Classifications

Vectors

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

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

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

Classifications

Vectors

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

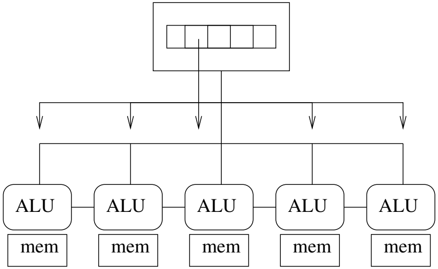
For reads: as the cores are all doing the same thing, if one requests a global shared value from a fixed shared memory location, then all of them are doing the same

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

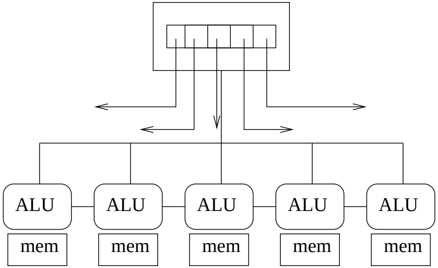


One read goes to all cores

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

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*

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

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 *k*th 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

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 *k*th core is often optimised by the hardware using *coalescence*

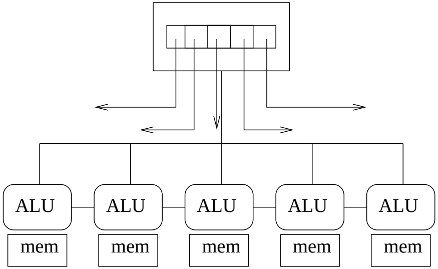
Using a wide bus (e.g., 512 bits) a *single* read operation can fetch multiple data (e.g., 16 integers) and put them all on the bus simultaneously

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

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

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

Classifications

Vectors

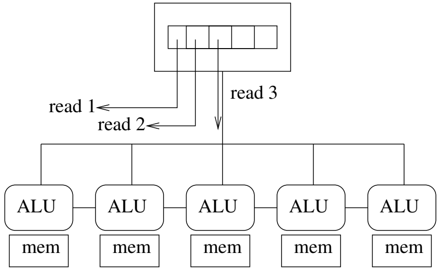
However, it needs data accesses in the program to be of certain patterns for this to work, e.g., linear access to an array

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 *k*th slot in an array could be coalesced

Classifications

Vectors

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

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

Classifications

Vectors

Similarly for writes: e.g., core *k* writing a value to the *k*th 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

Classifications

Vectors

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

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

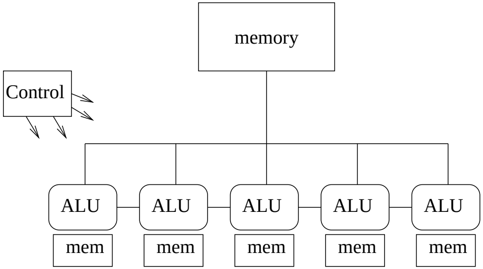
Exercise Consider the case of indirecting through a pointer to global memory (a) when each core points to the same location and (b) when each core points to a different location in the global memory

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

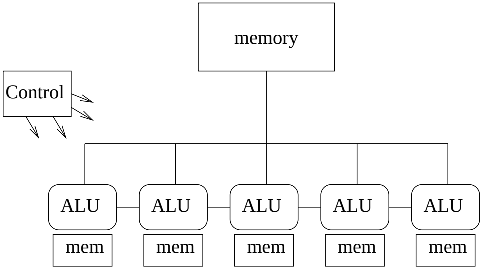


Neighbour connections

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

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

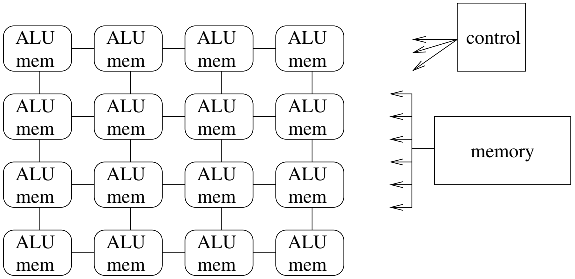
Arrays

An extension of the idea was the *array processor*

Classifications

Arrays

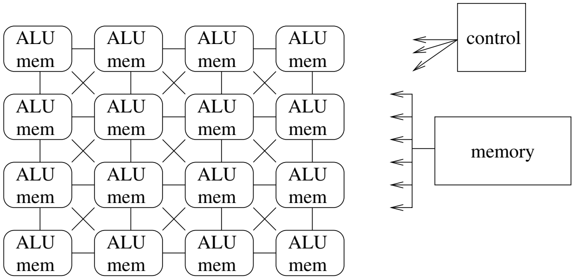
An extension of the idea was the *array processor*



Array processor

Classifications
Arrays

An extension of the idea was the *array processor*



Array with diagonal connections

Classifications

Arrays

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

Classifications

Arrays

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

Fast connections in two or more directions

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

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

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

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

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

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

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*

Classifications

Pipelines, Systolic Arrays

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

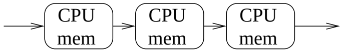
These generalise CPU instruction pipelines to processes

Classifications

Pipelines, Systolic Arrays

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

These generalise CPU instruction pipelines to processes



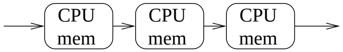
Process pipeline

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

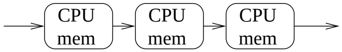
The CPUs are independent (MIMD/MPMD), each performing one step in the transformation of the input data

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

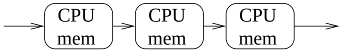
More often found in hardware to solve specific problems; not often found as a generic machine

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

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

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

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

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

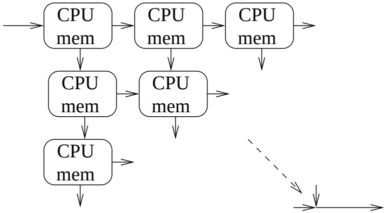
Used in graphics coprocessors as another form of parallelism

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?

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

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?

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

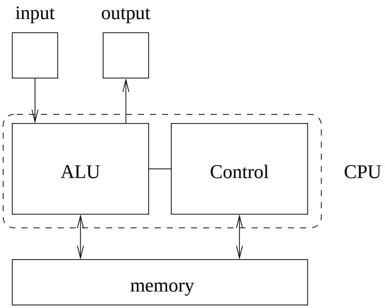
Is there a model that encapsulates multiprocessors in the same way?

There are many contenders, but no obvious winner

Classifications

Extensions of von Neumann

We have the original von Neumann 5 box model

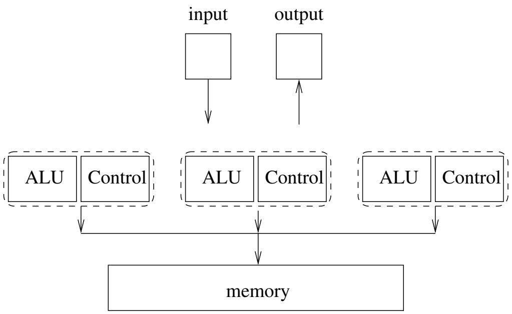


von Neumann 5 box model

Classifications

Extensions of von Neumann

Shared memory MIMD

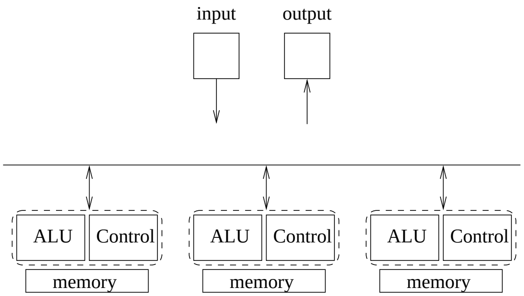


Shared memory box model

Classifications

Extensions of von Neumann

Distributed memory MIMD

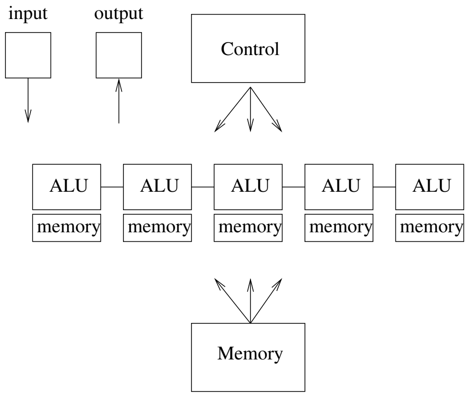


Distributed memory box model

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

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

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**

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

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:

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

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

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:

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

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

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

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

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

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

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

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**

Classifications

Extensions of von Neumann

Next: **BSP**

The *Bulk Synchronous Parallel* model

Classifications

Extensions of von Neumann

Next: **BSP**

The *Bulk Synchronous Parallel* model

This model takes communication time into account

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

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*

Classifications

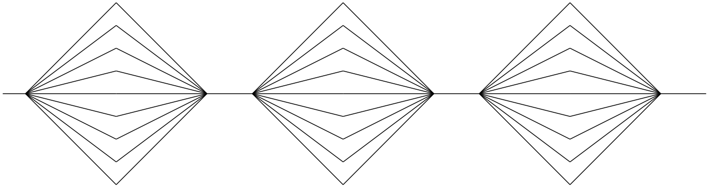
Extensions of von Neumann

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

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

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

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

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

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

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

So we need to look at how to analyse parallel algorithms

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

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”

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

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

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

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

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

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

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

Analysis

Speedup

Usually S_p is much smaller than p for several reasons

Firstly, there is communications overheads between processors

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

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

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

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

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!

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