take? Is it different from "`x = z`"?

A good programmer looking for a good, consistent performance from their code will still need to think hard

A poor programmer will think their life is easier

Classifications

Virtual Shared/Distributed Virtual Memory

The underlying system also needs to solve all the problems of cache coherence that shared memory hardware has, but now using the (relatively) slow messaging passing layer rather than custom-designed hardware

The NUMA aspect is so unpredictable that many programmers prefer to be in control and have an explicitly non-shared model

When you write `FetchDouble` you *know* it is going to be slow

Compare with "how fast is `x = y`?" in VSM

Classifications

Virtual Shared/Distributed Virtual Memory

The underlying communications layer in VSM might be implemented

- in the Operating System, such as *Mosix*. This means all standard system libraries and user code can be used unchanged and a cluster looks like a single big machine: a *single system image* (SSI)
- by the programming language and libraries, such as Cluster OpenMP or Unified Parallel C (see later), so the language may need a bit of learning by the programmer

Classifications

Virtual Shared/Distributed Virtual Memory

VSM is currently fairly rare in practice, though as NUMA techniques improve, people are starting to talk about *shared memory clusters* as being a viable and useful way to proceed

Latency numbers every programmer should know

L1 Cache hit	0.5 ns	0.5 sec one heart beat
Mutex lock/unlock	25 ns	25 sec making coffee
Main memory access	100 ns	100 sec brushing your teeth
Read 1MB from memory	250,000 ns	2.9 days a long weekend
Round trip within datacentre	500,000 ns	5.8 days a short holiday
Read 1MB from disk	30,000,000 ns	1 year
Send a packet California $\rightarrow$ Netherlands $\rightarrow$ California	150,000,000 ns	4.8 years two round trips to Mars

`https://gist.github.com/hellerbarde/2843375`

Classifications

The next class of architecture is one we have already touched on

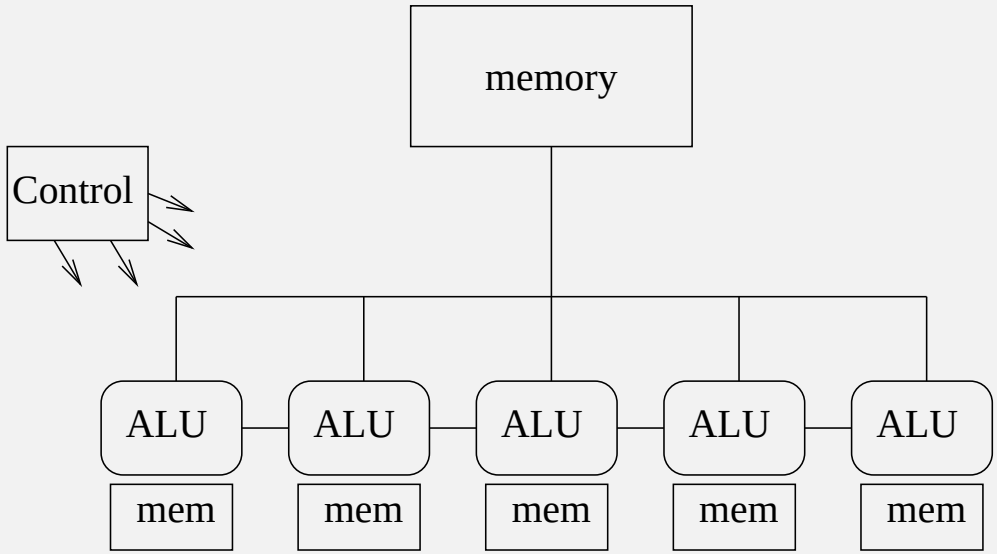
It has elements of both shared and distributed memory

It is used for data parallel computation

Classifications

Vectors

A *vector processor* is a SIMD collection of CPUs (actually ALUs), often with a chunk of global shared memory (and a single control unit)



Vector processor

Each processor also has its own chunk of local memory that it operates on

Classifications

Vectors

The local memory allows each ALU to work on a different set of values

Note: this is *not* cache, but simply per-ALU memory

Classifications

Cache vs Local

Cache memory: a fast local copy of a slower memory location. If a value of a variable is cached on different cores, we want all the caches to contain the same value for that variable

Local memory: per core memory (not always fast, by the way!) where we expect to have different values for a given variable in each

Classifications

Vectors

In a vector processor, the bottleneck to the shared memory still needs thinking about

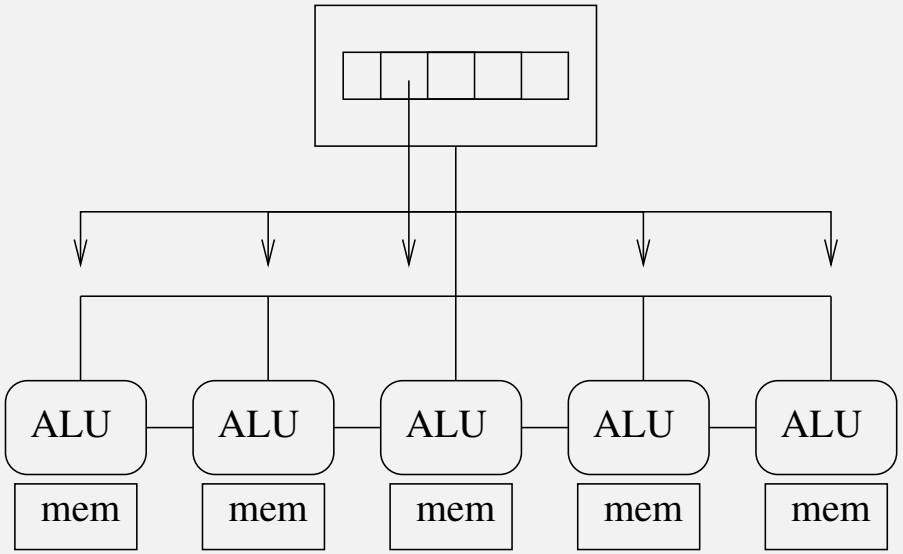
For reads: as the cores are all doing the same thing, if one requests a global shared value from a fixed shared memory location, then all of them are doing the same

So the memory system puts that single value on the bus and all the cores read it: no bottleneck

Sometimes called a *broadcast*

Classifications

Vectors

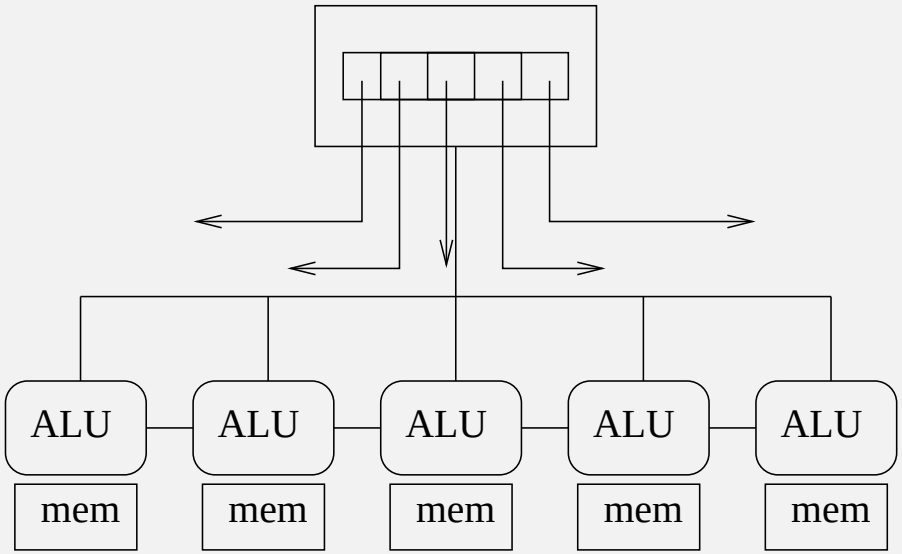


One read goes to all cores

Classifications

Vectors

However, as is often the case, it can be that each core wants a value from a different part of global memory. E.g., core *k* wants the *k*th element from a array



Reading a vector of values

Classifications

Vectors

In this case, it takes careful management, both by the hardware and by the programmer, to ensure the transfers use the shared memory bus efficiently

The case of sending the sending the k item to the kth core is often optimised by the hardware using *coalescence*

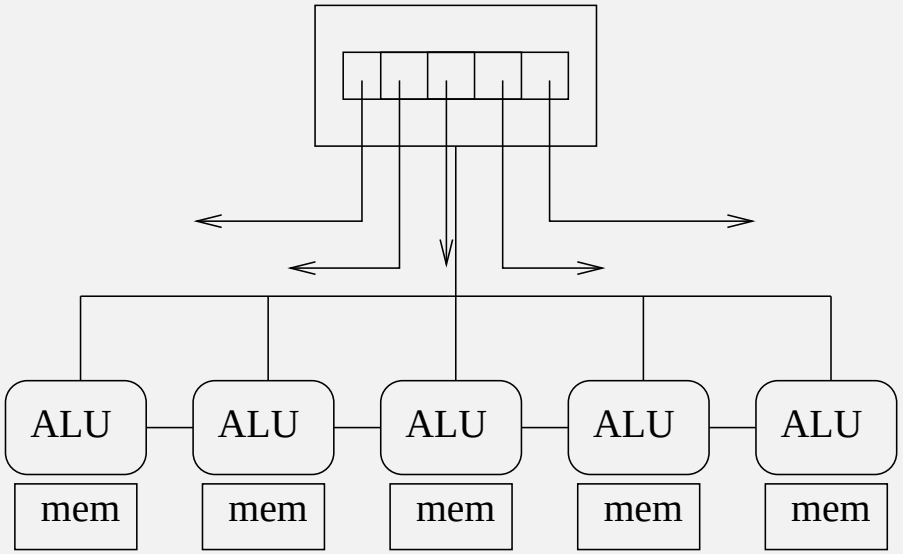
Using a wide bus (e.g., 512 bits) a *single* read operation can fetch multiple data (e.g., 16 integers) and put them all on the bus simultaneously

Each core reads the value it wants

The next 16 values are sent in the next transfer; and so on

Classifications

Vectors



A single fat read goes to multiple cores

Classifications

Vectors

However, it needs data accesses in the program to be of certain patterns for this to work, e.g., linear access to an array

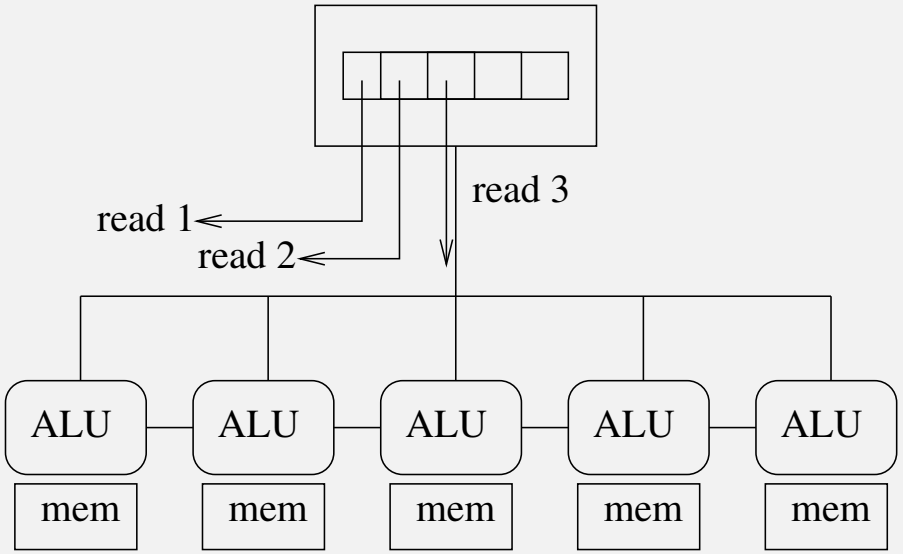
The kinds of access pattern allowed for coalescence are dependent on what the hardware supports, but are generally picking some subset of a contiguous chunk of the shared memory

Otherwise, the reads cannot be coalesced and might require many (e.g., 16) individual reads: much slower

E.g., proc k wants value k^2 from the array

Classifications

Vectors



Awkward distribution done in multiple reads

Classifications

Vectors

Similarly for writes: e.g., core k writing a value to the kth slot in an array could be coalesced

Multiple writes to a single location make no sense and are often disallowed by the system

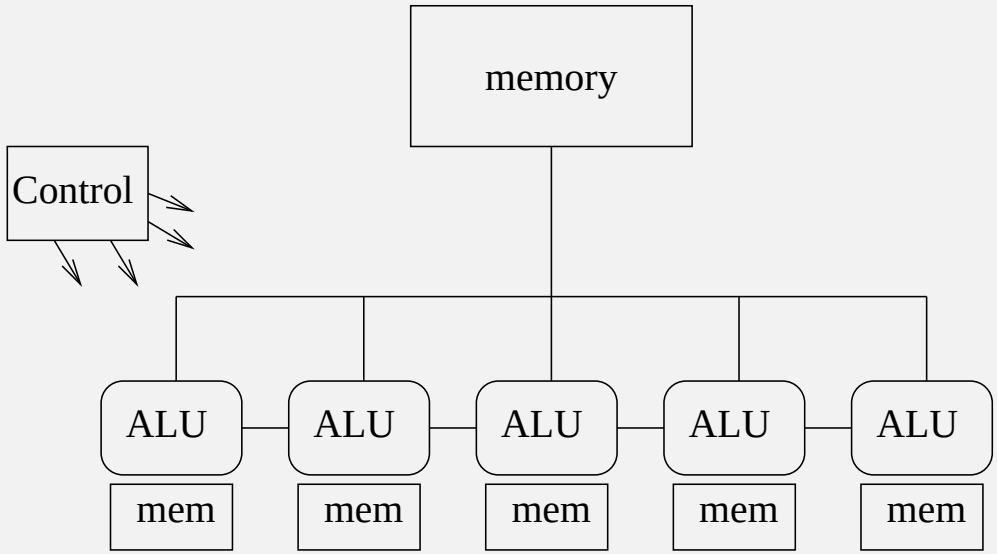
Exercise Consider the case of indirecting through a pointer to global memory (a) when each core points to the same location and (b) when each core points to a different location in the global memory

Exercise Consider the case of indirecting through a pointer to local memory (a) when it's pointing to the same location on all cores and (b) when it's pointing to a different location on each core

Classifications

Vectors

Often there is fast direct communications between neighbouring CPUs



Neighbour connections

This allows data to shuffle up and down the vector very quickly: many problems (e.g., differential equations solving) work on data and neighbour data in this way

Classifications

Arrays

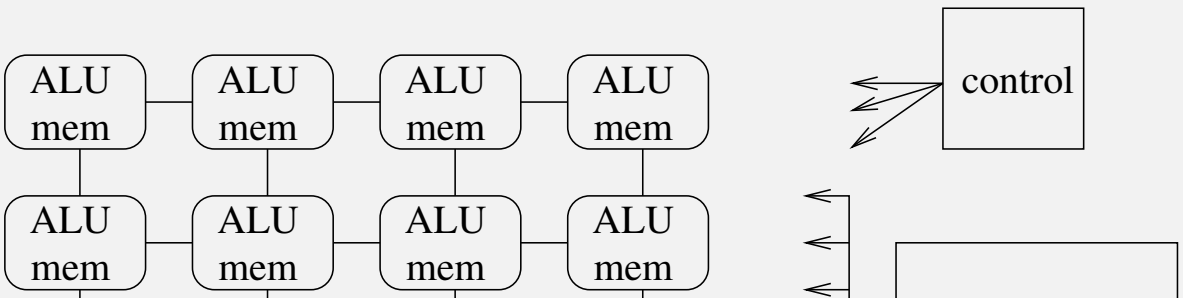
Clearly, vector processors are SIMD and not suitable for MIMD or even SPMD

Vector processors appeared early in parallel computing as they are relatively easy to build: ALUs are relatively easy to build and replicate, while control units are complex and hard

Classifications

Arrays

An extension of the idea was the *array processor*



Classifications

Arrays

The CPUs are in SIMD lockstep as before, but now in an array

Fast connections in two or more directions

This fits well with 2 dimensional differential equation problems

More expensive than vector processors and much less common

Classifications

Arrays

Early array processors were very simple, but they became bigger as technology advanced

		CPU	#CPUs	mem/CPU
DAP	1979	1 bit	4k	4k bits
CM	1983	1 bit	64k	few kB
MPP	1983	1 bit	16k	1 kB
MasPar	1990	4 bit	16k	16kB
MasParII	1992	32 bit	64k	64kB

DAP: ICL Distributed Array Processor
CM: Connection Machine (pretty lights)
MPP: Goodyear Massively Parallel Processor

Classifications

Arrays

Despite being very wimpy processors, this was made up by having so many of them

Their throughput (results achieved per second) is quite respectable

They work very well for certain kinds of problem (e.g., weather forecasting), but are not suited to many other kinds of problems

Vector/array processing processors are important due to their influence on the design of GPUs

Classifications

Arrays

Shared, distributed and vector processors are the three major architectures used today

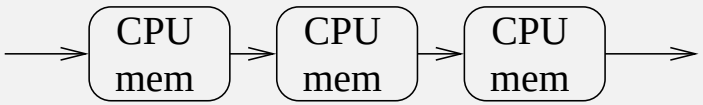
But others have been tried, with varying levels of success

Classifications

Pipelines, Systolic Arrays

Similar looking to vector processors, but actually quite different, are *systolic arrays*

These generalise CPU instruction pipelines to processes



Process pipeline

The CPUs are independent (MIMD/MPMD), each performing one step in the transformation of the input data

More often found in hardware to solve specific problems; not often found as a generic machine

Exercise Could this be classified MISD?

Classifications

Pipelines, Systolic Arrays

For example, a graphics card might want to do clipping of polygons, then colouring, then shading

Each step separate, but compute intensive

Just as pipelining instructions in a processor allows instructions to be processed faster, pipelining these kinds of computations allows pixels to be computed faster

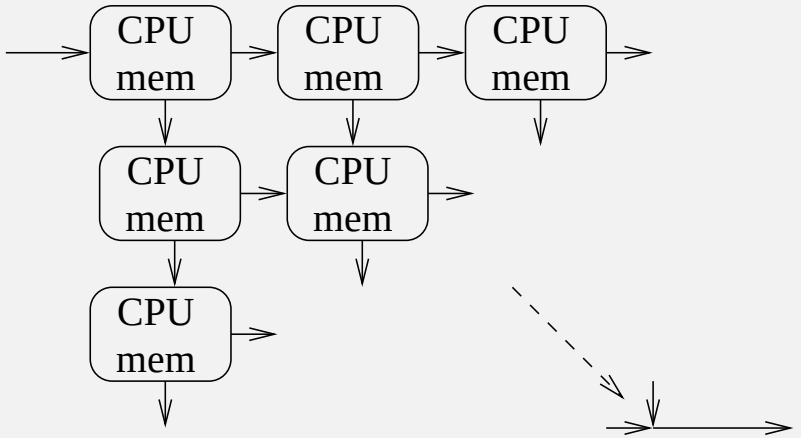
Used in graphics coprocessors as another form of parallelism

Part of the reason why digital TV is delayed relative to realtime is that the encoding of the picture goes through a big pipeline before it is transmitted: there is an inherent *latency* in pipelines

Classifications

Pipelines, Systolic Arrays

Systolic arrays are the obvious extension



Systolic array

but it is unclear if these were ever built

Classifications

Extensions of von Neumann

So why do all these varieties of parallel architecture exist?

There is essentially just one way uniprocessor machines are built: the von Neumann model

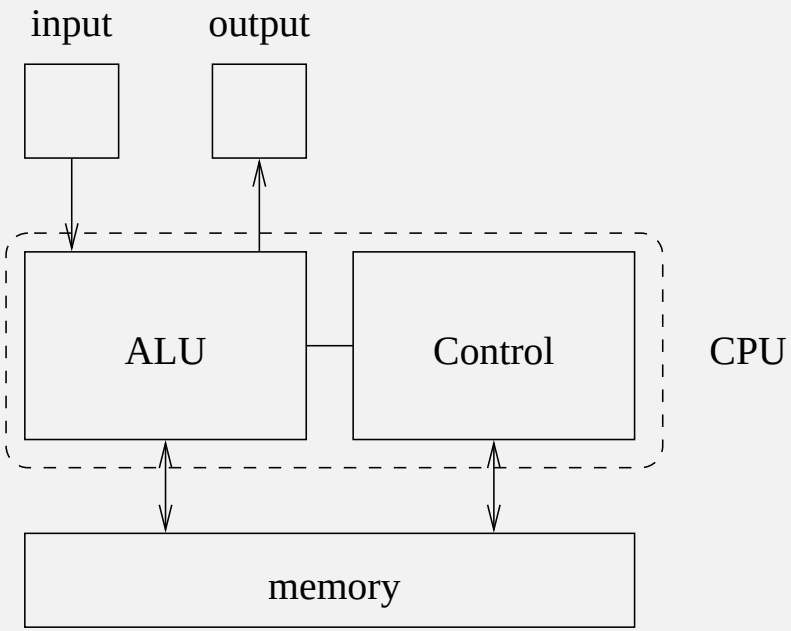
Is there a model that encapsulates multiprocessors in the same way?

There are many contenders, but no obvious winner

Classifications

Extensions of von Neumann

We have the original von Neumann 5 box model

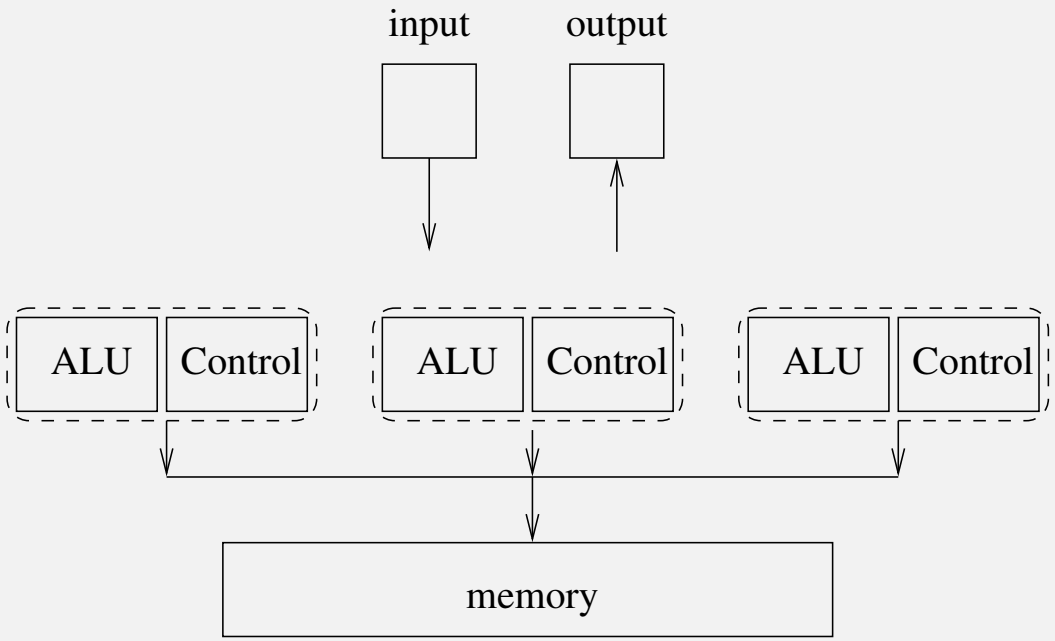


von Neumann 5 box model

Classifications

Extensions of von Neumann

Shared memory MIMD

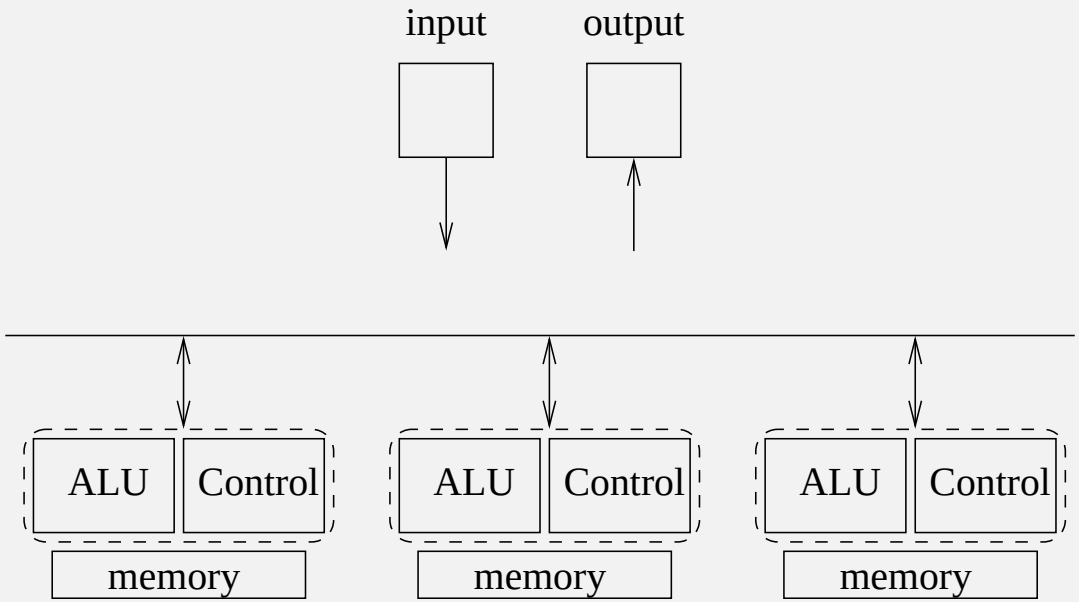


Shared memory box model

Classifications

Extensions of von Neumann

Distributed memory MIMD

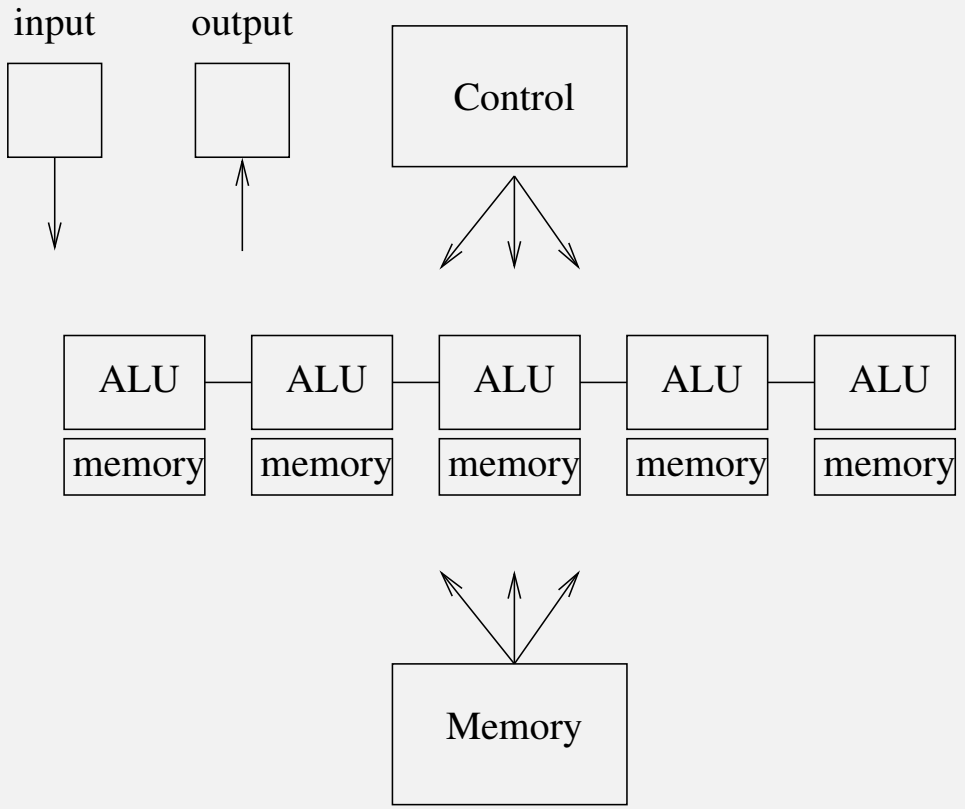


Distributed memory box model

Classifications

Extensions of von Neumann

Vector processor



Vector processor memory box model

Classifications

Extensions of von Neumann

Perhaps there just isn't a single extension of von Neumann that is suitable as a one-size-fits-all solution

Or perhaps we just haven't thought of it yet?

Classifications

Extensions of von Neumann

There are several theoretical models whose aim is to guide the design of parallel algorithms and allow the analysis of them

As with von Neumann, the idea is that you

- write your program in accordance with the model
- the model maps well onto all kinds of real hardware
- therefore your program maps well onto all kinds of real hardware

Classifications

Extensions of von Neumann

Firstly: **PRAM**

The *Parallel Random Access Machine* model idealises a parallel computer as shared memory MIMD, concentrating on the memory bottleneck

You have a choice of how memory can be accessed:

- *Exclusive Read Exclusive Write* (EREW). Each memory location can only be read or written by *one* processor at a time. The simplest architecture
- *Concurrent Read Exclusive Write* (CREW). Each memory location can be read by many processors simultaneously, but written by just *one* processor at a time (c.f. global memory in a vector processor)

Classifications

Extensions of von Neumann

- *Concurrent Read Concurrent Write* (CRCW). Each memory location can be read or written by *many* processors simultaneously. Not a realistic model
- *Exclusive Read Concurrent Write* (ERCW). The fourth combination, never used.

Classifications

Extensions of von Neumann

PRAMs make many further simplifying assumptions, including:

- Memory is symmetric: every location is accessed at the same speed. Decreasingly realistic
- There are an unlimited number of processors: there's always another processor if you need it. Seems unrealistic, but not so bad as you think as most programs are unable to make use of the hardware that we currently have
- Memory is unlimited. This assumption is also often made in analysis of uniprocessor algorithms

Classifications

Extensions of von Neumann

In the early days of Computer Science, there were many clever algorithms invented to deal with the lack of available memory

And, to some extent, memory is still limited in some modern architectures that have very large numbers of CPUs so proportionally each has only a small share of memory

And people want to run programs on datasets of ever-increasing size

Classifications

Extensions of von Neumann

So you analyse your program, counting numbers of memory accesses it makes (according to which of EREW/CREW/CRCW you have chosen) and this gives you a measure of the time your program will take to run

This is primarily a MIMD model, but you can analyse SIMD using it

It totally ignores important realities like NUMA and other overheads, such as communication time in a distributed memory system

But it gives you a rough idea and it is extensively used in analysis of parallel algorithms: we'll have some examples later

Classifications

Extensions of von Neumann

Next: **BSP**

The *Bulk Synchronous Parallel* model

This model takes communication time into account

It assumes processors with local memory communicating over a network

Good for distributed, but can be used for shared memory where you just have smaller communication costs

Classifications

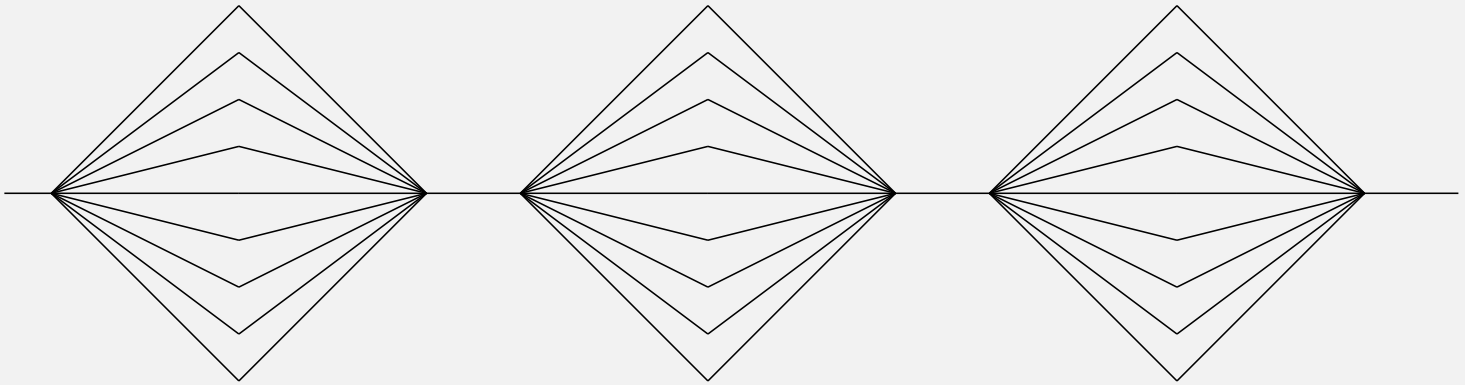
Extensions of von Neumann

A computation is modelled as a sequence of *supersteps*

- each processor does some computation (MIMD, but could be SIMD)
- each processor does some communication
- each processor waits at a global *barrier* until everybody has finished their communications. This is the “bulk synchronous” part
- repeat

Classifications

Extensions of von Neumann



BSP supersteps

Classifications

Extensions of von Neumann

Processing is simplified in this way to give us a chance of being able to make an analysis

Fortunately, many real computations are not too far from this shape

More realistic than PRAMs, but harder work to get analyses out of it

But those analyses tend to be a better match to realistic hardware

Classifications

Extensions of von Neumann

And so on for many other models, some practical, some not

For example, parallel Turing machines and *Communicating Sequential Processes* (CSP) amongst others. Both better at describing the nature and limitations of parallel programs than for investigating how well they work

But the fact remains that there is not one simple theoretical model that works well for all kinds of parallel processing

This might be the source of the confusion in parallel hardware, but we have to live with it

Analysis

So we need to look at how to analyse parallel algorithms

Analysis of parallel algorithms is like analysis of sequential algorithms, just more complicated

Later we shall see statements like “this takes time $O(n^2)$ using $O(p)$ processors”

But we shall start with a few simple measures that we can use to indicate how well our parallel algorithms are working

They are quite crude, but effective

Analysis

Speedup

They mostly measure the parallel algorithm in comparison with a corresponding sequential algorithm

Or a parallel *implementation* with a corresponding sequential implementation: by timing actual running code

We have seen that having p processors won't necessarily make our program run p times as fast

The *speedup* using p processors is

$$S_p = \frac{\text{time on a sequential processor}}{\text{time on } p \text{ parallel processors}}$$

Ideally we'd like $S_p = p$, but this never happens

Analysis

Speedup

Usually S_p is much smaller than p for several reasons

Firstly, there is communications overheads between processors

This might be fairly small for shared memory, or large for distributed memory, but it is present

Time spent communicating is time not spent computing

Analysis

Speedup

So more communications (data movement) will tend to lead to smaller speedups

For example, speedups on distributed memory machines can be reduced as the cost of communications is quite high

But speedups can improve for a larger computation where the *relative* cost of communications drops

Remember clusters are used for large problems where the emphasis is on size, not speed

Analysis

Slowdown

In really bad cases, $S_p < 1$, i.e., our parallel program goes *slower* than our sequential program even though we've thrown all this expensive hardware at it!

This is more common than we'd like

Analysis

Speedup: Amdahl's Law

Now there is the natural upper bound of $S_p \leq p$: we wouldn't expect to get more speedup than the number of processors we have

But it turns out that the number of processors is generally not the limiting factor on speedup: there is another fundamental restriction on speedup that is often overlooked

Amdahl's Law reveals a natural upper bound on the speedup that is theoretically possible even before we add in implementation overheads

Analysis

Speedup: Amdahl's Law

Suppose we have a problem of which 90% can be run in parallel, leaving 10% sequential code

For example, we have to read data before we can process it: a necessary sequentiality. Similarly for writing after processing. Or the add after the square in $x^2 + 1$

So there's always *some* sequentiality

But in the best possible case, using an unlimited number of processors, we might be able to get the parallel part down to as close to zero time as we like

We still have the 10% sequential part

Analysis

Speedup: Amdahl's Law

So the speedup is

$$S_\infty = \frac{\text{time on a sequential processor}}{\text{time on parallel processors}} = \frac{100}{10} = 10$$

A speedup of 10 even on an infinite number of processors

It doesn't even matter what the problem is, or what hardware we have

Analysis

Speedup: Amdahl's Law

This is Amdahl's Law:

> *Every program has a natural limit on the maximum speedup it can attain, regardless of the number of processors used*

Analysis

Speedup: Amdahl's Law

We can quantify Amdahl's Law:

Let $T = T_{\text{seq}} + T_{\text{par}}$ be the time spent in the sequential and parallel parts of our problem on a sequential processor

Then the *maximum* speedup S_p using p processors on the parallel part is

$$S_p \leq \frac{T_{\text{seq}} + T_{\text{par}}}{T_{\text{seq}} + T_{\text{par}}/p}$$

where we have perfectly parallelised the parallel part

Analysis

Speedup: Amdahl's Law

Thus there is a natural upper limit on how fast programs can go

Most do I/O, which must be serialised (made sequential)

Further, as $p \to \infty$, we get

$$S_\infty \leq \frac{T_{\text{seq}} + T_{\text{par}}}{T_{\text{seq}}}$$

so there is a limit even given infinite processing power

This limit is determined by the time taken in the sequential part of the computation

Analysis

Speedup: Amdahl's Law

To see this consider the fraction $x = T_{\text{seq}}/(T_{\text{seq}} + T_{\text{par}})$ which is the proportion of the sequential part within the whole

Note that $0 \leq x \leq 1$, and that rearranging the above gives

$$S_p \leq \frac{1}{x + (1 - x)/p}$$

And so

$$S_\infty \leq \frac{1}{x}$$

is bounded

Analysis

Speedup: Amdahl's Law

Note that Amdahl **does not say anything about how the speedup varies with** p

All Amdahl says is that an upper limit exists

Your program may not get anywhere close to that limit and often in real programs, does not

Analysis

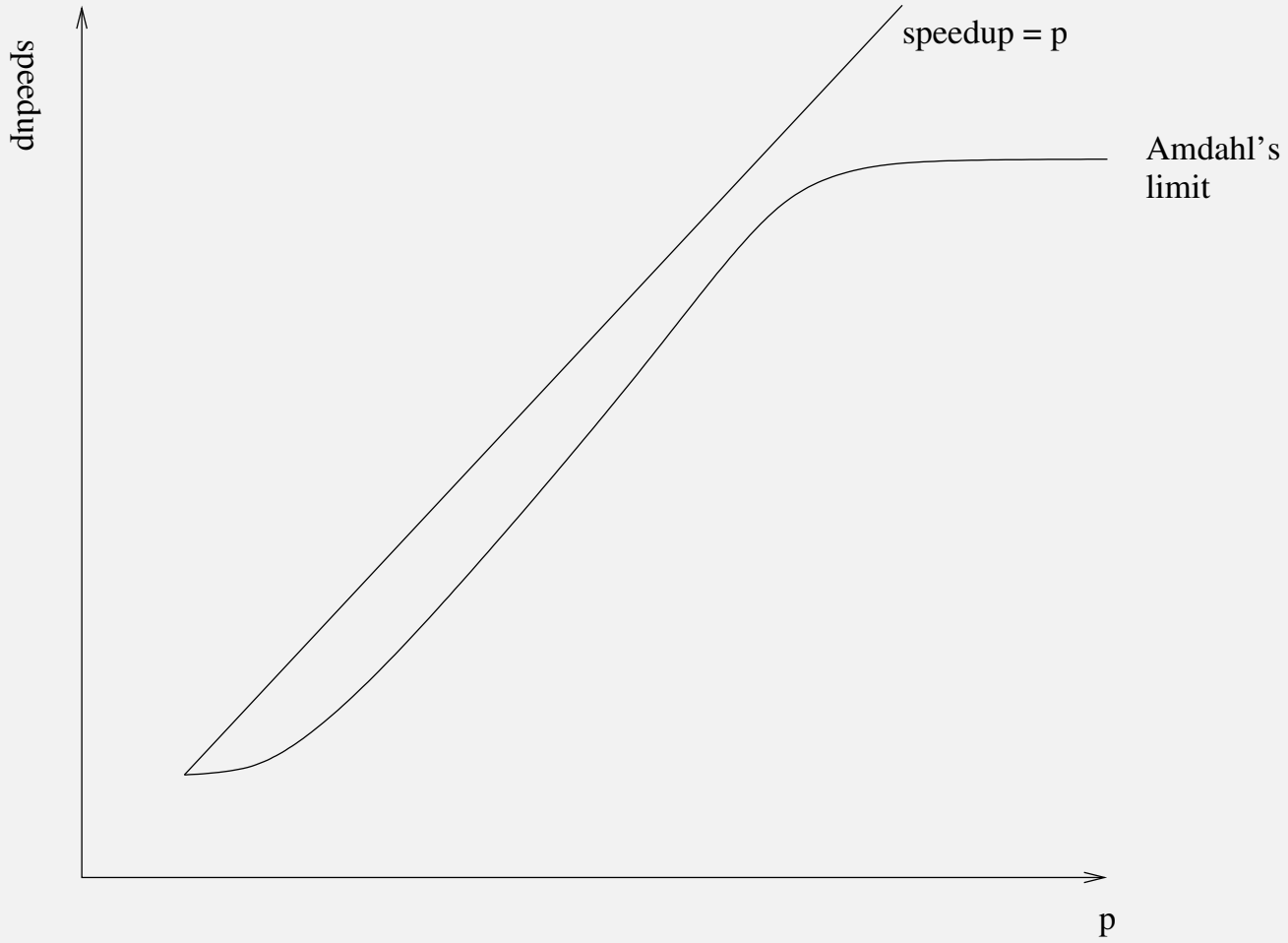
Speedup: Amdahl's Law

In real programs, there are many other factors that affect speedup, so that the speedup may well vary all over the place as p increases

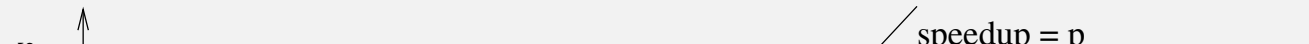
It can even decrease as p gets larger

Analysis

Speedup: Amdahl's Law



Speedup in theory



Analysis

Speedup: Amdahl's Law

To emphasize: all we know is that actual speedup is below Amdahl's limit

Exercise Show that if $0 \leq x \leq 1$, then

$$\frac{1}{x + (1 - x)/p} \leq p$$

Exercise What is the maximum speedup of a program that is 100% sequential?

Analysis

Speedup: Gustafson's Law

Amdahl's law is real: there is a natural limit on speedup *for a given problem*

But there's another point of view

Gustafson pointed out that in real life larger machines tend to attract larger problems

Amdahl assumes a fixed size of problem

Gustafson's Law (occasionally called *Gustafson-Barsis's Law*) gives us another limit

Analysis

Speedup: Gustafson's Law

Suppose we have a problem of size n

$$S_p(n) \leq \frac{1}{x_n + (1 - x_n)/p}$$

where $S_p(n)$ is the speedup on p processors for a problem of size n; x_n is the fraction of the computation spent sequentially

Gustafson argues: as n gets larger, the sequential part relatively decreases, so $x_n \to 0$ (p is fixed)

So

$$S_p(\infty) \leq p$$

i.e., we now get a speedup limit that is the "perfect" speedup p — on an infinitely sized problem

Analysis

Speedup: Amdahl's Law, Gustafson' Law

Both Amdahl and Gustafson are correct: they just apply to different cases of scaling

Amdahl: fixed problem, scaling processing power (sometimes called *strong scaling*)

Gustafson: fixed processing power, scaling problem

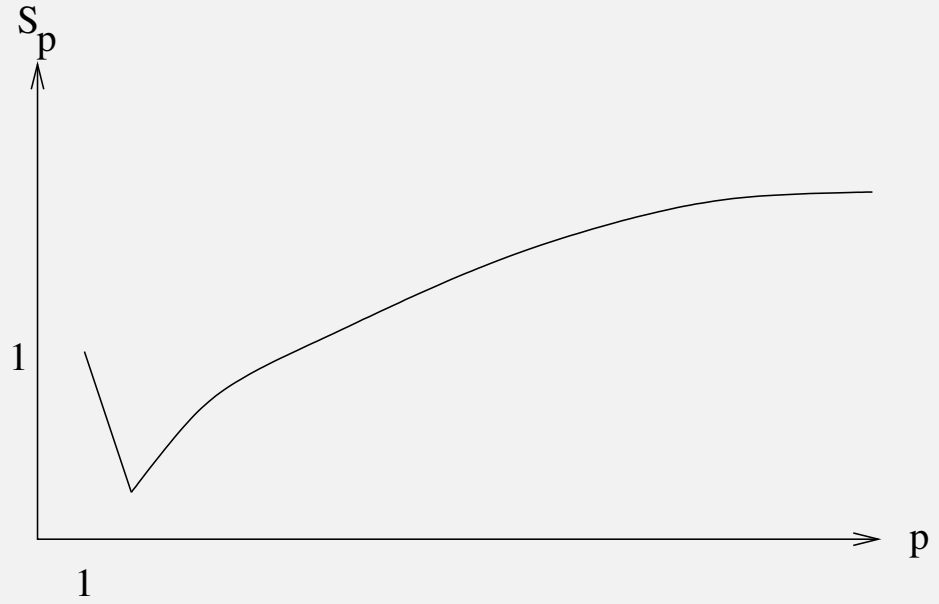
This should convince you that even a simple measure like speedup can be problematic!

But it does re-emphasise the fact that parallelism is not about making things faster, but about making things larger

Analysis

Speedup

Speedup is a simple measure, often proving that your parallel program is slower than it ought to be



Typical speedup curve

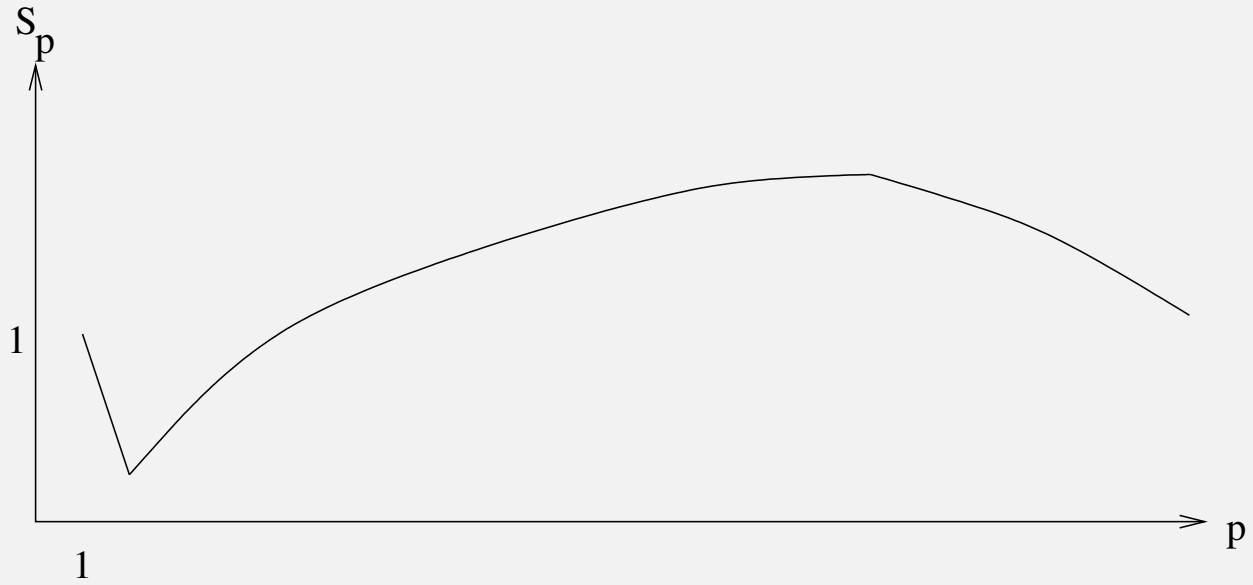
Sometimes it takes p to be surprisingly large before you even catch up with the uniprocessor time with $S_p = 1$ (sometimes never!)

Analysis

Speedup

Very common is the low start, a modest increase, then a tailing off

But taking it further



Adding more processors

We might eventually find adding processors makes it slower!

Analysis

Speedup

This is usually due to increased communications between the processors adding more overhead but not more speedup, perhaps due to Amdahl

Of course, it's not always this bad, but it's quite common!

It does mean there is often an optimum number of processors for a given size of problem that achieves the best speedup

Of course, these are only typical behaviours: a given program may behave quite differently from all of this

Analysis

Speedup

Exercise Consider what might be the difference between a sequential implementation of something and a parallel implementation running on one processor

Analysis

Superlinear Speedup

You will get used to seeing $S_p < p$

On the other hand, it is possible that $S_p > p$

This seemingly impossible condition is called *superlinear speedup*

It is quite rare in real life, but it really can happen that a program runs more than p times as fast on p processors

This can happen for a variety of reasons, some technological, and some more philosophical

Analysis

Superlinear Speedup

The first technological reason is due to cache memory

Cache memory is a lot faster than main memory so if you can fit your problem entirely in cache, it will run faster

For example, a Core i7: perhaps 200 cycles to access main memory, compared to 2 cycles for a L1 cache hit

p processors might have *p* times the cache of a single processor, so a problem spread across the processors might well fit in the larger amount of cache available

Of course, this takes a certain kind of low-communication, easily dividable problem to work; and the right hardware

Analysis

Superlinear Speedup

Note: modern CPUs tend to share cache across multiple cores, so it is unlikely p cores has p times as much cache

(This helps with cache coherence!)

Analysis

Superlinear Speedup

Another (more philosophical) reason is due to the way speedup is defined

$$S_p = \frac{\text{time on a sequential processor}}{\text{time on } p \text{ parallel processors}}$$

What are we comparing against what?

Here is an example to illustrate the issue

We have bubblesort running on a uniprocessor: we wish to make it run on a parallel machine

Analysis

Superlinear Speedup

One way of doing this is:

- split the data into equal halves
- bubblesort each half in parallel
- merge the two sorted lists together

This is 2-way parallelism

The middle step can be itself parallelised recursively: split into two, bubble and merge, giving 4-way parallelism

Depending on the number of processors we have, we can keep recursively dividing

Analysis

Superlinear Speedup

This seems like a reasonable way to implement bubblesort on a parallel machine

What is the speedup? We need to find out how long each version takes to run

Normal bubblesort takes time $n^2/2 + O(n)$ comparisons in the average case to sort n items

So bubblesorting the two halves (in parallel) takes time

$$(n/2)^2/2 + O(n/2) = n^2/8 + O(n)$$

Analysis

Superlinear Speedup

Merging n values takes $O(n)$, giving a total of

$$n^2/8 + O(n) + O(n) = n^2/8 + O(n)$$

time

This gives speedup

$$S_2 = \frac{n^2/2 + O(n)}{n^2/8 + O(n)} \approx 4$$

Already superlinear!

Analysis

Superlinear Speedup

On 4 processors we could repeat: the speedup we get is $S_4 \approx 16$

Clearly this a wonderful algorithm

If we were to implement it, we would truly see these speedups

What is happening?

Analysis

Superlinear Speedup

Consider the same subdividing algorithm on a *single processor*

Time to bubblesort halves: $2 \times (n^2/8 + O(n)) = n^2/4 + O(n)$; time to merge $O(n)$; total $n^2/4 + O(n)$

"Speedup"

$$S_1 = \frac{n^2/2 + O(n)}{n^2/4 + O(n)} \approx 2$$

So we win even on a uniprocessor

Analysis

Superlinear Speedup

What is happening is that bubblesort is a really poor sorting algorithm on average

By subdividing and merging we are converting it into a different kind of sort: if we recurse all the way we have actually implemented a *merge* sort

Merge sort has complexity $O(n \log n)$

Analysis

Superlinear Speedup

The point of this is that by converting bubblesort to be parallel in this way we are fundamentally changing it

This is an extreme case, but in general we must be care when computing speedups that we are comparing like with like

It may not always be possible to have a suitable parallel version of an algorithm: in such a case “speedup” is not meaningful

In most real cases we don’t get this effect, but it’s worth being aware that it can happen

Analysis

Speedup

Some people go further and define speedup as

$$S_p = \frac{\text{time of the best possible sequential algorithm}}{\text{time on } p \text{ parallel processors}}$$

but this has its own problems, not least that we might not know the best possible sequential way of doing things

And we now might be comparing two completely unrelated algorithms

Analysis

Speedup

In a similar vein, another reason for getting superlinear speedups is that the original, sequential, program was poorly written

Perhaps the programmer spent more time thinking about the parallel version, or gained more experience from writing the sequential version, making it substantially better code than the sequential version

This is much the same as the “transform bad algorithm to better algorithm” above, but is now “transform bad code to better code”

So, again, we are not really comparing like with like

Analysis

Speedup

And occasionally we see superlinear speedup due to randomness

If the data contains random numbers, or there is something that adds an elements of randomness to the run time we can get a superlinear speedup

This time due to the parallel version “getting lucky” and hitting a special case that finishes early relative to your measured sequential version

So also not comparing like with like

You would need to ensure each run had the same randomness to be properly comparable; or run many times and take an average time

Analysis

Speedup

In conclusion: speedup is a nice and simple, easy to understand measure: but we have to take care over what it is telling us

Some problems are *pathologically parallel*, meaning they fall easily into parallel parts that have a minimum of communication

For such problems it is easy to get good speedups

E.g., graphics rendering, weather forecasting, parameter sweeping, etc. Often they are data parallel problems

Other problems fare less well — in terms of speed — from parallelisation!

Analysis

Efficiency

If we are lucky enough that S_p increases with p we can make our program get faster by adding more processors

But at what cost?

If we can double the speed of a program using 4 processors we feel we are doing better than if we used a different approach that needed 8 processors for the same speedup

Efficiency measures this

Analysis

Efficiency

Efficiency is speedup per processor:

$$E_p = \frac{S_p}{p} = \frac{\text{time on a sequential processor}}{p \times \text{ time on } p \text{ parallel processors}}$$

Usually $0 \leq E_p \leq 1$, and is often written as a percentage

$E_p = 0.5$ (50%) means we are using only half of the processors' capabilities on our computation; half is lost in overheads or idling while waiting for something

$E_p = 1$ (100%) means we are using all the processors all the time on our computation

$E_p > 1$ indicate superlinear speedup: we are using more than 100% of the processors!

Analysis

Efficiency

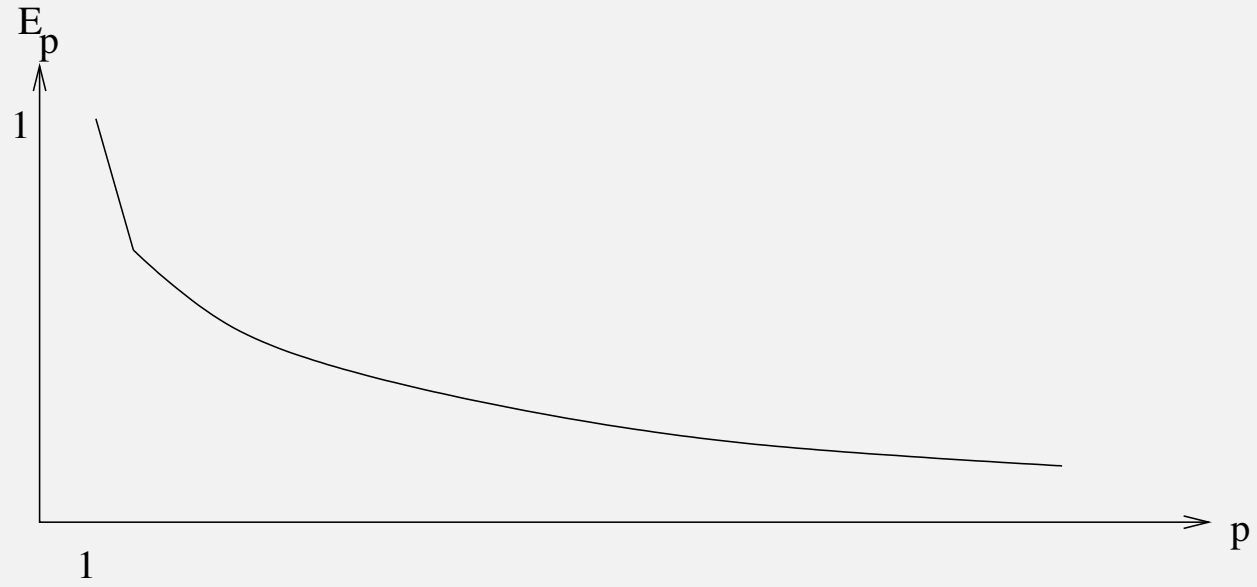
Efficiency is useful when we need to gauge the cost of a parallel system: the higher the efficiency the better the utilisation of the processors

When $E_p < 1$ this indicates that somewhere at some point a processor not working on the computation. Perhaps it is occupied in communication; or possibly just lying idle waiting

Analysis

Efficiency

Typical efficiency graph on a fixed size problem:



Efficiency graph dropoff

Analysis

Speedup and Efficiency

As an example of calculating speedup and efficiency we consider a pipeline (systolic array)

Systolic array

Data moves from one processor to the next being transformed at each stage: we assume one time step per transform

This could equally be a CPU instruction pipeline

Analysis

Speedup and Efficiency

A p-stage pipeline will take p time steps to fill; after that it produces one result per time step

So it can produce n results in $p + n - 1$ time steps

A sequential system will take np time steps to do the p steps on the n computations

Analysis

Speedup and Efficiency

The speedup is

$$S_p = \frac{np}{p+n-1} = \frac{p}{(p-1)/n+1}$$

As time passes, the number of tasks n gets large, and $S_p \to p$

A p-stage pipeline has a speedup is less than p, but that gets closer to p as time progresses

Also, the speedup starts low (for $n = 1$, $S_p = p/(p+1-1) = 1$) and increases over time, getting closer and closer to p

Analysis

Speedup and Efficiency

The efficiency is

$$E_p = \frac{S_p}{p} = \frac{n}{p+n-1} = \frac{1}{(p-1)/n+1}$$

As n gets large, $E_p \to 1$

Eventually we are (close to) using all the processors all the time: perfect efficiency!

Also, the efficiency starts low (for $n = 1$, $E_p = 1/(p+1-1) = 1/p$) and increases over time

Analysis

Speedup and Efficiency

Pipelines are a really good way of making something parallel: both great speedup and great efficiency

As long as we can keep the pipeline full: in a CPU each time we take a jump the instruction pipeline breaks, empties and needs to refill

To keep high efficiency we need to avoid this: thus the complications in the designs of modern processors that are aimed at keeping the pipeline full

(Things like speculative evaluation and branch prediction, using many transistors. . .)

Analysis

Other measures

Speedup and Efficiency are simple, but useful measures of a parallel system, as long as you take care over using them

There are many other measures that are occasionally used, but they are of lesser importance

Exercise Some people use the phrase “negative speedup” rather than “slowdown”. Why is that incorrect?

Analysis

Karp-Flatt

Sometimes people use the *Karp-Flatt metric* as a measure of an implementation to see how well it is doing

This is essentially an empirical measure of the sequential fraction of a computation (important for the Amdahl limit)

$$e = \frac{\frac{1}{S_p} - \frac{1}{p}}{1 - \frac{1}{p}}$$

where S_p is the measured speedup and p the number of processors

Analysis

Karp-Flatt

A larger e indicates a larger sequential part

If we have perfect speedup, $S_p = p$, and $e = 0$

If we have no speedup, $S_p = 1$, and $e = 1$

If we have slowdown, e.g., $S_p = 1/2$, and $e \approx 2$

(If we have superlinear speedup, $S_p > p$, and $e < 0$)

Exercise Calculate Karp-Flatt for the pipeline. What does it tell us?

Analysis

Karp-Flatt

Note that Karp-Flatt will give you an estimate for the sequential time *for a given implementation*

It does not tell us the sequential limit for the *problem*

After all, you might just have a poor implementation

A big Karp-Flatt value is often an indication you need to re-think your code

Analysis

Work Efficient

Next: a parallel algorithm is *work efficient* (*cost efficient*) if the number of operations it performs is no more than the sequential algorithm

For example, a parallel algorithm might duplicate some operations on separate processors as it is more convenient, or reduces communications

The *parallel overhead* is

$$T_o = pT_p - T_s$$

where T_s is the sequential time and T_p is the parallel time

Analysis

Work Efficient

This measures the amount of extra work we are doing to get the parallelism

A measure of the extra energy expended in the parallel algorithm or implementation

And the cost of the overheads (e.g., communication) when we measure a real implementation

Exercise Calculate the parallel overhead for the pipeline. What does it tell us?

Analysis

Isoefficiency

Another question is “how scalable is this algorithm?”

Here we ask for a relationship between p, the number of processors and n the size of the problem for a given efficiency

If we increase p, how much to we have to increase n to maintain a given efficiency?

Analysis

Isoefficiency

Increasing p will generally decrease efficiency (Amdahl)

Increasing n will generally increase efficiency (Gustafson)

A poorly scalable algorithm will need to increase n a lot to maintain efficiency as we increase p

This relationship is called the *isoefficiency*, and expresses n as a function of p

It quantifies the balance between Amdahl and Gustafson

Analysis

Isoefficiency

Computing the isoefficiency can be a bit fiddly, but often it is easiest to start by looking at the parallel overhead

We have efficiency $E = T_s/pT_p$ and overhead $T_o = pT_p - T_s$. Combining these:

$$E = \frac{T_s}{p\left(\frac{T_o+T_s}{p}\right)} = \frac{T_s}{T_o + T_s} = \frac{1}{1 + T_o/T_s}$$

So to keep E constant, we need to keep T_o/T_s constant

Analysis

Isoefficiency

So we must have

$$T_s = cT_o$$

for some constant c

As both T_s and T_o depend on n and p, this equation generally gives us enough to solve for n in terms of p

Analysis

Isoefficiency

Example. The p-stage pipeline had efficiency

$$E = n/(p + n - 1)$$

on a problem of size n

The overhead

$$T_o = pT_p - T_s = p(p + n - 1) - np = p^2 - p$$

independent of n

This fixed overhead again tells us it is a good idea to keep the pipeline full!

Analysis

Isoefficiency

We want $T_s = cT_o$ which is

$$np = c(p^2 - p)$$

We solve for n

$$n = c(p - 1)$$

Thus the isoefficiency is

$$n = O(p)$$

Analysis

Isoefficiency

This is linear in p: if we double p we need only double n to maintain efficiency

So this tells us pipelines are very scalable

Analysis

Measures Conclusion

There are many ways we can measure if our parallel program is performing well, or poorly

But we do need to be careful that we are making meaningful comparisons of parallel and sequential algorithms

Exercise Compute these measures for summing n numbers using p processors

Analysis

Bandwidth and Latency

While we are thinking about measurement of parallel systems we need to make a quick comment about *bandwidth* and *latency* as they play an important role in the way we regard communications overhead

Bandwidth is the number of bytes per second transmitted over some medium

Latency is how long we have to wait for the data to arrive

Analysis

Bandwidth and Latency

Bandwidth is often mentioned in descriptions of things as it is easy to visualise (a rate of flow)

However, latency is often just as important in parallel systems

Bandwidths these days are pretty high: Mb and Gb rates are common

Latencies of milliseconds may *seem* small, but relatively speaking they are the big problem

Analysis

Bandwidth and Latency

Example A memory bus (DDR5) might have 400Gb/sec bandwidth and latency 100ns.

Fast, but processors are faster! Data might arrive at a prodigious rate when it does arrive, but a processor could do a lot of work while it was waiting for the first byte to arrive

This is why processors have lots of complex and clever caching to avoid going off-chip

Analysis

Bandwidth and Latency

Example A local network (10Gb Ethernet) might have bandwidth 10Gb/sec and latency 100μs

This is how nodes in a cluster are often connected

Again we are in the range of hundreds of thousands of instructions while waiting

And this does not include the CPU overhead of going through the Operating System to send and receive the packets from the network

Analysis

Bandwidth and Latency

The latency affects coding strongly: it may be worthwhile doing duplicate computations if that is faster than fetching a value

In large parallel systems compute power is cheap and plentiful, but communications are slow and expensive

This is why when we implement parallel code we really need to concentrate on the communications more than the computations

Analysis

Bandwidth and Latency

It is quite easy to increase bandwidth

Doubling the width of a bus will double the bandwidth, but do nothing to the latency

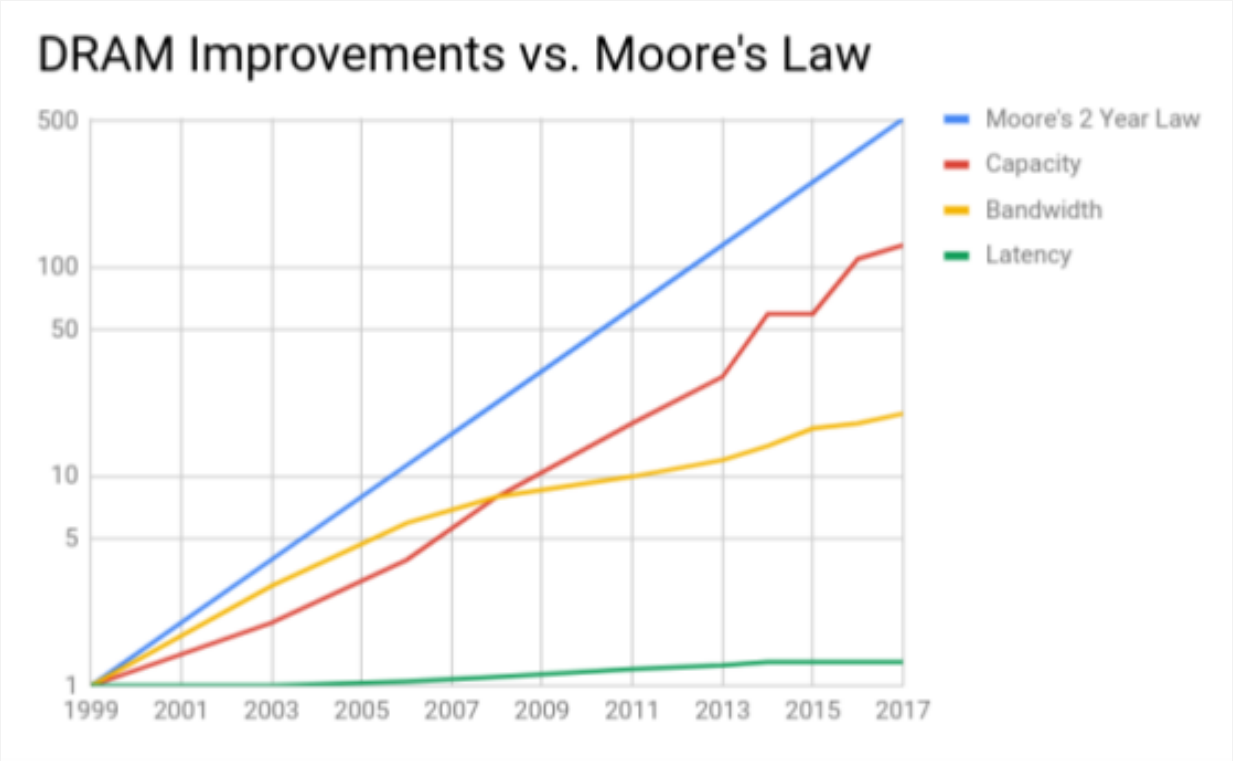
We might get a huge bandwidth by strapping a USB stick to a pigeon: the latency would not be so good, though!

For a long time *sneakernet* was the best way to transmit large volumes of data

Exercise Read about how data was transmitted to generate the recent (2019) image of a black hole

Analysis

Note: Moore says sizes of RAM are increasing, but latencies are far behind



Sizes of RAM over time

Graph from Kevin K. Chang, PhD., CMU 2017

Analysis

Bandwidth and Latency

Latency is often limited by Physics: the speed of light is a big factor on latency these days

Thus, like Amdahl, latency is another natural limit on parallel computation

Particularly on distributed architectures

Shared Memory Systems

We now move on to look at shared memory and distributed memory systems in more detail, in particular the issues that arise in software and programming

We start with shared memory MIMD as people think it seems more similar to SISD than distributed memory is, and so is "easier"

We will look at simple programs that have multiple *threads of control*, i.e., parts of the process are running simultaneously on separate processors

Shared Memory Systems

Note: a single program might use several processes, and each process might contain several threads

Separate processes have separate (virtual) memory address spaces (my memory location 42 is different from your memory location 42)

Multiple threads in the same process (generally) share the same (virtual) address space (my memory location 42 is the same as your memory location 42)

Here we consider the shared part, i.e., threads within a process

Shared Memory Systems

Suppose we want to count the number of positive values in a list of numbers

```
count = 0;
for (i = 0; i < 100; i++) {
    if (val[i] > 0) { count = count + 1; }
}
```

In C or C++ or Java or whatever

It's not really worthwhile parallelising this in real life (**Exercise** why?), but let's try

Shared Memory Systems

We could split this into two blocks

1

```
for (i = 0; i < 50; i++) {
    if (val[i] > 0)  count = count + 1;
}
```

2

```
for (i = 50; i < 100; i++) {
    if (val[i] > 0)  count = count + 1;
}
```

and by magic to be discussed later have blocks 1 and 2 run in parallel on separate processors, sharing the variables (i.e., shared memory)

Shared Memory Systems

1

```
for (i = 0; i < 50; i++) {
    if (val[i] > 0) {
      count = count + 1;
    }
}
```

2

```
for (j = 50; j < 100; j++) {
    if (val[j] > 0) {
       count = count + 1;
    }
}
```

Note we want to share `val` and `count`, but not the loop variables!

No communication or interaction between the threads: instant speedup of 2?

Shared Memory Systems

It may run twice as fast, but sometimes will give the wrong answer!

Sometimes it will give a value of `count` that is too small

The problem is the *shared resource*, the variable `count`

We have two separate threads reading and updating the value

Shared Memory Systems

Occasionally, just occasionally, the following happens

1

```
read the value of count
into a CPU register
add 1
store the value
```

2

```
read the value of count
into a CPU register
add 1
store the value
```

Shared Memory Systems

So both read a value, 10, say. Both add 1 to get 11. Both store 11.

Even if we don't have hardware that supports simultaneous reads and writes (we might have EREW) it can still go wrong

1

```
read the value of count
add 1
store the value
...
```

2

```
...
read the value of count
add 1
store the value
```

Shared Memory Systems

The parallel version is simply an incorrect program

This is another example of a *race condition* where an unexpected or overlooked timing in the execution produces an incorrect result

It is a *data race*: an unsynchronized, concurrent access to data involving a write

Read-only data is always safe to share: nothing can go wrong

But when a write (or multiple writes) is involved, things can go badly wrong

Shared Memory Systems

And notice this can even happen on a single processor, when multiple threads are being timeshared by the OS

The OS may choose to deschedule thread 1 in between its read and write; and schedule thread 2 that reads the old value

Exercise And it might give even worse counts: think why

So this is a concurrency error, and not just a parallelism error

Shared Memory Systems

The race may or may not happen according all kinds of external events that might affect the timing of the execution of the updates

So the program may often be right, and occasionally wrong

Or the program may often be wrong, and occasionally right

The program might always give the correct answer on your machine, but give the wrong answer on your customer's machine

Exercise Compare with deadlocks

Shared Memory Systems

Note: the “obvious solution” of having separate `count1` and `count2` introduces a new, separate, problem we shall address later: for now we need to consider shared resources

Races

Philosophy Exercise A race condition is only a bug if the non-determinism is undesirable. Discuss

Shared Memory Systems

The myriad ways of avoiding race conditions are what keep programmers and theoreticians in their jobs

And the people designing debugging tools

Some debugging tools exist which will find simple errors like the above, but in general we have to rely on programmers finding the bugs by thinking

Shared Memory Systems

Race Condition Detection Tools

Some tools to help detect race conditions:

- Intel Parallel Inspector, a Visual Studio plugin
- Helgrind, a Valgrind plugin
- Data Race Detection (DRD), another Valgrind plugin

Ideally, the programming language itself would prevent you from writing code with races (see later for examples)

Experience tells us it is hopeless to rely on the programmer to get it right!

Shared Memory Systems

Areas of code that use a shared resource are called a *critical region* (also called a critical *section*)

In the above example, the increments of `count` form a (small) critical region

A critical region comprises any pieces of code that access a resource that might be updated in parallel

So, in this example, *any* region of code that updates `count` is critical

So these pieces of code have to be carefully thought out to avoid race conditions

Shared Memory Systems

Such critical regions are rife in parallel programs and appear in many different guises

Sometimes you can run a program 100 times and get the right answer, but on the 101st time it is wrong

Such events can have a very low probability, making them hard to debug by “run it and see if it works”

But they do happen, so you have to find them by hard thought instead

Shared Memory Systems

Locks

The problem is that two (or more) threads are trying to update something at the same time (update $=$ read, modify, write)

In between the read and the write another thread might have gone behind the first's back and updated the thing itself

Shared Memory Systems

Locks

The simplest solution to stop multiple threads updating a resource is to allow only *one* thread at a time to do an update on a shared resource

If a second thread wishes to update while a first has already started, the second is forced to wait until the first has finished

This will ensure correct updates by avoiding the update overlap we saw earlier

Note, though, the second thread will have to wait: this is an inefficiency and if that happens a lot the system as a whole will be slower than it ought

Concurrency Primitives

Locks

We are forcing the bits of code in the critical region into executing sequentially, which Amdahl tells us is bad for speedup

But the sequential execution is essential for the code to be correct

So we need to make critical regions as small and fast as possible

Concurrency Primitives

Locks

One simple way of enforcing this *mutual exclusion* on critical regions is the use of *locks*

Also called: mutexes. Some confused people use *semaphores* (see later), but these are better employed for other problems

A lock is a simple flag that says “Please wait, this region is busy”

Concurrency Primitives

Locks

We must surround all critical regions that update a given shared resource with a grab and release of the lock:

```
get lock                          get lock
do stuff on a resource            other stuff on same resource
release lock                      release lock
```

If a second thread tries to grab the lock it will be made to wait until the lock is released by the first thread

In this way we can ensure that two updates never overlap

Concurrency Primitives

We will get either

```
get lock
do stuff on a resource
release lock
```

```
try to get lock
(wait)
(wait)
get lock
other stuff on same resource
release lock
```

or

```
try to get lock
(wait)
(wait)
get lock
do stuff on a resource
release lock
```

```
get lock
other stuff on same resource
release lock
```

No parallelism on access to the resource!

Concurrency Primitives

Locks

Note that *every* piece of parallel code in the program that updates that resource will have to have to be wrapped in the grab of the lock

If we miss protecting *any* occurrence of a parallel update, the whole thing is broken

This is clearly a good source of bugs

Locks are a very crude method to prevent race conditions, but they are widely used

Concurrency Primitives

Locks

This also applies to more than two threads, of course

The first grab of the lock will succeed, the others will have to wait until the lock is released

If more than one thread tries to grab the lock at the same instant, just one will succeed. The others will have to wait

If there are several threads waiting on a lock, just one will get the lock when it is released: the other threads continue to wait

Concurrency Primitives

Locks

Also, most implementations of locks are *not fair* in the sense that *any* one of the waiting threads will get the lock, there's no first-in-first-out enforced

This is because (a) it's extra overhead for the OS to implement such a FIFO and (b) most programs don't need it, so why have an overhead that most programs don't want?

The threads are likely arriving at the lock in a non-deterministic order, so what's the sense in preserving that random order?

Concurrency Primitives

Locks

Also, **it's bad practice for the programmer to rely on the order of things happening in a parallel system**

If certain things need to happen in a certain order, the programmer must write code to ensure that this happens

You can't rely on luck, or that they usually happen in the right order

Also note that specifying orders on events is another form of sequentiality, which we would like to minimise

Concurrency Primitives

Locks

Often, the wait on the lock is implemented and enforced by the operating system, which might deschedule the waiting thread to free up the CPU for something else to run

With this kind of lock implementation, a thread takes no CPU time while locked

Thus the overhead of this lock is the CPU time it takes for the OS to deschedule and later reschedule the thread (not trivial!)

Concurrency Primitives

Spinlocks

In contrast, sometimes the lock wait is implemented as a *busy wait*: the thread keeps trying in a tight (busy) loop to grab the lock, continually burning CPU cycles

These are called *spinlocks*

The argument is that critical regions should be small to maintain efficiency, so it will only be a short time before the lock will be released

And by the time the OS has descheduled the waiting thread the lock could already be free, so instead just keep busy trying

Concurrency Primitives

Spinlocks

This is good for when responsiveness is more important than CPU cost, e.g., real-time systems, but too expensive for many systems

Note that spinlocks use CPU cycles, thus occupying the CPU, while blocking locks release the CPU so it can potentially used for something else

Concurrency Primitives

Spinlocks

You should take care over using spinlocks rather than blocking locks

They assume that the holding thread only holds the lock for a brief time: but the holding thread can be preempted by the OS at any time

Thus preventing release of the lock for an arbitrarily long period of time

Concurrency Primitives

Spinlocks

Exercise And read about the cache-thrashing behaviour that occurs if the spinlock is not implemented carefully

> *. . . do not use spinlocks in user space, unless you actually know what you're doing. And be aware that the likelihood that you know what you are doing is basically nil*
>
> Linus Torvalds

Concurrency Primitives

Locks

A hybrid implementation will spin for a short while, then pass to the OS: trying to get the best of both approaches

Though there is still great debate over the best approach

Concurrency Primitives

Locks

To use a lock, in pseudocode:

```
countlock = make_a_new_lock();
...
get_lock(countlock);          get_lock(countlock);
count = count + 1;            count = 2*count;
free_lock(countlock);         free_lock(countlock);
```

Remember we must put a grab and release of the `countlock` around *all* updates to `count` in code where there might be more than one thread wanting to update the value

Concurrency Primitives

Locks

For most programming languages it is the responsibility of the programmer to spot all the shared resources that need this treatment and to write correct code to enforce exclusive access

Getting this wrong (e.g., overlooking an update to `count` and not putting in the lock) is the source of one of the most common bugs in parallel programming

Particularly for programmers trained in sequential programming; for sequential programs *all* accesses are already sequential!

Concurrency Primitives

Locks

Also, be careful not to over-lock

We don't need locks when there can only be one thread updating `count`, e.g., in a non-parallel part of the code, or we are already in some protected larger critical region

Over-locking is safe, but simply wastes time and thereby reduces speedup

Concurrency Primitives

Locks

Locks are definitely needed when we update (read then modify) the value of a variable

The question arises regarding whether we need a lock around a simple read of a multi-byte value, such as a 32-bit (4 byte) integer

It is easy to believe some bytes of a value might be written while half-way through being read, resulting in a mix of the bits of the old and new values

Called read (or write) *tearing*

Concurrency Primitives

Locks

However, for most (non-embedded) machine architectures these days it is likely (not certain!) to be safe to read simple values like integers or doubles that fit in a register: the hardware read is atomic (another side effect of the caching mechanism)

Though you do need to be careful on strange machine architectures, or with compilers that try to be too clever (For hackers: think about non-aligned accesses)

Certainly, though, for reading all of a larger object or structure, a lock will be necessary to ensure consistency across the multiple machine reads it takes to read in the whole structure

Concurrency Primitives

Locks

```
int x, y;
...
y = x;
```

Usually safe as reads of `int`s are generally atomic

Concurrency Primitives

Locks

```
// Also OO classes or objects
struct rational {
  int num, den;
};
struct rational r, s;
...
r = s;
```

Possibly unsafe, as it could take two machine reads to get all of `s`, which might be modified halfway through by another thread

Unlikely, but you can't rely on that

Analogously for the write of `r`

Concurrency Primitives

Locks

Exercise For C geeks. There is an aliasing problem with bit fields in a `struct`

```
struct {
   int a: 5;
   int b: 3;
}
```

where an update to field `a` might be implemented as a read of a byte, modifying the bits of `a`, then writing a byte. Investigate

Exercise What about a 128-bit `long long int` on a 64-bit machine?

Concurrency Primitives

Locks

What about when we need to use more than one lock?

Of course, we can and should have separate locks in order to protect separate resources: we *could* use `countlock` to protect updates to another shared variable `sum`, but that would prevent one thread updating `count` while another is updating `sum`, which is perfectly safe to do

The only real reason to share a lock like this would be in when there are severe memory limitations: but lock implementations tend to use only a little memory per lock

Concurrency Primitives

Locks

But we do need to be careful about what we protect from what as it all has a cost

Getting and releasing a lock can be relatively cheap (in some architectures and operating systems; expensive in others) but it is not free: it is an overhead to be taken into account and avoided if you can

In many implementations these days the cost of getting an uncontended lock (not already locked) is cheap, while the cost of getting a lock that is already held is expensive

So the common (you hope) case is cheap

Concurrency Primitives

Locks

Also note, locks can be used to protect anything, not just variables, e.g., whole function calls or whole loops. But we should try too keep the regions small

```
get_lock(mux);
someone_elses_dodgy_code();
free_lock(mux);
```

Another reason to use a single lock could be that the code you want to protect is so complicated you are not clear on how to proceed!

Concurrency Primitives

Locks

Locks are a simple, low level mechanism

Too low level: they are easy to use incorrectly

Suppose we have a couple of variables `x` and `y` we are protecting with mutexes `mx` and `my` respectively. We want to swap their values; elsewhere replace them both by their average

```
tmp = x;
x = y;
y = tmp;
```

```
av = (x+y)/2;
x = av;
y = av;
```

Concurrency Primitives

Locks

To make this safe we have to use both locks

```
get_lock(mx);
get_lock(my);
tmp = x;
x = y;
y = tmp;
free_lock(my);
free_lock(mx);
```

Concurrency Primitives

Locks

Some pages of code later

```
get_lock(my);
get_lock(mx);
av = (x+y)/2;
x = av;
y = av;
free_lock(mx);
free_lock(my);
```

Spot the bug!

Concurrency Primitives

Locks

This will probably work most of the time, but occasionally just hangs doing nothing

Sometimes we will get

1

```
get_lock(mx);
get_lock(my); (waits)
```

2

```
get_lock(my);
get_lock(mx); (waits)
```

This is simple deadlock, another race condition

Concurrency Primitives

Locks

A very easy error to make, but often very difficult to find, particularly as the locks of `mx` and `my` may be widely separated in the code, or in someone else's code

The use of locks requires a great deal of careful management when the code gets large

Exercise Why wouldn't having another mutex `mxy` to protect both `x` and `y` solve things?

Concurrency Primitives

Locks

If we want to use a lock in portable code, we can use a library specification like *POSIX*

This is a standard that covers a large number of functions, specifying their use and behaviour

Concurrency Primitives

POSIX `pthread`

The `pthread` section on the POSIX specification contains several functions that we shall soon be looking at:

- Locks: `pthread_mutex_ init`, `lock`, `unlock`, `destroy`
- Barriers: `pthread_barrier_ init`, `wait`, `destroy`
- Condition Variables: `pthread_cond_ init`, `wait`, `signal`, `broadcast`, `destroy`
- Semaphores: `sem_ init`, `post`, `wait`, `destroy`
- Management: `pthread_ create`, `join`

And many others

Concurrency Primitives

POSIX `pthread`

For example, `pthread_create` (we shall come back to this later)

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

is how to create a new thread: it takes an *attribute* (always `NULL` for our purposes), a function of one argument to start executing, and a value to pass as the argument to that function

It returns a *thread identifier* in the first argument

Concurrency Primitives

POSIX `pthread`

Documentation for POSIX pthread functions is available everywhere, online and possibly on your own computer

For example, on Linux you can use manual pages, e.g.,
`man pthread_create`
to get detailed information

Concurrency Primitives

POSIX Locks

A real example of locks, as defined by the POSIX standard, where they are called mutexes

```
#include <pthread.h>
pthread_mutex_t mutex;
```

An (uninitialised) mutex

Concurrency Primitives

POSIX Locks

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t
                             *restrict attr)
```

Initialises the mutex pointed at by the first argument, returns a 0 that indicates success or non-0 to indicate failure

POSIX locks come with various attributes: the default (NULL) is normally what you want

```
pthread_mutex_t mut;
if (pthread_mutex_init(&mut, NULL) != 0) { ...error... }
```

Concurrency Primitives

POSIX Locks

There is a alternative static way to initialise mutexes if all you need is a basic lock:

```
// declare and initialise
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Concurrency Primitives

POSIX Locks

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

The main grab and free functions

It is an error to try and unlock a mutex that is held by another thread: the thread that locks must be the thread that unlocks

This is a POSIX specification designed to make locks widely implementable of a variety of architectures

And this is not a limitation: it is a desired behaviour. If you allowed another thread to unlock a mutex you can bet this would be misused by some programmers thus opening a new opportunity to write buggy code

Concurrency Primitives

POSIX Locks

“It is an error”: some implementations return an error value, while others (depending on the OS) have undefined behaviour

Some versions of mutexes also allow *recursive* (or *reentrant*) locking, where a thread that already owns a lock can lock it again; it needs to do the same number of unlocks to free the lock

Non-recursive versions just self-deadlock, or have undefined behaviour

Concurrency Primitives

POSIX Locks

On fairness of POSIX mutexes:

Posix says “the scheduling policy shall determine which thread shall acquire the mutex” if more than one is waiting

This allows implementations to take `pthread_attr_setschedpolicy` and thread priorities into account: we shall not talk about that here!

Concurrency Primitives

POSIX Locks

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Like `pthread_mutex_lock` but return immediately (without getting the lock) if the lock was already held. It returns a value of 0 if it got the lock, a non-zero otherwise

This function is occasionally useful, but less than you might believe, as the result doesn't quite mean what people think it means (sequential assumptions. . .)

Concurrency Primitives

POSIX Locks

It doesn't say "the mutex *is* locked", but really says "the mutex *was* locked"

It gives the instantaneous state of the lock at the time of the `trylock` function call: it is possible that by the time the calling thread looks at the value that was returned by `trylock` the lock is already free

Concurrency Primitives

POSIX Locks

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

It's good to clear up when you no longer need the mutex as this may free up some system resources

Concurrency Primitives

POSIX Locks

Example code:

```
#include <pthread.h>
...
pthread_mutex_t m;
/* ought to check values returned by these calls */
pthread_mutex_init(&m, NULL);
...
pthread_mutex_lock(&m);
... <CR> ...
pthread_mutex_unlock(&m);
...
pthread_mutex_destroy(&m);
```

We can lock and unlock a mutex as often as we wish: we would typically create it once and use it many times before tidying up

Concurrency Primitives

POSIX Locks

The properties of POSIX locks are specified just to the point to make them useful: in a portable program you can't rely on any feature not explicitly mentioned

For example, calling `destroy` on an uninitialised lock; or calling `init` on an already-initialised lock; or destroying a lock while another thread holds it; or using a bitwise copy of a lock structure; and so on

Remember that a lot of machines don't have the nice predictable architecture of a PC

And even PC architectures are very complicated these days

Concurrency Primitives

POSIX `pthread`

Exercise Read about `pthread_spin_lock` and `pthread_rwlock`

Advanced Exercise Think about mutexes in the context of async programming, where we have concurrency (but not necessarily parallelism) and we require threads never to block

How to make threads

Now we have been introduced to POSIX, we need to take a little diversion from talking about primitives to cover something essential to parallelism

Namely, how do we create new threads to run?

As always, a simple idea that can have unexpected consequences

We shall look at the POSIX mechanism

Concurrency Control

POSIX

Creating threads:

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

Link with `-lpthread`

This looks ugly, but is quite simple in practice: it creates a new thread running the function `start_routine` on the argument `arg`

Concurrency Control

POSIX

It returns a thread identifier in argument `thread`. This can be used to do things to the thread

`attr` is a thread attribute: you probably will never need more than the default (NULL), but occasionally you might (stack size; detached thread)

The `start_routine` names a function of one argument that the thread will start executing when it begins running

The `arg` is the argument passed to the function (a pointer)

Concurrency Control

POSIX

Roughly:

```
void *hello(void *n)
{
  printf("hello %d\n", *(int*)n);
  return n;
}

int main(void)
{
  int m;
  pthread_t thr;

  m = 1;
  // should check return value from create ...
  pthread_create(&thr, NULL, hello, (void*)&m);
  ...
}
```

Concurrency Control

POSIX

`pthread_create` returns (pretty much) immediately with an error code, 0 indicating success

It makes a new thread that runs separately from the main thread

Possibly simultaneously with the main thread, depending on the number of cores and the OS's scheduling

Concurrency Control

POSIX

It runs the function `hello` with argument a pointer to `m`

It does this concurrently with the `main` function, which continues to run

The `start_function` will generally call lots of other functions to perform whatever the thread needs to do

Ugly type casting is common in C

Threads

Aside

This also works on uniprocessor systems: the threads are scheduled in a similar way to processes

You can debug a concurrent program on a sequential machine, but it may not exhibit some of the more subtle race conditions or deadlocks as the threads won't truly be running in parallel

Threads

Aside

You can make more threads than there are cores: for example, run 10 (or 1000) threads on a 4 core machine

And the OS will schedule between the threads

A thread that is blocked (e.g., waiting on a lock) typically would not be scheduled, so it uses no CPU cycles

The question remains whether that is worth it or not to have more threads than cores, as both creating threads and OS scheduling eats up CPU time

A common error is to create hundreds of threads and then wonder why everything is running slowly

Threads create concurrency, not parallelism

Threads

Aside

Incidentally, using threads as a way of structuring your program can sometimes be a good approach, even if you are not concerned with parallelism

For example, have a GUI running on one thread and the computation it controls on another thread

Called *structure by process*

Concurrency Control

POSIX

More realistically we type cast in the create:

```
void hello(int *n)
{
  printf("hello %d\n", *n);
}

int main(void)
{
  int m;
  pthread_t thr;

  m = 1;
  pthread_create(&thr, NULL, (void*(*)(void*))hello, (void*)&m);
  ...
}
```

Concurrency Control

POSIX

How about two new threads?

```
void hello(int *n)
{
  printf("hello %d\n", *n);
}

int main(void)
{
  int m;
  pthread_t thr1, thr2;

  m = 1;
  pthread_create(&thr1, NULL, (void*(*)(void*))hello, (void*)&m);
  m = 2;
  pthread_create(&thr2, NULL, (void*(*)(void*))hello, (void*)&m);
  ...
}
```

Concurrency Control

POSIX

This creates two threads, both running the same code, namely `hello`, but on separate threads. Each thread has its own stack, thus its own copy of `n`

Unfortunately, it is buggy code!

As usual, it may appear to run correctly several times, printing `"hello 1"` and `"hello 2"` (in either order!)

But sometimes it prints `"hello 2"` and `"hello 2"`

Concurrency Control

POSIX

This is another case of sequential assumptions not following into parallel code: another race condition

It *looks* like we update `m` in between the two new threads

But the new threads are in parallel, running *asynchronously* with the main thread

Concurrency Control

POSIX

What we expect is

main	**1**	**2**
creates **1**	1 starts running	
	reads m=1	
updates m	prints 1	
creates **2**		2 starts running
		reads m=2
		prints 2

Concurrency Control

POSIX

What might happen is

main	**1**	**2**
`creates` **1**		
`updates m`	`1 starts running`	
`creates` **2**	`reads m=2`	`2 starts running`
	`prints 2`	`reads m=2`
		`prints 2`

If thread 1 starts running slightly later

In fact, this is quite likely, as creating a new thread takes a fair amount of time

Concurrency Control

POSIX

There are three threads in the program: the two running `hello` and the one running `main`

The threads are *sharing* the variable `m` (via the pointers), so the behaviour of the program is dependent on what order the threads happen to access `m`. This is again bad programming, a data race

Be very careful about the values you pass into the thread

Concurrency Control

POSIX

We can fix that race by not sharing:

```
void hello(int *n) {
  printf("hello %d\n", *n);
}

int main(void) {
  int m1, m2;
  pthread_t thr1, thr2;

  m1 = 1;
  pthread_create(&thr1, NULL, (void*(*)(void*))hello, (void*)&m1);
  m2 = 2;
  pthread_create(&thr2, NULL, (void*(*)(void*))hello, (void*)&m2);

  return 0;
}
```

Concurrency Control

POSIX

But now we (still) have another race condition, which fortunately is easier to spot

We *might* see both hellos, but more likely is we will see nothing at all

Again, the `main` thread *continues to run* and `main` might return before the new threads have had chance to get started

In C, when the `main` function returns the *whole* process exits, and all of the threads are terminated, possibly before they have had chance to print

Concurrency Control

POSIX

To fix this the initial thread should wait for the other threads to finish

```
int pthread_join(pthread_t thread, void **retval);
```

This blocks the calling thread until the named thread exits

This is the main use of the thread identifiers: joining threads (waiting for threads to finish)

A thread can end by returning from its initial function or by calling `pthread_exit(void *retval);`

Concurrency Control

POSIX

The thread can return a value, which is a pointer. This will be copied into where `retval` in `pthread_join` points

Use `NULL` if you don't need a return value

Be careful not to return a pointer to something on the stack of the exiting thread!

Any thread can wait for any other thread to terminate, as long as it knows the thread's id (the `pthread_t`)

Concurrency Control

POSIX

```
int main(void)
{
  int m1, m2;
  pthread_t thr1, thr2;

  m1 = 1;
  pthread_create(&thr1, NULL, (void*(*)(void*))hello, (void*)&m1);
  m2 = 2;
  pthread_create(&thr2, NULL, (void*(*)(void*))hello, (void*)&m2);
  pthread_join(thr1, NULL);
  pthread_join(thr2, NULL);
  return 0;
}
```

Concurrency Control

POSIX

- If any thread calls `exit()` anywhere, the entire process dies: the `exit` function means “exit process”
- if any thread calls `pthread_exit()` anywhere, that thread dies
- if any thread returns from its initial function, that thread dies
- there is no hierarchy of threads, all threads are equal and independent once created

Concurrency Control

POSIX

The only thing to watch out for is the thread running `main`, because in C the `main()` function has an implicit `exit()` after its end. So if it finishes, the entire process subsequently dies

Exercise (For later) Think about what coding would be needed if we wanted always to get `hello 1` printed first and `hello 2` second

Exercise Then generalise to *n* threads

Concurrency Control

POSIX

Advanced Exercise The following code might cause a segmentation violation. Why?

```
int main(void)
{
  int m1, m2;
  pthread_t thr1, thr2;

  m1 = 1;
  pthread_create(&thr1, NULL, (void*(*)(void*))hello, (void*)&m1);
  m2 = 2;
  pthread_create(&thr2, NULL, (void*(*)(void*))hello, (void*)&m2);
  return 0;
}
```

Concurrency Control

Threads

It's not just C that invites these kinds of racy bugs, but they are common to all library-based parallelisms used in sequential languages

And to sequential-trained programmers

There is nothing in the C language itself to stop parallel stupidities as it was designed as a sequential language

As were many other languages in popular use today

Concurrency Primitives

Atomic Update

Back to primitives

The problem with updates is that there is more than one operation involved: first read, then modify, then store

Another thread may access the shared resource in between the read and store

This leads us to another approach to the update race condition by having indivisible *atomic update*

Concurrency Primitives

Atomic Update

This where the hardware supplies a special instruction to, say, increment an integer as a single atomic operation (read-add 1-store)

This must be in the hardware: the increment instruction must prevent other modifications of that value while it is being incremented

The hardware sorts out the sequentialisation in the case of simultaneous (or near-simultaneous) update by different threads

The operation is guaranteed not to be interrupted or interleaved with other threads

Concurrency Primitives

Atomic Update

Note that “atomic” does not mean “fast”

Depending on the cpu architecture, a single atomic instruction might take possibly hundreds of cpu cycles to execute

The hardware might need to sort out memory buses, or cache coherence, or pausing other cores trying to do a simultaneous update, or other low-level stuff

Concurrency Primitives

Atomic Update

Atomics are indeed a reasonable approach, used by many, but they have limitations

- Atomic instructions are hard to build in the context of the complexity of caching and so on in modern architectures
- you would need an atomic instruction for each kind of update you might want to do
- getting a high-level language compiler to generate code using that instruction will not be straightforward
- they can be slow to execute

Concurrency Primitives

Atomic Update

You do see machine instructions in modern CPUs to do some selection of atomic increment and decrement of integers, add, subtract, logical and, logical or, swap a value in a register with a value in memory, swap two values in memory, and a couple of conditional tests but usually nothing much more than those

Instead, the best approach is to use a more flexible machine instruction that you can build on to make more generic higher-level solutions (see “test and set” and friends, later)

Indeed, we shall soon see how a lock implementation might be built from atomic operations

Concurrency Primitives

Atomic Update

Do not use atomics for the coursework

To use them effectively you need more more detail that we can't go into right now

Concurrency Primitives

Atomic Update

Exercise For hardware geeks: atomic operations often lock an entire cache line, and can stall the CPU for hundreds of clock cycles while the caches synchronise, so they can slow you down more than you think. Read about this

Exercise For hardware geeks: compare the cost of using a lock against the cost of using an atomic update (the answer can depend on the pattern of access)

Exercise Effective use of atomics involves understanding *memory consistency orderings*. Read about this

Exercise Some programming languages offer atomic datatypes, e.g., Java, C++, Rust. These usually eventually just call the machine instruction atomics. Read about this

Concurrency Primitives

Implementation of Locks

A little more to say about locks. . .

How are locks implemented?

They are a flag: say an integer, or even just one bit

We might use 1 to indicate locked, and 0 to indicate unlocked

Concurrency Primitives

Implementation of Locks

```
int lock = 0;

void get_lock()
{
   while (lock == 1) {
     deschedule();
   }
   lock = 1;
}
```

i.e., test the flag. If it is already 1, wait; else we can grab it by setting the flag to 1

Spot the bug!

Concurrency Primitives

Implementation of Locks

There is another update race condition

1	**2**
`test flag: OK`	`test flag: OK`
`set flag`	`set flag`

And now both calls to `get_lock` succeed and both threads proceed to enter the critical region

Concurrency Primitives

Implementation of Locks

In between the testing of the flag and the setting of the flag all kinds of other things might happen

Code lines that are textually next to each other like this are widely separated in some sense: what we want is the testing and setting to be atomic

That is the test and the set are inseparable: nothing can get between them

This is another kind of critical region, so we could solve it by using locks. . .

Concurrency Primitives

Implementation of Locks

Fortunately we don't have to go into an infinite regression as there are two kinds of solution: hardware and software

Hardware designers understand mutual exclusion, so the instruction sets of all modern processors have an instruction specifically designed for this

For example the *compare and swap* instruction

Concurrency Primitives

Implementation of Locks

Intel has `cmpxchgb` that atomically operates on a register and a byte in memory

> *CMPXCHG r/m8, r8*
> *Compare AL with r/m8. If equal, ZF is set and r8 is loaded into r/m8. Else, clear ZF and load r/m8 into AL. This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically*

Concurrency Primitives

Implementation of Locks

In C, its action is like

```
int compare_and_swap(int *reg, int *mem, int new)
{
    if (*reg == *mem) {
       *mem = new;
       return 1;  /* got lock */
    }
    *reg = *mem;
    return 0;     /* fail */
}
```

but the entire thing is done *atomically*

Concurrency Primitives

Implementation of Locks

Using this:

```
int flag = 0;
...
int reg = 0;
// try to set flag to 1
while (compare_and_swap(&reg, &flag, 1) == 0) {
  reg = 0; // try again
}
<CR>
flag = 0;
```

This implements a busy wait

You should spend some time going through this!

Concurrency Primitives

Implementation of Locks

Instructions found in other architectures include `test_and_set` and an atomic `swap`

Early architectures did not have such instructions, so software versions were devised

These include: Dekker, Dijkstra and Lamport

They are very subtle as they must construct an atomic effect from non-atomic code

Exercise Go and read about these

Concurrency Primitives

Synchronisation

Now we look at some other problems

Consider our original counting code with a shared variable `count`. A simple solution might be to make `count` non-shared:

1

```
for (i = 0; i < 50; i++) {
    if (val[i] > 0)
       count1 = count1 + 1;
}
count = count1 + count2;
```

2

```
for (j = 50; j < 100; j++) {
     if (val[j] > 0)
          count2 = count2 + 1;
}
```

There is now another, different, problem with this code!

Concurrency Primitives

Synchronisation

The problem now is *when is the* `count = count1 + count2` *executed?*

To be correct, it has to happen after both the loops have finished: any earlier will give a wrong answer

It will definitely happen after loop **1** has finished, but what about loop **2**?

We can't rely (in a MIMD architecture) on the two loops on different cores running at the same time and finishing at the same time

Timings in the system may have the two loops running in any conceivable arrangement of before, after or overlapped

Concurrency Primitives

Synchronisation

1

```
for (i = 0; i < 50; i++) {
    if (val[i] > 0)
       count1 = count1 + 1;
}
count = count1 + count2;
```

2

```
for (j = 50; j < 100; j++) {
     if (val[j] > 0)
           count2 = count2 + 1;
 }
```

Concurrency Primitives

Synchronisation

So we must explicitly write code to ensure the final sum only happens when both loops are finished

This is a *synchronisation* between the two threads

It may mean thread **1** waiting for thread **2**

Another sequentialisation!

Concurrency Primitives

Synchronisation

More subtly: if this code is executed more than once (perhaps counting more than one array), thread **2** ought to wait for thread **1** before starting!

It is possible that **1** is slow or paused for some reason, when **2** might do its bit and come around again on the next call to the count code, do the count on some other data, updating `count2` as it goes

Finally **1** awakes and gets the wrong `count2`

This does happen and is a source of bugs

Concurrency Primitives

Semaphores

Semaphores can be used for thread synchronisation

Typically, we might have some thread that can only continue its work when one (or more) others have finished doing something, maybe computing some inputs for the thread to process

It can wait on a semaphore, again a simple flag, until another thread sets the flag. Then it knows it can continue

Note that even though both locks and semaphores are flags, they are very different things! Beware it is common for people to confuse the two

Concurrency Primitives

Semaphores

Semaphores are manipulated by two atomic operations P and V that symbolically act atomically as:

```
P(s): while (s == 0) {
         suspend();
      }
      s = 0;
```

```
V(s):  s = 1;
       if any process waiting on s
          unblock one
```

Concurrency Primitives

Semaphores

On finding $s = 0$ a thread will suspend itself; when awoken it will re-attempt to set the semaphore: and it will often succeed, unless a third thread comes along and gets the semaphore first

Like locks, semaphores are *not fair* on which thread will be awoken if more than one is waiting

Other names for P are: wait, up, lock, enter, open

Other names for V are: signal, down, unlock, exit, close

P stands for “proberen”, V for “verhogen”, which are Dutch for “test” and “increase”

Concurrency Primitives

Semaphores

Semaphores synchronise across threads:

```
do something                    
wait(s)                         prepare data
read data                       signal(s)
                                carry on

                                prepare data
do something                    signal(s)
wait(s)                         carry on
read data
```

Thread 1 waits until thread 2 has prepared some data before reading it

The signal and wait might happen in any order

Concurrency Primitives

Counting Semaphores

The above are called *binary* semaphores as the idea can be trivially extended into *counting* semaphores

```
P(s): while (s == 0) {
         suspend();
      }
      s = s - 1;
```

```
V(s):  s = s + 1;
       if any process waiting on s
          unblock one
```

When initialised with the value n, this will allow n threads to open the semaphore before blocking

Concurrency Primitives

Counting Semaphores

This allows region access control when there can be one than one, but fewer than some limit in the region simultaneously

For example, if there are 5 places at a dining table we can allow no more than 5 people in the room at a time

Or 4 if they are philosophers. . .

Concurrency Primitives

Semaphores

Mutual exclusion with semaphores happens to be easy:

```
wait(s);
<CR>
signal(s);
```

Wait for the semaphore; signal it's free when you are done

But don't do this: it's better to use locks here. Semaphores are more general than locks: they allow a thread to suspend itself and be awoken by another thread when some condition is true

Concurrency Primitives

Semaphores

Mutexes: the thread that sets the flag must be the thread that clears the flag

Semaphores: the thread that sets the flag will generally be different from the thread that clears the flag

Semaphores should be used *across* threads, mutexes must not

The locking effect is in some sense incidental: more useful is using semaphores to synchronise

Concurrency Primitives

POSIX Semaphores

POSIX semaphores:

```
#include <semaphore.h>
sem_t sem;
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_trywait(sem_t *sem);
```

“post” for signal

Concurrency Primitives

POSIX Semaphores

Exercise Add a semaphore to the `count1`/`count2` example to get thread 1 to wait for thread 2 before doing the final sum

Exercise Then add another semaphore to get thread 2 to wait for thread 1 before starting

Concurrency Primitives

Barriers

Another synchronisation primitive is *barriers* (occasionally called *rendezvous)*

A barrier stops threads from continuing until some required number of threads have all hit the barrier; then they can all continue together

This allows us to synchronise parts of the program: recall supersteps

Concurrency Primitives

Barriers

Suppose we have a list of numbers we want to square then add in pairs

```
for (i = 0; i < 100; i++) {
    v[i] = v[i]*v[i];
}
for (i = 0; i < 100; i++) {
    s[i] = v[i] + v[99-i];
}
```

We can parallelise this by having (say) 4 threads; each thread squares a block of values; then they add a block of values

Concurrency Primitives

Barriers

```
1               2               3               4
v[0]^2          v[25]^2         v[50]^2         v[75]^2
v[1]^2          v[26]^2         v[51]^2         v[76]^2
v[2]^2          v[27]^2         v[52]^2         v[77]^2
...             ...             ...             ...
v[24]^2         v[49]^2         v[74]^2         v[99]^2
v[0]+v[99]      v[25]+v[74]     v[50]+v[49]     v[75]+v[24]
v[1]+v[98]      v[26]+v[73]     v[51]+v[48]     v[76]+v[25]
...             ...             ...             ...
v[24]+v[75]     v[49]+v[50]     v[74]+v[25]     v[99]+v[0]
```

Concurrency Primitives

Barriers

```
1                                   2                                   3...
for (i = 0; i < 25; i++) {          for (j = 25; j < 50; j++) {
    v[i] = v[i]*v[i];                   v[j] = v[j]*v[j];
}                                   }
for (i = 0; i < 25; i++) {          for (j = 25; j < 50; j++) {          ...
    s[i] = v[i] + v[99-i];              s[j] = v[j] + v[99-j];
}                                   }
```

Again, the above might work sometimes, or many times, but it is buggy

Concurrency Primitives

Barriers

The problem here is again that the threads may not all be running at the same speed: perhaps one thread is interrupted and descheduled by the OS; or memory access is not uniform speed; or many other factors

So we can't rely on all the threads finishing their squares at precisely the same time: one thread might finish its block and start adding using values not yet finished squaring

Another synchronisation problem

Concurrency Primitives

1	2	3	4
v[0]^2	v[25]^2	v[50]^2	
v[1]^2	v[26]^2	v[51]^2	
v[2]^2	v[27]^2	v[52]^2	v[75]^2
...	...	...	v[76]^2
...	...	...	...
v[24]^2	v[49]^2	v[74]^2	v[97]^2
v[0]+ **v[99]**	v[25]+v[74]	v[50]+v[49]	v[98]^2
v[1]+v[98]	v[26]+v[73]	v[51]+v[48]	**v[99]^2**
...	...	...	v[75]+v[24]
...	...	...	...
v[24]+v[75]	v[49]+v[50]	v[74]+v[25]	v[97]+v[2]
			v[98]+v[1]
			v[99]+v[0]

This is how we get the wrong answer: again just because the lines of code for the adds follows the lines of code for the squares make us believe every add happens after every square

Concurrency Primitives

Barriers

We need to synchronise all the threads at the end of the squares before allowing them to continue with the adds

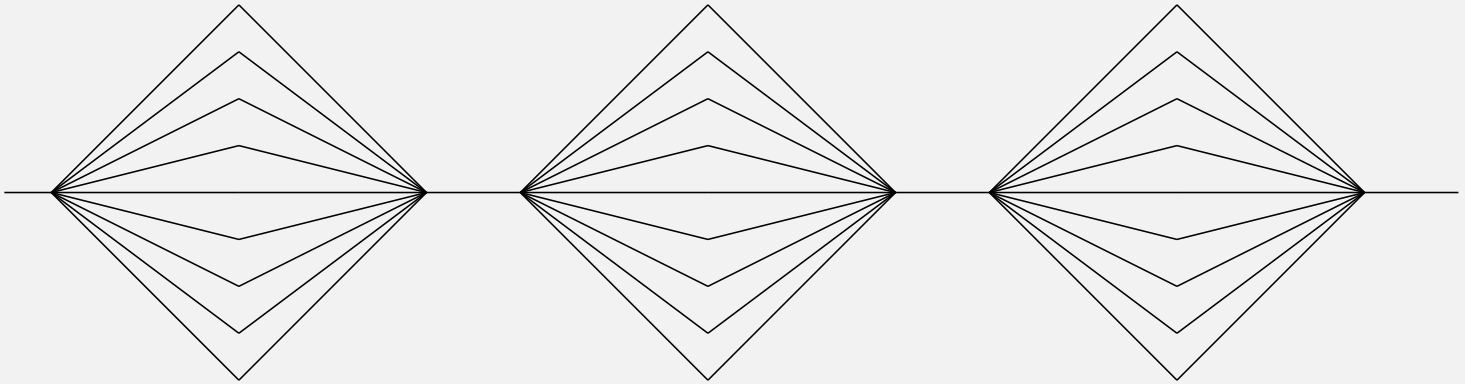
```
                          b = make_barrier(4);
<parallel squares> <parallel squares> <parallel squares> ...
barrier_wait(b);   barrier_wait(b);   barrier_wait(b);   ...
<parallel adds>    <parallel adds>    <parallel adds>    ...
```

Only when all 4 threads have reached the barrier can they all proceed

Concurrency Primitives

Barriers

Barriers are good for the superstep style of programming



Supersteps

But beware: as a barrier synchronises many threads, there is potentially a lot of waiting going on: we can't progress faster than the slowest thread

Thus barriers are best when all the threads are doing roughly the same amount of work

Concurrency Primitives

POSIX Barriers

```
#include <pthread.h>
pthread_barrier_t barrier;
int pthread_barrier_init(
    pthread_barrier_t *restrict barrier,
    const pthread_barrierattr_t *restrict attr,
    unsigned count);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

A barrier can be reused immediately after it has released its threads; it has a fixed value of *n* set when it is initialised

Exercise Have a look at the return value from `pthread_barrier_wait`

Concurrency Primitives

POSIX Barriers

Exercise Fix the `count1`/`count2` problem with barriers

Exercise Both semaphores and barriers are about synchronisation. Think about how you might implement barriers using semaphores

Exercise Think about how you might implement semaphores using barriers

Concurrency Primitives

Condition Variables

One last primitive we are going to look at is *condition variables*

As the name suggests, it is a way a thread can wait until some condition is true

The idea is that one or more threads can wait on a condition variable until another signals that the required condition is now true

The signal can either let just *one* thread continue, or be a *broadcast* that lets all waiting threads continue

Condition variables are normally associated with a mutex, and are used *inside* a critical region protected by that mutex

Concurrency Primitives

Condition Variables

1

```
get_lock(mx);
<CR>
condvar_wait(cv, mx);
(wait)
<CR>
free_lock(mx);
```

2

```
get_lock(mx);
<CR>
condvar_signal(cv);
free_lock(mx);
```

`condvar_wait` releases the mutex and waits on the condition variable

When the other thread signal signals and releases the mutex, the first thread regains the mutex and continues within the critical region

Concurrency Primitives

Condition Variables

The condition variable allows thread 1 to “step outside” the critical region, letting another thread to enter and do something

Conditions variables combine mutual exclusion and synchronisation

Again, not fair on which thread gets to continue if more than one is waiting

With a `broadcast` all other threads are marked as ready to run, but only one will regain the lock; the others will blocked on the lock as normal

One will get the lock when the first thread releases it; and so on

Concurrency Primitives

POSIX Condition Variables

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *restrict cond,
            const pthread_condattr_t *restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *restrict cond,
            pthread_mutex_t *restrict mutex);
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
            pthread_mutex_t *restrict mutex,
            const struct timespec *restrict abstime);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Concurrency Primitives

POSIX Condition Variables

As an example of the kind of grungy detail that parallelism has to address: POSIX recognises that there is a nasty implementation detail that would otherwise make implementing condition variables impractical

The specification for `pthread_cond_signal` says

> *The* `pthread_cond_signal()` *function shall unblock at least one of the threads that are blocked on the specified condition variable* `cond`

"at *least* one": there is a (rare) problem of *spurious wakeups* that is in general too expensive to avoid

Concurrency Primitives

POSIX Condition Variables

This just means you have to be a bit formulaic about the use of condition variables and always have a *condition* to test before continuing

1

```
iteration = 0;
get_lock(mx);
<CR>
it = iteration;
while (it == iteration)
    condvar_wait(cv, mx);
<CR>
free_lock(mx);
```

2

```
get_lock(mx);
<CR>
iteration++;
condvar_signal(cv, mx);
free_lock(mx);
```

Thread 1 might get awoken spuriously but it doesn't want to continue until the next iteration

Concurrency Primitives

POSIX Condition Variables

In general you would test for whatever condition you were waiting for: thread 2 sets the condition, thread 1 should test for it

Condition variables are very useful, but a bit of a pain to use

Concurrency Primitives

Concurrency Primitives

We have called these things *primitives*, but we can implement them in terms of each other

Exercise Do this

All eventually go back to the underlying hardware or software support

“Primitive” is actually a good description as they are all very low level

Concurrency Primitives

Concurrency Primitives

And they do have a cost, thus their use does limit the speedup available

Their overhead can be divided into two parts

(a) the time spent blocked as a necessary part of its function, e.g., wait on a lock

(b) the time spent in executing the code of the primitive

Note part (a) isn't really a limitation of the primitive: it's necessary if it is to work at all. It is (b) that the implementation of a primitive seeks to minimise

Concurrency Control

Higher Level

Semaphores, locks, barriers, etc., and even threads are likened to assembler: low-level, fast, fine control, but very likely to encourage buggy programs

While many programmers are happy using them, others need higher level solutions

These come in many forms

Concurrency Control

Higher Level

Concurrency control can be supported in a high-level language as

- added in to an existing language, in library support. We have seen some of this already: the POSIX examples
- fudged into the syntax of an existing language
- part of the initial design of a new language

We shall be looking at all of these approaches

Concurrency Control

Higher Level

There is a lot of sequential code out there that people would like to run faster on parallel hardware

While there is a lot of effort being put into automatic analysis of code to discover and exploit parallelism, the results are sporadic

Functional languages offer a decent hope here, but not much code is functional style

So code needs to be rewritten to make best advantage of parallelism

The hope (and economics) is we can take existing code using an existing language and modify it

Concurrency Control

Libraries

It's not a good way of doing things, but rewriting from scratch is just too expensive

Of course, new projects ought to be written with parallelism in mind from their start

Also, there are lots of programmers with extensive expertise in languages like C, Java and C++ — meaning such programmers are cheaper to employ

So we are led to the approach of taking, say C, and adding parallelism to it

The easiest way is to leave the language itself untouched, just adding a library of functions that do parallelism

Concurrency Control

Libraries

For example, the POSIX `pthread` approach

Note: We have been using C and the POSIX library to illustrate points, but this library technique applies to all sensible languages

But you can't just add a parallel library to a sequential language and hope everything is OK

Concurrency Control

Threads again

Modern compilers and modern hardware both try their best to execute your code as fast as possible

But in doing so, they can break parallel code

For example, some compiler optimisations can break parallel code

And some hardware optimisations can break parallel code

Concurrency Control

Compiler Reordering

Modern compilers often reorder code to make things more efficient

For example, main memory access is (relatively) slow, so if the value of a variable is needed, the compiler might try to start loading it earlier than the code might suggest

Concurrency Control

Compiler Reordering

Given code

```
y = 2;
x = z;
x += y; // need to wait for z before we can do this
```

The compiler might spot it can start loading `z` earlier, so there is less of a wait before it can do the increment:

```
x = z;
y = 2; // do this without waiting for z to be loaded
x += y;
```

The effect is the same, but it goes a little faster. The compiler in effect rewrites your code

Concurrency Control

Compiler Reordering

This could break things. Consider

```
A                                          B
while (cont == 0) {/* nothing */}          x = 42;
print x;                                   cont = 1;
```

where the intent was to have thread A to wait for thread B to set the `cont` flag before continuing to print 42

A compiler only seeing the code for B may conclude that the variables `cont` and `x` are independent and so (perhaps for whatever reason) it can rearrange the code as

```
cont = 1;
x = 42;
```

Concurrency Control

Compiler Reordering

Similarly for A: it is possible that the read of `x` can done before the loop

Note: *never* write code like this in the hope that it might work: it is simply buggy code! Use a semaphore or equivalent

The problem is that there is a hidden relationship between the variables `x` and `cont` that is in the mind of the programmer, but is not expressed in the code

Concurrency Control

Compiler Reordering

Example. Consider the code:

```
int a = 0;
int b = 0;
```

```
A
a = 42;
printf("%d\n", b);
```

```
B
b = 42;
printf("%d\n", a);
```

Explain how it might print `0` twice, even though it appears we always print after an update

Concurrency Control

Compiler Reordering

Thus, to be correct, the programmer needs to inform the compiler not to do these kinds of “optimisations”

Languages like C and Java have a `volatile` keyword:

```
volatile int cont;
```

tells the compiler not to mess around with such variables and assume that external operations might change their value

Concurrency Control

Compiler Reordering

But `volatile` was introduced for hardware/peripheral-related reasons and is *not* a way of fixing concurrency issues as they don't solve the whole problem, as the *hardware* needs telling, too

Summary: **don't** use `volatile` to try to solve parallelism problems, as is sometimes recommended

Concurrency Control

Hardware Reordering

Second problem: as it's not just the compiler that reorders things

Modern CPUs use *out of order execution* on machine instructions to improve efficiency in superscalar architectures, where the *processor* can reorder instructions as it sees fit

For example, in the machine code for

```
x = y + z;
w = 2*u;
```

Since loading from memory takes a long time the CPU might decide to start loading `u` *before* doing the sum

Again, this reduces the overall time the code takes to run as the multiply does not have to wait as long for `u` to arrive

Concurrency Control

Hardware Reordering

So, even given un-reordered code or machine code equivalent loading registers

```
cont = 1;                    load $r1, 1
x = 42;                      load $r2, 42
```

the CPU might *while running* decide the loads look independent and load `x` (`$r2`) first

Out of order execution is common in modern architectures

Concurrency Control

Thus we also need special code like

```
while (cont == 0) {/* nothing */}
memory_fence();
print x;
```

```
x = 42;
memory_fence();
cont = 1;
```

(details vary according to language and compiler) that tell the compiler *and* processor not to reorder things

Firstly, the compiler will know not to try to move reads or writes across the call to `memory_fence()`

Secondly, the `memory_fence()` would compile to a specific special machine instruction that tells the CPU's out of order mechanism not to move read or writes across this boundary

The first fence says not to read `x` too early, while the other says don't assign `cont` before `x`

Concurrency Control

Memory Consistency

In fact, in modern machine architectures you *must* use some primitive like a fence, or something that uses a fence (e.g., a semaphore), to ensure the intended behaviour

Memory fences work (when you remember to use them) but prevent some correct optimisations. Thus more subtle mechanisms are also used

Exercise The above stops both reads and writes from being moved forward or back. Fences also come in variants that only block movement forward; or only movement back. Read about these

Concurrency Control

Third problem: other memory effects

It is possible (in some machine architectures) for thread A to read the wrong value of `x`, *even if there is no out-of order execution*

It could be that B writes `x` and then writes `cont`; and A reads `cont` before reading `x`

But, due to caching (or other weirdness) it can be that B's write to `cont` reaches A before its write to `x`

So A reads the new value of `cont` but the old value of `x`, as its view of `x` has not yet been updated

Concurrency Control

Memory Consistency

The specification for a parallel language needs a *memory model* to describe how memory reads and writes are visible to multiple processors

This involves the use of special language constructs and special memory access operations to inform the compiler and hardware about what kinds of reordering are allowable and what kinds of *memory consistency* across processors are needed

Concurrency Control

Memory Consistency

And this is the problem: languages like C (and C++, and Java, and . . .) were conceived before memory models were necessary

So they didn't have them

Updates to the language standards are trying to retrofit memory models, but sometimes it's very difficult to fit new ideas into an old language

Further, programmers need to be (re)trained to understand these things

Concurrency Control

Memory Consistency

So, for example, the programmer may decide that some reads or some writes may be reordered, while others should not

Generally, the programmer must understand the issues involved and use the right constructs in the right places

Allowing just enough flexibility for the compiler/hardware to be efficient, while still correct code

Thus allowing the system to reduce synchronisation and increase parallelism

Concurrency Control

Memory Consistency

Fortunately for us, if we use primitives (locks, semaphores, and so on) and higher-level constructs they will look after the details for us

As long as we use them!

So: if you have a cross-thread relationship, use a parallelism mechanism, don't just wing it

Concurrency Control

Memory Consistency

Exercise Read about memory consistency. Including: memory fences, *strict consistency*, *strong consistency*, *causal consistency*, *weak consistency*, *sequentially consistent*, *acquire-release*, *relaxed*, *consume*, etc.

Exercise Read about how modern C and C++ standards address the memory consistency issue

Exercise Read about the difference between Java's memory model and C/C++'s model (and what `volatile` does in each)

Concurrency Control

Memory Consistency

Exercise Read about the difference between the Intel (x86) memory model and the Arm memory model

Exercise And read about the memory problems that Apple's Arm M1 and later chips have in trying to support old x86 code via an instruction translator (Rosetta)

Concurrency Control

Threads

So now you have the tools to hand: thread creation to run things concurrently/in parallel, and primitives to control races

An important note on the cost of thread creation: they are not free!

But, in a good OS implementation, they are relatively cheap

Depending on the operating system, it can take hundreds or thousands of CPU instructions to create or destroy a thread

For the “hello” examples above it probably would not be worth creating new threads, but be faster to run the `printf`s sequentially

(But, remember, raw speed is not necessarily the target for parallelism)

Concurrency Control

Threads

A rough test on my PC indicates that the overhead of creating and joining one thread is about the same amount of time as doing 2000 floating point operations

Exercise That is for a particular OS and a particular CPU. Find out how long it takes to create a thread on your computer and OS

Concurrency Control

Threads

You have to judge whether it is worthwhile paying the creation overhead

And there is the additional cost of *context switching* between threads when there are more threads than processors

The thread model of parallelism leads one to write programs with large numbers of threads

Probably more than there are processors in the system, particularly when you take into account the threads in the other processes running in the system

Concurrency Control

Threads

This means that threads need to be scheduled, just like processes

And this has a cost, just like processes

It is easy to make so many threads that the OS starts thrashing

You need to be careful about how many threads to create!

Typically, creating a (POSIX) thread when you need it, and then destroying it when done is costly and not a good approach

The objective is to give a thread as much computation as possible, perhaps repeated or multiple tasks

Concurrency Control

Thread Pools

Trying to address the cost of thread creation and deletion leads some people to the *thread pool* model of parallelism

Your program creates a pool of threads (not too many, not too few!) once and reuses them multiple times

Each thread is given a task as is necessary; it does it and then goes back for another task

Concurrency Control

Thread Pools

You pay the cost of creation just once at the start (and destruction just once at the end), rather than once per thread use

Though there is a cost in the pool task management mechanisms

But these threads have a long life, and do many things

Concurrency Control

Thread Pools

Apple's *Grand Central Dispatch* (GCD) does thread pooling at a higher level: system-wide

The OS manages threads across all processes running, not just within each process

More on GCD later (in particular, its costs), but note this is in contrast to the model of each *program* creating and destroying threads as it needs them, as we were doing previously

Concurrency Control

POSIX

As mentioned previously, POSIX pthreads are a very popular library-based mechanism to support parallelism (actually: concurrency)

We have just scratched the surface of POSIX

There are lots of other functions described by the POSIX standard: try
`man -k pthread`
and
`man 7 pthreads`
on Linux for an overview

Concurrency Control

Non-POSIX

Windows has something similar to POSIX threads: different names for the functions, but similar enough to be confusing

They do provide an implementation of POSIX threads, but MS would rather you use their own thread library: MS are not interested in portability across OSs

Apple macOS, like Linux, has good POSIX coverage

Concurrency Control

Other Threads

It is worthwhile mentioning that there are many other kinds of threads, mostly invented to try to overcome the costs of (a) thread creation/deletion and (b) context switching between threads

They have names like *fibres*, *coroutines*, *protothreads*, *microthreads*, *light-weight processes* and so on

Concurrency Control

Other Threads

For example, some languages, e.g., Go (“goroutines”) and Erlang (“processes”), have very *lightweight threads* as part of the language

These are scheduled by the language runtime across system threads

They are very cheap to create, and allow thousands or millions of “threads” to be active

They encourage the use of massive threading at the cost of overhead from a more complicated language runtime

More discussion of Go and Erlang later

More Libraries

We were discussing library-based parallelism

Taking a sequential language and using a parallel library

But this has the dangers of the sequential language not understanding parallelism and mis-optimising

But library-based parallelism is very popular: particularly if we avoid shared memory

More Libraries

Another important library-based solution is the *Message Passing Interface* (MPI) and we shall look at this later when we talk about distributed memory systems

We shall just note here that MPI is an example of one library-based technique that is quite popular: write code that is sequential, or modestly parallel, but call library functions that do what we want to achieve that are parallel—and written by somebody else

Another example, the *Basic Linear Algebra Subprograms* (BLAS)

More Libraries

The BLAS are a (standard for a) collection of functions that implement various algorithms in linear algebra: vector sums; matrix multiplication; vector dot products; etc. for various representations of these datatypes

Implementations are written by people who really understand what they are doing in terms of making the best use of hardware: in particular parallel hardware

If you write your application to use the BLAS your code will be using this expertise

If someone comes out with an improved implementation of the BLAS that goes twice as fast, your code will automatically go twice as fast (in the BLAS bit)

More Libraries

They really can be a factor of two difference *on the same hardware*

BLAS libraries are typically tuned to the version of the processor in your machine, taking into account cache sizes; memory speeds and so on

The GotoBLAS, written by Goto, are recognised as being particularly good

His implementation contains chunks of processor-specific assembler and pays particular attention to the sizes of blocks of data, matching them carefully to cache sizes

More Libraries

Many other libraries exist: for example, the *template* approach

This is a standard header file with a library of code behind it that introduces a bunch of new classes to aid parallel computation

For example, C++ AMP (Accelerated Massive Parallelism) from Microsoft defines some parallel container types with methods that act concurrently on them

E.g., `concurrency::parallel_for_each(...)`

The details are hidden from the programmer, who gets a fairly simple API to work with

More Libraries

There are many other template libraries for C++ (a language very suited to this approach):

- Parallel Patterns Library (PPL) from Microsoft
- Thrust from Nividia
- Intel Threading Building Blocks (TBB)
- Boost
- Etc.

But you do need to be careful using them: they do make writing parallel code simpler, but they don't necessarily prevent you from using them incorrectly!

Concurrency Control

Monitors

The next approach to parallelism we shall look at is to have constructs as part of the language

For example, a *monitor* is a language construct that combines mutual exclusion and synchronisation in a way that can be easier to use than the concurrency primitives

```
monitor Name
   local variable declarations
   func fun1(args)  body
   func fun2(args)  body
   ...
end
```

The actual syntax will vary by language

Concurrency Control

Monitors

Mutual exclusion is enforced by

> *only one thread at a time may be executing any function inside a given monitor*

So, if one thread is executing `fun1` and another thread tries to execute `fun2`, it will have to wait until the first thread exits the monitor

Concurrency Control

Monitors

So there is mutual exclusion on the local variables and within the dynamic scope of the functions in the monitor, i.e., mutual exclusion continues even if `fun1` calls a function defined outside the monitor

The mutual exclusion finishes when the thread of control exits the (top level) monitor function

Clearly, monitors will be implemented with locks, but this conveniently hidden from the programmer using them

Concurrency Control

Monitors

Synchronisation is provided by the use of condition variables

`wait(c);` and `signal(c);`

The associated lock is the monitor mutual exclusion lock, and is implicit

Just like the POSIX version, `wait()` will drop the monitor lock to allow other threads access; and try to regain it when it resumes

Concurrency Control

Monitors

We can easily implement a lock using a monitor:

```
monitor Lock
  int flag = 0;
  condition c;
  lock() { while (flag == 1) wait(c); flag = 1; }
  unlock() { flag = 0; signal(c); }
end
```

The monitor lock provides the atomicity we need in the definition of `lock`

Concurrency Control

Monitors

Monitors help with management of mutual exclusion, but the usual nesting deadlock is still possible. For monitors `m1` and `m2`:

```
monitor m1
  fun1() { ... fun2() ...}
  ...
end
```

```
monitor m2
  fun2() { ... fun1() ... }
  ...
end
```

1

`fun1` in monitor `m1` calls
`fun2` in monitor `m2` (waits)

2

`fun2` in monitor `m2` calls
`fun1` in monitor `m1` (waits)

Concurrency Control

Monitors

Modularity might even encourage this error, though monitors are high enough level to be easy to analyse automatically so there are source code tools to spot this

They require careful use and are not a universal solution!

Concurrency Control

Java Monitors

Monitors clearly fit well with object oriented languages: for example, Java implements monitors on a per-object level:

```
class foo {
  private int n = 0;
  public synchronized int inc() { n++; }
  public synchronized int dec() { n--; }
  ...
}
```

Methods with the `synchronized` keyword are within a per-object monitor, i.e., one per instance of `foo`

Concurrency Control

Java Monitors

Only one of `inc` and `dec` can be executing on a given instance of `foo` at a time

Condition variables: `wait()`, `notify()` and `notifyAll()`

Class methods (`static`) can be synchronised, too, locking the class but not its instances

Concurrency Control

Monitors

Monitors are fairly easy to use, but are somewhat large grained: the whole of each monitor, for example *all* methods marked `synchronized` in a Java object

```
class foo {
  private int n = 0, m = 0;
  public synchronized int incn() { n++; }
  public synchronized int decn() { n--; }
  public synchronized int incm() { m++; }
  public synchronized int decm() { m--; }
}
```

Concurrency Control

Monitors

To have separate locks on some of the methods requires code refactoring (or see below): You can do this, but this is driving the code towards complexity

Similarly, it is a bit fiddly to decide on what functionality goes into which monitor: if you are not careful you end up with all your code in one big monitor—sequential!

Concurrency Control

Monitors

Exercise What about the following?

```
class foo {
  private int n = 0, m = 0;
  public synchronized int incn() { n++; }
  public synchronized int decn() { n--; }
  public synchronized int incm() { m++; }
  public synchronized int decm() { m--; }
  public synchronized int swap() { int s = m; m = n; n = s; }
}
```

Concurrency Control

Java Monitors

Java recognises that monitors are sometimes too large, so it allows synchronising of *statements* (rather than whole methods) as a way of providing finer gain control

```
public class locket {
  private Object nlock = new Object();
  private int n = 0;
  public void inc() {
    synchronized(nlock) { n++; }
  }
  public void dec() {
    synchronized(nlock) { n--; }
  }
}
```

Concurrency Control

Java Monitors

`synchronized` takes an arbitrary object as argument

A class can have as many of these as it likes in addition to the implicit one provided by the class monitor

This is fine, but we have just reinvented mutexes!

But in a more convenient form: you can't forget to lock or unlock these

Concurrency Control

Java Monitors

Incidentally, Java also has a library of *atomic* datatypes, e.g., `AtomicInteger` with a few methods, that does the obvious thing

But these are tiresome to use as Java does not have operator overloading, like C++: thus `n.incrementAndGet()` rather than overloading ++ and using the simpler `++n`

Concurrency Control

Conditional Critical Regions

Exercise A similar, but simpler, kind of idea is *conditional critical regions*, where a semaphore is associated with blocks of code (the critical regions)

```
let s = Semaphore::new(1);
...
region s {
  // critical region
  ...
  await <some condition>
  ...
}
```

```
region s {
   ...
   <set condition>
   ...
}
```

Read about this (e.g., in Ada).

Parallelism Languages

The logical approach to parallel programming is to use a language that was designed from the start to support parallelism

There have been very many attempts at creating new languages with explicit support for parallelism

For example, Occam, Strand, Erlang, Linda, SALSA, SISAL, Parlog, Charm++, NESL, Go, Rust as just a few from a huge list

We should have time to look at one or more of these towards the end of the Unit

Some of these languages are quite difficult to learn and use effectively

Language Modification

A conservative approach to getting these kinds of parallel support is to take an existing language, like C, and tweak *the language* to add parallelism

Then, so the theory goes, you can tap into the existing expertise in that language and extend it to parallel systems

This is true to a certain extent, but it still tries to layer parallel ideas over a sequential foundation

Parallelism should not be an afterthought, but should really be part of the foundation

Language Modification

The main example we shall be looking at is *OpenMP* (Open MultiProcessing)

This takes C (or C++) and add some new constructs to notate parallel execution

By hiding the low-level primitive locking and synchronisation they aim to provide an easier way of writing parallel programs

And minimise the kinds of errors the primitives invoke

OpenMP

OpenMP fits nicely into the superstep model of computation

While you shall *not* be using OpenMP for the coursework, some of you might want to use it for your FY Project

OpenMP

Here is a simple loop

```
for (i = 0; i < 10; i++) {
  sq[i] = n + i*i;
}
```

With OpenMP annotation

```
#pragma omp parallel for
for (i = 0; i < 10; i++) {
  sq[i] = n + i*i;
}
```

The `#pragma omp` indicates that we want the loop to be run in parallel

`#pragma` is a general C mechanism, not limited to OpenMP

OpenMP

When this is run, the loop is split into some number of chunks, running on some number of threads

The OpenMP runtime system determines the number of chunks and number of threads

That is, it makes a choice when the code is run

And the numbers of chunks and threads may differ on different runs

OpenMP

Typically the number of chunks is the same as the number of threads, which is the same as the number of processors in the system, but it need not be

And each chunk typically iterates close to

$$\frac{\text{size of loop}}{\text{number of chunks}}$$

times

OpenMP

Also important is that the runtime creates parallel code with a *private* version of `i` per thread

Each thread wants its `i` to range, in parallel, over different values, e.g., 0–2, 3–5, 6–8, 9

Or maybe 0–2, 3–5, 6–7, 8–9; or something else

The runtime decides, and potentially might choose a different split in different runs

The `parallel for` construct knows the loop variable must be private

But the variables `n` and `sq` are *shared* across the threads

OpenMP

Note:

- we do not give a number of threads
- the creation and destruction of threads is all hidden from us: it may create and destroy threads on each occurrence of a `#pragma omp`; or it may use a thread pool
- the compiler determines we need a per-thread variable `i`
- by using the construct we are assuring the compiler that it *is* safe to do the loop in parallel and there are no data (or other) races.
  If the loop was
  ```
  av[i] = av[i] + av[i-1];
  ```
  it would blindly do this in parallel
- so OpenMP provides a simple *mechanism*, but no *analysis*

OpenMP

Exercise Convince yourself why the following is wrong:

Convert

```
for (i = 0; i < 10; i++) {
  av[i] = av[i] + av[i-1];
}
```

to

```
#pragma omp parallel for
for (i = 0; i < 10; i++) {
  av[i] = av[i] + av[i-1];
}
```

OpenMP

Another example:

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[])
{
#pragma omp parallel
  printf("Hello world, I am thread %d\n",
         omp_get_thread_num());
  return 0;
}
```

Guesses for the output?

OpenMP

Running on an 8 core machine:

```
Hello world, I am thread 0
Hello world, I am thread 6
Hello world, I am thread 5
Hello world, I am thread 4
Hello world, I am thread 3
Hello world, I am thread 1
Hello world, I am thread 7
Hello world, I am thread 2
```

OpenMP

Note:

- the `printfs` are in no particular order; running the same code again gives a different order output
- the `printfs` are separate, the outputs are not mixed. This is because this implementation has internal locks on output streams
- We see all of the `printfs`: OpenMP has an implicit barrier at the end of each construct (superstep). This means the main thread (or rather, the `pragma parallel`) waits for all threads to finish before moving on and executing the next line (`return` in this example)

OpenMP

There are several OpenMP pragmas

```
#pragma omp parallel for
for (...) { }
```

The loop variable is made private per-thread; by default all other variables are shared between the threads

OpenMP

```
#pragma omp parallel sections
{
#pragma omp section
  {
    printf("Hello world, I am thread %d\n",
           omp_get_thread_num());
  }
#pragma omp section
  {
    printf("hi there, I am thread %d\n",
           omp_get_thread_num());
  }
}
```

This executes on (maybe) just two threads, one thread per section

OpenMP

The sections need not contain similar code

Exercise But ideally should contain codes that take roughly the same time to execute. Why?

OpenMP

```
#pragma omp parallel
{
#pragma omp for
#pragma omp sections
#pragma omp barrier
#pragma omp masked
#pragma omp critical
...
}
```

A general parallel section that contains more specific ways of parallelising

OpenMP

`barrier` is an explicit barrier

`masked` marks code that will only be executed by threads that match the mask

`critical` marks a critical region that will be executed by exactly one thread at a time (a monitor or mutex)

OpenMP

```
#include <stdio.h>

int count = 0;

void inc() {
#pragma omp critical
  count++;
}

int main(int argc, char* argv[])
{
#pragma omp parallel
  inc();

  printf("count = %d\n", count);
  return 0;
}
```

Prints the number of threads (bad code!)

OpenMP

Each parallel pragma can take extra arguments for fine control:

```
#pragma omp parallel for [shared(vars), private(vars),
firstprivate(vars),lastprivate(vars),
default(shared|none), reduction(op:vars), copyin(vars),
if(expr), ordered, schedule(type[,chunkSize])]
```

- `shared` a list of variables that are shared between the threads (default: all variables except the loop variable)
- `private` a list of variables that are private to each thread; default for a loop variable
- `nowait` remove the implicit barrier at the end of the section
- `reduction(op:vars)` private variables that are *reduced* using the `op` at the end

OpenMP

```
int i;
#pragma omp parallel reduction(+:i)
  i = omp_get_thread_num();
printf("i = %d\n", i);
```

Each thread gets its own private `i`; at the end of the section all copies are reduced to the single value of `i` by +

So, maybe, $0 + 6 + 5 + 4 + 3 + 1 + 7 + 2 = 28$

Reductions turn out to be commonly needed in parallel programs

OpenMP

There are several useful functions

- `int omp_get_num_threads(void)` returns the number of threads in this parallel region
- `int omp_get_thread_num(void)` returns a per-thread unique number
- `int omp_get_max_threads(void)` the maximum number of threads available (often defaults to the number of cores)
- `void omp_set_num_threads(int)` set the number of threads OpenMP can use
- `int omp_get_num_procs(void)` number of processors in this system

OpenMP

And lots more functionality

For example, setting the environment variable `OMP_NUM_THREADS` before running the program sets the default number of threads

```
OMP_NUM_THREADS=7 ./prog
```

OpenMP is widely supported. For example, to compile under GCC:

```
cc -fopenmp -Wall -o prog prog.c
```

OpenMP

OpenMP is clearly naturally associated with shared memory

There is a distributed memory version from Intel, called *Cluster OpenMP*

There is an undercurrent of “if your program doesn’t work well on normal OpenMP, then it won’t work well on Cluster OpenMP”

OpenMP

OpenMP

- is an evolving standard
- is easy to use; you can modify existing programs incrementally
- hides messy threads fiddling
- needs compiler support, unlike pthreads (but is supported by the mainstream compilers, in particular GCC, Clang and MSVC)
- is dependent on good implementation of the compiler: if you pass control of the parallelism to a compiler you need that compiler to be good at it
- is very large and complicated in scope
- still allows trivially buggy programs

OpenMP

Exercise Would the coursework be easier using OpenMP?

Cilk Plus

Of course, OpenMP is not the only way of tweaking C

Cilk Plus is somewhat similar in that it adds annotations and is based on fork and join

But as new keywords in C, not as pragmas (mostly)

Cilk Plus is intended as an extension to C++, but works for C, too

You may come across other versions named “Cilk” and “Cilk++”

We may have time to talk about Cilk later

Shared Memory

This concludes our discussion of the shared memory world

For now

Distributed Memory

We now turn to distributed memory programming

We could use interfaces like threads or OpenMP and have an underlying or virtualising infrastructure that converts them to message passing between processors over a network

Good programmers don't like that as it hides the source of the cost of distributed parallelism from the programmer, making it harder to design and write efficient programs

So most distributed programs are explicitly message passing, or have some other way of making the cost of an operation more clear

Distributed Memory

The big player in this field is *Message Passing Interface* (MPI)

You may hear about

- PVM: Parallel Virtual Machine, a predecessor to MPI
- SHMEM: SHared MEMory, only on Cray (SGI) machines
- UPC: Unified Parallel C, a supposed successor to MPI

MPI

MPI is what Big Science uses, when terabytes of data crunching is needed

And remember distributed systems are not good for small programs due to the overhead of the messaging outweighing the parallelism gained

MPI runs the same program on multiple processors (SPMD), but definitely not in lockstep

The processes communicate via messages

MPI

MPI is “simply” a library of functions to do messaging; you can use it with normal (unmodified) C, Fortran, etc.

Even Java, Python and other languages less suited to high performance systems

MPI is actually a standard with several competing implementations

Code written to the standard should run on any implementation

But frequently doesn’t

The MPI standard specifies a huge number of functions, covering a wide range of different types of messaging

MPI

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv)
{
  int rc, myrank, nproc, namelen;
  char name[MPI_MAX_PROCESSOR_NAME];

  rc = MPI_Init(&argc, &argv);
  if (rc != MPI_SUCCESS) {
     printf ("Error starting MPI program\n");
     MPI_Abort(MPI_COMM_WORLD, rc);
  }

  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

continued

MPI

```
  if (myrank == 0) {
    printf("main reports %d procs\n", nproc);
  }

  namelen = MPI_MAX_PROCESSOR_NAME;
  MPI_Get_processor_name(name, &namelen);
  printf("hello world %d from '%s'\n", myrank, name);

  MPI_Finalize();
  return 0;
}
```

MPI

Notes:

- `MPI_Init(&argc, &argv);` Set up the system: you must always do this. A batch processing system (e.g., SLURM) starts the processes on all the processors, while `MPI_Init` sets up the connections between them
- Later versions of MPI allow `MPI_Init(NULL, NULL)` but the above is preferable as it provides more information to the MPI system
- `rc` Always check to make sure it worked
- `MPI_COMM_WORLD` The system can be sub-divided into subsets of processors called *communicators*. The `WORLD` communicator is all processors; `MPI_COMM_SELF` refers to just the calling processor

MPI

- `MPI_Comm_rank` Each process in a communicator has a unique rank within that communicator: this is just an integer from 0 to *size of the communicator* -1. So, for `WORLD` the rank ranges from 0 to *total number of processors* -1
- `MPI_Comm_size` Get the size of the communicator
- `if (myrank == 0)` All processors run the same code (SPMD). This is how we get different things happening on different processors
- `MPI_Finalize` All procs must always call this to tidy up their MPI state

MPI

Compile using `mpicc`:

```
mpicc -Wall -o hellompi hellompi.c
```

MPI

Batch file `runnit.slm`:

```
#!/bin/sh
#SBATCH --account=cm30225
#SBATCH --partition=teaching
#SBATCH --job-name=HelloMPI
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8

mpirun ./hellompi
```

MPI

The lines of note here are:

- `--nodes=2` we want two nodes
- `--ntasks-per-node=8` we will be using just 8 of the 44 cores on each node

Recall we had:

```
if (myrank == 0) {
  printf("main reports %d procs\n", nproc);
}

namelen = MPI_MAX_PROCESSOR_NAME;
MPI_Get_processor_name(name, &namelen);
printf("hello world %d from '%s'\n", myrank, name);
```

MPI

Output:

```
hello world 3 from 'ip-AC125409'
hello world 5 from 'ip-AC125409'
hello world 4 from 'ip-AC125409'
hello world 11 from 'ip-AC125408'
hello world 6 from 'ip-AC125409'
hello world 9 from 'ip-AC125408'
hello world 1 from 'ip-AC125409'
hello world 15 from 'ip-AC125408'
hello world 7 from 'ip-AC125409'
hello world 12 from 'ip-AC125408'
hello world 2 from 'ip-AC125409'
hello world 10 from 'ip-AC125408'
main reports 16 procs
hello world 0 from 'ip-AC125409'
hello world 14 from 'ip-AC125408'
hello world 13 from 'ip-AC125408'
hello world 8 from 'ip-AC125408'
```

MPI

Notes:

- `ip-AC125408` and `ip-AC125409` are the names of the two nodes that happened to be allocated; the next run may well get different nodes
- Processes 0–8 are on `ip-AC125409` while processes 9-15 are on `ip-AC125408`, but it might happen the other way around
- `ntasks-per-node` is important here as sometimes you want fewer MPI tasks on a node than there are cores on that node: an MPI task can itself be multithreaded (not your coursework!)

MPI

- Output in a random order, even for the “main reports 16 procs” which we might think happens first!
- We *do* see “main reports” before “hello world 0”, though!
- MPI has a mechanism for routing prints on any node back via the network to a single point: this results in all kinds of timing variations in output

MPI

- MPI is SPMD, so this code is *not* synchronised across processors
- For example, when proc 0 is doing its `printf` the other processors may well already be doing `MPI_Get_processor_name`
- Or perhaps still `MPI_Comm_size`
- But many MPI function calls do have a built-in synchronisation and block the calling processor until all processors involved in that call are done
- Each MPI "task" is a separate *process*, not sharing anything with any other task: in particular, not sharing any variables (e.g., `myrank`), even if the tasks happen to be on the same node

MPI

Exercise Does adding a `MPI_Barrier` after the “main reports” conditional *ensure* the message comes out first?

MPI

In the batch file, `mpirun` sets up the processors and processes involved

Depending on the MPI implementation, this might be clever and sort out the best transport between them, e.g., in memory for processors on the same node and on the network for processors on different nodes

Or it might simply use network connections, regardless

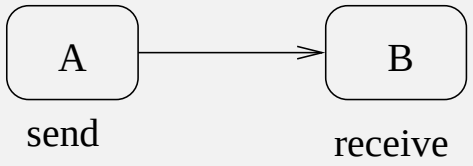
The programmer uses the same MPI functions to send messages whatever the underlying mechanism

MPI

One-to-one messaging

MPI is about sending messages between processes

A basic use scenario is when one processor wants to send a message (some data) to another



Simple message send

Processor A sends data (integers, floats, strings, etc.) to B

A can use a *send* function, while B uses a *receive* function

MPI

One-to-one messaging

```
int n[5];
...
if (myrank == 0) {
  MPI_Send(n, 5, MPI_INT, 1, 99, MPI_COMM_WORLD);
}
else if (myrank == 1) {
  MPI_Status stat;
  MPI_Recv(n, 5, MPI_INT, 0, 99, MPI_COMM_WORLD, &stat);
}
```

We suppose A has rank 0, B rank 1 in `WORLD`

MPI

One-to-one messaging

`MPI_Send` uses

- `n` A pointer to a memory location containing the data; can be a single variable or (more likely) an array of values
- `5` The number of items to send
- `MPI_INT` The type of the items
- `1` The rank of the destination
- `99` A *tag* As there can be many messages flying around you can tag them with specific integers. This allows you match up a particular send with a particular receive
- `MPI_COMM_WORLD` The rank is within this communicator

MPI

One-to-one messaging

`MPI_Recv` uses

- `n` A pointer to a memory location where to store the data: it need not be the same place as A (`n` in our example) as B is a separate process
- `5` The number of items to read
- `MPI_INT` The type of the items
- `0` The rank of the source
- `99` The *tag* on the message you are waiting for: use `MPI_ANY_TAG` if you don't care
- `MPI_COMM_WORLD` The communicator
- `stat` A structure contains the status of the transfer, in particular the source and tag; and the error type in case of an error

MPI

Messaging Types

Types include
`MPI_CHAR`, `MPI_SHORT`, `MPI_INT`, `MPI_LONG`, `MPI_FLOAT`,
`MPI_DOUBLE`, `MPI_BYTE`
among several others

MPI

Messaging Types

`MPI_Send` and `MPI_Recv` are *blocking*, meaning `MPI_Send` waits until the data has been copied out of the buffer `n` into the messaging subsystem. The array `n` in A can be safely reused immediately after the `MPI_Send` call returns

Note the data itself may not yet have reached or have been read by B

Or even sent yet by A; all we know is that is has been copied out of `n`

Naturally, `MPI_Recv` waits until the data is safely copied into its buffer

MPI

Messaging Types

This provides a weak synchronisation between A and B

All we know is that B has to wait for A: nothing more than that

B gets the data after A produced it

Beyond this synchronisation we can say little about what the relationship between A and B is

For example, A won't know when B actually gets the data; B doesn't know when A sent the data

MPI

Asynchronous messaging

In a distributed system you have to be aware of the *asynchronous* nature of communication

As messages take a significant time to be transmitted a send and a receive are certainly non-simultaneous

In comparison, in a shared memory system, once a value is written to a variable, that value is available essentially instantly everywhere (ignoring caching and speed of light issues!)

MPI

MPI also provides

- `MPI_Ssend` Waits until the destination has started to receive the message: a stronger synchronisation, not often needed
- `MPI_Isend` Send, but don't wait and carry on processing. A separate thread or DMA subsystem will asynchronously copy out and send the data. You have to be careful about reusing the buffer too soon ("I" for "immediate")
- `MPI_Irecv` Indicate a buffer where data should be read into, but don't wait for it; the data will be copied asynchronously into the buffer at some point in the future
- `MPI_Wait` Block until a given non-blocking send or recv has completed

And lots more

MPI

Synchronisation

Simple synchronisation can be achieved by
`MPI_Barrier(MPI_Comm comm);`

This blocks until all the processes in the communicator have reached the barrier

Note that the processes involved in the barrier are specified by the communicator; compare with pthread barriers that wait for any *n* threads that happen to arrive

`MPI_Barrier` is rarely needed as (a) many of the other MPI functions (`MPI_Send`, `MPI_Recv` etc.) also synchronise already and (b) SPMD programs generally have less of a need for barriers anyway

If you find yourself using `MPI_Barrier`, think again!

MPI

A quick note on messages:

Messages in MPI are *reliable*, *in order*, but *not fair*

Reliable: messages don't get lost in the network

In order: if A sends message 1 then message 2 to B, then B will get message 1 before message 2: messages from one source to the same destination do not overtake each other

However, a message from A to B may be overtaken by a later message from C to B: there is no guarantee of order on messages from different sources (e.g., A to B is over the network, but C to B is in shared memory)

MPI

As usual, “not fair” means “not guaranteed fair”. Mostly things will happen in the expected orders, but you should not rely on it

If you need a specific order, use tags

A blocking receive with a tag will wait until a message with that tag arrives, even if other messages are ready waiting

MPI

Multiple participant messaging

The above send and receive are point-to-point messages, namely one source and one destination

MPI provides many more general kinds of messaging

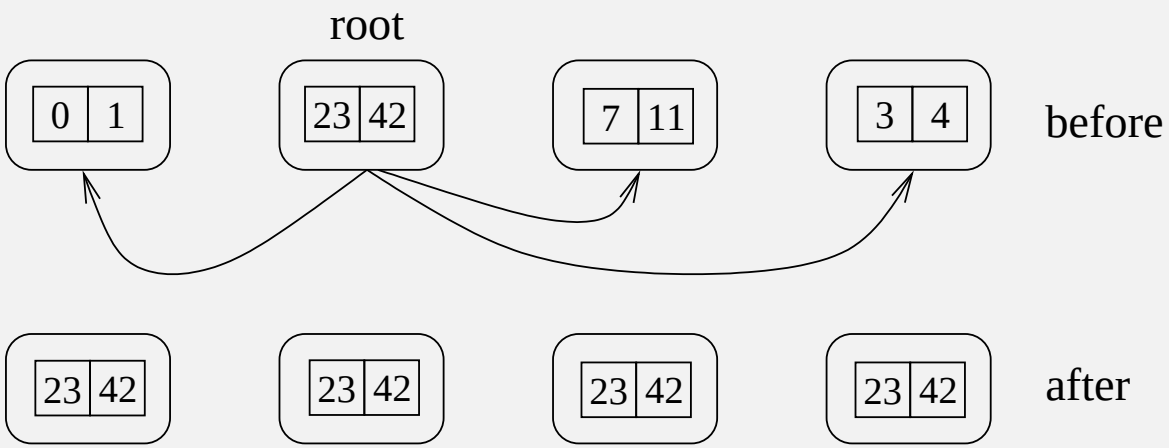
Point-to-point turns out to be much less useful than you might think

MPI

Broadcast:

```
MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,
int root, MPI_Comm comm);
```

The buffer of data is sent from the process with rank `root` to *all* processes in the communicator



MPI broadcast

MPI

Note: all processes, including the receivers, should call `MPI_Bcast` with the same value for `root`

The destination buffer can be different on each processor, but is typically the “same” buffer (in an SPMD sense)

MPI

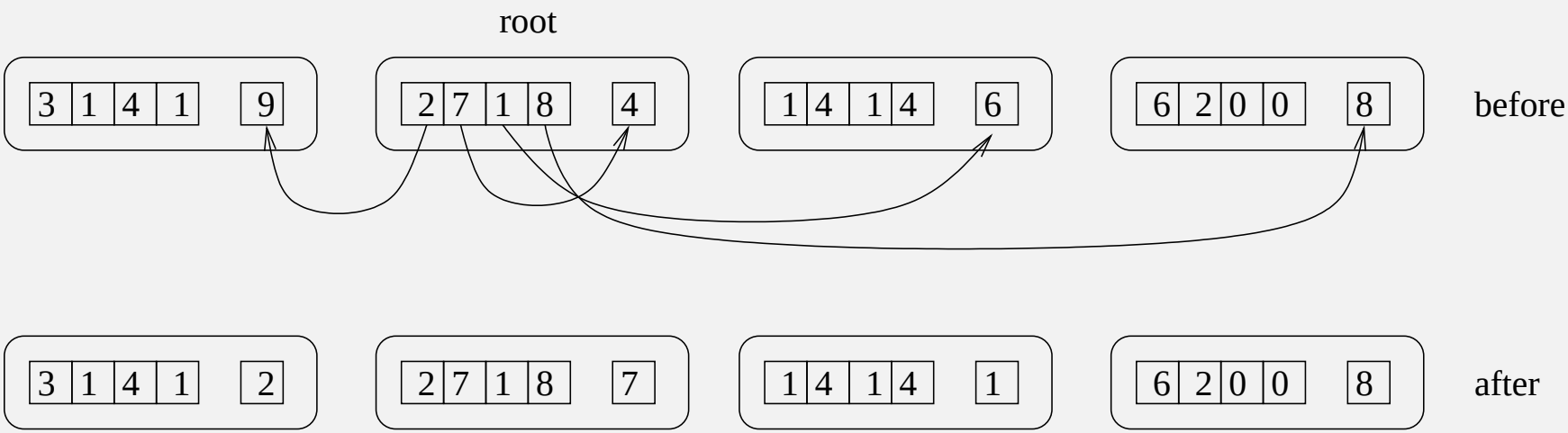
```
int n[2];
if (myrank == 1) {
  n[0] = 23;
  n[1] = 42;
}
...
MPI_Bcast(n, 2, MPI_INT, 1, MPI_COMM_WORLD);
```

All processes will now have the same values for their versions of `n`

MPI

```
MPI_Scatter(void* sendbuf,int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype
recvtype, int root, MPI_Comm comm);
```

This takes the data `sendbuf`, an array, in processor with rank `root`, and sends `sendcount` items from the array to each other processor (and to itself) to end up in `recvbuf`



Scattering single values

MPI

The processor with rank 0 (in the specified communicator) gets the first `sendcount` items from `sendbuf`; processor 1 gets the next `sendcount` items; and so on

Just as in `broadcast`, every processor executes `SCATTER` with the same `root`

Note: `recvtype` can be different from `sendtype`, but you had better be sure you understand what you are doing

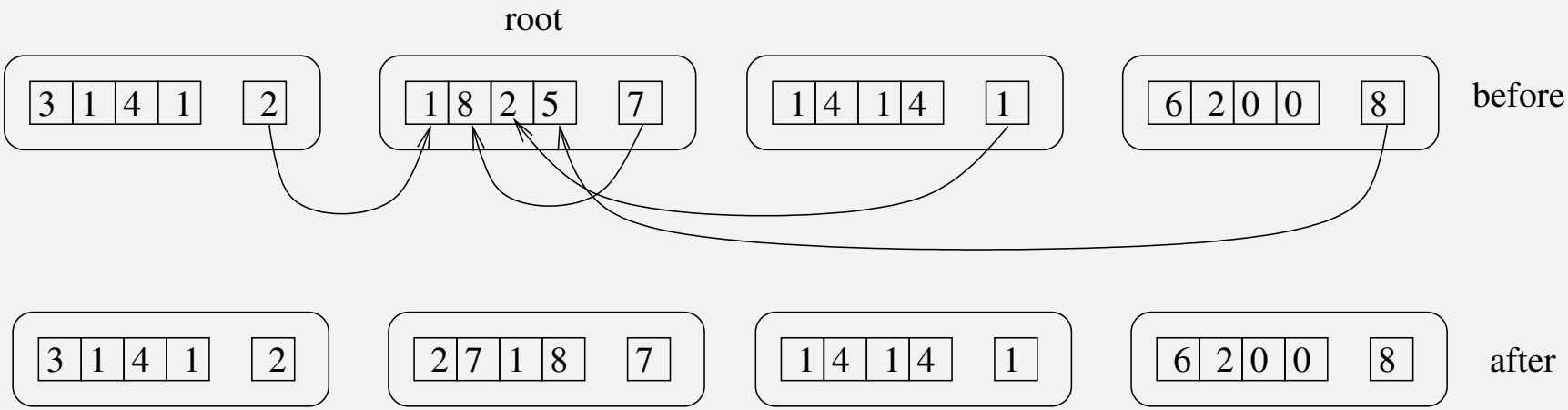
`recvcount` can be different from `sendcount`, but you had better be sure you understand what you are doing

Don't do that!

MPI

```
MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype
recvtype, int root, MPI_Comm comm);
```

Takes `sendcount` elements of data `sendbuf` from each processor and puts them in the array `recvbuf` on processor `root`



Gathering single values

MPI

`MPI_Gather` is the “opposite” of `MPI_Scatter`

The `recvbuf` on the root processor is filled, in order, with the specified number of items from processors rank 0, 1, etc.

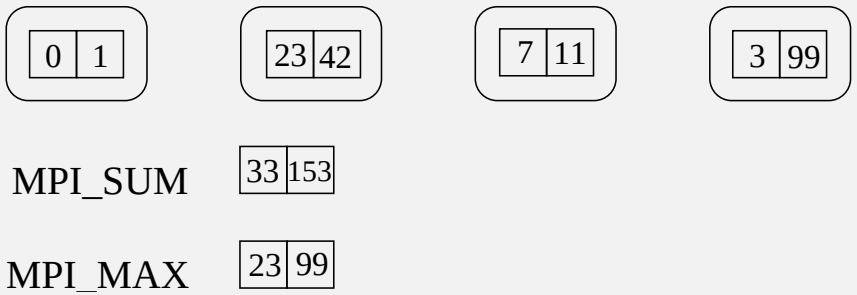
Type and counts can vary across processors

But don’t do that

MPI

```
MPI_Reduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
```

Applies a reduction of operation `op` to each value in `sendbuf`, putting the result(s) into `recvbuf` on processor `root`



MPI reduce

MPI

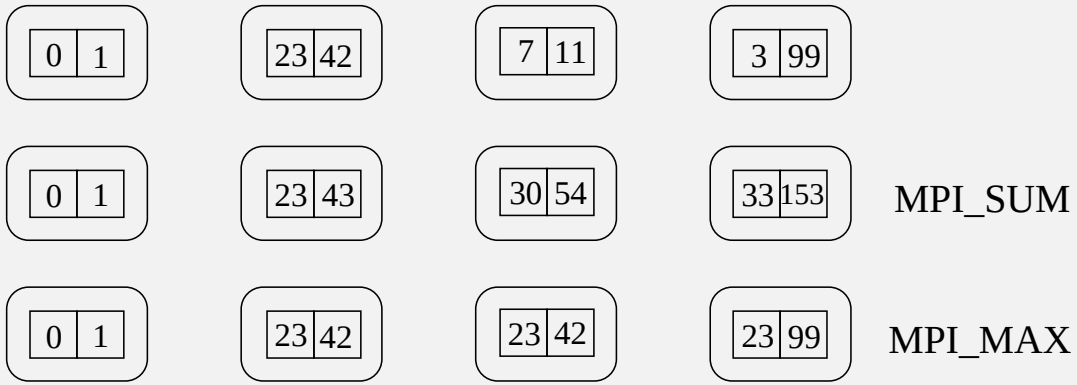
Operations include
`MPI_MAX`, `MPI_MIN`, `MP_SUM`, `MPI_PROD`, `MPI_LAND` (logical AND), `MPI_LOR` (logical OR)
amongst others

You can also define your own reduction operators

MPI

```
MPI_Scan(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

A *prefix scan* of the source `sendbuf`. Processor of rank *i* gets the reduction of values from processors 0 . . . *i* stored in its `recvbuf`



MPI scan

Prefix scans turn out to be a very useful tool in parallel algorithms

MPI

As usual with MPI, there are many other combinations of blocking and non-blocking messages possible

Note these functions are **not cheap**: they hide a lot of messaging, which you should be aware of when you are using them

For example, a `MPI_Bcast` of a large datastructure can be very slow

MPI

For timing, `MPI_Wtime()` returns a “high precision” elapsed time in seconds on the calling processor

It returns a `double`, with precision as given by `MPI_Wtick()`

This might be, say, 0.000001 (1 microsecond)

MPI

MPI also provides

- defining new MPI datatypes including arrays and structures;
- means of creating communicators;
- processor groups (communicators contain one or more groups);
- processor topologies (ways of arranging processors into particular geometric shapes that might fit a certain problem or hardware);
- more kinds of scatter/gather/reduce/scan;
- all-to-all broadcasts;
- and so on

MPI

MPI is used extensively out there in the big world of Real Science

It is very well suited for when there is so much computation needed that the overhead of a bunch of messages is well worth paying

The large (100k core) clusters will be running jobs using MPI

MPI scales very well to large systems

MPI

And, of course, you can mix shared and distributed memory: running shared memory OpenMP tasks communicating across nodes via MPI

Don’t use OpenMP in the coursework: that should be pure MPI

MPI

MPI requires you to make sure all your MPI function calls are coordinated across the processes: every processor must call the appropriate same or matching functions at the appropriate times

This the programmer's problem: it's a bug if you get it wrong

MPI

For example, you can still easily deadlock. Suppose A and B wish to exchange messages:

A

```
MPI_Recv(...);
...
MPI_Send(...);
```

B

```
MPI_Recv(...);
...
MPI_Send(...);
```

This is slightly more obvious when it happens since MPI is SPMD and has a single program source

Careful use of message tags helps structuring

As is common, MPI provides easy mechanism but no analysis

MPI

In fact, for this case, MPI provides `MPI_Sendrecv` which combines a send with a receive that is guaranteed not to deadlock

A

```
MPI_Sendrecv(...);
```

B

```
MPI_Sendrecv(...);
```

This function is recommended in cases of swapping data

And it can connect any pair of processes; is not limited to simple swapping between two processes. For example, A sends to B but receives from C; while B sends to C but receives from A; etc.

MPI

Using MPI requires careful thought about messages to get the maximum efficiency out of the system

For example, we might be able to overcome message latency by judicious use of non-blocking sends and receives

Rather than waiting for a receive to complete, we carry on working on some other part of the computation: later, when the receive has completed, we can go back to that part of the computation

MPI

This requires careful programming, but can give good results

Sometimes not

In general (not just distributed computing), overlapping communication and computation is a good thing to do

But hard to program and easy to make errors

Exercise You wish to make a cup of tea and a sandwich. Do you

(a) make the sandwich then start boiling the kettle; or
(b) start boiling the kettle then make the sandwich?

MPI

Also:

- messaging has a high overhead, so MPI only really works well on very large programs
- it is hard to program effectively: simple programs are easy to write, but efficient programs usually need experienced programmers
- there are a huge number of variations of messaging: quite often you can replace several calls to MPI functions with one, more complex, MPI function that is more efficient overall

MPI

- you need a careful balance of MPI function calls and data movement: you would generally aim to use as few MPI calls as possible, but sometimes moving less data with more calls can be better than moving large amounts of data with fewer calls
- it is not naturally dynamic: the number of processors is effectively fixed and cannot vary during the execution of the program. This excludes efficient execution of some kinds of program (later versions of MPI do include `MPI_Comm_spawn` but it's not easy to use)

MPI

MPI has succeeded for many reasons

- An open standard, inviting several competing implementations
- Thus implementations tend to be optimised and efficient
- MPI is simple in concept, so straightforward to program (not necessarily *easy* to program. . .)
- MPI is flexible as it contains lots of kinds of communication
- MPI is supported by many languages and environments
- MPI scales well to very large problems

The MPI standard is still being developed and updated

MPI

Exercise Read about UPC, a (not popular) alternative to MPI, that presents a virtual shared NUMA architecture

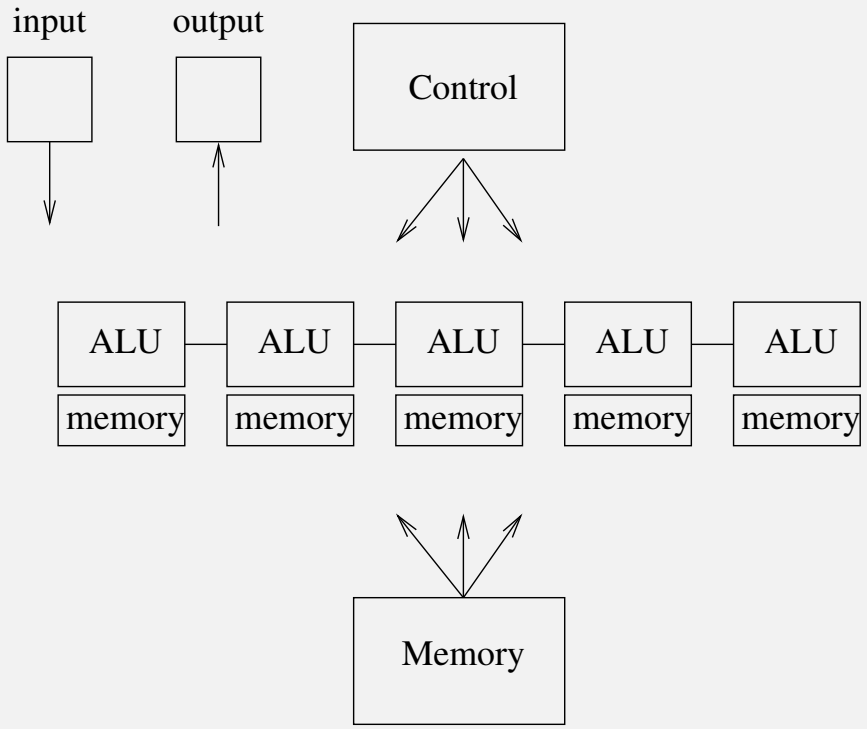
Vector and Array Processors

Moving on from distributed: the next major architecture to consider is SIMD

Recall: these have many processors all executing the same thing on different data

First we need to recall the SIMD architecture and go through the issues it brings

Vector and Array Processors



SIMD box model

All processors are controlled by just one Control unit, so are all executing the same instruction

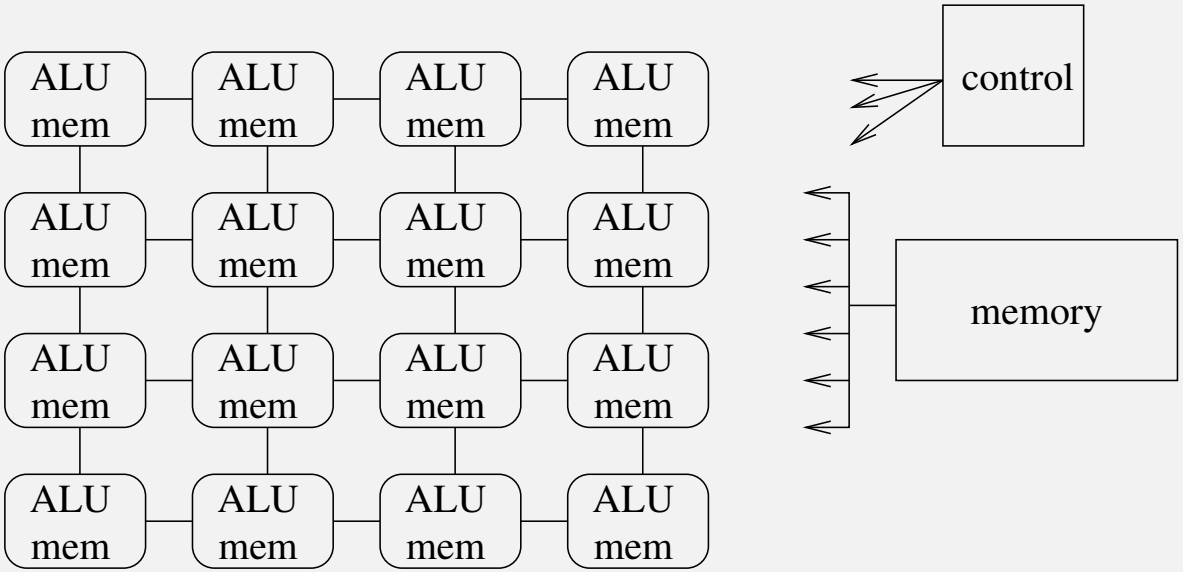
This is data parallelism

Vector and Array Processors

There is a shared chunk of *global memory* and each processor has its own chunk of *private memory*

Processors can be strung linearly in a *vector* or in a square mesh as an *array*

Vector and Array Processors



Array processor

Of course, you can use an array as a vector or a vector as an array, with a modest loss of efficiency

Vector and Array Processors

Vector processors appeared quite early on in computer architectures (1960s) and were a mainstay in 1980s supercomputers (Crays), as they are a relatively simple extension of the uniprocesor

Array processors have come into fashion and gone away again several times

GPUs owe a lot to array processor design: more on this later

Vector and Array Processors

The basic idea of SIMD is that we can parallelise loops like

```
for (i = 0; i < 1024; i++) {
  c[i] = a[i] + b[i];
}
```

as

```
 in parallel do c[i] = a[i] + b[i];
```

Exercise Go back and look at OpenMP

Vector and Array Processors

The important points being

- all elements in the arrays are being treated identically
- there is no interference between any of the operations
- there are no dependencies across iterations of the loop

So no races, thus no serialisation of the operations is needed

Vector and Array Processors

What if there *are* conflicts? For example

```
for (i = 1; i < 1024; i++) {
  a[i] = a[i] + a[i-1];
}
```

Here, the new value of `a[i]` depends on the value of `a[i-1]`; which will have been updated in the previous iteration of the loop

In comparison

```
in parallel do a[i] = a[i] + a[i-1];
```

takes the original value of `a[i-1]`

Vector and Array Processors

Starting with `a` = 1, 1, 1, 1; the sequential loop gives

1 2 1 1
1 2 3 1
1 2 3 4

While the parallel version gives

```
 1  1  1  1
    1  1  1  +
-------------
 1  2  2  2
```

This is due to the nature of the original loop: it is actually a *prefix scan* operation

Prefix scans can be done SIMD, but when parallelising code you have to be aware that is what is happening!

Vector and Array Processors

Having given a warning, SIMD processing is very powerful

Vectors and arrays with thousands of processors are common

If your problem is data parallel, it can get huge speedups by running SIMD

If you can get your data to the individual processors fast enough

Vector and Array Processors

In SIMD the processing power is not the problem: it's the data movement

With thousands of processors, CPU is essentially free

The major way to lose efficiency is through data movement

Vector and Array Processors

As usual, the bus bandwidths between the processors and between the global memory and the processors is much less than you might wish

The total *aggregate* bandwidth, adding together all the individual bandwidths of all the buses can be huge, but this is a useless statistic (thus is given by marketing)

Careful overlapping of communications and processing is the way to make these systems work at their best efficiency

Thus, for example, rather than waiting for a read from memory to return a value, go away and do some other computation while the read is being processed

This kind of asynchronous programming improves efficiency but is much harder to do and to get right

Vector and Array Processors

Back to the SIMD architecture: now is the point where need to talk about an interesting feature of SIMD processing

The main feature of SIMD is that all processors are doing the same thing. . .

. . . so how can conditionals work?

Here is an example, written using a fictional SIMD C

Vector and Array Processors

Suppose we have a `get_proc()` function ("get processor number") that returns the index of the processor:

```
int me;
me = get_proc();
...
```

This allows us to distinguish between processors; the value of `me` is different on each processor

We could use `me` to index into a vector, so each processor operates on a different element

```
v[me] = (v[me - 1] + v[me + 1])/2.0;
```

Vector and Array Processors

So what does this code do?

```
int me, n;

me = get_proc();

if (me > 512) {
  n = 1;
}
else {
  n = -1;
}
```

Vector and Array Processors

Instinctively you think it sets `n` in processors above 512 to 1 and in the other processors `n` is set to -1

And this is what it does do

But a SIMD machine executes the same code in all processors, so how can it execute the `n = 1` assignment on some and the `n = -1` assignment on others?

Vector and Array Processors

It doesn't: at any point in time each processor is executing the current instruction

or doing nothing at all

Processors can be *inhibited*, meaning not participating in the current instruction

There is a per-processor inhibit flag to say whether this processor is on or off

This is how we get different code paths on different processors

Vector and Array Processors

We must modify our description of SIMD machines:

> *Each processor either executes the same instruction as the others; or does nothing at all*

Vector and Array Processors

Returning to the code

```
if (me > 512) {
  n = 1;
}
else {
  n = -1;
}
```

This is executed as follows:

- All processors execute the test in the `if`
- In those processors for which the test fails, the inhibit flag is set
- All processors move to the `n = 1`; the inhibited processors do nothing while the others execute the assignment

Vector and Array Processors

- All processors move to the `else`; all inhibit flags are inverted
- All processors move to the `n = -1`; the inhibited processors do nothing while the others execute the assignment
- All inhibit flags are cleared
- All processors move on to after the `if`

Both branches of an `if` always taken by all processors!

Vector and Array Processors

Overlay

Vector and Array Processors

The time taken for an `if` is the sum of the times of both branches

Quite different from sequential code

Reality is a little more complicated: think about nested `if`s

There is actually a *stack* of inhibit flags!

Exercise Think this through for yourself!

Vector and Array Processors

This seems like poor use of our processors if lots of them are inhibited

True, so SIMD code should be written to minimise conditional branches

But with thousands of CPUs, processing power is cheap, so inhibiting some of them is not as bad as it seems, as long as it is not overdone

```
if (me > 512) foo();
else bar();
```

is not good code: all of `foo` must be executed before `bar` can start, so there is a large amount of inhibition

Vector and Array Processors

Inhibition applies to all conditional code, like loops:

```
int i, n;
...
for (i = 0; i < n; i++) {
  ...
}
```

All processors start the loop

As `i` increases, some processors pass their exit test and are inhibited; other processors continue executing; *all processors continue looping*

Note no processor starts executing after the loop until *all* processors have exited

Vector and Array Processors

Loops must wait until all processors have completed: they take time the maximum of the individual processors

SIMD loops are most efficient when all the loops are of the same size

Similarly for all conditional constructs: if there is a choice all processors will take all the choices, but some are appropriately inhibited

Vector and Array Processors

Connection Machines had a lightbulb per processor: initially they set it so the light was on when the processor was active

After a while they fixed it so the light was on when the processor was inhibited. . .

We shall return to SIMD programming with CUDA, later, when we talk about parallel languages

End of Architectures

We have seen a variety of machine architectures, but primarily people use:

- shared memory
- distributed memory
- SIMD

Quite often, all at once!

It is time to move from the machines to the code running on them

Parallel Algorithms

We now turn to parallel *algorithms*

We shall approach them in two ways

- general principles
- specific examples

The first will look at a few general techniques and some classic problems in parallelism

The second will be a couple of specific algorithms, such as a parallel sort

Parallel Algorithms

Divide and Conquer

Perhaps the simplest way to parallelise a problem is *divide and conquer*

- subdivide the problem into smaller parts
- process the parts in parallel
- merge the results back together

Of course, this only applies if you have a problem that you *can* subdivide!

And it works best if the parts are independent of each other: less communication

Parallel Algorithms

Divide and Conquer

For example, summing n values becomes

- subdivide the values into smaller chunks, sending the chunks to separate processors
- each processor sums its chunk (process in parallel)
- return the results to the main processor and add the values together (merge)

Parallel Algorithms

Divide and Conquer

Question: how big should the chunks be?

Too small and we spend all our time in communication overhead; plus the merge step gets bigger

Too large, thus fewer chunks, and we might not get the parallelism we want

Parallel Algorithms

Granularity

This is the question of *granularity*, or “chunk size”

A big problem in programming parallelism is deciding on the choice of granularity of a sub-problem, for exactly the reasons given above

Computing a single sum is a small grain; while averaging a row of a large matrix is a big grain

The former you might not want to parallelise; the latter you would

Parallel Algorithms

Granularity

Grain size: the size of a chunk

You will see “small grain” and “large grain”; alternatively “fine grain” and “coarse grain”

Granularity: the ability of a problem (data or computation) to be divided into fine or only coarse grains

Some programs may only admit a coarse granularity

Some may admit a fine grain, but should we split it up into small grains?

Parallel Algorithms

Granularity

Fine: more parallelism, more communications

Coarse: less parallelism, less communications

Parallel Algorithms

Granularity

It's the grey area in the middle that is the issue: how large should a grain be before we consider running it in parallel?

The answer: it depends

On everything, but particularly the ratio of computation time to communications speed on the particular hardware we have

Parallel Algorithms

Granularity

For fast communications (shared memory, perhaps) we would chop our problem up into relatively small grains

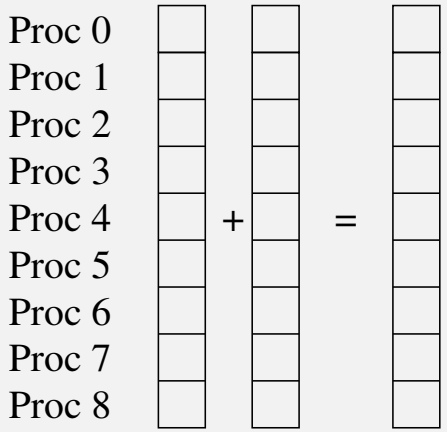
For slow communications (distributed memory, perhaps) the sub-problems need to be larger before we benefit from parallelising

Often, the best way of working it out is just to try some test programs and measure the result

Parallel Algorithms

Granularity

An example: adding together two large vectors, maybe on shared memory, maybe on distributed memory



Adding vectors

The simple fine grain allocation of one add per processor might not be the best if communications costs dictate otherwise

Parallel Algorithms

Granularity

For example, if the time it takes to get the data to the individual processors is large we would want to reduce the data movement

And in current memory architectures, it could take roughly the same amount of time to move one byte as it takes to move 10 or 100 or 1000 bytes

Time $=$ fixed overhead in setting up the transfer $+$
variable overhead in doing the transfer

Parallel Algorithms

Granularity

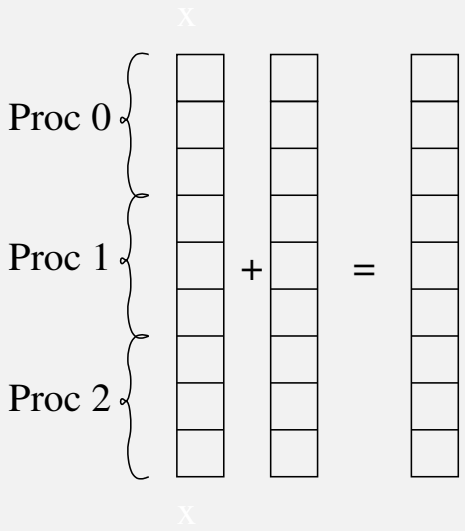
Thus: if we need to move data, move it in large chunks

So, typically, we would have each processor would take a selection of elements and add them sequentially

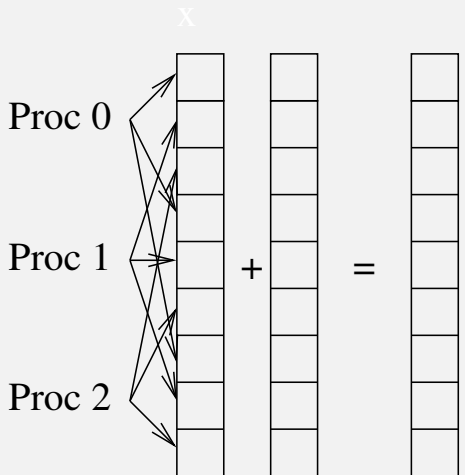
Larger grains of computation

Parallel Algorithms

Granularity



Adding contiguous blocks



Parallel Algorithms

Divide and Conquer

The size of the grain we need will dictate the number of chunks we chop the problem into

How many sub-problems should we have on each core?

It is sometimes recommended that you have a “few” sub-problems per processor

This allows you to overlap communications with computation

While a sub-problem is waiting for some data, the processor can continue computing on another sub-problem

Parallel Algorithms

Divide and Conquer

How many is “a few”?

It depends

GPUs like to have *very many* many sub-problems per cores: as graphics problems need to push a lot of data around the processors would need to hang around doing nothing while waiting for data a lot: unless they have lots of other sub-problems to work on

Parallel Algorithms

Divide and Conquer

Back to divide and conquer of adding numbers: isn't the merge step "add the values together" just another instance of the original question?

Yes, so a lot of divide and conquer methods are deeply recursive (not all, though)

Parallel Algorithms

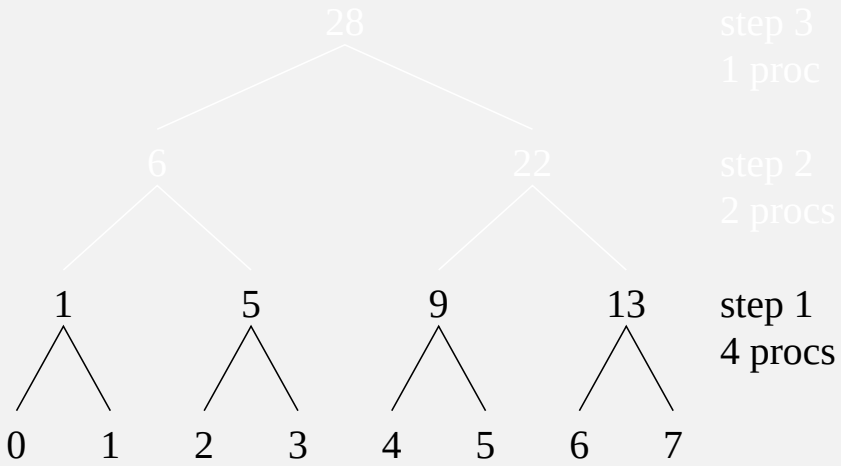
Divide and Conquer

This summation problem is usually regarded as

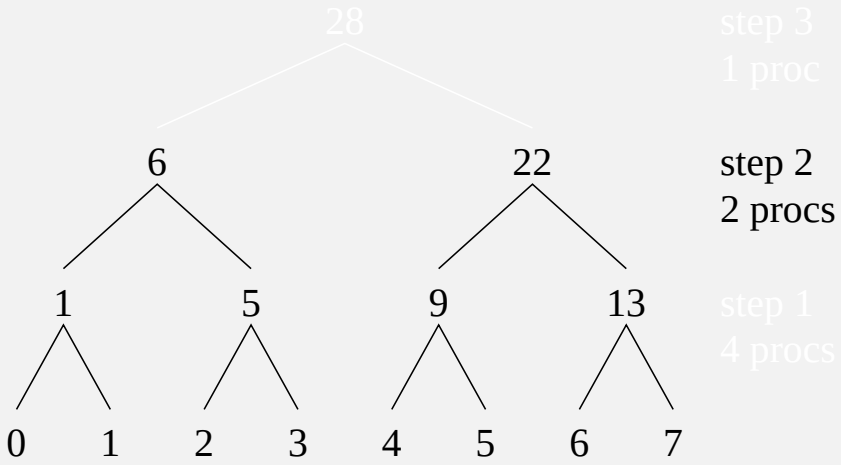
- if the number of values is small then
- add them directly, sequentially
- return the sum
- else divide them into two chunks
- recursively sum the parts in parallel
- add the two results
- return the sum

Parallel Algorithms

Divide and Conquer



Add pairs



Add pairs of sums

Parallel Algorithms

Divide and Conquer

We can compute the speedup and efficiency of this. We ignore communications overhead, so essentially we are using a PRAM model

Time on a sequential processor: 7

Time on this parallel system: 3

Speedup $= 7/3 = 2.33$

Efficiency, using 4 processors: $2.33/4 = 58\%$

Note we are only using all the processors in the first step: thereafter there is increasing amounts of idle hardware

Parallel Algorithms

Divide and Conquer

Divide and conquer is a good approach as long as you use it carefully

It is natural and easy to understand

It is fairly easy to program

It scales well to very large problems

But not all problems break up arbitrarily like this

And merging the parts can be as hard as the original problem

Parallel Algorithms

Divide and Conquer

It is a good technique to use in sequential systems, too

Recall merge sort (divide and conquer) is much better than bubble sort

Bubble sort isn't parallelisable in any meaningful way (while still remaining essentially a bubble sort)

The Fast Fourier Transform is a prime example of a good sequential application of divide and conquer

Parallel Algorithms

Divide and Conquer

Of course splitting up isn't always the best option when you have a big problem. Counselling often works.
Anonymous. CM30225 exam, January 2011

Parallel Algorithms

Provider/Consumer

Terminology: we shall describe a method that previously was called “master/slave”: if you need to look it up, you will find it under this name

Until a generally agreed replacement terminology is decided, we shall be calling it “provider/consumer”

Parallel Algorithms

Provider/Consumer

Divide and conquer is a way of arranging the problem. We now look at a way of arranging the control of the processing

Provider/consumer is a technique where there is a single main thread that determines what many consumer threads do

For example, to do a large matrix multiplication, the main thread could get many consumer threads to do sub-parts of the operation

When the consumers are done the main thread can continue

Parallel Algorithms

Provider/Consumer

Provider/consumer aligns naturally with divide and conquer, but usually not in a recursive way: in most uses the consumers don't use sub-consumers

Note: these ideas are not mutually exclusive, but they tend to overlap somewhat

Parallel Algorithms

Provider/Consumer

Provider/consumer is also related to the *server farm*, where a (large) collection of machines waits for problems to be sent to them

For example, to do a search Google might send out sub-parts of the search to a collection of machines, and then collate the results

In any case, in provider/consumer there is an asymmetry of control: one thread controlling several others

Parallel Algorithms

Manager/Worker

Provider/consumer is superficially quite similar to *manager/worker*, also called *bag of tasks*

In this, there is a global set of problems to process held by the manager and the workers request a problem from the manager as they need

A different control than provider/consumer

This allows easy *load balancing* on the workers

Parallel Algorithms

Load Balancing

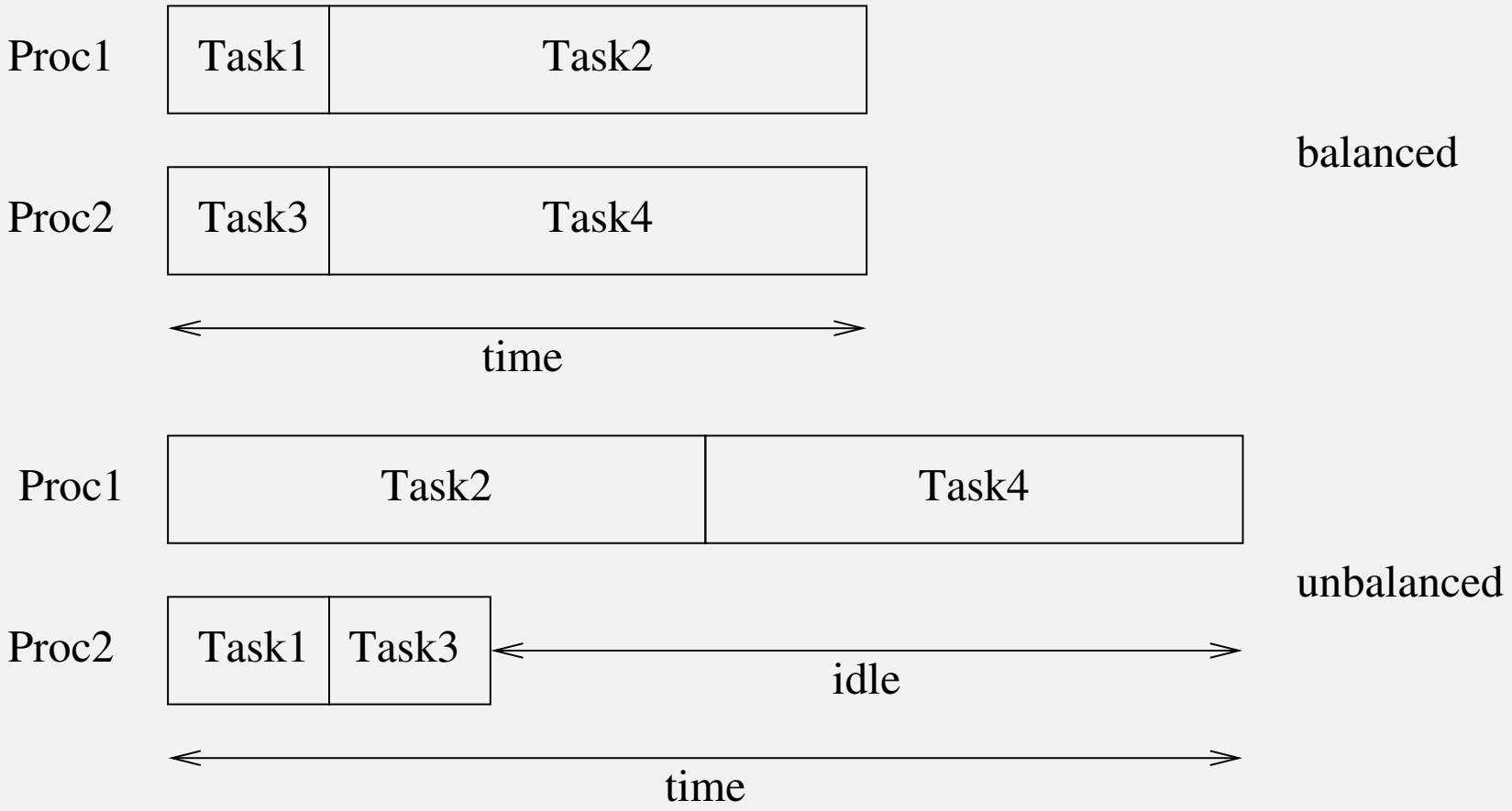
Load balancing is one thing to do to approach a good efficiency

For example, if we have two big (time consuming) problems and two small ones, and two processors it makes sense to give each processor one big and one small

If we give one processor both big problems and the other both the little ones it is clear our speedup and efficiency will both be lower as the second processor will soon be idling while we wait for the first to finish

Parallel Algorithms

Load Balancing



Balanced and unbalanced computations

Parallel Algorithms

Load Balancing

Load balancing tries to spread out the workload in a sensible fashion

It requires us to have some idea of how big each sub-problem is, namely a good estimate of their granularity

Theory tells us that this is impossible in general, but for the most part in practice we can make a decent guess

Many large problems are quite regular in structure and as so fairly amenable to this kind of analysis, but there are many irregular problems that are not so easy

Parallel Algorithms

Load Balancing

And even if we have a good idea of the size of each task, finding an even balance can be difficult (the *multiprocessor scheduling problem* is NP-hard)

Note that load balancing applies to more than just CPU cycles: there's memory, network bandwidth and any other limited resource

And these play off against each other: it may be worthwhile to put two sub-problems on the same processor if they need to swap data and this will reduce communications overheads

Load balancing is quite similar to process scheduling in operating systems: but now we might be working with large distributed systems

Parallel Algorithms

Manager/Worker

The manager/worker model is good because it is somewhat self-balancing on average

A worker that happens to get a small task will soon be back for another task

Provider/consumer might have to take some care over which tasks it supplies to where

Though this is not a problem if all sub-tasks are the same size. Provider/consumer is good for this case and might be simpler to implement than manager/worker

Parallel Algorithms

Thread Pools

A way of implementing manager/worker is to use thread pools

We have a pool of threads that take tasks from one or more managers

After each task, a thread goes back to the manager for a new task

We mitigate the overhead of thread creation/deletion

The thread pool can be managed within the program, or system-wide by the OS

Parallel Algorithms

Thread Pools

If the pool is managed by the operating system it can have a global view of how the entire system's resources are being used

Threads can be passed to any program, again reducing the overall overheads

And the OS can increase or decrease the number of threads according to how the whole system is loaded

This requires OS support, of course: think of the issues of access to the program's address space by each thread

Parallel Algorithms

Thread Pools: GCD

This is the idea of Apple's "solution" to parallelism: *Grand Central Dispatch* (GCD)

Rather than programs creating their own threads, e.g., using pthreads, they use (and re-use) the OS's threads from a global thread pool

A program gets access to a pool thread by putting a task, e.g., a function call, on a *queue*

The worker threads pick tasks off the queues and execute them

Parallelism is obtained by having lots of worker threads taking tasks

Parallel Algorithms

Thread Pools: GCD

So GCD gets the automatic load balancing of manager/worker

GCD can also provide mutual exclusion

By creating and using a special queue called a *serial queue* a program indicates it wants just one thread to service this new queue

As only one thread executes tasks from this queue there can be no issues of interference between threads on that queue

Parallel Algorithms

Thread Pools: GCD

So, roughly speaking, code like

```
            fblock = make_lock();
get_lock(fblock);           get_lock(fblock);
foo();                      bar();
free_lock(fblock);          free_lock(fblock);
```

becomes

```
          fbqueue = make_serial_queue();
enqueue(foo, fbqueue);    enqueue(bar, fbqueue);
```

Parallel Algorithms

Thread Pools: GCD

There is no loss of parallelism by using a single thread to process the queue in this case, as the critical region has to be serialised anyway

Though you do need to be careful about making the function called as small as possible, for the usual reasons

Just as each critical resource needs its own lock, in GCD each critical resource needs its own serial queue

If a resource would need *two* locks, then you need two queues and put a function on the first queue that itself puts another function on the second queue that actually executes the required critical region

Somewhat fiddly

Parallel Algorithms

Thread Pools: GCD

Rather than placing functions in queues, Apple's implementation makes extensive use of *closures*, a feature they have added to their version of C

They call them *blocks*, but they are similar to lambdas in other languages

Of course, closures were imported from the functional programming style: as long as we have referential transparency the individual tasks can run completely independently

Parallel Algorithms

Thread Pools: GCD

Apple's claim is that queues are cheap to create and use, while threads and mutexes are expensive

They are less effusive on costs like mutual exclusion on the queue itself; costs of the OS deciding on which thread services which queue; costs of the virtual address mapping of the threads as they get assigned to processes; cost of creation and manipulation of closures; and so on

We are still waiting to see if the GCD paradigm is easy to use in real programs or not!

Parallel Algorithms

Thread Pools

While GCD uses thread pools at the OS level, the approach of a program implementing its own pool is quite common

Again, and this is true for all these concurrency paradigms, this only works well if your problem happens to fit well into the pool or manager/worker patterns

One of the many issues encountered when designing parallel programs is choosing the right parallelism pattern

Parallel Algorithms

Thread Pools

Exercise There is a Linux library `libdispatch` that implements (per process) GCD. Write some programs using it

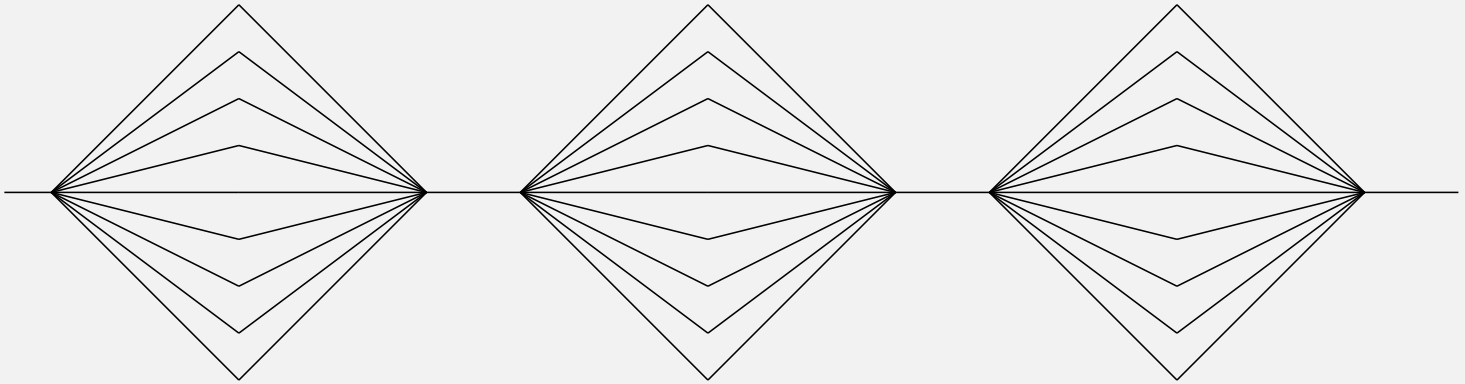
> *Disadvantages include that it is managed by Apple. No more needs to be said.*
> Anon, Jan 2023 CM30225 exam

Parallel Algorithms

Fork and Join

The next general structuring method to look at is *fork and join*

We have seen this before, as it is just the superstep



Superstep

Of course, we would like to make the sequential parts between the forks as small as possible

Parallel Algorithms

Fork and Join

This is quite popular, as many problems decompose this way

For example, multiply two matrices together *then* add in a third matrix

The processing forks to multiply the matrices using parallel sub-tasks, then joins after that

We could use barriers between the two phases

Parallel Algorithms

Fork and Join

Take care not to confuse the structure of fork and join with the creation and joining of threads

“Fork and join” describes the concurrency in the execution, not the mechanism for execution

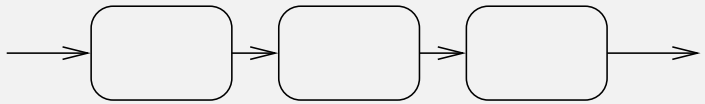
We might want to do the sub-tasks provider/consumer, or manager/worker or thread pool or whatever

It is very unlikely we would want to use `pthread_create` and `pthread_join` every time

Parallel Algorithms

Pipelines/Systolic

Another structuring method we have seen before is the *pipeline*, also called *systolic array*



Pipeline

Input data is transformed by several separate stages by several separate processors

A well-balanced pipeline (eventually) gives perfect speedup and efficiency

Parallel Algorithms

MapReduce

Finally, for now, we look at another concept imported from the functional style: *MapReduce*

This is a combination of a *map* and a *reduce*, and is a kind of divide and conquer

A map takes a function and a structure (a list or vector or tree or whatever) of data, and applies that function to each element in the structure

As long as there is no interference between the items of data, this is trivially parallelisable: stick different items of data on different processors and execute the function on each

Parallel Algorithms

MapReduce

The reduce step then gathers together all the sub-results and merges them together to produce the required answer

Depending on what kind of reduction we require, this can be extensively parallelised, too

E.g., the merge in a parallel sum being done in a tree-like way

E.g., the merge of URLs that result from a Web search can be done similarly, perhaps a sort in order of relevance

Other reductions might be less or more parallelisable

Parallel Algorithms

MapReduce

For example, given a vector of numbers compute the sum of the squares of the values

Map: do the squares in parallel

Reduce: add them together in parallel

Parallel Algorithms

MapReduce

Another example: Web search. The data is distributed in chunks across many machines

Map: a machine searches its own chunk

Reduce: merging and sorting the partial results

MapReduce is much used by Google for their various services, not just searching

Parallel Algorithms

MapReduce

This clearly scales well to huge systems!

This is helped a lot helped by the source data being stationary and sending the map function to the machine that hosts the data: a reversal of the way we normally think about things

MapReduce also copes well with less than 100% reliability of the hardware

Parallel Algorithms

Aside: Reliability

A quick word on reliability: modern machines are pretty reliable and we are not used to them breaking down too often

Huge clusters are a different proposition entirely

When you have 100s of thousands of machines in your system, you must plan for one to break down in the middle of your computation!

So another issue large systems and the algorithms that run on them have to contend with is machines failing

Parallel Algorithms

Aside: Reliability

For example, you might want to run the same sub-task on more than one processor for reliability: if one breaks you'll still get the result

At one point Hector, a UK academic cluster, was having a failure rate of one node per day

Parallel Algorithms

Classical Problems

We now turn to look at a few classical problems that are used to illustrate the issues that arise in designing parallel programs

The first is *readers/writers*, which looks at synchronisation in the shared use of data, in, for example, a database

Some processes may want to simply read data, a *reader*

Others might want to read and then update data, a *writer*

To ensure consistency in the data, a writer must have exclusive access to the database

(A simplification of reality, if you know anything about databases)

Parallel Algorithms

Readers/Writers

When there is no writer using the database, any number of readers can access it simultaneously

Note, as a consequence of exclusive access, a writer cannot access the database while there is any reader using it

One solution is to use simple primitives

Parallel Algorithms

Readers/Writers

```
int readers = 0;
rlock = make_lock();      // protect readers
wsem = make_semaphore(1);// sync writers

void reader()
{
  lock(rlock);
  readers++;
  if (readers == 1) wait(wsem);
  unlock(rlock);
  ... read ...
  lock(rlock);
  readers--;
  if (readers == 0) signal(wsem);
  unlock(rlock);
}
```

```
void writer()
{
  wait(wsem);
  ... write ...
  signal(wsem);
}
```

Parallel Algorithms

Readers/Writers

The `rlock` is to protect the count of the number of readers

The `wsem` synchronises the readers and writers: a writer must wait until all readers have left, and a reader must wait until a writer has left

`if (readers == 1) wait(wsem);` the first reader in sets the write semaphore

`if (readers == 0) signal(wsem);` the last reader out releases the semaphore

This works, but has a problem

Parallel Algorithms

Readers/Writers

The problem is that this code is unfair in the way it treats readers and writers

A writer can be excluded for an arbitrarily long time while readers come and go

- reader 1 arrives and sets the `wsem`
- a writer arrives; it waits on `wsem`
- reader 2 arrives; it can continue
- reader 1 leaves
- reader 3 arrives; it can continue
- reader 2 leaves
- and so on

Parallel Algorithms

Readers/Writers

This is called *readers' preference*

The continuing stream of readers conspire to keep out the writer: the readers never signal the `wsem`

With low probability, but it happens

This is *starvation* of the writer

Parallel Algorithms

Readers/Writers

We might try to fix the writer starvation by having a writer pending count, and have readers wait if there is a writer (or some suitable number of writers) waiting

Exercise Do this

But now we have a writers' preference and readers can be starved

Parallel Algorithms

Readers/Writers

Making this fair for both readers and writers is harder than you think

Though having a readers' preference is not as bad as you might think, as typical code has more reads than writes

Exercise Go and read up on the many suggested solutions to readers/writers

Exercise Read about the POSIX `pthread_rwlock`

Exercise Read about *read-copy-update* (RCU) and its choice of compromises

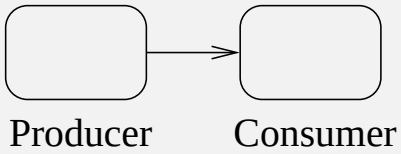
Exercise Think about how you might use GCD queues

Parallel Algorithms

Producers/Consumers

The next classical problem looks at how two or more processes can communicate: passing data between processes

For example, how a manager might feed data to a worker



Producer/Consumer

If the producer sends directly to the consumer, this would require a synchronisation between them for every data item

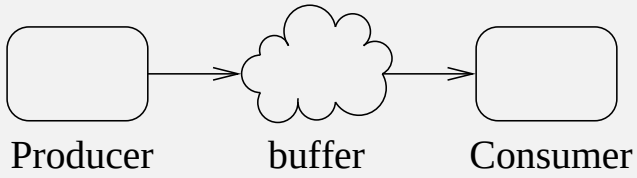
And it would require the consumer to process data at the same rate as the producer produces it (as in a pipeline)

Exercise Compare with MPI

Parallel Algorithms

Producers/Consumers

So, typically, there is a *buffer* between them



Buffered Producer/Consumer

This is just some area of memory in a shared memory system; or a message queue for a distributed memory system

Parallel Algorithms

Producers/Consumers

The advantage is that we can *decouple* the producer and consumer

- each can work at their own rate, until the buffer fills or empties
- there is less synchronisation, thus less waiting around
- the producer and consumer are now working *asynchronously*: not synchronising on every message

Parallel Algorithms

Producers/Consumers

When the producer produces data, it writes it into the next free place in the buffer

Unless the buffer is full, when the producer must wait until a place becomes free by the consumer reading some data

Symmetrically, when the consumer want to consume data, it reads it from the next position in the buffer

Unless the buffer is empty, when the consumer must wait until some data arrives by the producer writing it

So there *is* synchronisation, but only when necessary, dictated by the size of the buffer

We need to see how to manage this synchronisation

Parallel Algorithms

Producers/Consumers

For example, a buffer of size 1, using two semaphores, called `empty` and `full`

```
              empty = make_semaphore(1);
               full = make_semaphore(0);
producer() {                  consumer() {
  produce data                  wait(full);
  wait(empty);                  take from buffer
  insert in buffer              signal(empty);
  signal(full);                 consume data
}                             }
```

Parallel Algorithms

Producers/Consumers

A simple extension to a buffer of size *n* is to use counting semaphores `data` and `free` with `free` initialised to *n*

```
    free = make_counting_semaphore(n);
    data = make_counting_semaphore(0);
producer() {                  consumer() {
  produce data                  wait(data);
  wait(free);                   remove from buffer
  append to buffer              signal(free);
  signal(data);                 consume data
}                             }
```

Parallel Algorithms

Producers/Consumers

But this works only if appending to and reading from the buffer are independent operations

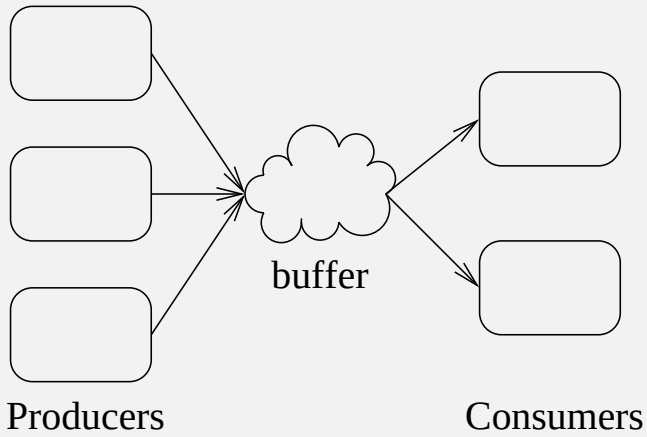
In this code as written, the producer and consumer might be acting simultaneously on the buffer: we need to make sure the update does not have a data race

So, for example, might want a lock on the buffer, or make sure the buffer can otherwise safely support a simultaneous read and write (e.g., for a hash table this might be difficult)

Parallel Algorithms

Producers/Consumers

And things get more interesting when there is more than more producer, or more than one consumer



Multiple Produces/Consumers

Parallel Algorithms

Producers/Consumers

Now concurrent access to the buffer is really a problem

We might use a lock to do this

```
                free = make_semaphore(1);
                data = make_semaphore(0);
               buffy =  make_lock();
producer() {                    consumer() {
  produce data                    wait(data);
  wait(free);                     get_lock(buffy);
  get_lock(buffy);                take from buffer
  insert in buffer                free_lock(buffy)
  free_lock(buffy);               signal(free);
  signal(data);                   consume data
}                               }
```

Parallel Algorithms

Producers/Consumers

Exercise Prove that this cannot deadlock

Using one lock means that we cannot insert into the buffer at the same time as reading from it

This is often an unnecessary restriction, e.g., the buffer is an area of memory where we can read one element at the same time as writing a different one

Again, this might not be possible if the buffer was some more sophisticated kind of datastructure

Parallel Algorithms

Producers/Consumers

So, often we have two locks, one for the insert position and one for the remove position

And we have to be careful when they coincide, e.g., when the buffer is full or empty

Parallel Algorithms

Producers/Consumers

Implementations of buffers tend to be either

- linked lists (unbounded size)
- fixed arrays, used circularly

In any case, the buffers are usually actually *queues*, namely first in first out

Parallel Algorithms

Producers/Consumers

More advanced use of queues is possible

If you have just **one** producer, you can implement a *lockless* insert into the queue: namely the insert end does not need a lock (or other synchronisation mechanism)

The “gap” between testing for a space in the buffer and inserting is not a problem as no-one else is inserting data

You still have to think carefully about the interaction of this with the removal of data

Parallel Algorithms

Producers/Consumers

Symmetrically, if there is just **one** consumer, it is possible to have a lockless read

These require *extremely* careful programming, but can be useful in reducing overheads

Consequently, it is possible to implement a single producer/single consumer entirely lock-free

Exercise Find out how to do this (it involves memory barriers!)

Parallel Algorithms

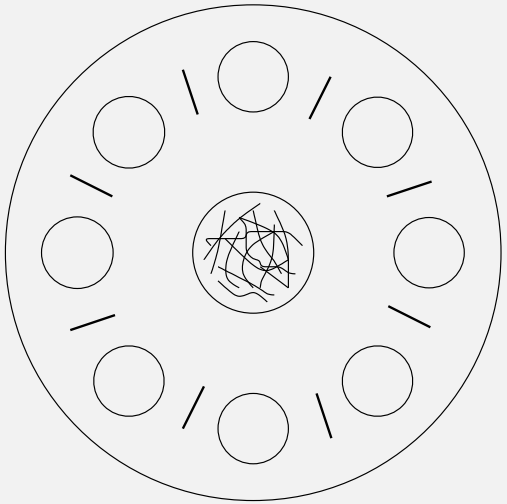
Dining Philosophers

Another old and famous problem: the *Dining Philosophers*

Often used to illustrate problems of resource contention in operating systems, it can be used to help understand problems in concurrency, too

Parallel Algorithms

Dining Philosophers



Dining Philosophers

We have five philosophers wanting to eat spaghetti, but there are only five chopsticks to go round

Parallel Algorithms

Dining Philosophers

The life of a philosopher is

- think
- sit
- take chopsticks
- eat
- drop chopsticks
- leave
- repeat

Parallel Algorithms

Dining Philosophers

A philosopher sits at any free position, but can only use the two neighbouring chopsticks

They require two chopsticks to be able to eat!

If a chopstick is already in use, the philosopher must wait until it is free

Parallel Algorithms

Dining Philosophers

This problem shows

- mutual exclusion of the chopsticks
- deadlock if all the philosophers sit down simultaneously and grab the left chopstick: they will all then have to wait on their right chopstick
- starvation, as four of the philosophers might conspire to keep out the fifth

Parallel Algorithms

Dining Philosophers

Mutual exclusion of the chopsticks is easily provided by having a mutex for each chopstick

```
lock chopstick[5];
```

Then philosopher *i* grabbing and dropping the chopsticks is

```
lock(chopstick[i]);
lock(chopstick[(i+1)%5]);
eat();
unlock(chopstick[(i+1)%5]);
unlock(chopstick[i]);
```

Parallel Algorithms

Dining Philosophers

But, as we know, this can deadlock if all philosophers grab (say) the left chopstick simultaneously

Simply alternating left-then-right grab with right-then-left grab won't fix it; neither will picking a random chopstick first

The classical solution is to have a counting semaphore, initialised to 4, to limit the number of simultaneously sitting philosophers

Parallel Algorithms

Dining Philosophers

```
lock chopstick[5];
place = make_counting_semaphore(4);
...
philosopher(int i) {
  while (1) {
    think();
    wait(place);
    lock(chopstick[i]);
    lock(chopstick[(i+1)%5]);
    eat();
    unlock(chopstick[(i+1)%5]);
    unlock(chopstick[i]);}
    signal(place);
  }
}
```

Parallel Algorithms

Dining Philosophers

Exercise Prove this cannot deadlock

Exercise Think about fixing starvation

Exercise Solve the Dining Philosophers using monitors

Exercise Solve the Dining Philosophers using GCD

Parallel Algorithms

Sorting

We now turn to some concrete examples of parallel algorithms, beginning with sorting

Clearly, a merge sort is amenable to divide and conquer

- divide data into two equal chunks
- recursively merge sort each half in parallel
- merge the two sorted lists together

Parallel Algorithms

Sorting

For example, $n = 8$. The division is trivial, so we concentrate on the merge:

								t	p
3	1	4	1	5	9	2	6		
1	3	1	4	5	9	2	6	**2**	4
1	1	3	4	2	5	6	9	**4**	2
1	1	2	3	4	5	6	9	**8**	1
							Total:	**14**	

t is the time to merge sort that line; p the number of processors

Parallel Algorithms

Sorting

It is easy to calculate the time this takes on n values (PRAM: assume we have enough processors and ignore communications costs)

- The last merge takes time n
- The step before takes time $n/2$ (twice, in parallel)
- The step before takes time $n/4$ (four times, in parallel)
- etc.

Total time is $T(n) = n + n/2 + n/4 + \cdots + 2 = 2n - 2 = O(n)$

Parallel Algorithms

Sorting

The sequential merge sort takes time $O(n \log n)$, giving a speedup of

$$S = O(n \log n / n) = O(\log n)$$

using $O(n)$ processors ($n/2$ in this case)

This increases with n, but not very quickly, and is a lot smaller than n

It uses $O(n)$ processors, for an efficiency of

$$E = O(\log n / n)$$

The efficiency drops to 0 as n gets large

Parallel Algorithms

Sorting

If we have just p processors, this becomes

$$T_p(n) = O\left(n + \frac{n}{p}\log\frac{n}{p}\right)$$

as we have sequential merge sorts of p chunks of size n/p, plus $(n/p)O(p) = O(n)$ steps to merge them in parallel

We get

$$S_p(n) \approx p$$
$$E_p(n) \approx 1$$

for large n and fixed p

Exercise Work this example through for yourself

Parallel Algorithms

Sorting

So: for a fixed number of processors we can get good a speedup, but if we let the number of processors get large our relative speedup gets quite poor

Seems counterintuitive until you think about it, but it means we have to have lots of data relative to the number of processors to get a good speedup

Alternatively: if we have a lot of processors, most of them are going to be idle most of the time: we only use all of them in the first step; and even fewer in subsequent steps

Exercise Think about this result in the context of Amdahl and Gustafson

Parallel Algorithms

Sorting

The most famous sequential sort (after bubble) is *quicksort*

Similar to mergesort, in that it is a divide and conquer method, but different in how it divides

- pick a value, the *pivot*, from the data
- partition the data into two chunks: values bigger than the pivot; values less than the pivot
- recursively quicksort the two chunks
- return the sorted lower chunk; the pivot; the sorted higher chunk

Parallel Algorithms

Sorting

The partition phase is a bit fiddly to parallelise, but the recursive sorts are clearly parallelisable

It works well with manager/worker: as each sub-partition is created it becomes a new task

Also, the tasks are entirely independent with no communications between them once created

Though we do need to join the sorted partitions back together

Parallel Algorithms

Sorting

Parallel quicksort is very similar in time complexity to mergesort: it takes time $O(n)$ with $O(n)$ processors in the average case

And time $O(n + (n/p)\log(n/p))$ with p processors

As usual, quicksort relies on decent pivots: this translates directly to the need to get good load balancing of the sub-tasks

Parallel Algorithms

Sorting

Heapsort: another $O(n \log n)$ (sequential) sort, is valued as it has very stable behaviour: no bad cases

But there doesn't seem to be a good way of parallelising it as the swaps in the heap creations and destructions need to pass in unpredictable ways through the entire dataset

Parallel Algorithms

Sorting

Bucket sort parallelises well: this splits the data into several buckets, then recursively sorts the buckets

Example. Sorting CDs. Have one bucket per letter of the alphabet. It is quick to put CDs in the correct buckets

Clearly, an extension of the merge sort, it has very similar properties

Parallel Algorithms

Sorting

Parallel sorting algorithms exist that take parallel time $O(\log n)$, but require $O(n^2 / \log n)$ processors: very inefficient

Other sorts exist that take time $O(\log n)$ time and $O(n)$ processors: sounds better?

Some of these you need to be sorting upwards of 10^{22} items to be faster than simpler sorts with apparently worse complexities, like the *bitonic* sort, with time $O(\log^2 n)$

Parallel Algorithms

Sorting

The bitonic sort, a divide and conquer method somewhat related to merge sort and shell sort, takes time $O(\log^2 n)$ on $O(n)$ processors

It takes $O(n \log^2 n)$ sequentially, so having a speedup of $O(n)$

This sounds good, until you realise this is a parallelisation of a slightly sub-optimal sequential sort

Comparing against a $O(n \log n)$ fast sort, we see bitonic has speedup $O(n/\log n)$; still not too bad

But the important thing is that it is practical for realistic sizes of n

Exercise Go and read up on bitonic sort

Parallel Algorithms

Sorting

And there are many other sorts

The literature for parallel sorts is huge, as it is a problem that is easy to understand, but hard to solve

Particularly when you start to factor communications costs into your time complexities

Parallel Algorithms

Sorting

Exercise It has been claimed that `MapReduce` can sort "a petabyte of data in a few hours". Find out about how it does this

Exercise Related to sorting is the problem of finding the maximum value in a dataset. Discuss how this might be parallelised and its time complexity

Exercise Then find the middle value in a dataset

Parallel Algorithms

Searching

The other classical problem is searching

This is very datastructure dependent, but can parallelise very well

For example, if the data are spread over many machines, searching for an item is as simple as getting each machine to search its chunk

When any machine finds the item, they can all stop

Or, if multiple results are wanted, there can be a reduce step

Parallel Algorithms

Searching

If the data is distributed sensibly over p processors, the chunks will be of size n/p and take n/p time to search for a naïve linear search

Thus parallel searching can give perfect speedup $n/(n/p) = p$

But linear search is far from a good sequential search

Again, we get a good speedup since we start from a poor place

Parallel Algorithms

Searching

Searching in a tree takes time $O(\log n)$, so if we can perfectly distribute sub-trees across p processors, we can search them in parallel time $O(\log(n/p))$ for a speedup $O(\log n/\log(n/p))$

Sounds good? Well, consider the speedup for large n:

$$\begin{aligned} O(\log n/\log(n/p)) &= O(\log n/(\log n - \log p)) \\ &= O(1/(1 - \log p/\log n)) \\ &\to 1 \text{ as } n \to \infty \end{aligned}$$

Here the problem is that tree search is so good that the benefit you get from spreading it across p processors is small, and gets smaller as the dataset increases in size

Parallel Algorithms

Searching

And these algorithms rely on everything being nice and uniform and randomly accessible and ignoring communications costs

For example, if the searches cluster around the data on a single machine, we could write a sequential search that takes advantage of that fact, and our parallel search would not be much faster

Parallel Algorithms

Searching

Also, the datastructure must be able to be evenly spread

Lists and trees, that have restrictions on the order you access their elements, are harder to access in this random manner

Of course, Google does this in a big way, using MapReduce, showing that searching petabytes of data can be done in fractions of a second

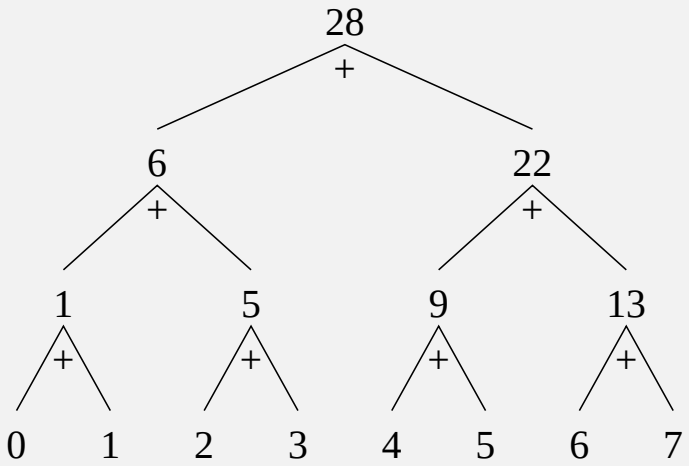
Again, we find that parallelism allows us to go bigger, rather than faster

Parallel Algorithms

Reduction

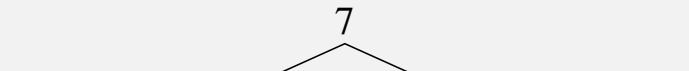
Next: parallel reduction

Reduction has a natural parallelisation using a tree



Tree reduction sum

Reducing a list of values using summation (read bottom up)



Parallel Algorithms

Reduction

This takes $O(\log n)$ steps to reduce n values, using $O(n)$ processors

Sequential time: $n - 1$ operations, giving speedup

$$S = O(n/\log n) \text{ using } O(n) \text{ processors}$$

This is not much less than n, as $\log n$ grows only slowly with n

Parallel Algorithms

Reduction

Efficiency

$$E = O(1/\log n)$$

which slowly drops as n increases

Parallel Algorithms

Reduction

For p processors, divide the data into p chunks of size n/p

Time to reduce a chunk (sequential): $O(n/p)$
Time to reduce the chunks: $O(\log p)$

Total

$$O\left(\frac{n}{p} + \log p\right)$$

Parallel Algorithms

Reduction

Speedup

$$S_p = \frac{n}{n/p + \log p} = \frac{p}{1 + (p \log p)/n}$$

which approaches p as n gets large

Likewise, the efficiency approaches 1 for large n

Similar to previous examples, if you allow yourself an indefinite number of processors, the speedup will be greater, but at a high cost, i.e., low efficiency

For a fixed number of processors, you get a fixed bound on the speedup, but you will be using the hardware very efficiently as the dataset get large

Parallel Algorithms

Reduction

There are a couple of issues, however

In real implementations we need to worry about the cost of data movement between processors: reduction inherently needs to move data around

Probably a small cost for a shared memory system, but it can easily be much larger than the cost of the reduction operation if you are not careful

So parallel reduction on, say, a distributed memory machine, is only worthwhile for large datasets

Or a very costly reduction operation

This is grain size, again

Parallel Algorithms

Reduction

The other issue is about reduction in general, not just in parallel. Reduction relies on the associativity of the reduction operation

Reduce the list $(1, 2, 3, 4)$ using $-$

Do we mean

$$((1 - 2) - 3) - 4 = -8$$

a *left* reduction

Or

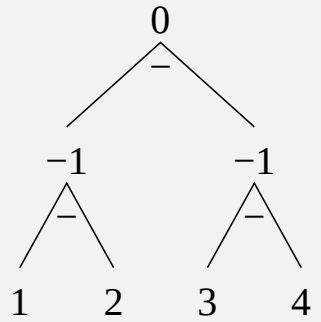
$$1 - (2 - (3 - 4)) = -2$$

a *right* reduction?

Parallel Algorithms

Reduction

And a tree reduction will give



Tree Reduction

Or something else entirely depending on where the data ended up in the tree

Parallel Algorithms

Reduction

The simple answer is not to do reductions using non-associative operations, even sequentially

However, there are many useful reduction operations, including $+$, $*$, max, min, left$(a, b) = a$ and so on

Parallel Algorithms

Reduction

Reduction appears as an operation in many languages, e.g., JavaScript `array.reduce(op)` to reduce the `array` with the `op`:

```
((array[0] op array[1]) op array[2]) op ...
```

Thus amenable to automatic parallelisation, if the operation is associative and independent of the array (e.g., not if the `op` updates the array)

Parallel Algorithms

Prefix Scan

Closely related to reduction is the *prefix scan*: $(1, 2, 3, 4)$ with $+$ returns

$$(1, 3, 6, 10)$$

So: `(array[0], array[0] op array[1], array[0] op array[1] op array[2], ...)`

The partial reductions, usually left associated

Parallel Algorithms

Prefix Scan

This can also be done in $O(\log n)$ steps (on n processors)

Even though it seems you need to compute $1 + 2$ before computing $1 + 2 + 3$ before computing $1 + 2 + 3 + 4$, thus serialising the whole thing

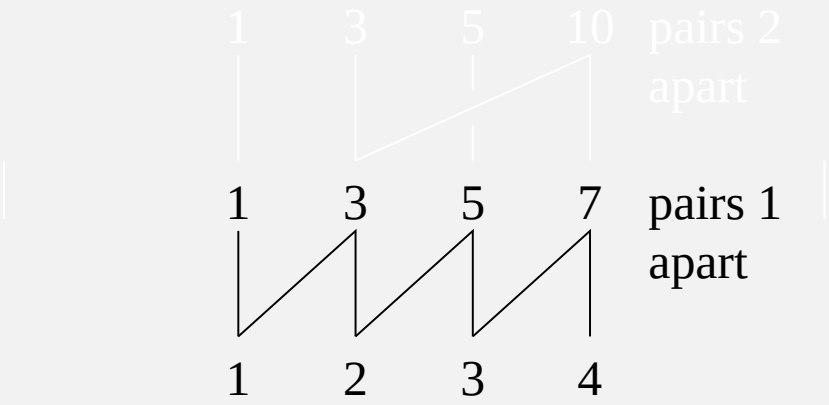
But this is sequential thinking!

For example, you can compute $3 + 4$ at the same time as $1 + 2$; and then $(1 + 2) + 3$ in parallel with $(1 + 2) + (3 + 4)$

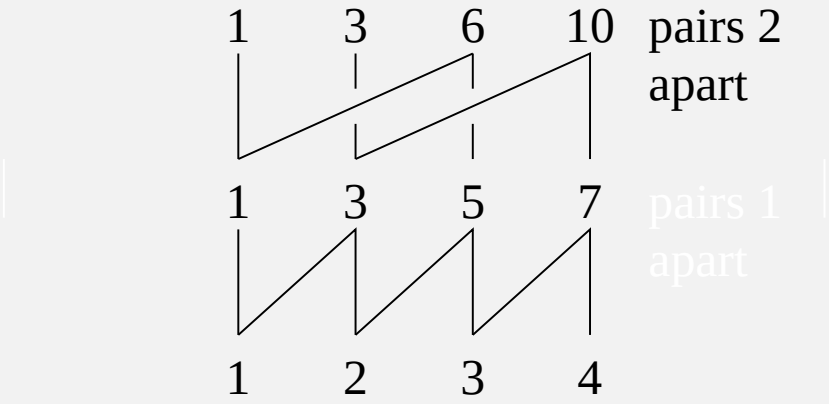
We can proceed in a tree-like sequence of combination of pairs of values

Parallel Algorithms

Prefix Scan



Prefix Scan 1 apart



Prefix Scan 2 apart

Parallel Algorithms

Prefix Scan

First step is to sum `array[i] = array[i] + array[i-1]` in parallel

Then double the distances:
`array[i] = array[i] + array[i-2]`

Then double the distances:
`array[i] = array[i] + array[i-4]`

And so on, for $\log n$ steps on $O(n)$ processors: this gives us all the prefix sums, including the total reduction as the last element

Parallel Algorithms

Prefix Scan

When limited to p processors we can produce a scan in time

$$O\left(\frac{n}{p} + \log p\right)$$

Scan has the same issues as reduce, namely data travel and associativity

Parallel Algorithms

Prefix Scan

Scan appears to give us more answers than reduce for the same amount of work!

It's not: for a start, reduce uses at most $n/2$ processors, while scan uses up to $n - 1$

Parallel Algorithms

Prefix Scan

But more importantly, reduce halves the number of active processors in each step, while scan uses more processors more of the time. It uses $n - 2^r$ active processors in step r, so it *ends* with about $n/2$ active processors

They both complete in the same amount of time so they have the same speedup, but scan is more efficient

Meaning scan uses more hardware more of the time (and therefore takes more energy)

We can see that reduce has quite a lot of slack in parallel!

Parallel Algorithms

Prefix Scan

Note that both scan and reduce work well on a SIMD architecture

They work on distributed memory, too, but we have to watch the cost of the messaging

MPI includes several scan operations including
`MPI_MAX`, `MPI_MIN`, `MP_SUM`, `MPI_PROD`, `MPI_LAND` (logical AND), `MPI_LOR` (logical OR)
amongst others

Exercise Write a parallel prefix scan in OpenMP

Exercise In fact there is a better, work efficient, more complicated algorithm that only needs $n/2$ processors. Look it up

Parallel Algorithms

FFT

The Fast Fourier Transform (FFT) is one of the basic algorithms in CS, known by everybody who knows anything about CS

The Discrete Fourier Transform (DFT) takes a sequence of n (complex) numbers and returns a sequence of n numbers

If the input numbers represent a signal, the DFT values represent the constituent frequencies of that signal

$$y_k = \sum_{j=0}^{n-1} x_j e^{-2\pi ijk/n}, \text{ for } 0 \leq k < n$$

The n values x_i are input; the n values y_i are output

Parallel Algorithms

FFT

This has two obvious elements of parallelism:

- each y_k can be computed independently, for a n-way parallelism
- each summation can be done as a tree, for a $\log n$-way parallelism
- taking total time $O(\log n)$ on $O(n^2)$ processors

But, instead let us look at a sequential divide and conquer version

Parallel Algorithms

FFT

This sum can be computed as presented: summing n values for each of n values y_k, thus taking time $O(n^2)$

However, if n is even, then we get a nice recursive presentation by splitting the sum into evens and odds

Parallel Algorithms

FFT

$$
\begin{aligned}
y_k &= \sum_{j=0}^{n-1} x_j e^{-2\pi ijk/n} \\
&= \sum_{j=0}^{n/2-1} x_{2j} e^{-2\pi i(2j)k/n} + \sum_{j=0}^{n/2-1} x_{2j+1} e^{-2\pi i(2j+1)k/n} \\
&= \sum_{j=0}^{n/2-1} x_{2j} e^{-2\pi ijk/(n/2)} + e^{-2\pi ik/n} \sum_{j=0}^{n/2-1} x_{2j+1} e^{-2\pi ijk/(n/2)}
\end{aligned}
$$

Decomposition of Fourier Transform

This is just two half-size DFTs

Parallel Algorithms

FFT

For n a power of 2 we can repeat recursively, leading to the *Fast Fourier Transform*, a way to implement the DFT

In fact, the FFT is an unwinding of the recursion into an iteration that runs slightly faster, but is harder to understand

The FFT takes sequential time $O(n \log n)$, which is a huge improvement over $O(n^2)$; e.g., for $n = 1{,}000{,}000$, this is about 20,000,000 against 1,000,000,000,000

But, for our purposes, we can see this as a simple divide and conquer, thus easily parallelisable

Parallel Algorithms

FFT

The parallelisation of the FFT works in a way very similar to what we have seen before and has complexity $O(\log n)$ on $O(n)$ processors, and $O(\log p + (n/p)\log(n/p))$ on p processors

As the FFT is such an important algorithm, much has been written about it and its parallel variants, in particular matching it to the various kinds of hardware (SIMD, pipeline, shared memory, etc.)

Parallel Algorithms

And So On

There are very many other parallel algorithms: just think of the large literature on sequential algorithms that exists

We have just looked at a couple, but everything that you have done in the past sequentially will probably have a parallel counterpart

Some algorithms will map best to shared memory, some distributed, some SIMD, and so on

Some will be sensitive to the topology of the architecture (full connect, torus, etc.), others work well regardless

Still more will not work well in parallel at all

Exercise Look some up!

Topics

We now look at a few topics in parallel computing

Each year this unit is given may cover different topics so don't be too worried if past exam papers ask questions on things that were not covered this year

Hardware

We have seen that there are many kinds of parallelism

But there has been hardware support for parallelism for much longer than you might think

Even in sequential CPUs!

Hardware

Bit level

Recall from the 1st Year Architecture unit about adders: adding together two binary words

Serial Adders work one bit at a time, propagating the carry up the words as they do

Simple hardware, simple to implement

Parallel Adders work on all the bits in parallel

More complex and expensive hardware, but faster

A simple example, but this illustrates how parallelism trades complexity for speed

Hardware

Pipelines

Again from Architecture: instructions are executed faster by using a pipeline

This is parallelism by overlapping the fetch$\rightarrow$decode$\rightarrow$fetch arguments$\rightarrow$execute$\rightarrow$store result cycle

Hardware

fetch→decode→args→exec→store→fetch→decode→args
→exec→store→fetch→decode→args→exec→store→fetch
→decode→args→exec→store...

becomes

fetch→decode→ args → exec → store
fetch →decode→ args → exec →store
fetch →decode→ args →exec→store
fetch →decode→ args →exec→store
...

Again, more complexity for speed

It also shows how simple CPU clock speed is *not* a good indicator of speed of processing

A pipelined CPU will produce results faster than a non-pipelined CPU of the same clock speed

Hardware

Coprocessors

Early chips were too small to fit everything on them

So some operations were offloaded to a separate chip, a *coprocessor*

At one point, a popular design was to put floating point operations on a coprocessor and only have integer arithmetic on the main processor chip

The coprocessor was specialised for floating point and could do little else

This allowed a weak form of parallelism: ship an operation (say a square root) off to the coprocessor, and while it is chewing on that, the main processor can carry on with something else in parallel

Hardware

Coprocessors

Floating point eventually migrated onto the main chip (using lots of transistors!), but coprocessors are still hugely popular

Graphics cards (GPUs) are coprocessors, originally specialised to pixel crunching

And now they are commonly used as *general purpose GPUs* (GPGPU) and are turning out to be important in highly parallel computation

We shall return to GPGPUs

Exercise Read about *Tensor Processing Units* (TPUs)

Hardware

Superscalar

To employ those extra transistors, engineers starting putting multiple arithmetic units on the chip

For example, two add units

The processor can now do two adds at the same time

Simultaneous execution of whole instructions is called *superscalar*

Pipelining is parallel execution of *parts* of the instruction cycle

Hardware

For example, the two adds in

```
x1 = y1 + z1;
x2 = y2 + z2;
```

can be done at the same time

However, the two adds in

```
x1 = y1 + z1;
x2 = x1 + z2;
```

cannot be done at the same time

The CPU needs to sort out the dependencies to determine if it can do simultaneous multiple operations

Hardware

Out of Order

This can be improved with careful *instruction scheduling* by the processor, to let it do *out of order execution*

For example, the code

```
x1 = y1 + z1;
a1 = x1*y1;
x2 = y2 + z2;
```

is equivalent in results to

```
x1 = y1 + z1;
x2 = y2 + z2;
a1 = x1*y1;
```

but on a CPU with two add units the latter can do the two adds in parallel

Hardware

Out of Order

A processor that does out of order execution will scan the instruction stream, analyse the upcoming operations and their dependencies, and reorder them suitably

Implementing this in the hardware uses a lots of transistors, and so keeps the engineers happy

Compiler writers can help somewhat by generating machine code that is easier for the hardware to analyse

But, mostly, this is a hardware feature

Hardware

Out of Order

But we have already seen how out of order execution can break parallel code if we are not careful

Hardware

Out of Order

Hard Exercise (come back to this later). Suppose we have initial values `x = 0` and `y = 1`. Two parallel threads on hardware that does out of order execution:

Thread 1	Thread 2
`y = 3;`	`if (x == 1) {`
`x = 1;`	`  y = 2*y;`
	`}`

What are the possible final values of `y`?

Example taken from the Rust website; also see
`https://en.wikipedia.org/wiki/Memory_ordering`

Hardware

Hyperthreading

The next stage is to duplicate the state-bearing parts of the processor, namely the program counter, the registers and other related stuff

This allows two (generally two, sometimes more) simultaneous threads (streams of instructions) to share the available hardware

There will be some conflicts between the threads if they both try to use a computational unit (say a division) when there is only one unit of that type on the chip

In that case one thread will have to pause and wait

Hardware

The main argument for hyperthreading is that if one hyperthread has to wait for something (e.g., a memory access) the other can run and keep the core busy

The idea of having more threads of execution than hardware so that there is always a thread ready to run becomes very important later

Hyperthreading gives the illusion of a multicore system, but is not truly multicore

The amount of repetition in the architecture will imply some limits on how effective this is and how much parallelism can be gained, as will the pattern of memory accesses by the code

Hardware

Some say that two hyperthreads are worth about 1.5 cores, due to the amount of interference between the threads

Downsides are that the hyperthreads can fight over the core's cache memory

For some tasks hyperthreading can reduce overall performance

And there are security issues where information can leak (via the cache) from one hyperthread to its pair

Most High Performance systems turn off hyperthreading (a bigger share of the memory cache is more important than more threads)

Hardware

SWAR

Next: the idea of SIMD/vector processing has been adopted in a small way in the instruction sets of some processors

It arose from multimedia processing, graphics in particular

Some operations (e.g., computing pixel colours) are data parallel

Now we can regard a 64 bit register as

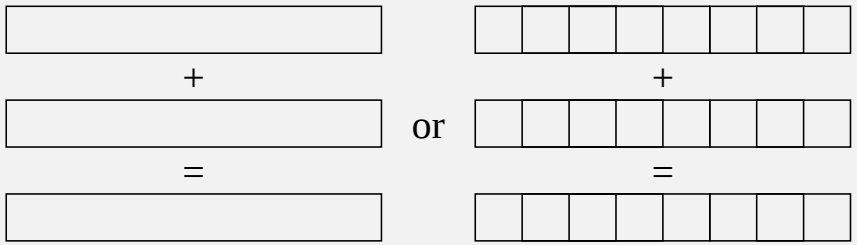
- a 64 bit register
- two 32 bit registers
- four 16 bit registers
- eight 8 bit registers

Hardware

SWAR

An instruction is provided to (for example) add together eight 8 bit values in those registers in parallel

Another to add four 16 bit values in parallel, etc.



SIMD Within A Word

Hardware

SWAR

This is *SIMD within a register* (SWAR)

We are treating the register as a (small) vector processor

This was found to be very effective for data parallel graphics processing

Intel provide these instructions in their MMX (Multi Media Extensions), SSE (Streaming SIMD Extensions), SSE2, SSE3, SS4, AVX (Advanced Vector Extensions, 128 bit registers), AVX2 (256 bit registers) extensions

Similarly others from other manufacturers (AMD, Arm, etc.)

Hardware

SWAR

Now, most code is written in a sequential fashion, e.g., looping over 8 values rather than code to add 8 values simultaneously

In fact, few languages support SWAR operations directly, so there has to be some mechanism for getting to SWAR from conventional code

The process of converting sequential operations to SWAR is called *vectorisation*

Hardware

SWAR

We need compiler support to generate these SWAR instructions: it needs to spot that rather than generating eight instructions to add eight 8-bit numbers, it should generate one instruction to add them in SWAR

Compilers have always been far behind hardware: an architecture might provide an eight-way multiply instruction, but that is *only useful if you can get a compiler to generate code to use it*

Or get the programmer to writer the assembler by hand

For a compiler spotting that a loop can be converted into SWAR vector instructions is very hard

Hardware

SWAR

For example, the multiplies in the code

```
char x[20], y[20];
for (i = 0; i < 20; i++) {
  y[i] = x[i]*x[i];
}
```

might be compiled as *three* $(8 + 8 + 4)$ 8-way SWAR multiply instructions

Plus a bunch of other stuff to get the values in and out of the right places in the register

Hardware

SWAR

Making good compilers is harder than you think and has been a major drag on the effective use of modern hardware

A lot of code to use these kinds of instructions still has to be written by hand, in assembler

Hardware

SWAR

In procedural code, we tend to write loops: the compiler would have to analyse it carefully to determine if SWAR would be useful (e.g., no value depends on an earlier value in the loop)

In contrast, in the functional style we write code like “do this operation on these data” (map), which is much easier to analyse as the operation is explicitly separate from the iteration

Hardware

SWAR

Exercise Think about the code

```
char x[], y[];
for (i = 0; i < n; i++) {
  y[i] = x[i]*x[i];
}
```

where the loop limit is variable

Exercise Then think about the functional version

```
y = x.map(square);
```

Hardware

VLIW

The transition of CPUs from *complex instruction set computer* (CISC) to *reduced instruction set computer* (RISC) architectures was based on advances in compiler technology

The idea was to move complexity out of the hardware and into the software

Rather than using complicated instructions poorly, we use simple instructions effectively: by streamlining the instruction set we can run things faster

This is strongly reliant on the compiler being good enough to understand and exploit the details of the RISC architecture

But this is easier than a compiler trying to make best use of a complicated CISC architecture

Hardware

VLIW

The same idea was touted for the *very long instruction word* (VLIW)

Design a processor with many repeated arithmetic units—lots of add units, lots of multiply units and so on

Have instructions that are *very long*, e.g., 128 bits or more

The instructions are composites of the simple operations, e.g., two adds, a subtract and a multiply could be bundled together in a single instruction

Hardware

VLIW

The compiler composes these instructions and makes sure there are no nasty interactions between the sub-instructions, e.g., none of the inputs to the sub-instructions are the outputs of any others of the sub-instructions

The compiler does the hard work of sorting out interactions, leaving the hardware to blast on at full speed without checking or doing any reordering

The compiler is promising to the hardware that nothing bad is going to happen if the hardware blindly executes the instructions as given

Hardware

VLIW

Moreover, the chip uses less energy as it does not have the silicon to do instruction dependency analysis and reordering and the like

The analysis and reordering was done by the compiler

This appeared in the Bulldog compiler (early 1980s) and the Multiflow computer (late 1980s)

It didn't turn out to be terribly practical or popular

Compilers were not sufficiently clever to untangle enough instruction dependencies to get good hardware utilisation

Hardware

VLIW

VLIW was briefly revived by Intel in their *Itanium* processor (2001)

They called it *Explicitly Parallel Instruction Computing* (EPIC), a limited form of VLIW

It, too has flopped

Possibly due to their classic x86 chips being too entrenched, but also their compiler was never quite up to the job

Hardware

VLIW

It still pops up here and there: some AMD Radeon graphics chips have a VLIW architecture, though their newer architectures reverted to more traditional RISC

VLIW may well re-emerge in the future when compilers have progressed further: though more likely it will be overtaken by other kinds of hardware parallelism

Hardware

VLIW

Exercise Think about the

```
char x[], y[];
for (i = 0; i < n; i++) {
  y[i] = x[i]*x[i];
}
```

example with VLIW

Hardware

Multicore

Next we have full replication of arithmetic units, control and registers: true multicore

Two or more full CPUs on the same chip

Often regarded as the first emergence of hardware parallelism

But, as we have seen, it's not

Hardware

Early multiprocessor machines were unicore chips side by side on the same motherboard

Modern multicore processors, having cores on the same chip, can share things like on-chip cache memory and other chip infrastructure

Also there is faster inter-core data transfer: no need to go off-chip. Off-chip transfers run at the bus speed, much slower than the chip speed

Hardware

Multicore

Large machines tend to be multiple multicores: e.g., two 24-core chips on a motherboard; a total of 48 threads of execution

Or 96 if 2-way hyperthreading is enabled

This is slightly *asymmetric*: some cores are a little “closer” to each other than the others

Hardware

All of the above

These things are not mutually exclusive

A typical large installation these days is a CLUMP

- a cluster
- of multiple processors
- each having multiple cores
- which might have hyperthreads
- and SWAR instructions
- on a pipelined architecture
- with parallel instructions
- sometimes with a coprocessor or two on the side

It is very hard to make efficient use of all that!

More on Threads

We return to the idea of threads

POSIX threads is just one example of many different approaches to threads

And just one example of the many different *kinds* of threads

TBB

We shall look briefly at Threading Building Blocks (TBB) as it contains some interesting ideas

It is a standard C++ template library, needing no specific compiler support

It provides things like concurrent containers and concurrent operations as well as the usual atomics and synchronisations

TBB Concurrent Operations

```
#include <tbb/tbb.h>
#include <iostream>

using namespace tbb;
using namespace std;

void hi(int n) {
  cout << "hello: " << n << endl;
}

int main() {
  parallel_for<int>(0, 10, hi);

  return 0;
}
```

TBB Concurrent Operations

Though you quickly realise you should have written

```
std::mutex m;

void hi(int n) {
   m.lock();
   cout << "hello: " << n << endl;
   m.unlock();
}
```

But not a single `pthread_create` in sight!

TBB Concurrent Containers

Containers are things like vectors, queues and hash tables

You have to take care over concurrent access to these as pushing value to a stack at the same time as another thread is popping a value is an easy route to races

Thus TBB provides safe datastructures that get the details right (we hope!)

TBB Work Stealing

The interesting thing about TBB is that is uses *work stealing* to manage parallelism

In something like a `parallel_for` there are a lot of tasks to be scheduled across the available threads

Each thread has a queue of tasks that are ready to be run (actually a *double ended queue*, or *deque*)

When a new task is spawned it is pushed onto the end of the spawning thread's queue

("Spawn" is the terminology for creating a new task)

TBB Work Stealing

When a thread completes a task it pops a task off the **end** of its queue and runs that next

That is, the most *recently* created task for that thread

If its queue is empty, the thread *steals* a task off the **start** of another thread's queue and runs that

That is, the *oldest* created task for that thread

Thus keeping all threads busy as long as there are tasks to do

TBB Work Stealing

Note that pushing and popping a task off your own queue is a relatively cheap operation, so the overhead is kept small for this case, which you hope is the common case

In other words, when there is no opportunity for more parallelism as every thread is already busy doing its own tasks, the overhead is minimal

The overhead of stealing a task is greater, but this only happens when a thread would otherwise be idle and has time to spare

TBB Work Stealing

So: if a thread has work to do it does its most recently created task first, thus preserving locality of execution: the next task executed is “nearest” to one just finished

And if a thread has nothing to do it takes the oldest task off another thread, thus disrupting its locality as little as possible

Exercise It’s *much* more complicated than this, of course. Read about the details

Exercise Work though how work stealing might execute the `parallel_for` example

TBB

Benefits of TBB:

- easy-to-write parallelism (for a good C++ programmer)
- is very flexible and extensible (e.g., `parallel_for` works for any type that you can iterate over)
- purely a library, so you can use a standard compiler
- and is easy to update with new versions of the library
- it provides sophisticated constructs like pipelines and general graph parallelism
- contains a large number of features

TBB

Drawbacks:

- the code needs some reasonably advanced C++ constructs (e.g., functors) get the most benefit
- little checking on the correctness of your use of the constructs: it provides *mechanism* but no *analysis*
- it is tied to C++
- and thus not easily interoperable with other languages
- contains a large number of features

Exercise Read about the large number of other features that TBB provides, particularly ranges for load balancing

Cilk Plus

Cilk Plus also has a task-based view of computation (like TBB), rather than thread based

This means the programmer thinks about what tasks need to be done, and Cilk Plus thinks about the best way of assigning those tasks to threads

It targets roughly the same area as OpenMP

And similar to OpenMP, the number of threads used and the threading mechanisms are mostly hidden from the programmer

Cilk Plus

```
int fib (int n) {
    if (n < 2) return n;
    int x, y;
    x = cilk_spawn fib(n-1);// fork
    y = fib(n-2);
    cilk_sync;                  // join
    return x+y;
}
```

(from the Cilk Plus website)

Cilk Plus

- Cilk Plus has just three main keywords: `cilk_spawn`, `cilk_sync` and `cilk_for`
- So is much simpler than OpenMP
- And more lightweight to use
- And seemingly less flexible: but Cilk Plus provides other mechanisms for more advanced control
- Ignoring the keywords leaves a valid equivalent sequential C program

A `cilk_for` indicates a parallelisable `for` loop

There is an implicit `cilk_sync` at the exit of every function that contains a spawn

Cilk Plus

Cilk Plus also employs work stealing of tasks, but in a more subtle way than TBB

In the code

```
cilk_spawn fun1();
fun2();
```

the *current* thread actually starts executing `fun1()`

Cilk Plus

In more detail:

- when the current thread reaches the `cilk_spawn` it saves the current continuation (i.e., the point in the code just before the `fun2()`) on its continuation stack
- it then starts executing `fun1()`
- when done with that, it pops the continuation stack and starts executing what it finds there: `fun2()` in this example

Cilk Plus

An idle other thread can steal a continuation and start executing it

Thus leading to the initially surprising behaviour that `fun2()` might get stolen, not `fun1()`

In contrast with TBB, where the current thread pushes `fun1()` and so it is that that can be stolen

TBB implements *child stealing*;
Cilk Plus has *continuation stealing*

Cilk Plus

Manipulating continuations is why Cilk Plus needs compiler support. Child stealing as implemented by TBB is implementable in C++ directly as it is essentially just pushing and popping functions on a queue

The difference is that continuation stealing has better memory use patterns than the child stealing and so tends to give more efficient parallelism

Exercise Child stealing can have unlimited memory use, while continuation stealing does not. Read about this

Cilk Plus

Whatever the relative merits, OpenMP and Thread Building Blocks have wide recognition while Cilk Plus is quite niche

In fact, Intel now has deprecated Cilk Plus in favour of their TBB, which being a purely library-based mechanism is easier to support, despite being potentially worse in runtime behaviour

Exercise Read about the many other parts of Cilk Plus, such as *vector sections*

Exercise Work through how continuation stealing might execute the `parallel_for` example

Exercise Compare Cilk Plus, OpenMP, and TBB

Cilk Plus and OpenMP

Exercise Later versions of OpenMP supports *tasks*, which are quite similar in use to Cilk Plus:

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
#pragma omp task shared(x)
  x = fib(n-1);
  y = fib(n-2);
#pragma omp taskwait
  return x+y;
}
```

Read about tasks, and compare with Cilk Plus

Yet More Threads

We now give, as an alternative view to POSIX, a sketch of how threads are natively supported in a few languages, though this could be argued to be more properly in the “design of a language” part of the unit

First, C++

C++ Threads

While C++ can use POSIX threads it has defined — as part of the language specification — its own threads

Which are often implemented on top of POSIX threads, but are more C++ in the way they are used

The C++ specification replicates the usual primitives, including thread creation, mutexes, condition variables and so on, but tidying things up a bit to make them more ergonomic and C++-like

Described as “a restricted/simplified subset of POSIX functionality”

C++ Threads

```
#include <iostream>
#include <thread>
#include <mutex>
#include <string>

std::mutex mut;

void show(const std::string msg, int *n) {
  std::cout << msg << " ";
  // create a lock guard object on the mutex; ownership of
  // the guard is the lock
  std::lock_guard<std::mutex> lock(mut);
  *n += 1; // protected critical region
}
// lock guard deleted at end of scope by
// normal C++ destructor method; thus releasing lock
```

C++ Threads

```
int main() {
  int m = 0;

  std::thread thr1(show, "hello", &m);
  std::thread thr2(show, "world", &m);

  thr1.join();
  thr2.join();

  std::cout << "\nm = " << m << "\n";

  return 0;
}
```

C++ Threads

Producing

```
hello world
m = 2
```

or

```
world hello
m = 2
```

C++ Threads

C++ threads, while mostly similar to POSIX, are closely tied into the rest of the design of C++, thus certain behaviours are better defined

For example, it is not clear how C++'s exception mechanism interacts with POSIX threads, while C++ threads specify a behaviour

And they are portable even if there is no (or poor) POSIX support, e.g., Windows

C Threads

In a similar way, the C11 standard for C also has some language support for threads, though it is optional and not universally supported, e.g., not supported by MS at the moment

It defines types `thrd_t`, `mtx_t`, `cnd_t` and so on

It is essentially pthreads with everything that might be non-portable across *all* architectures removed

C++ threads are widely used, but C11 threads are not, even though they are supported by `gcc` and `clang`

Perhaps ingrained use of pthreads, or lack of perception of benefit of using C11 threads?

Exercise Read about `threads.h` and `stdatomic.h`

Java Threads

Next: Java. It's all based on objects, of course

There are two basic ways to create threads in Java:

- as an instance of a subclass of the `Thread` class
- by providing a method for the `Runnable` interface

Java Threads

```
public class Hello extends Thread {
    public void run() {
        System.out.println("Hello world!");
    }
    public static void main(String args[]) {
        Hello t = new Hello();
        t.start();
    }
}
```

Your classes need to be subclasses of the Thread class

The initial function is the `run` method, which will be called when we execute `start` inherited from `Thread`

A thread can be created, but won't start running until we invoke its `start` method: sometimes separating creation from execution is useful

Java Threads

This way is somewhat constricting in use, as it requires you to design your classes around the `Thread` class

So Java gives an alternative way by providing a `Runnable` interface, which you can add to your existing classes

Java Threads

```
public class Hello implements Runnable {
    ...
    public void run() {
        System.out.println("Hello world!");
    }
    public static void main(String args[]) {
        Thread t = new Thread(new Hello());
        t.start();
    }
}
```

`Runnable` requires a `run` method

The new instance of our class is passed to the `Thread` constructor, which has a `start` method as before

Java Threads

There are `join` methods on `Thread` that wait for thread completion: `join()` and `join(long ms)` and `join(long ms, int ns)`

Simply returning from `main` waits for threads (actually: non-*daemon* threads)

Explicitly calling `System.exit` does not wait

Java

Java also has higher-level support for parallelism in constructs like *parallel streams* that run concurrently

These fall into the class of “sequential code using parallel operations written by someone else”

Though they still have the problem of being non-trivial to use correctly

Exercise Read about *Akka*, a Scala/Java framework for concurrency based on *actors*

Python

And Python. . .

Python was designed without parallel support, and typical implementations of the Python interpreter are strongly not-parallel

Python supports concurrency, but not parallelism

Python

From the docs:

> *The Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the global interpreter lock or GIL, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.*

Python

So, practically speaking, doing anything in Python is necessarily wrapped by a lock

You can get some benefit from using process-based parallelism (`import multiprocessing`), where each process has its own separate Python interpreter, but this is quite heavyweight

The best approach is to call parallel library code written in C, for example

JavaScript

JavaScript is another language that has single threaded interpreters

Exercise Read about how it uses *Web Workers* to provide parallelism

Go

Go (Golang) has its own kind of threads

Here threads are called *goroutines*, and are very lightweight (minimal creation overhead) and are managed by the Go runtime

Note the management is by the Go runtime, not the OS

The Go runtime gets parallelism by scheduling the goroutines across OS threads

Creating new goroutines is very easy — actually encouraged — and you can create “1000s” of goroutines

And it is OK for them to be short lived

Go

Creating a new goroutine:

```
go fun(x+y, x-y)
```

evaluates the arguments and then creates a new asynchronous goroutine running `fun` with the values of those arguments

Go

However:

- Go provides no particular protection against races; it does provide mutexes and so on, but the programmer must remember to use them (or avoid sharing mutable state)
- the runtime that manages the goroutines is quite complex, so Go is less amenable to small or embedded systems
- Go is a garbage collected language, so has that complexity in the runtime, too, e.g., having to stop *all* threads during a GC

Exercise Find out about the current state of Go with regards to GC and parallelism

Go

Go is a well-designed, popular language, but in terms of parallelism is stuck in the mindset of taking a sequential language and adding parallelism and hoping things will be OK

Parallelism is *not* an add-on!

All these languages (Go, C++, Java, C, etc.) provide mechanism, but no (or insufficient) analysis for concurrency

Topics: Linda

Linda: an adaption of the thread pool/worker idea

Like these, it is task based, with threads choosing tasks and executing them

However, now, the tasks have extra structure to guide the choice

And the view of the system is flipped from thinking of it as a thread pool to thinking of it as a task pool

Linda

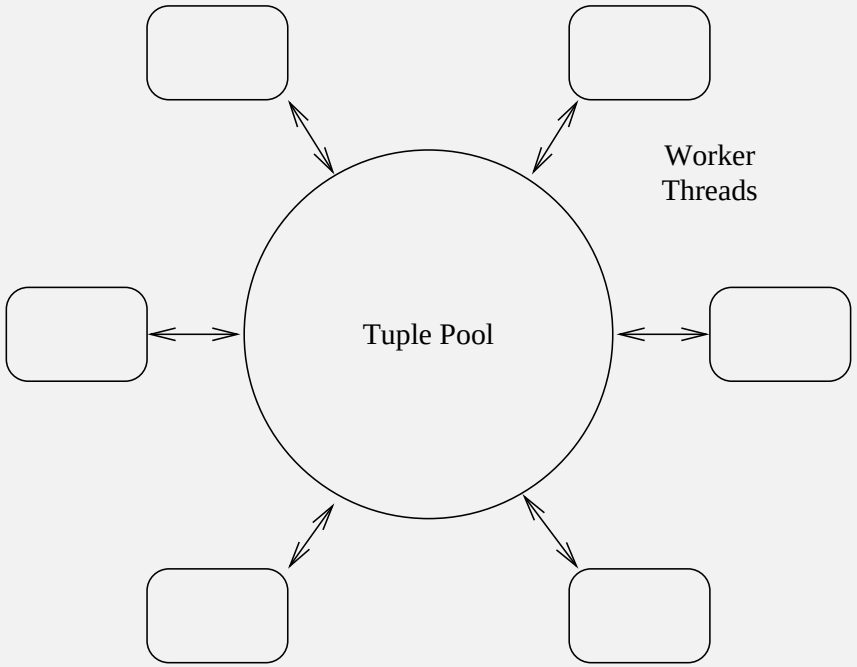
The world is based around *tuples*: these are simple (short) ordered sequences of values

E.g., `[1, "hello"]` and `[2, "goodbye"]`

And there is a global pool or *tuplespace* containing these tuples

Threads communicate via the pool by putting tuples in and taking tuples out

Linda



Linda pool

All communications via the pool

Linda

There are four operations the threads can execute:

- `out` send a tuple to the pool
- `in` get and remove a tuple from the pool
- `read` get but don't remove a tuple from the pool
- `eval` create a new thread

Linda

The important bit is how `in` and `read` work

The arguments to these are either (a) a literal constant (e.g., string, integer) or (b) a pattern variable, e.g., `?s`, or something suitable for the language you are using

A matching tuple is returned from the pool where a match is defined by

- the tuple is the same length and
- the contant literals match

Then the pattern variables are set to the corresponding values in the chosen tuple

Linda

For example, if the pool contains

```
[1, "hello"], [2, "goodbye"], [1, "world"]
```

then there are two matches for `[1, ?s]`, namely `[1, "hello"]` and `[1, "world"]`

One of these will be chosen, *non-deterministically*

If the former is chosen, then the variable `s` will be given the value `"hello"`

Linda

Note that the matching and choosing done by the pool: a possible bottleneck if the implementation of the Linda library is not careful

Of course, the pool might itself be a multithreaded system

If no matching tuple exists, the call will block until one arrives

If more than one thread simultaneously matches a tuple using `in`, exactly one will get the tuple

The action of match and removal for an `in` is atomic

Linda

If both threads use `read`, there is no problem, they both get a copy

If one uses an `in` and the other a `read`, it can go either way:

- `read` before `in`: they both get the tuple
- `in` before `read`: the `in` gets the tuple, the `read` doesn't

This non-deterministic outcome would normally be considered a programmer error

Linda

These subtleties mean you must be quite careful with Linda

A common paradigm is to use an initial *tag*, often an integer, as in `[1, "hello"]`, to impose some structure on the tuples

Linda

Dining Philosophers in Linda

We have five philosophers and shall prevent deadlock by only letting four sit at a time

Initial conditions:
`out("place ticket")` four times;
`out("chopstick", i)` for $i = 0 \ldots 4$
`eval(phil, i)` for $i = 0 \ldots 4$

Linda

```
defun phil(i) {
  while true {
    think()
    in("place ticket")
    in("chopstick", i)
    in("chopstick", i+1 mod 5)
    eat()
    out("chopstick", i)
    out("chopstick", i+1 mod 5)
    out("place ticket")
  }
}
```

This example contains no patterns, only constant literals

Note Linda does not eliminate the possibility of deadlock in badly written programs: just put the `(in "place ticket")` after the `in` of the chopsticks

Linda

Producers/Consumers is just as easy

```
defun producer(n) {
  out(n, make-product())
  producer(n + 1)
}
defun consumer(n) {
  var prod
  in(n, (? prod))     ; pattern
  consume-product(prod)
  consumer(n + 1)
}
```

We use a tag to ensure we consume values in the same order as they are produced (if that is important)

Linda

Exercise Think about the assessed coursework using Linda

Questions of granularity are just as important in Linda as elsewhere

Linda

Linda is easy to add to existing languages, usually as a library, occasionally with a minor tweak to the syntax for the patterns

Versions exist for C, Perl, Java and Prolog and others

Linda

Also

- the blocking semantics lead to unwanted non-determinism: see the `in` vying against the `read` above
- some implementations have non-blocking variants of `in` and `read`, but this just adds to the uncertainty
- the low-level, unstructured nature of Linda can lead to awkward code: every application needs some mechanism to structure the tuples (tags being the simplest)
- there is no fairness on selecting tuples: a tuple can be ignored indefinitely if there are others that can be chosen
- junk can collect in the pool: tuples put in but never taken out. This can slow down the matching
- the pool can be a bottleneck

Linda

Further

- detecting when the program needs to terminate is a problem: this could be done by putting a special “end of program” tuple in the pool; but then threads have the overhead of constantly checking for that tuple (and you need an non-blocking `read` to do so). Or have an extra field in every tuple that is a status flag, etc.
- *aliasing* is a problem: careful constructions of name schemes (tags, usually) are needed to ensure that tuples are not accidentally picked up by the wrong threads
- related is *temporal aliasing*, where information about the order tuples were put into the pool is lost: again an enumeration tag can fix this, but it has to be coded

Linda

So extensions of Linda exist, e.g., using multiple pools to structure, thus avoiding the first kind of aliasing

Now pools become first-class objects, and you can pass pools via pools to other threads

But, as always, this moves away from the initial simplicity of the Linda concept

Linda

Linda

- is a simple abstract model of parallelism
- can map reasonably well to different kinds of hardware (shared and distributed)
- is explicitly non-deterministic, with the non-determinism mostly well delineated
- is not suited for all kinds of problem
- is not widely used, but you do see Linda being mentioned now and again, mostly for coordination between other systems

Linda

For example, the computational chemistry (molecule simulation) package Gaussian uses OpenMP on a node and uses Linda between nodes

Topics: Parallel Languages

We now have a look at some languages that were designed specifically with parallelism in mind

- Occam (channels)
- Erlang (explicit parallelism)
- Go (explicit parallelism)
- Rust (explicit parallelism)
- SISAL (implicit parallelism)
- Strand (declarative)

Picked pretty much at random: by no means an exhaustive or even comprehensive list, many other languages exist

Occam

Occam was a language that was based on *Communicating Sequential Processes* (CSP) a theoretical model of parallel computation: a *process algebra* (c.f., Lambda Calculus)

CSP models processes that communicate by passing messages between themselves along *channels*

In the algebra there are various rules on combining processes and descriptions on how these combined objects behave

Then theoreticians get busy on proving that behaviours of various systems are equivalent (or not)

A note on channels

The channel concept is quite simple and so appears in many languages and systems

You put data in one end, it comes out the other end

The simplicity is probably why it appears in so many guises

They are good for structuring your code

But channels are as fast or slow as the underlying mechanism, e.g., network messages in MPI or shared memory in shared memory machines. They can't magic away the cost of communications

Occam

Occam was a realisation of CSP, designed hand-in-hand with the hardware it would run on: the *transputer*

The transputer (early 1980s) was going to be the future of parallel processing: a new hardware architecture explicitly supporting message passing between cores

Unfortunately, the level of technology of the time was not really up to the task: they had problems with clock speeds and heat management

There was no real advantage to using a transputer over existing, classical processors (like Intel), so it never managed to sell in numbers large enough to be successful

But the transputer was designed primarily to run Occam

Occam

Occam has explicit parallelism of tasks:

```
PAR
  f(x)
  g(y)
```

runs `f` and `g` concurrently

More unusually, Occam has explicit sequentiality:

```
SEQ
  f(x)
  g(y)
```

runs `f`, then `g`

This is because in CSP sequential composition of code is of equal note to parallel composition of code

Occam

Communication between processes is via channels

```
ch ! x
```

writes the value of `x` down the channel named `ch`

```
ch ? y
```

reads a value into `y` from the channel named `ch`

Both are blocking: the write will wait for the corresponding read; the read will wait for the corresponding write

Occam

Thus we get communication and synchronisation between threads

```
INT x:
CHAN INT ch:
PAR
  SEQ
    print("hello")
    ch ! 42
  SEQ
    ch ? x
    print(" world")
```

will print `"hello world"`

Occam

There is also non-deterministic choice

```
ALT
  in1 ? x
    SEQ
      x := x+1
      out1 ! x
  in2 ? x
    SEQ
      x := x-1
      out2 ! x
```

will wait until data arrives on channel `in1` or `in2` and will then execute the relevant section of code

If data arrives on both simultaneously, one branch will be taken non-deterministically

Occam

The only way for tasks to communicate is via channels

There is no concept of shared or distributed, so a program should work equally on shared or distributed memory

This is a bit like MPI messaging: it provides independence from the hardware

Occam

Plus loads more features: boolean guards (on ALT); timeouts on guards; priority ordered ALTs; functions; procedures; arrays; while loops; etc.

A program is a bunch of processes (threads in modern terms), joined by `PAR`s, that send data along channels to each other

By being closely related to CSP, there were opportunities to do proofs on Occam programs

Thus Occam can be said to provide both mechanism and analysis for concurrency

Occam

Occam never took off as transputers were not really up to it

Programmers never got the hang of it, either

It has, however had a long-lasting influence on the design of other modern languages

There was an extension: Occam-π. This was a realisation of the π-calculus, which is itself a generalisation of CSP, where channels and processes are first class objects, e.g., pass a channel down a channel

A good model to revisit in light of the current obsession with mobile processes

Big Exercise Implement Occam on top of MPI, or OpenMP

Occam

Exercise Read about the Xc language that is like C with distinct Occam flavour:

```
int main() {
   par {
      foo(0);
      bar(1);
      baz(3);
   }
   return 0;
}
```

Erlang

Erlang is a single assignment functional language, with explicit support for MIMD parallelism

A program can contain a large number of very lightweight threads: 20 million is possible they claim

Thus these threads do not correspond directly to OS threads, but are managed by the Erlang runtime (a VM; c.f. Go)

Having *no shared state*, the threads act more like OS processes than normal threads

Erlang

They do not share state because the processes (they call their threads “processes”) may be on distributed memory

Or two processes might be on the same local shared memory, but you cannot assume that

Also, this fits in nicely with the functional style: everything is local to the process and everything is referentially transparent

An important consideration is that the overheads of creation, destruction and context switching are very small: thus encouraging many small, short-lived, single-use processes

Erlang

An Erlang runtime will typically run one OS-style thread per core; each running an Erlang scheduler

These schedulers will choose and run the Erlang-style processes in a manager/worker fashion

Thus it avoids the overhead of OS thread creation/deletion

In one Erlang implementation, a process requires approximately 600 bytes of state

Thus enabling a large number of processes

Exercise Find out the memory overhead of a normal pthread in your favourite operating system

Erlang

Erlang threads communicate via messages like Occam/CSP, but they are *asynchronous*, unlike Occam/CSP

Again, messages works equally over shared and distributed memory

Also, Erlang does not have named channels, but each process has a “mailbox” where it receives all its messages

Alternative point of view: the process “name” is the name of the (only) channel to a process

Erlang

The messages can be values, tuples of values, or any other datatype, including closures (functions)

And there is pattern matching to fetch messages from the mailbox (a bit like MPI tags, but more general matching, so more like Linda)

Erlang

```
Otherproc ! { hello, 99 }
```

sends a tuple with *atom* (like a Lisp symbol) `hello` and the integer 99 to the process named by `Otherproc` (variables start with capital letters)

```
receive
  { hello, X } -> io:format("x was ~B~n", [X]);
  { bye, X } -> io:format("time to go~n", []);
  _ -> io:format("eh?~n", [])
end.
```

an underscore matches any message; this is like an ALT in Occam

Erlang

Creation of processes is via `spawn`

```
factrec(0) -> 1;
factrec(N) when N > 0 -> N*factrec(N-1).
fact(N, Ans) -> Ans ! factrec(N).

FactPid = spawn(fact, [5, self()]).

receive
  F -> io:format("factorial is ~B~n", [F])
end.
```

is clumsy code to make a new process running `fact` with arguments 5 and the process identifier (PID) of the current process

The `receive` causes the current process (`self()`) to wait for a message (from anyone), and stores it in `F`

Erlang

A PID is the way you refer to a process, in particular for sending a message to it

N.B. some liberties taken with Erlang modules here

Erlang

Erlang is quite popular in real systems as it has lots of useful features

For example, *Process Restart*, where a process is immediately restarted by the runtime if it crashes for any reason

This allows Erlang to cope with hardware failure and buggy code

In fact, Erlang has hot swap of code: code can be changed (fixed or updated) while the main program is running

Load balancing of processes is done by the runtime VM

Erlang

Originally designed by Ericsson to support (soft) realtime systems that can't be taken down for maintenance (like telephone exchanges), it has found use in other areas

Companies like Yahoo, Facebook, WhatsApp, Bet365, etc. use it for some element of their products

Somewhat an under-appreciated language

Exercise Have a look at
`http://learnyousomeerlang.com/content`

Go

Go we have seen before, so here's just a short discussion (in the context of these other parallel-aware languages)

It has goroutines, communicating via channels, similar to Occam and Erlang

Channels are type safe ("channel of int") and blocking

There is a `select` that acts like Occam's `ALT` waiting on multiple channels

Go

Synchronisation and communication are provided by channels

Libraries provide condition variables, mutexes, atomics and a variety of other low-level functionality

Channels are the recommended ways of passing data between threads; though you can also use shared variables

Though shared variables are not recommended as Go provides no inherent protection against the usual data races (if you don't remember to use mutexes and the like)

Go

From the Go website (worth repeating!):

> *Share memory by communicating; don't communicate by sharing memory.*

Go

Go has a race detector tool: compiling with `-race` checks memory accesses and spots unsynchronised accesses

This

- is run time detection
- slows the execution by an order of magnitude
- only finds races that actually happen in a run

Go

Go is used widely, with some vocal proponents

It was designed by people with a considerable amount of expertise, but doesn't bring anything new to the table in terms of tackling parallelism

> *. . . in fact, there isn't much to Go other than channels and goroutines!*

Stjepan Glavina

Rust

A language originally designed and developed by the Mozilla team, with the eventual aim of reimplementing the Firefox browser in Rust, but now a general-purpose language in its own right

A lot of the problems in many applications (including browsers) are to do with bad memory management

> *Around 70 percent of all the vulnerabilities in Microsoft products addressed through a security update each year are memory safety issues*
> Matt Miller, Microsoft security engineer, Feb 2019

Rust

So Rust is a *memory safe* language, meaning it can not have problems like dangling pointers (null pointers), uninitialised variables, use after free errors, or buffer overflows

Or, at least, it makes it very hard for the programmer to produce such bad code

Unlike many languages, such as C and C++, that make it very easy

Rust

Memory safety is not new: many memory safe (or nearly safe) languages have been devised, with various trade-offs to get this safety

For example, runtime checks on accesses to buffers; garbage collectors; and so on

A lot of these have runtime overhead, i.e., your program is safer, but runs more slowly

And they are not always completely successful, e.g., Java can have null pointers

Rust

Rust takes a different approach and tries to put as much checking as possible into the compiler: your code is safe, and fast

But the trade-off is this: it does this by having a concept of the *owner* of a memory location and tracking that ownership in the compiler

Rust

In an assignment `y = x;` the ownership of the memory referred to by `x` is transferred to `y`. It is now illegal/impossible to use the variable `x` in subsequent code

The **compiler** would flag any later reference to `x` as an error and refuse to compile

This helps with memory management, as the compiler can precisely track the lifetime of a value and so its memory can be deallocated automatically when the compiler can prove it is not longer accessible and without the need for a garbage collector

Thus avoiding the programming errors common to C-like languages and the runtime complexities of GC languages

Rust

Memory safety is a good thing in that you can't accidentally use a value that has been deallocated: but even better is that ownership also helps with data races

A data race can happen when one memory location is accessed by two threads, at least one doing a write

A read-only (const) value can be shared; a writable (mutable) value shouldn't be shared (c.f., RW locks)

The Rust compiler can use ownership to track a value and will spot an attempt to modify a shared value and refuse to compile

Rust

Thus making the programmer face up to the problem and fix it before the code will even compile

Rust developers call this “Fearless Concurrency” as the language itself prevents these kinds of data-race

Rust provides both mechanism and analysis for concurrency

This fixes data races: unfortunately the Rust compiler is not (yet?) able to spot non-data-race race conditions, like deadlock

Rust

Rust has a classical heavyweight thread approach, with a `thread` module that contains functions like `thread::spawn()`

This takes a closure as argument as the thing to execute

```
let threadid = thread::spawn(|| foo(x+1,y-1));
...
let val = threadid.join().unwrap();
```

Rust

What do we do if we need to mutate a shared value in different threads?

We can use a mutex to sequentialise the accesses

Mutexes can be shared across threads

Rust

A `Mutex` can be used to wrap any data:
`let mtx = Mutex::new(v);`

And now the *only* way of accessing the data that used to be in `v` is via the mutex: `let mut data = mtx.lock().unwrap();`

An important thing here is that the ownership of the value has passed to within the mutex `mtx`, and so is no longer available from the variable `v`

This prevents accidental direct access to the data, and this is checked and enforced by the compiler: we can't accidentally use `v`

And we can get mutable access to the data only when locked

Rust

There is no unlock method: the mutex automatically unlocks when the holder goes out of scope

Thus the programmer can't forget to unlock a mutex, **or access the data without using the mutex**

Rust

Rust also has barriers, condition variables, channels, etc.

As always, channels are still an excellent way for threads to communicate, but Rust's ownership model means sharing variables is no longer dangerous: the compiler simply won't let you share things unsafely

Rust

Rust is still being developed, but has already been taken up by many big companies and projects (e.g., by Google for Android, alongside Java and Kotlin; Microsoft are rewriting parts of Windows in Rust)

The ownership mechanism is a stumbling block to many programmers coming from other languages

Mostly those programmers who don't like the compiler telling them their code is broken: you need to get more things correct before you can compile code

But the learning curve is worth it for the safety achieved

Rust

Exercise For C++ geeks. The idea of tracking ownership (“move semantics”) has recently been adopted by C++, though its use is optional and not the default. Read about this

Exercise The Rust compiler guarantees that a mutable (writable) memory location can never be accessed by more than one thread at a time. How might the compiler use this knowledge to optimise operations on that memory location?

Rust

Shared mutable state is the root of all evil. Most languages attempt to deal with this problem through the 'mutable' part, but Rust deals with it by solving the 'shared' part.
From the Rust website

It may be harder to write Rust code than Java code, but it's a lot harder *to write* incorrect *Rust code than* incorrect *Java code*
"Llogiq on stuff" Feb 2016

SISAL

Another single assignment, functional language, this time with *implicit* parallelism

Streams and Iteration in a Single Assignment Language, as its name suggests has special regard for streams and iterations

It distinguishes carefully between loops where the computations in the loop body are independent (thus parallelisable, they call them *for-loops*) and those where they are not independent (they call these *iterations*)

SISAL

The for-loop looks like

```
for <range>
   <optional body>
returns <returns clause>
end for
```

All expressions in SISAL return one or more values

SISAL

An example:

```
for i in 1, n
   sqs := vals[i]*vals[i]
returns array of sqs
end for
```

returns an array of the squares of the values

The effect is like a new instance of `sqs` is made for each value of `i`, then the `array of` operator collects (a reduce operation) them into an array

SISAL

Other reductions are possible

```
for i in 1, n
   sqs := vals[i]*vals[i]
returns array of sqs,
        value of sum sqs
end for
```

returns two things: the array as before, and the sum of the squares; `sum` is another reduction operation

SISAL

The point here is that each squaring is independent

SISAL makes us write the loop in such a way to make this simple and evident

So it may choose to run this in parallel: automatic parallelisation

SISAL

SISAL was briefly popular in the mid-1980s when people were looking for ways for extracting parallelism automatically

It is an example of a *dataflow language*

These work on the idea that it is the data that should direct the processing

A spreadsheet is a simple example of the dataflow concept: change the value in a cell and this triggers various (re)computations, possibly running in parallel

SISAL is of academic interest, but is not used widely

Strand

A single assignment language reminiscent of Prolog with dataflow (again, mid to late 1980s), declarative

There is a single, shared global namespace and threads communicate by writing and reading variables

If a thread tries to read a variable before it is set, that thread will block

Thus we get both message passing and synchronisation

And so variables are also a bit like single-use channels

Strand

Strand only supports parallel composition: i.e., you cannot write sequentially

The dataflow between the variables is all the sequencing we get

And, conversely, if one expression does not depend on another, that can be run in parallel

Again allowing automatic parallelism

Strand

Code is a list of *rules* rather like Prolog:

```
clause :- guard, guard, ... | body
```

A program consists of many rules

Strand

All rules are eligible for execution at all times as long as all their guard conditions are satisfied

Guards can be evaluated in parallel

If a rule is selected, then a new process evaluates the body

If no rules match, then it's an error in your program

Strand

Rules:

```
consumer(X) :- X | eat(X).
producer(Y) :- Y := "food".
```

with program:

```
producer(Z), consumer(Z).
```

the variable `Z` acts as a shared “channel” between the producer and consumer

Strand

As always, there's much more to Strand than this: streams, foreign language interface (to call C, etc.), garbage collection, and so on

And, just like Prolog, not widely used

It's just not the way most programmers think!

Parallel Languages

Thus there are several ways a language design can avoid races:

- have no shared variables (e.g., Erlang)
- have no mutable shared variables (e.g., Rust)
- have no mutation (e.g., Haskell)
- have no parallelism! (e.g., JavaScript, Python)

Allowing unrestricted access to shared values (as we are used to in sequential programming) is a sure route to creating races

But having any the above restrictions in a language is guaranteed to irritate some programmers — they don't like being forced to write correct programs!

Parallel Languages

And so on. See Wikipedia!

- C*. Connection Machine, SIMD
- Cω. Cray, modified C, like data parallel Fortran
- Concurrent Euclid. Functional influenced descendant of Pascal
- Data Parallel Haskell.
- E. Secure distributed programming
- Ease. A CSP language
- Fortress. Secure Fortran, implicit parallelism
- Janus. “bag channels” pool-like communications

Parallel Languages

- Joule. Dataflow, like E
- Joyce. Pascal syntax, CSP
- Limbo. Channels
- Lucid. Dataflow
- MultiLisp. Scheme extension, arguments to function calls explicitly evaluated in parallel, lazy evaluation
- NESL. Precursor to Data Parallel Haskell
- Orc. Concurrent, non-deterministic
- Oz. Multiparadigm: dataflow and declarative
- Parlog. Parallel Prolog

Parallel Languages

- SALSA. Actor, runs on Java machine
- Sing#. Extension of C#. Message passing
- SPARK. Based on Ada
- SR. Message passing
- $*$Lisp. Connection Machine
- Turing$+$. Monitors
- XC. Explicit parallelism
- ZPL. Like C/C++, implicit parallelism.

Parallel Languages

Exercise Swift, Rust and Go are all “modern” languages, designed in the current era of parallel hardware. Compare their approaches to parallelism

Exercise Think about using all of OpenMP, MPI (and CUDA/OpenCl on GPUs) in a single program

Redo Assignment 1 using Swift, Rust, Go, CUDA, etc.

Topics: Time Warp

Time Warp is another way of parallelising programs, mostly applied in the area of discrete event simulation, using distributed architectures

For example, simulating the movement of molecules in a fluid; or packets in a network; or tanks on a battlefield

"Discrete Event" means we simulate the system for discrete ticks of time, $t = 0$, $t = 1$, etc., where each tick might be a second or a microsecond; even a nano- or picosecond for some physics and chemistry simulations

The objects in the system interact via events, e.g., one molecule hitting another or a packet entering a router

Time Warp

On the face of it, simulation should be good to parallelise, as it comprises objects (molecules, packets, tanks) that are mostly independent, but require the occasional interaction (molecules bounce off each other; packets overload a router; tanks fire missiles at each other)

As long as there isn't *too* much interaction (lots of missiles?)

Time Warp

Simulations can have millions of objects, and so must be done on a distributed memory machine: they simply won't fit on a shared memory machine

This is parallelism for reasons of size, not speed

So the events must be messages

Now the problem turns out to be synchronisation, but in *simulation* time

Time Warp

Two molecules A and B might be on separate processors; each moving through their simulated ticks of time

Molecule A might be trundling along, simulated at time $t = 100$, $t = 101$, etc., only to find it should have bounced off molecule B at time $t = 90$

The “collision” message may simply have taken ages to get from B to A across the cluster’s network

Or B’s processor is slower than A’s; or more heavily loaded; etc.

Time Warp

So, we could just synchronise at every time step?

Yes, but this would be very slow: just think of the synchronisation messaging needed

The secret is to let each processor work at its own pace and only synchronise when necessary

There are several ways of doing this, but the weirdest is the *Time Warp*

This is an *optimistic* method

Time Warp

Invented by Jefferson in the 1980s, it tries to extract as much parallelism as possible by each object optimistically simulating forward in time, only going back to fix things if they went too far

For the most part, most objects do not interact, so waiting (synchronising) on other objects is usually a waste of time

Every processor simulates at its own speed, under the optimistic assumption that there would have been no interaction or synchronising

Time Warp

So molecule A is simulated as fast as its processor permits, its time progressing $t = 100$, $t = 101$, etc.

Similarly for B on its processor $t = 89$, $t = 90$, etc.

Each molecule lives in its own bubble of time, independent of other molecules

Each object in the system has an *input queue* of messages yet to be processed from other objects: these messages will have timestamps in the future of the object

An object will repeatedly read the next message from its queue and act upon it. The object's current time is set to the timestamp on the message

Time Warp

Now sometimes, and we hope this is rare, there *is* interaction and we might have gone too far: molecule A is at time 102 but then receives a message with timestamp $t = 98$, so A should have bounced then

The bounce message simply didn't reach A soon enough

In this case the Time Warp mechanism says we should locally reverse time and roll A back to time 98, do the computation for the bounce, then continue forward

Time Warp

To do this *rollback* we either need reversible computations, or we keep a record of the past states of A and revert it to its state at time 98

A then can continue forward, after bouncing appropriately

If rollbacks do not happen too often, we get improved speedup as each processor can compute at full speed with no synchronisation

Time Warp

But there is a problem

Suppose A is optimistically computing forward and decides it hits molecule C at time $t = 101$

So it sends a message to C, saying "A hits C at $t = 101$"

Then B's message arrives, saying "B hits A at $t = 98$"

This bounce will probably divert A so that it would not have hit C

But the message has already been sent!

Time Warp

A's message to C is now an error

Time Warp fixes this with *anti-messages*

As part of the rollback process, if an object finds it sent a message in error it now sends a corresponding anti-message

A sends the anti-"A hits C at $t = 101$" message to C

Time Warp

Two things can happen in C:

- If the message arrives in C's future (C is still processing at time $t = 100$, say), the positive message is still waiting for C to read it. Thus the positive message can simply be removed from C's input queue of messages
- If the message arrives in C's present or past (C is processing at time $t = 104$, say), this triggers a rollback of C: it unwinds to time $t = 101$, and then proceeds forward, this time without the bounce off A

Time Warp

C's rollback might require anti-messages from C; which might trigger more rollbacks from other objects; and so on

In the worst case, there can be an *anti-message cascade* where more and more objects trigger rollbacks of other objects

The hope is that this is rare, and outweighed by the general forward progress from the optimistic computation

Time Warp

For the right kind of simulation, Time Warp is very effective

However, there are so many important details of implementation that it is hard to find a good implementation of Time Warp

For example, the management of past states. These are required for the rollbacks, but you can't keep them forever as they will eat up more and more memory as the computation proceeds

So implementations include a garbage collection of old, inaccessible states

Time Warp

If all objects *and all unread messages* are beyond time $t = 100$, then there cannot be a rollback to earlier times

So states older than $t = 100$ can then be discarded (garbage collected) across the entire system

So now we have the problem of finding the earliest timestamp in the system: another problem in its own right as this is information that is distributed across the entire system

We have to be careful that the messaging needed to find the earliest timestamp plus the time spent garbage collecting is not too large in itself

Time Warp

Time Warp is effective if

- interactions are rare
- the cost of rollback is low

But

- the cost of storing state can be high
- the overheads of garbage collection can be high

Other distributed simulation methods are more popular, e.g., conservative simulation: you only progress as far as you can prove is safe, which may be more appropriate than Time Warp where there is a moderate amount of interaction

Time Warp

There is a big literature on parallel discrete event simulation (PDES) as it is used by various large organisation, e.g., the US Army

Much research into Time Warp was funded by the US Army

They have very large battlefield simulations

Researchers were worried when explaining Time Warp to the Generals that, by talking about missiles and anti-missiles instead of messages and anti-messages that the Generals might get the wrong idea and require the invention of anti-missiles. . .

Topics: GPUs

Graphics co-processors have grown immensely in power in the last few years

Originally intended to offload graphical work from the main CPU they have become recognised as powerful processors in their own right and people have tried to tap into their potential

General-Purpose computing on Graphics Processing Units (GPGPU) has emerged as an important example of parallel processing

So hardware, originally intended to support gamers, is now being used in general purpose computations

GPU-based computing appears strongly in the Top 500 largest computers in the world

GPUs

GPUs naturally do certain things very well: in particular data-parallel pixel rendering (colouring, shading and so on)

The computations you typically do on pixels can be quite intensive, but are fairly restricted in nature

And the data-parallel nature of the computations on the millions of pixels on your screen is very relevant

Over time GPUs became more and more programmable as they needed to do more and more complex manipulations

Graphics libraries (like OpenGL and DirectX) that were originally developed to draw pictures eventually supported programmable sequences of operations via *shader languages* such as GLSL and HLSL (aka Cg)

GPUs

So people soon realised that GPUs are powerful multicore SIMD processors, but just tuned for certain intensive data-parallel computations

GPU companies like NVIDIA and AMD/ATI have seen the possibilities of using this power and now put hardware into their GPUs specifically to help GPGPU computations

This means putting in hardware to support generic computation, not just graphics oriented stuff

GPUs

And NVIDIA have also produced a language, *Compute Unified Device Architecture* (CUDA), to aid in the general programming of these devices

There is also an open standard, *Open Computing Language* (OpenCL), that is not vendor based

CUDA is quite popular right now, but only runs on NVIDIA cards

OpenCL is strong, and is supported by NVIDIA, AMD, Intel and ARM amongst others

GPUs

CUDA

CUDA looks a lot like C and C++

Dangerously close, as there are several important differences between CUDA and these languages

CUDA is a modified C/C++ with a *syntactic* addition to notate parallel execution and various semantic additions to support parallelism

It requires a special compiler, provided by Nvidia

In contrast, OpenCL is a library that runs on plain C or C++ (and any other language that can call C functions)

GPUs

Architecture

The language reflects the hardware architecture

A GPU has several *multiprocessors* each containing a bunch of SIMD cores: thus a GPU is a MIMD of SIMD

It works best when there are thousands of threads, even if there are only hundreds of cores

This is to overlap communications with computation: a core that would be waiting for some data can pick up another thread and work on it instead on doing nothing

Memory access in GPUs is relatively **very slow**, so there would be a lot of waiting otherwise

GPUs

Architecture

Threads in a GPU are hardware managed and extremely lightweight, meaning they have tiny creation and scheduling overhead

Thus there is no need to worry about making and destroying large numbers of threads

Very different from normal CPU threads

Exercise Why don't normal CPUs do the same: have hardware support for threads?

GPUs

Architecture

GPUs have very complicated architectures, both for threading and memory

We shall describe them using CUDA terminology

OpenCL has a separate set of words for the same things

GPUs

CUDA

There is a hierarchical management of the threads

- A *kernel* is some code running on the *device* (GPU)
- A *grid* is the collection of all threads in a kernel
- A grid contains one or more *thread blocks*
- A thread block contains a number of threads: all blocks in a grid contain the same number of threads

All threads in a grid execute the same kernel

These are not all SIMD, but are arranged in bunches, called *warps*, of SIMD threads within the blocks

NVIDIA calls this “Single Instruction Multiple Thread” (SIMT)

GPUs

CUDA

For example, threads 0–31 are in one warp and 32–63 are in another warp

Warps are the basic SIMD chunk

This means it is better to gather threads that take the same branches of an if or loop as they will be processed together:

```
if (threadid < 32) {...} else {...}
```

is better than

```
if (threadid % 2 == 0) {...} else {...}
```

GPUs

CUDA

A block (of multiple warps) is the basic chunk that gets scheduled on a multiprocessor; the multiprocessor then executes the warps, as many as it can at a time as the hardware permits

While threads within a warp are SIMD, separate blocks of threads might be executed at different times: a kind of SPMD of SIMD, though the SPMD nature is generally not really usable

Warps within a block might be executed at the same time or at different times depending on the number of cores per multiprocessor and the number of schedulers per multiprocessor

GPUs

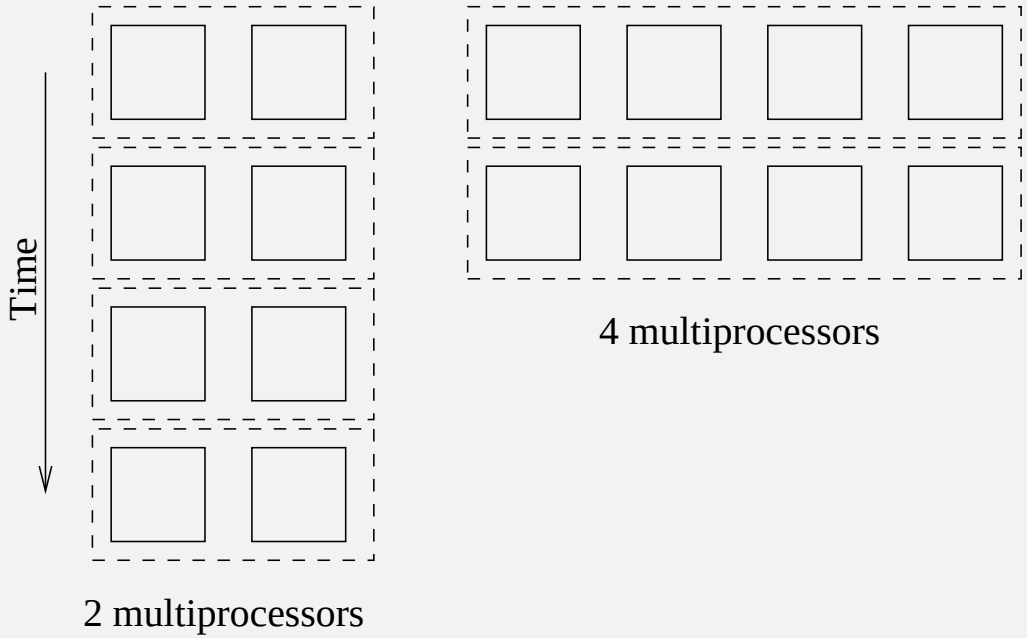
CUDA

Having many warps and many blocks means the system can adapt at runtime to the number of multiprocessors available in the hardware

Suppose we have 8 blocks in our grid

GPUs

CUDA



Processing CUDA blocks

This naturally and automatically obtains more parallelism when there are more multiprocessors. So it makes sense to have lots more blocks than multiprocessors

GPUs

CUDA

All the blocks in a given grid have the same number of threads

Blocks are indexed in the grid in one, two or three dimensions (programmer's choice)

`blockIdx.x` returns the block index for a 1D arrangement

`blockIdx.x` and `blockIdx.y` return the block indices for a 2D arrangement

`blockIdx.x`, `blockIdx.y` and `blockIdx.z` return the block indices for a 3D arrangement

You specify the size and number of dimensions when creating the grid

GPUs

CUDA

The threads within a block are indexed in one, two or three dimensions

- `threadIdx.x`
- `threadIdx.x`, `threadIdx.y`
- `threadIdx.x`, `threadIdx.y`, `threadIdx.z`

You specify the size and number of dimensions of the blocks when creating the grid

GPUs

CUDA

Each thread has its own CPU-style state and registers used in the normal way for function local variables and temporary results; the hardware has a fixed number of registers (32768, say) which are shared amongst the threads in a block

Each thread has a chunk of **slow** local memory (`__local__`)

This is accessible only by the thread

Registers are what you need to use if you want fast access, but registers are limited in number, and `__local__` memory might be needed if the compiler can't fit the data into registers

GPUs

CUDA

Each block has a chunk of **fast** shared memory (`__shared__`)

This is accessible by all the threads in the block and can be used to communicate between threads in a block

GPUs

CUDA

A grid has a big chunk of **slow** global shared memory

This is accessible to all the threads in all the blocks and is the way to communicate between threads in different blocks

Importantly, access to each of these areas of memory is at radically different speeds

Access to registers is a bit faster than block shared memory (a few cycles to access); both are *much* faster than global shared and thread local memory (hundreds of cycles to access)

So you need to take care on where you place data

GPUs

CUDA

A typical CUDA source program contains a mix of code to be run on the CPU and code to be run on the GPU

This can be in the same source file: GPU kernels are marked by `__global__`

The code is pretty much normal C/C++, but with some restrictions

Note, when executing, code and data on the GPU are separate from code and data on the CPU

Values are passed from CPU to GPU as arguments of CUDA kernel calls; or as explicit cpu-memory-to-gpu-memory copies

GPUs

CUDA

CUDA has dimension types that are used to specify sizes and shapes of grids and blocks

`dim3 B(w, h, d)` defines `B` to be a 3D $w \times h \times d$ shape object

`dim3 G(n, m)` defines `G` to be a 2D $n \times m$ shape object

Just use an integer for 1D

GPUs

CUDA

If `fun` is a kernel (i.e., GPU function), we can call it from the CPU code by

```
fun<<<G,B>>>(arg1, arg2, ...);
```

to run `fun` on a grid containing blocks arranged as `G`; the blocks containing threads arranged as `B`

This creates $n \times m \times w \times h \times d$ threads, each running `fun`

(And copies the code for the kernel to the GPU; copies the argument values to the GPU; starts the GPU scheduler; and so on)

GPUs

CUDA

Each thread is uniquely indexed by `threadIdx` and `blockIdx` and can use these values to decide what to do

You can choose dimensions and sizes of grids and blocks to suit your problem: you should not be shy of 1000s of threads

In fact, one of the issues when writing a CUDA program is figuring how to choose your blocks and distribute your data amongst them

For example, the amount of shared memory per block is very limited, so this may affect how you choose blocks

GPUs

Properties of a typical gamer's card (2020):

name	'GeForce RTX 3080'
totalGlobalMem	10GB
maxThreadsPerBlock	1024
maxRegistersPerBlock	65536
clockRate	1.44 GHz
multiProcessorCount	68 processors
CoreCount	8704 (128 per multiprocessor)
warp size	32 threads
processing:	25 TFlop single
	783 GFlop double (1/32)
power	320W

GPUs

Properties of a compute oriented GPU card (2015):

name	'GeForce GTX K20X'
totalGlobalMem	6039339008
sharedMemPerBlock	49152
maxThreadsPerBlock	1024
maxRegistersPerBlock	65536
maxThreadsDim	1024 x 1024 x 64
maxGridSize	2147483647 x 65535 x 65535
clockRate	0.73 GHz
multiProcessorCount	14 processors
CoreCount	2688 (192 per multiprocessor)
warp size	32 threads
processing:	3935 GFlop single
	1310 GFlop double (1/3)
power	235W

GPUs

December 2017: NVIDIA Titan V

CUDA Cores	5120
Tensor Cores	640
Transistors	21.1 billion
Power	250W
Single precision	12.4 TFLOPS
Double precision	6.1 TFLOPS
Half precision	24.6 TFLOPS

Half precision they call “deep learning FLOPS”

Tensor cores are specialised to 4×4 matrix half-precision fused multiply add ($AB + C$) computations, also for AI

GPUs

CUDA

The main point of GPUs is they have a large number of cores: the RTX 3080 above has 8704 cores in 68 multiprocessors

GPUs

CUDA

There is a lot of global memory, but this is substantially slower (100s of cycles to access) than the block shared memory (maybe 2 cycles)

Though modern GPUs do cache global shared memory: access time is a couple of cycles for a cache hit (though the cache is of limited size, of course)

There is also a chunk of global *constant* memory (`__constant__`), which is read-only but faster to access than the read-write global memory

And some read-only *texture* memory, whose development arose from the needs of graphics

GPUs

CUDA

Constant memory is actually a different way of accessing global memory, but the mechanism (to make it fast access) limits the amount of constant memory available, e.g., to 64K bytes

Similarly texture memory is global memory accessed in a strange way, via a *texture reference* object

A texture reference can be associated with an area of global memory and then that memory is read via the reference

GPUs

CUDA

The weird stuff:

- the index into the texture memory is a floating point number: the value at index 3.14142, say, is interpolated appropriately by the hardware between the values for indices 3 and 4
- the index can be *normalised* to the interval 0.0 to 1.0. Then the index 0.5 corresponds to the index half-way along the array
- this can be done for 1, 2 or 3 dimensional arrays

It is possible to ignore the clever stuff and just use textures as a fast(er) way to read global memory

GPUs

CUDA

	Speed	Access	Scope	Size	Lifetime
register	v fast	r/w	thread	10s	thread
local	slow	r/w	thread	GBs	thread
shared	fast	r/w	block	KBs	block
global	slow	r/w	grid	GBs	application
constant	cached	r	grid	KBs	application
texture	cached	r	grid	KBs	application

N.B. the thread, block and grid/kernel lifetimes are typically all the same; a typical application will have many kernel calls

GPUs

CUDA

Memory affects the execution of threads

Thread blocks are scheduled by the hardware on multiprocessors and more than one block can be simultaneously scheduled on a multiprocessor, thus sharing its resources, particularly shared memory and registers

So the pattern of use of shared memory can put a limit on the number of blocks in the grid, thus a limit on the rate of execution

Similarly, there is a limit on the number of threads per block: up to 65536 in one of the above GPUs

GPUs

CUDA

GPUs offers a huge amount of processing power at low cost, but in a way that is extremely sensitive to memory access

It is easy to get started with CUDA as it is basically C, but you do have to be very aware of the properties of memory

GPUs

CUDA

Modern GPUs support *unified memory spaces*

This allows you to use a single virtual address space for both host and device memory and not worry which is which (a bit like VSM)

A hidden mechanism copies data between CPU and GPU as necessary

Exercise Is this a good idea?

(Shortly we will see some systems that have physically shared memory)

GPUs

Memory

Next, there is the extra problem shared by all coprocessors: memory bandwidth between the main CPU and the coprocessor

Copying data in and out of the GPU is significantly time consuming

So we need to worry about data movement between the GPU and the main CPU

And, if possible, overlap data transfers with GPU and CPU computation

And overlap CPU and GPU computations

GPUs

Memory

We often forget that the system also has to copy the *code*, ie., the kernels, to the GPU memory, too

The cost of this is usually small relative to the cost of copying data, but it's another reminder that the GPU's memory is separate from the CPU's

But a recent trend is to integrate the GPU onto the same package as the CPU (or vice-versa!)

Using lots of transistors!

GPUs

Memory

For example, AMD's Kaveri is a CPU+GPU on the one chip

4 CPU cores and 512 GPU cores that share cache and main memory

Of course, this changes all the memory access vs. compute balances, so needing you to revise your code

This is an example of a *Heterogeneous System Architecture* (HSA)

GPUs

Memory

The idea is more of a symmetry between the CPU and GPU: the GPU is not just a coprocessor

The GPU can now pass tasks back to the CPU to do

Accompanying this is a new low-level virtual architecture *HSA Intermediate Layer* (HSAIL) that will be used to implement higher-level abstractions like OpenCL

In a similar way, Apple's M1 architecture has CPU and GPU *and memory* on the same chip, further confusing the memory vs. compute costs question

GPUs

CUDA

Back to CUDA

Here is an example of trivial CUDA code, `prog.cu`

(Checking return values and tidying up omitted for brevity)

CUDA

```
#include <stdio.h>
__global__ void setarray(int p[])
{
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  p[k] = k*k;
}
int main(void)
{
  int i, *dm, m[1024];
  cudaMalloc(&dm, 1024*sizeof(int));
  setarray<<<16,64>>>(dm);
  cudaMemcpy(m, dm, 1024*sizeof(int),
             cudaMemcpyDeviceToHost);
  for (i = 0; i < 1024; i++)
    printf("m[%d] = %d\n", i, m[i]);
  return 0;
}
```

GPUs

CUDA

This starts 16 blocks, each containing 64 threads, each thread runs the kernel `setarray`

Each invocation of `setarray` gets the same pointer to some global memory allocated on the GPU

Each computes a different value for the index `k`, and each sets a different element of the array

This assignment is a memory bottleneck that will take a relatively long time to complete

GPUs

CUDA

CUDA programmers try to mitigate the memory bottleneck by ensuring there are lots of threads

Within a block, a warp of 32 threads is scheduled to run

These run (in SIMD) until they would have to wait for a lengthy memory access to complete: the assignment to `p` in the example

Rather than simply waiting for the memory, this warp is put aside *while the memory access is still progressing* and another warp (from this block or another block on the same multiprocessor) is scheduled to run instead

GPUs

CUDA

Thus keeping the multiprocessor busy computing

When the memory access has completed, the original warp can be run again

All these scheduling decisions and actions are done by the hardware!

Exercise Compare with hyperthreading as a way of keeping CPUs busy

GPUs

CUDA

Thus we want a lot of threads to schedule between as they run then wait for memory

If we don't have enough threads the cores will be idle during their wait for memory

Ideally each block should have a multiple of 32 threads, whenever possible, to get the most from the multiprocessor

For example, running just 16 threads means half of the warp is lying idle

GPUs

CUDA

Additionally, multiprocessors are given whole blocks to execute

So we want at least as many blocks as multiprocessors, to keep all the hardware busy

Thus it's good to have lots of threads per block and lots of blocks per multiprocessor to provide lots of choice of warps to schedule

GPUs

CUDA

How many blocks and how many threads per block?

It depends on how the program accesses memory: e.g., the use of shared resources like block shared memory might be a factor

GPUs

CUDA

From the NVIDIA documentation:

- How many blocks?
  - At least one block per SM to keep every SM occupied
  - At least two blocks per SM so something can run if block is waiting for a synchronization to complete
  - Many blocks for scalability to larger and future GPUs
- How many threads?
  - At least 192 threads per SM to hide read after write latency of 11 cycles (not necessarily in same block)
  - Use many threads to hide global memory latency
  - Too many threads exhausts registers and shared memory
  - Thread count a multiple of warp size
  - Typically, between 64 and 256 threads per block

GPUs

CUDA

The programmer might want to experiment to find the best combination of numbers of blocks and threads per block for the particular GPU they are running on

There are profiling tools and spreadsheets available to help you make this decision

And to add to the complexity: later versions of CUDA allow multiple different kernels to run concurrently (i.e., it schedules between kernels), so supplying more blocks and more threads to keep the hardware busy

CUDA kernels run asynchronously from the CPU

GPUs

Memory Coalescence

And the *pattern* of global memory access is vital, too

The memory bus has a high latency, but a large bandwidth

We have to wait a long time for bytes to arrive; but then they arrive in large chunks

Memory is set up to deliver, say, 64 bytes at a time (512 bit bus)

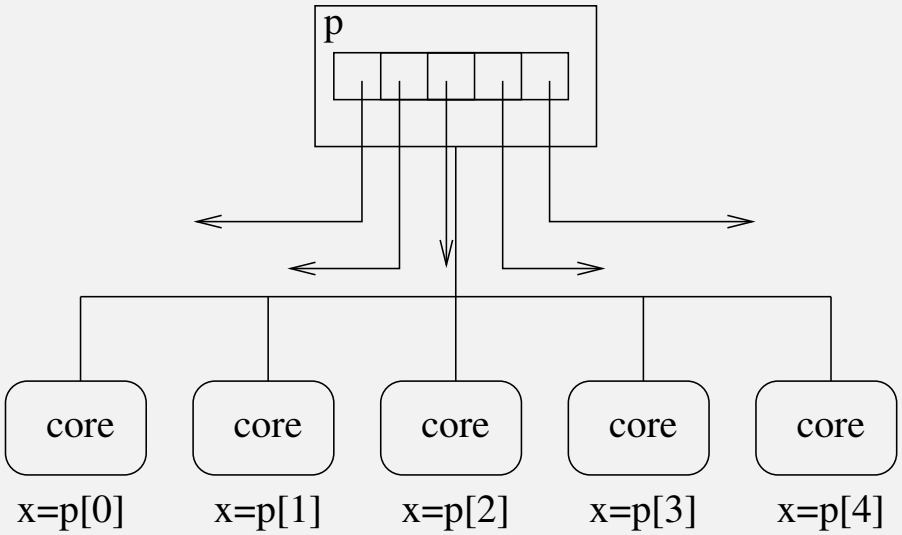
And programs often ask for large chunks of data in parallel, e.g., working in parallel on an array

64 bytes is 16 (half-warp) four-byte integers or 16 single precision floats

So a warp could be satisfied by just two reads

GPUs

Memory Coalescence




```
x = p[me]
```

If the reads are nicely arranged, a single read supplies many cores simultaneously: this is memory access *coalescence* (as discussed earlier in vector architectures)

GPUs

Memory Coalescence

As long as your code can do this

There are many rules imposed by the hardware to make this kind of memory access coalescence work

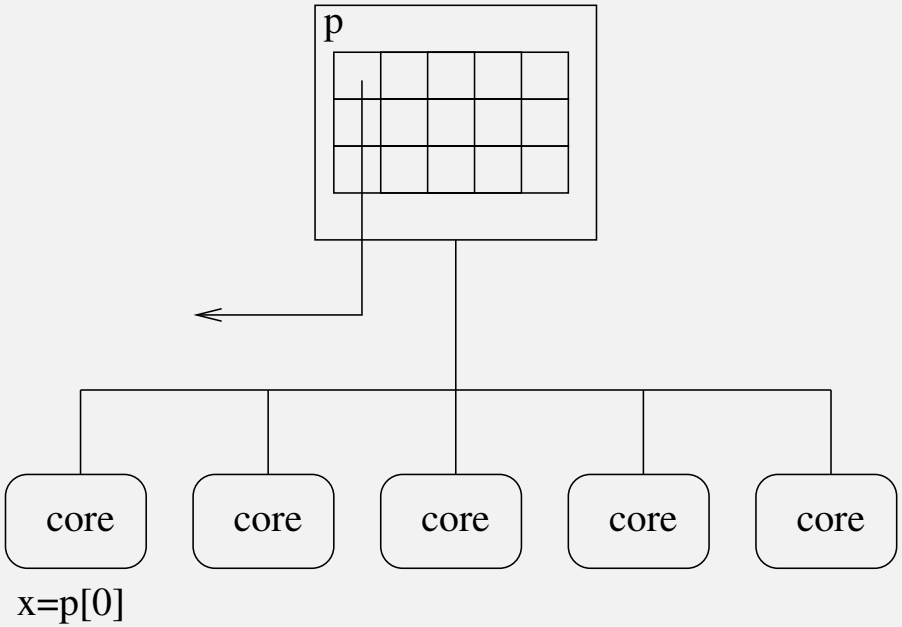
Such as alignments of areas of memory; the order in which neighbouring cores access memory; and so on

If you get it right, reading 16 integers in parallel is as fast as reading a single integer

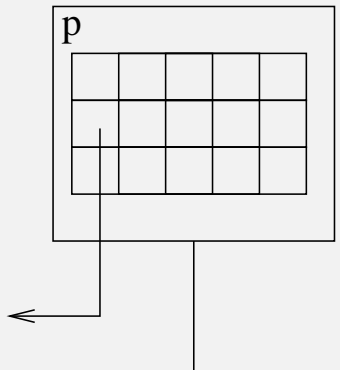
If you get it wrong, it can be 16 times as slow

GPUs

Memory Coalescence




```
x = p[16*me]
```



GPUs

Memory Coalescence

In this case, it might be faster to read coalesced chunks of memory into the block shared memory, and then have cores read their values from there

Awkward coding, but this is how you can get good performance

CUDA

```
#include <stdio.h>
__global__ void setarray(int p[])
{
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  p[k] = k*k;
}
int main(void)
{
  int i, *dm, m[1024];
  cudaMalloc(&dm, 1024*sizeof(int));
  setarray<<<16,64>>>(dm);
  cudaMemcpy(m, dm, 1024*sizeof(int),
             cudaMemcpyDeviceToHost);
  for (i = 0; i < 1024; i++)
    printf("m[%d] = %d\n", i, m[i]);
  return 0;
}
```

GPUs

CUDA

Back to the example: `dm` is the address of a chunk of memory on the device

The device memory is separate from the CPU memory, so we need special functions to allocate memory on the device

And we need explicit copies to get the data in and out of the coprocessor

GPUs

Memory

As always, data copies are time consuming, so we want to minimise them relative to computation time

We are used to the idea that the overhead can be so large that it is faster to do a computation sequentially on the CPU rather than send it to the GPU

The reverse is also true: if the data are on the GPU, it can be faster overall to use one of the wimpy GPU cores for a computation rather than copy back and forth to the CPU

This kind of computation vs. data movement judgement happens a lot when programming GPUs

GPUs

CUDA

In this example, we have only 16 blocks, so this would not be so good for a coprocessor with, say, 20 streaming multiprocessors

Real code would either simply have more blocks, or would interrogate the device to see how many multiprocessors it has and adjust accordingly

Exercise but you wouldn't want more than 32 blocks in our small example. Why?

GPUs

GPUs are becoming an ever more important method of computation

Even in phones: ARM's Mali GPU now has OpenCL support

GPUs are good for phones as they give a good amount of processing power for only a small amount of energy used

GPUs

OpenCL

OpenCL takes a wider view of computation than CUDA

While CUDA is explicitly about GPU computation, OpenCL tries to abstract away from the hardware and provide the programmer with a generic programming interface, independent of the underlying hardware

It tries hard not to assume there is a GPU coprocessor specifically, but just some “compute resource” coprocessor

OpenCL is provided as a library that is callable from standard C (and other languages), thus not needing a special compiler

GPUs

OpenCL

Things that CUDA has special syntax for (in particular kernel setup and launch) are done via normal function calls in OpenCL

OpenCL kernel code is kept in separate files from the C/C++ CPU code

Kernel code is read, compiled and executed by calling functions in the CPU code

Much like the shader code in OpenGL and the like

GPUs

OpenCL

In being generic, it is harder to use than CUDA, which does one thing well

CUDA can produce fast code, particularly if tuned to the specific hardware

But the hardware must be an NVIDIA card

Current OpenCL compilers produce code that runs universally but at sometimes uninspiring speeds (so code still needs the machine-specific tuning that OpenCL was supposed to avoid)

And there are features in the OpenCL programming model that reveal that the designers were still thinking of GPUs underneath the supposed genericity

GPUs

And there are many others. For example AMD have their *Radeon Open Compute platform* (ROCm) infrastructure

They have a language *Heterogeneous-Compute Interface for Portability* (HIP) that is very similar to CUDA and runs on AMD and NVIDIA hardware

In fact, they have a CUDA to HIP translator to aid porting code

GPUs

And Intel have *OneAPI*, with *Data Parallel C++*, (DPC++) also intended to be multi-architectural

They also have CUDA code migration tools

But it is clear each of CUDA, HIP and OneAPI are “best used” with the hardware of their respective developers

GPUs

In development is *WebGPU*, a JavaScript API for graphics and compute, providing a uniform Web interface to whatever is running underneath

Exercise Read about these

GPUs

Microsoft have their own versions of everything, of course

Their *DirectCompute* is not a million miles from CUDA, but is based on their DirectX suite

It runs on NVIDIA and AMD cards

But the portability to other operating systems is an open question

They also have C++ Accelerated Massive Parallelism (C++ AMP), an annotated version of C++ that is reputedly much easier to write code for

This is more like an OpenMP for GPUs

GPUs

OpenACC

In fact, there is also OpenACC, which is essentially OpenMP for GPUs

`pragma` annotations indicate code can be run on a GPU

```
#pragma acc parallel loop
  for (int i = 0; i < n; ++i) {
    z[i] = x[i] + y[i];
}
```

runs the loop on the GPU. The programmer does not have to think about copying data back and forth or writing and calling kernels

Exercise Is that a good or a bad thing?

GPUs

OpenACC

Similar to other systems, simply ignoring the `pragma` and running on the CPU will produce equivalent results

OpenACC does for accelerators (co-processors) what OpenMP does for multi-core

OpenMP and OpenACC pragmas can sit side-by-side in the same code

In fact, OpenACC is supposed to merge with OpenMP at some point, but progress seems slow

A freely available version of OpenACC for NVIDIA GPUs is available and GCC also supports it (but only on Nvidia and AMD): this may help OpenACC to become more popular

GPUs

GPUs have a great future ahead of them as they are excellent at certain kinds of problem, when programmed by really good programmers

There are CUDA bindings for Python and Java (of course), so you don't have to use C

Another item to note is that GPUs use (relatively) very little energy for the amount of processing they deliver

In a world where supercomputer centres spend more on electricity than they do on the computers themselves, the operations per watt that GPUs provide turns out to be very attractive

See the current Top 500

GPUs

If we were starting from scratch, we probably wouldn't design a GPU in the way it is

Just like the original CPU was based on existing integrated circuits that engineers noticed could be made programmable, the GPU is based on graphics co-processors that engineers noticed could be made programmable

So the accidents of history brought us to where we are today

GPUs

As previously mentioned, we are currently seeing multicore processors merging with GPUs

This is repeating the historical precedents of coprocessors merging with main processors

One processor that has had a lot of attention recently is the Apple M1

Coprocessors

Apple M1

The Apple M1 is a 4CPU+4CPU+GPU+NPU+memory ARM architecture system on a chip (SoC), using 16 billion transistors

- 4 fast CPU cores
- 4 energy efficient CPU cores
- 8 GPU cores (24,576 threads)
- up to 16GB memory in a *unified memory* architecture
- 16 core neural processing unit (NPU) (11 trillion ops/sec)
- a digital signal processor (DSP)
- an image processing unit (ISP)
- a video encoder/decoder

This takes coprocessing to new levels!

Coprocessors

Apple M1

The various units share memory in the unified memory architecture

This is 16GB memory, on the same chip

Note that on-chip memory is fast(er), but not expandable

Advanced Exercise Read about the memory consistency features used by the M1 to support compatability with x86 code

Coprocessors

Incidentally, Intel has its *Gaussian and Neural Accelerator* (GNA) integrated into the main CPU chip

Initially for support of speech recognition, it could probably be used for more general deep learning

ARM Mali

At the low-power end of the scale, ARM have their Mali core

“Core” in the sense of a chunk of silicon design that can be incorporated into other system chips

With current generations having up to 32 processing cores, this is a GPU you will find in your phone

It supports OpenCL and 64-bit floating point

As well as doing some graphics. . .

The End

End of Lectures

Future sessions will be problems classes, and going through past papers