Vector and Array Processors

Back to the SIMD architecture: now is the point where need to talk about an interesting feature of SIMD processing

Vector and Array Processors

Back to the SIMD architecture: now is the point where need to talk about an interesting feature of SIMD processing

The main feature of SIMD is that all processors are doing the same thing…

Vector and Array Processors

Back to the SIMD architecture: now is the point where need to talk about an interesting feature of SIMD processing

The main feature of SIMD is that all processors are doing the same thing…

… so how can conditionals work?

Vector and Array Processors

Back to the SIMD architecture: now is the point where need to talk about an interesting feature of SIMD processing

The main feature of SIMD is that all processors are doing the same thing…

… so how can conditionals work?

Here is an example, written using a fictional SIMD C

Vector and Array Processors

Suppose we have a `get_proc()` function ("get processor number") that returns the index of the processor:

```
int me;
me = get_proc();
...
```

This allows us to distinguish between processors; the value of `me` is different on each processor

Vector and Array Processors

Suppose we have a `get_proc()` function (“get processor number”) that returns the index of the processor:

```
int me;
me = get_proc();
...
```

This allows us to distinguish between processors; the value of `me` is different on each processor

We could use `me` to index into a vector, so each processor operates on a different element

Vector and Array Processors

Suppose we have a `get_proc()` function ("get processor number") that returns the index of the processor:

```
int me;
me = get_proc();
...
```

This allows us to distinguish between processors; the value of `me` is different on each processor

We could use `me` to index into a vector, so each processor operates on a different element

```
v[me] = (v[me - 1] + v[me + 1])/2.0;
```

Vector and Array Processors

So what does this code do?

```
int me, n;

me = get_proc();

if (me > 512) {
  n = 1;
}
else {
  n = -1;
}
```

Vector and Array Processors

Instinctively you think it sets `n` in processors above 512 to 1 and in the other processors `n` is set to -1

Vector and Array Processors

Instinctively you think it sets `n` in processors above 512 to 1 and in the other processors `n` is set to -1

And this is what it does do

Vector and Array Processors

Instinctively you think it sets `n` in processors above 512 to 1 and in the other processors `n` is set to -1

And this is what it does do

But a SIMD machine executes the same code in all processors, so how can it execute the `n = 1` assignment on some and the `n = -1` assignment on others?

Vector and Array Processors

It doesn't: at any point in time each processor is executing the current instruction

Vector and Array Processors

It doesn’t: at any point in time each processor is executing the current instruction

or doing nothing at all

Vector and Array Processors

It doesn’t: at any point in time each processor is executing the current instruction

or doing nothing at all

Processors can be *inhibited*, meaning not participating in the current instruction

Vector and Array Processors

It doesn’t: at any point in time each processor is executing the current instruction

or doing nothing at all

Processors can be *inhibited*, meaning not participating in the current instruction

There is a per-processor inhibit flag to say whether this processor is on or off

Vector and Array Processors

It doesn't: at any point in time each processor is executing the current instruction

or doing nothing at all

Processors can be *inhibited*, meaning not participating in the current instruction

There is a per-processor inhibit flag to say whether this processor is on or off

This is how we get different code paths on different processors

Vector and Array Processors

We must modify our description of SIMD machines:

> *Each processor either executes the same instruction as the others; or does nothing at all*

Vector and Array Processors

Returning to the code

```
if (me > 512) {
  n = 1;
}
else {
  n = -1;
}
```

Vector and Array Processors

Returning to the code

```
if (me > 512) {
  n = 1;
}
else {
  n = -1;
}
```

This is executed as follows:

Vector and Array Processors

Returning to the code

```
if (me > 512) {
  n = 1;
}
else {
  n = -1;
}
```

This is executed as follows:

- All processors execute the test in the `if`

Vector and Array Processors

Returning to the code

```
if (me > 512) {
  n = 1;
}
else {
  n = -1;
}
```

This is executed as follows:

- All processors execute the test in the `if`
- In those processors for which the test fails, the inhibit flag is set

Vector and Array Processors

Returning to the code

```
if (me > 512) {
  n = 1;
}
else {
  n = -1;
}
```

This is executed as follows:

- All processors execute the test in the `if`
- In those processors for which the test fails, the inhibit flag is set
- All processors move to the `n = 1`; the inhibited processors do nothing while the others execute the assignment

Vector and Array Processors

- All processors move to the `else`; all inhibit flags are inverted

Vector and Array Processors

- All processors move to the `else`; all inhibit flags are inverted
- All processors move to the `n = -1`; the inhibited processors do nothing while the others execute the assignment

Vector and Array Processors

- All processors move to the `else`; all inhibit flags are inverted
- All processors move to the `n = -1`; the inhibited processors do nothing while the others execute the assignment
- All inhibit flags are cleared

Vector and Array Processors

- All processors move to the `else`; all inhibit flags are inverted
- All processors move to the `n = -1`; the inhibited processors do nothing while the others execute the assignment
- All inhibit flags are cleared
- All processors move on to after the `if`

Vector and Array Processors

- All processors move to the `else`; all inhibit flags are inverted
- All processors move to the `n = -1`; the inhibited processors do nothing while the others execute the assignment
- All inhibit flags are cleared
- All processors move on to after the `if`

Both branches of an `if` always taken by all processors!

Vector and Array Processors

Proc	0	1	2	...	513	514	515	...
inhibit	F	F	F		F	F	F	
	`n`	`n`	`n`		`n`	`n`	`n`	
	0	0	0	...	0	0	0	...

Vector and Array Processors

Proc	0	1	2	. . .	513	514	515	. . .
inhibit	T	T	T		F	F	F	
	`n`	`n`	`n`		`n`	`n`	`n`	
`if` (`me` $>$ `512`)	0	0	0	. . .	0	0	0	. . .

Vector and Array Processors

Proc	0	1	2	...	513	514	515	...
inhibit	T	T	T		F	F	F	
	`n`	`n`	`n`		`n`	`n`	`n`	
`n = 1`	0	0	0	...	1	1	1	...

Vector and Array Processors

Proc	0	1	2	...	513	514	515	...
inhibit	F	F	F		T	T	T	
	`n`	`n`	`n`		`n`	`n`	`n`	
`else`	0	0	0	...	1	1	1	...

Vector and Array Processors

Proc	0	1	2	...	513	514	515	...
inhibit	F	F	F		T	T	T	
	`n`	`n`	`n`		`n`	`n`	`n`	
$\mathtt{n} = -1$	-1	-1	-1	...	1	1	1	...

Vector and Array Processors

Proc	0	1	2	...	513	514	515	...
inhibit	F	F	F		F	F	F	
	`n`	`n`	`n`		`n`	`n`	`n`	
after	-1	-1	-1	...	1	1	1	...

Vector and Array Processors

The time taken for an `if` is the sum of the times of both branches

Vector and Array Processors

The time taken for an `if` is the sum of the times of both branches

Quite different from sequential code

Vector and Array Processors

The time taken for an `if` is the sum of the times of both branches

Quite different from sequential code

Reality is a little more complicated: think about nested `if`s

Vector and Array Processors

The time taken for an `if` is the sum of the times of both branches

Quite different from sequential code

Reality is a little more complicated: think about nested `ifs`

There is actually a *stack* of inhibit flags!

Vector and Array Processors

The time taken for an `if` is the sum of the times of both branches

Quite different from sequential code

Reality is a little more complicated: think about nested `ifs`

There is actually a *stack* of inhibit flags!

Exercise Think this through for yourself!

Vector and Array Processors

This seems like poor use of our processors if lots of them are inhibited

Vector and Array Processors

This seems like poor use of our processors if lots of them are inhibited

True, so SIMD code should be written to minimise conditional branches

Vector and Array Processors

This seems like poor use of our processors if lots of them are inhibited

True, so SIMD code should be written to minimise conditional branches

But with thousands of CPUs, processing power is cheap, so inhibiting some of them is not as bad as it seems, as long as it is not overdone

Vector and Array Processors

This seems like poor use of our processors if lots of them are inhibited

True, so SIMD code should be written to minimise conditional branches

But with thousands of CPUs, processing power is cheap, so inhibiting some of them is not as bad as it seems, as long as it is not overdone

```
if (me > 512) foo();
else bar();
```

Vector and Array Processors

This seems like poor use of our processors if lots of them are inhibited

True, so SIMD code should be written to minimise conditional branches

But with thousands of CPUs, processing power is cheap, so inhibiting some of them is not as bad as it seems, as long as it is not overdone

```
if (me > 512) foo();
else bar();
```

is not good code: all of `foo` must be executed before `bar` can start, so there is a large amount of inhibition

Vector and Array Processors

Inhibition applies to all conditional code, like loops:

```
int i, n;
...
for (i = 0; i < n; i++) {
  ...
}
```

Vector and Array Processors

Inhibition applies to all conditional code, like loops:

```
int i, n;
...
for (i = 0; i < n; i++) {
  ...
}
```

All processors start the loop

Vector and Array Processors

Inhibition applies to all conditional code, like loops:

```
int i, n;
...
for (i = 0; i < n; i++) {
  ...
}
```

All processors start the loop

As `i` increases, some processors pass their exit test and are inhibited; other processors continue executing; *all processors continue looping*

Vector and Array Processors

Inhibition applies to all conditional code, like loops:

```
int i, n;
...
for (i = 0; i < n; i++) {
  ...
}
```

All processors start the loop

As `i` increases, some processors pass their exit test and are inhibited; other processors continue executing; *all processors continue looping*

Note no processor starts executing after the loop until *all* processors have exited

Vector and Array Processors

Loops must wait until all processors have completed: they take time the maximum of the individual processors

Vector and Array Processors

Loops must wait until all processors have completed: they take time the maximum of the individual processors

SIMD loops are most efficient when all the loops are of the same size

Vector and Array Processors

Loops must wait until all processors have completed: they take time the maximum of the individual processors

SIMD loops are most efficient when all the loops are of the same size

Similarly for all conditional constructs: if there is a choice all processors will take all the choices, but some are appropriately inhibited

Vector and Array Processors

Connection Machines had a lightbulb per processor: initially they set it so the light was on when the processor was active

Vector and Array Processors

Connection Machines had a lightbulb per processor: initially they set it so the light was on when the processor was active

After a while they fixed it so the light was on when the processor was inhibited…

Vector and Array Processors

Connection Machines had a lightbulb per processor: initially they set it so the light was on when the processor was active

After a while they fixed it so the light was on when the processor was inhibited…

We shall return to SIMD programming with CUDA, later, when we talk about parallel languages

End of Architectures

We have seen a variety of machine architectures, but primarily people use:

- shared memory
- distributed memory
- SIMD

End of Architectures

We have seen a variety of machine architectures, but primarily people use:

- shared memory
- distributed memory
- SIMD

Quite often, all at once!

End of Architectures

We have seen a variety of machine architectures, but primarily people use:

- shared memory
- distributed memory
- SIMD

Quite often, all at once!

It is time to move from the machines to the code running on them

Parallel Algorithms

We now turn to parallel *algorithms*

Parallel Algorithms

We now turn to parallel *algorithms*

We shall approach them in two ways

Parallel Algorithms

We now turn to parallel *algorithms*

We shall approach them in two ways

- general principles

Parallel Algorithms

We now turn to parallel *algorithms*

We shall approach them in two ways

- general principles
- specific examples

Parallel Algorithms

We now turn to parallel *algorithms*

We shall approach them in two ways

- general principles
- specific examples

The first will look at a few general techniques and some classic problems in parallelism

Parallel Algorithms

We now turn to parallel *algorithms*

We shall approach them in two ways

- general principles
- specific examples

The first will look at a few general techniques and some classic problems in parallelism

The second will be a couple of specific algorithms, such as a parallel sort

Parallel Algorithms

Divide and Conquer

Perhaps the simplest way to parallelise a problem is *divide and conquer*

Parallel Algorithms

Divide and Conquer

Perhaps the simplest way to parallelise a problem is *divide and conquer*

- subdivide the problem into smaller parts

Parallel Algorithms

Divide and Conquer

Perhaps the simplest way to parallelise a problem is *divide and conquer*

- subdivide the problem into smaller parts
- process the parts in parallel

Parallel Algorithms

Divide and Conquer

Perhaps the simplest way to parallelise a problem is *divide and conquer*

- subdivide the problem into smaller parts
- process the parts in parallel
- merge the results back together

Parallel Algorithms

Divide and Conquer

Perhaps the simplest way to parallelise a problem is *divide and conquer*

- subdivide the problem into smaller parts
- process the parts in parallel
- merge the results back together

Of course, this only applies if you have a problem that you *can* subdivide!

Parallel Algorithms

Divide and Conquer

Perhaps the simplest way to parallelise a problem is *divide and conquer*

- subdivide the problem into smaller parts
- process the parts in parallel
- merge the results back together

Of course, this only applies if you have a problem that you *can* subdivide!

And it works best if the parts are independent of each other: less communication

Parallel Algorithms

Divide and Conquer

For example, summing *n* values becomes

Parallel Algorithms

Divide and Conquer

For example, summing *n* values becomes

- subdivide the values into smaller chunks, sending the chunks to separate processors

Parallel Algorithms

Divide and Conquer

For example, summing *n* values becomes

- subdivide the values into smaller chunks, sending the chunks to separate processors
- each processor sums its chunk (process in parallel)

Parallel Algorithms

Divide and Conquer

For example, summing *n* values becomes

- subdivide the values into smaller chunks, sending the chunks to separate processors
- each processor sums its chunk (process in parallel)
- return the results to the main processor and add the values together (merge)

Parallel Algorithms

Divide and Conquer

Question: how big should the chunks be?

Parallel Algorithms

Divide and Conquer

Question: how big should the chunks be?

Too small and we spend all our time in communication overhead; plus the merge step gets bigger

Parallel Algorithms

Divide and Conquer

Question: how big should the chunks be?

Too small and we spend all our time in communication overhead; plus the merge step gets bigger

Too large, thus fewer chunks, and we might not get the parallelism we want

Parallel Algorithms

Granularity

This is the question of *granularity*, or “chunk size”

Parallel Algorithms

Granularity

This is the question of *granularity*, or “chunk size”

A big problem in programming parallelism is deciding on the choice of granularity of a sub-problem, for exactly the reasons given above

Parallel Algorithms

Granularity

This is the question of *granularity*, or “chunk size”

A big problem in programming parallelism is deciding on the choice of granularity of a sub-problem, for exactly the reasons given above

Computing a single sum is a small grain; while averaging a row of a large matrix is a big grain

Parallel Algorithms

Granularity

This is the question of *granularity*, or “chunk size”

A big problem in programming parallelism is deciding on the choice of granularity of a sub-problem, for exactly the reasons given above

Computing a single sum is a small grain; while averaging a row of a large matrix is a big grain

The former you might not want to parallelise; the latter you would

Parallel Algorithms

Granularity

Grain size: the size of a chunk

Parallel Algorithms

Granularity

Grain size: the size of a chunk

You will see “small grain” and “large grain”; alternatively “fine grain” and “coarse grain”

Parallel Algorithms

Granularity

Grain size: the size of a chunk

You will see “small grain” and “large grain”; alternatively “fine grain” and “coarse grain”

Granularity: the ability of a problem (data or computation) to be divided into fine or only coarse grains

Parallel Algorithms

Granularity

Grain size: the size of a chunk

You will see “small grain” and “large grain”; alternatively “fine grain” and “coarse grain”

Granularity: the ability of a problem (data or computation) to be divided into fine or only coarse grains

Some programs may only admit a coarse granularity

Parallel Algorithms

Granularity

Grain size: the size of a chunk

You will see “small grain” and “large grain”; alternatively “fine grain” and “coarse grain”

Granularity: the ability of a problem (data or computation) to be divided into fine or only coarse grains

Some programs may only admit a coarse granularity

Some may admit a fine grain, but should we split it up into small grains?

Parallel Algorithms

Granularity

Fine: more parallelism, more communications

Coarse: less parallelism, less communications

Parallel Algorithms

Granularity

It’s the grey area in the middle that is the issue: how large should a grain be before we consider running it in parallel?

Parallel Algorithms

Granularity

It’s the grey area in the middle that is the issue: how large should a grain be before we consider running it in parallel?

The answer: it depends

Parallel Algorithms

Granularity

It's the grey area in the middle that is the issue: how large should a grain be before we consider running it in parallel?

The answer: it depends

On everything, but particularly the ratio of computation time to communications speed on the particular hardware we have

Parallel Algorithms

Granularity

For fast communications (shared memory, perhaps) we would chop our problem up into relatively small grains

Parallel Algorithms

Granularity

For fast communications (shared memory, perhaps) we would chop our problem up into relatively small grains

For slow communications (distributed memory, perhaps) the sub-problems need to be larger before we benefit from parallelising

Parallel Algorithms

Granularity

For fast communications (shared memory, perhaps) we would chop our problem up into relatively small grains

For slow communications (distributed memory, perhaps) the sub-problems need to be larger before we benefit from parallelising

Often, the best way of working it out is just to try some test programs and measure the result