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

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*

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)

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)

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*

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*

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

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*

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)

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

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

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

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”

Background

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

HPE is Hewlett Packard Enterprise

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

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

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

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

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

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

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

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

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

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

Classifications

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

A simple classification was devised by Flynn (1966)

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

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

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

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

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

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

Classifications

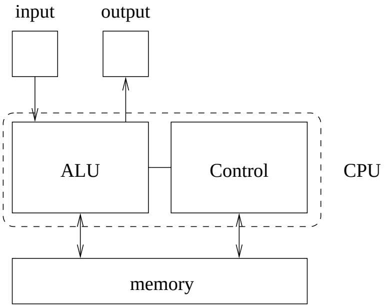
Of course, there are many more classifications we need to look at

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.



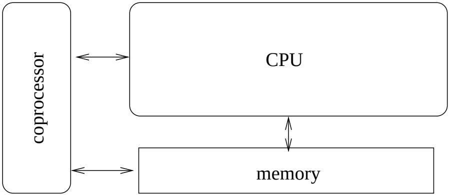
von Neumann Architecture

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

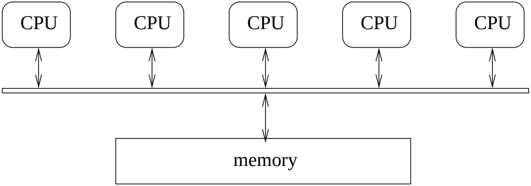
Multiprocessor

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

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

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

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

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

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

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

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

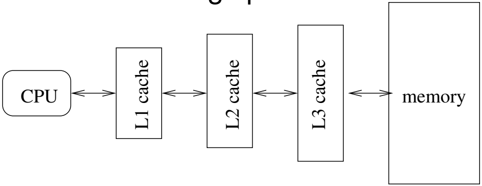
Classifications

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

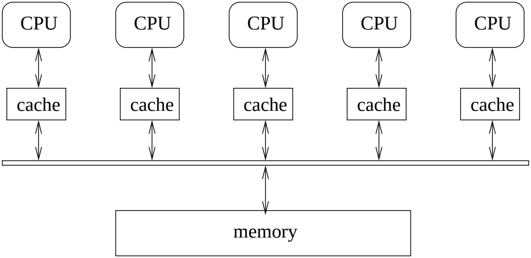


Levels of cache

Classifications

Shared Memory

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



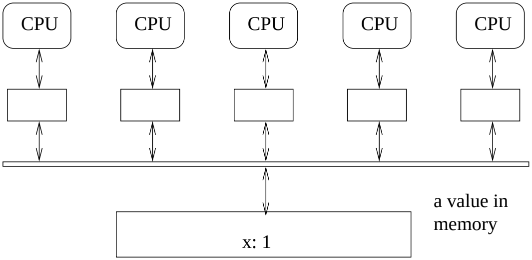
Memory caches

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

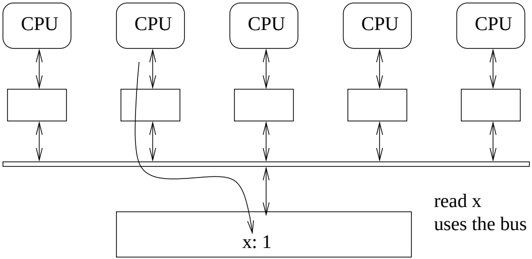


A value in memory

Classifications

Shared Memory

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

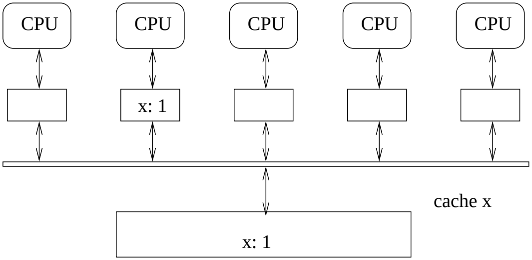


Read value

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

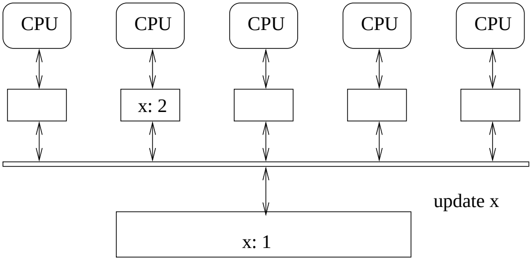


Copy in cache

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

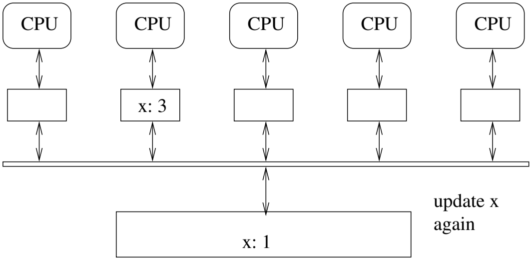


Update x (in cache)

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

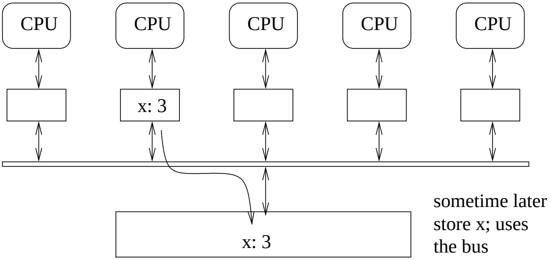


Update x again

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



Store x later

Classifications

Shared Memory

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

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

Classifications

Shared Memory

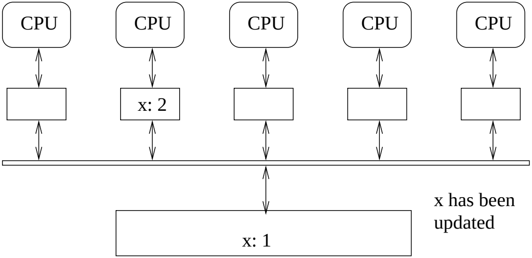
This reduces pressure on the shared bus: but now we have the problem of *cache coherence*

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

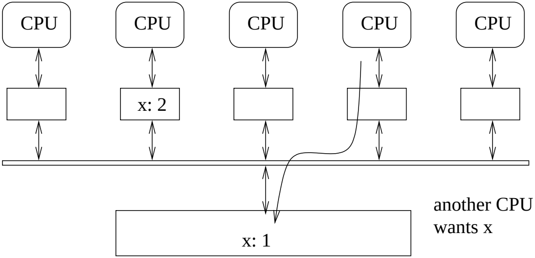
Shared Memory



x has been updated in cache

Classifications

Shared Memory



Another CPU wants x

Classifications

Shared Memory

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

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

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

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

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!

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

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

Classifications

Shared Memory

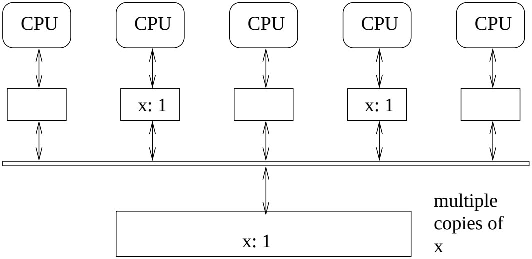
There are other ways to fail, too

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

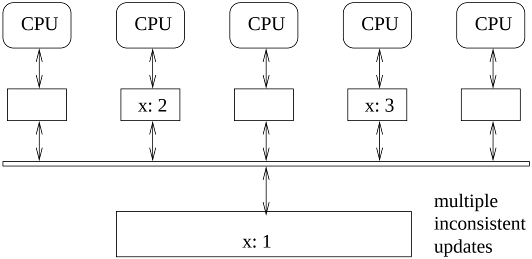
Shared Memory



Multiple copies of x

Classifications

Shared Memory



Multiple inconsistent copies

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

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

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)

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

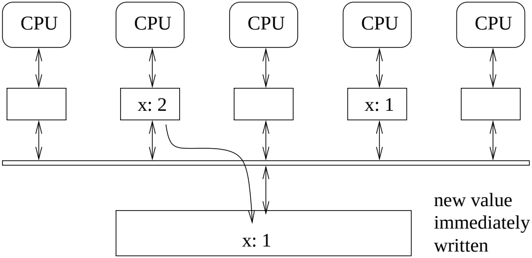
This might be done in several ways

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

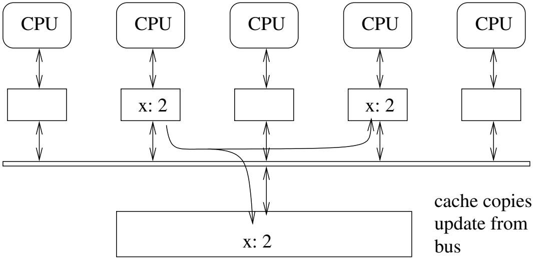
Shared Memory



New value immediately written to memory

Classifications

Shared Memory



Caches copy update from bus

Classifications

Shared Memory

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

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

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

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

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

Classifications

Shared Memory

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

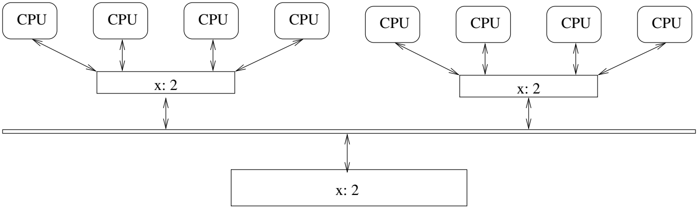
You could use very fast buses and main memory: not a solution due to cost

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

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

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