Classifications

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

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

Each processor also has its own chunk of local memory that it operates on (data parallel).
The local memory allows each ALU to work on a different set of values
The local memory allows each ALU to work on a different set of values

Note: this is *not* cache, but simply per-ALU memory
Cache memory: a fast local copy of a slower memory location. If there are caches on different cores, we want them all to contain the same value for a given variable.
Classifications
Cache vs Local

Cache memory: a fast local copy of a slower memory location. If there are caches on different cores, we want them all to contain the same value for a given 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.
In a vector processor, the bottleneck to the shared memory still needs thinking about
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 the shared memory, then all of them are doing the same.
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 the shared memory, 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.
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.
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.
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 $k$ item to the $k$th core is often optimised by the hardware using *coalescence*.
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 $k$ 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 send them to the relevant cores (16 of them in one transfer).
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 $k$ 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 send them to the relevant cores (16 of them in one transfer).

The next 16 in the next transfer; and so on.
However, it needs data accesses in the program to be of certain patterns for this to work.
Classifications

Vectors

However, it needs data accesses in the program to be of certain patterns for this to work otherwise, the reads cannot be coalesced and might require many (e.g., 16) individual reads: much slower
However, it needs data accesses in the program to be of certain patterns for this to work.

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.
Similarly for writes: core $k$ writing a value to the $k$th slot in an array
Classifications

Vectors

Similarly for writes: core $k$ writing a value to the $k$th slot in an array

Multiple writes to a single location make no sense and are often disallowed by the system
Similarly for writes: core $k$ writing a value to the $k$th slot in an array

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
(a) when it’s pointing to the same location on all processors and
(b) when it’s a pointing to a different location on each processor
Often there is fast direct communications between neighbouring CPUs.
Often there is fast direct communications between neighbouring CPUs

This allows data to shuffle up and down the vector very quickly: many problems (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
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 fairly easy to build and replicate, while control units are hard
An extension of the idea was the array processor
An extension of the idea was the *array processor*
An extension of the idea was the *array processor*
The CPUs are in SIMD lockstep as before, but now in an array
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
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.
Early array processors were very simple, but they became bigger as technology advanced.
Early array processors were very simple, but they became bigger as technology advanced.

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>CPUs</th>
<th>mem/CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAP</td>
<td>1979</td>
<td>1 bit</td>
<td>4k</td>
</tr>
<tr>
<td>CM</td>
<td>1983</td>
<td>1 bit</td>
<td>64k</td>
</tr>
<tr>
<td>MPP</td>
<td>1983</td>
<td>1 bit</td>
<td>16k</td>
</tr>
<tr>
<td>MasPar</td>
<td>1990</td>
<td>4 bit</td>
<td>16k</td>
</tr>
<tr>
<td>MasParII</td>
<td>1992</td>
<td>32 bit</td>
<td>64k</td>
</tr>
</tbody>
</table>

DAP: ICL Distributed Array Processor
CM: Connection Machine (pretty lights)
MPP: Goodyear Massively Parallel Processor
Despite being very wimpy processors, this was made up by having so many of them
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.
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.
Similar looking, but actually quite different are *systolic arrays*.
Similar looking, but actually quite different are *systolic arrays*

These generalise instruction pipelines to processes
Similar looking, but actually quite different are *systolic arrays*

These generalise instruction pipelines to processes

The CPUs are independent (MIMD/MPMD), each performing one step in the transformation of the input data
Similar looking, but actually quite different are *systolic arrays*

These generalise instruction pipelines to processes

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
Similar looking, but actually quite different are *systolic arrays*

These generalise instruction pipelines to processes

![Diagram of CPU and memory](image)

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

Each step separate, but compute intensive.
For example, a graphic 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 graphic 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
For example, a graphic 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
Systolic arrays are the obvious extension

but examples are rare to non-existent