MPI

Using MPI requires careful thought about messages to get the maximum efficiency out of the system

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

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

MPI

This requires careful programming, but can give good results

Sometimes not

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

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

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:

MPI

Also:

- messaging has a high overhead, so MPI only really works well on very large programs

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

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

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

MPI

MPI has succeeded for many reasons

- An open standard, inviting several competing implementations

MPI

MPI has succeeded for many reasons

- An open standard, inviting several competing implementations
- Thus implementations tend to be optimised and efficient

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

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

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

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

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

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

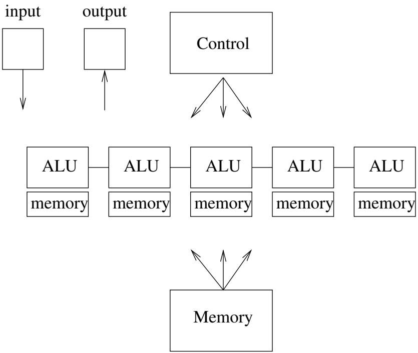
Vector and Array Processors

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

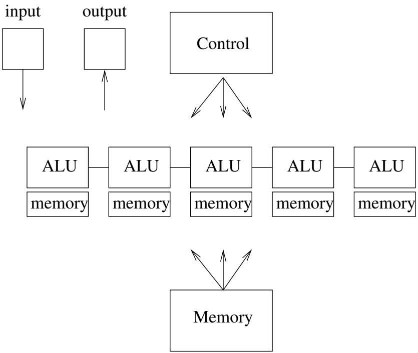
Vector and Array Processors



SIMD box model

All processors are controlled by just one Control unit, so are all executing the same instruction

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

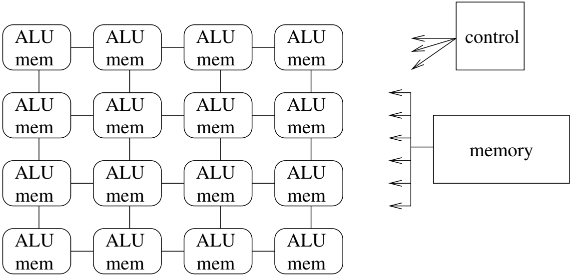
There is a shared chunk of *global memory* and each processor has its own chunk of *private memory*

Vector and Array Processors

There is a shared chunk of *global memory* and each processor has its own chunk of *private memory*

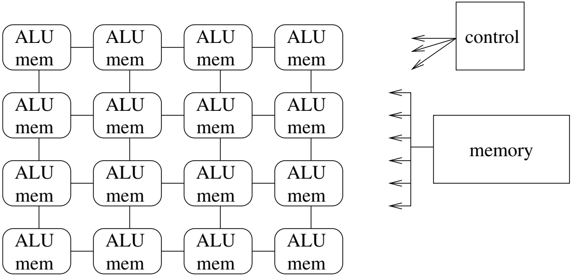
Processors can be strung linearly in a *vector* or in a square mesh as an *array*

Vector and Array Processors



Array processor

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

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

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];`

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

Vector and Array Processors

The important points being

- all elements in the arrays are being treated identically

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

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

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

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

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];`

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

Vector and Array Processors

Starting with `a` = 1, 1, 1, 1; the sequential loop gives

1	2	1	1
1	2	3	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

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	

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

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

Vector and Array Processors

Having given a warning, SIMD processing is very powerful

Vectors and arrays with thousands of processors are common

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

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

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

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

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)

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

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

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