Classifications

We need to classify these kinds of parallelism

- Single Instruction, Single Data (SISD). Traditional, von Neumann, uniprocessor machines
- Single Instruction, Multiple Data (SIMD). As in a vector processor. Multiple processors all doing the same operation, but on different datastreams
- Multiple Instruction, Multiple Data (MIMD). Multiple processors doing different things to different datastreams.

What most people (wrongly) think parallel computing is all about
Classifications

We need to classify these kinds of parallelism

A simple classification was devised by Flynn (1966)
Classifications

We need to classify these kinds of parallelism

A simple classification was devised by Flynn (1966)

- Single Instruction, Single Data (SISD). Traditional, von Neumann, uniprocessor machines
Classifications

We need to classify these kinds of parallelism

A simple classification was devised by Flynn (1966)

- Single Instruction, Single Data (SISD). Traditional, von Neumann, uniprocessor machines
- Single Instruction, Multiple Data (SIMD). As in a vector processor. Multiple processors all doing the same operation, but on different data streams
We need to classify these kinds of parallelism

A simple classification was devised by Flynn (1966)

- Single Instruction, Single Data (SISD). Traditional, von Neumann, uniprocessor machines
- Single Instruction, Multiple Data (SIMD). As in a vector processor. Multiple processors all doing the same operation, but on different datastreams
- Multiple Instruction, Multiple Data (MIMD). Multiple processors 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.
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

Flynn’s classification is nice and simple, so people have extended it further, in particular sub-dividing MIMD
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 processors in *lockstep*, so every processor is executing the same instruction. SPMD runs the same program (on different data) on a MIMD machine, with each processor going their own way, particularly on loops and conditionals
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 processors in *lockstep*, so every processor is executing the same instruction. SPMD runs the same program (on different data) on a MIMD machine, with each processor going their own way, particularly on loops and conditionals.

- Multiple Program Multiple Data (MPMD). A MIMD machine not running SPMD. So each processor 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.
Of course, there are many more classifications we need to look at.

We can think of how the parts of the architecture are connected.
A uniprocessor or sequential processor is the traditional von Neumann architecture of a single CPU, memory, etc.

A hugely successful model that enabled the computer revolution to take place
A *coprocessor* is a non-general processor used as a worker by the processor

Currently very popular in the form of graphics cards
A *multiprocessor* is a loose term applying to most parallel architectures, except possibly SIMD (which usually doesn’t have multiple full processors)
A multiprocessor has *shared memory* when the processors access memory on a shared bus.

Processors share each other’s data: if one processor modifies the value of a variable, the other processors see that change.
A multiprocessor has *shared memory* when the processors access memory on a shared bus.

Processors share each other's data: if one processor modifies the value of a variable, the other processors see that change.

In reality, the shared bus can be a lot more complicated, e.g., a tree or ring structure.
This is possibly what most people think of as a typical parallel architecture
This is possibly what most people think of as a typical parallel architecture.

Unfortunately, it has a lot of problems as an architecture.
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.
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 processors all trying to access memory simultaneously it gets much worse.
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
So shared memory machines try to cut down the traffic on the bus by using caches.

Each processor has its own chunk of cache memory: this cuts down on use of the bus.
If a processor 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.

x: 1

a value in memory
If a processor 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.
If a processor 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.
If a processor 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.
If a processor 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.
If a processor 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.
This reduces pressure on the shared bus: but now we have the problem of *cache coherence*
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.
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 processor wants to read the value before the updated version has been written back, it will get the old value
Classifications

Shared Memory

CPU CPU CPU CPU CPU
x: 1
x: 2
x has been updated

x: 1
Classifications

Shared Memory

CPU
CPU
CPU
CPU
CPU

x: 1

another CPU
wants x

x: 2
Even worse, dependent on timing, you don’t know if the first CPU has written the value back or not
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
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.
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
Not what we want, as we can’t control the vagaries of hardware operation.

You might get the right answer hundreds of times; it doesn’t mean your program is correct!
Not what we want, as we can’t control the vagaries of hardware operation

You might get the right answer hundreds of times; 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
There are other ways to fail, too
There are other ways to fail, too

Others processors might be doing the same: reading and updating the value. Thus there can be several differing copies of what is supposed to be the same variable in different caches.
There are other ways to fail, too

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

When one processor updates the variable the other processors will still be using their own in their caches.
Classifications
Shared Memory

CPU
x: 1

CPU
x: 2

CPU
x: 3

multiple inconsistent
updates

CPU
x: 1
The *cache coherence* problem is the issue of trying to make sure all cached copies of a variable are kept in sync.
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.
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, when 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, when 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 as every write must go through the bus.
Classifications

Shared Memory

new value immediately written
Classifications

Shared Memory

CPU

x: 2

Cache copies

update from

bus

CPU

x: 2

CPU

x: 2

CPU

x: 2

CPU
Classifications
Shared Memory

But this is better than you might imagine as typical code reads values much more than it updates values.
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.
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.
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. Shared state is bad design (more on this later).
Other solutions might be to try to balance the memory/cpu speed disparity
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.
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:

Does this solve the problem?
Unfortunately, *symmetric* shared memory does not scale well, perhaps under 100 cores, with complex hardware support in the caches.
Unfortunately, *symmetric* shared memory does not scale well, perhaps under 100 cores, with complex hardware support in the caches.

Intel’s *Xeon Phi* has 72 cores, with sophisticated cache coherence and a fast ring bus connecting processors to each other and to memory.
Unfortunately, *symmetric* shared memory does not scale well, perhaps under 100 cores, with complex hardware support in the caches.

Intel's *Xeon Phi* has 72 cores, with sophisticated cache coherence and a fast ring bus connecting processors to each other and to memory.

(Some might argue this is the slow processor approach... )
Unfortunately, *symmetric* shared memory does not scale well, perhaps under 100 cores, with complex hardware support in the caches.

Intel’s *Xeon Phi* has 72 cores, with sophisticated cache coherence and a fast ring bus connecting processors to each other and to memory.

(Some might argue this is the slow processor approach. . . )

And you might also see arguments that the Phi is not truly symmetric.
Exercise. Read about cache coherence mechanisms: snoopy caches; directory based; snarfing
Symmetric shared memory is the model that current small machines (multicore PCs) use
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