Parallel Algorithms

Reduction

There are a couple of issues, however

Parallel Algorithms

Reduction

There are a couple of issues, however

In real implementations we need to worry about the cost of data movement between processors: reduction inherently needs to move data around

Parallel Algorithms

Reduction

There are a couple of issues, however

In real implementations we need to worry about the cost of data movement between processors: reduction inherently needs to move data around

Probably a small cost for a shared memory system, but it can easily be much larger than the cost of the reduction operation if you are not careful

Parallel Algorithms

Reduction

There are a couple of issues, however

In real implementations we need to worry about the cost of data movement between processors: reduction inherently needs to move data around

Probably a small cost for a shared memory system, but it can easily be much larger than the cost of the reduction operation if you are not careful

So parallel reduction on, say, a distributed memory machine, is only worthwhile for large datasets

There are a couple of issues, however

In real implementations we need to worry about the cost of data movement between processors: reduction inherently needs to move data around

Probably a small cost for a shared memory system, but it can easily be much larger than the cost of the reduction operation if you are not careful

So parallel reduction on, say, a distributed memory machine, is only worthwhile for large datasets

Or a very costly reduction operation

There are a couple of issues, however

In real implementations we need to worry about the cost of data movement between processors: reduction inherently needs to move data around

Probably a small cost for a shared memory system, but it can easily be much larger than the cost of the reduction operation if you are not careful

So parallel reduction on, say, a distributed memory machine, is only worthwhile for large datasets

Or a very costly reduction operation

This is grain size, again

Parallel Algorithms

Reduction

The other issue is about reduction in general, not just in parallel. Reduction relies on the associativity of the reduction operation

Parallel Algorithms

Reduction

The other issue is about reduction in general, not just in parallel. Reduction relies on the associativity of the reduction operation

Reduce the list $(1, 2, 3, 4)$ using $-$

Parallel Algorithms

Reduction

The other issue is about reduction in general, not just in parallel. Reduction relies on the associativity of the reduction operation

Reduce the list $(1, 2, 3, 4)$ using $-$

Do we mean

$$((1 - 2) - 3) - 4 = -8$$

a *left* reduction

The other issue is about reduction in general, not just in parallel. Reduction relies on the associativity of the reduction operation

Reduce the list $(1, 2, 3, 4)$ using $-$

Do we mean

$$((1 - 2) - 3) - 4 = -8$$

a *left* reduction

Or

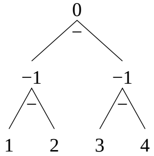
$$1 - (2 - (3 - 4)) = -2$$

a *right* reduction?

Parallel Algorithms

Reduction

And a tree reduction will give

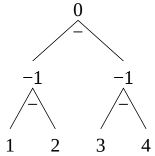


Tree Reduction

Parallel Algorithms

Reduction

And a tree reduction will give



Tree Reduction

Or something else entirely depending on where the data ended up in the tree

The simple answer is not to do reductions using non-associative operations, even sequentially

The simple answer is not to do reductions using non-associative operations, even sequentially

However, there are many useful reduction operations, including $+$, $*$, max, min, left$(a, b) = a$ and so on

Parallel Algorithms

Reduction

Reduction appears as an operation in many languages, e.g., JavaScript `array.reduce(op)` to reduce the `array` with the `op`:

`((array[0] op array[1]) op array[2]) op ...`

Reduction appears as an operation in many languages, e.g., JavaScript `array.reduce(op)` to reduce the `array` with the `op`:

```
((array[0] op array[1]) op array[2]) op ...
```

Thus amenable to automatic parallelisation, if the operation is associative and independent of the array (e.g., not if the `op` updates the array)

Parallel Algorithms

Prefix Scan

Closely related to reduction is the *prefix scan*: $(1, 2, 3, 4)$ with $+$ returns

$$(1, 3, 6, 10)$$

Closely related to reduction is the *prefix scan*: $(1, 2, 3, 4)$ with $+$ returns

$$(1, 3, 6, 10)$$

So: `(array[0], array[0] op array[1], array[0] op array[1] op array[2], ...)`

Closely related to reduction is the *prefix scan*: $(1, 2, 3, 4)$ with $+$ returns

$$(1, 3, 6, 10)$$

So: `(array[0], array[0] op array[1], array[0] op array[1] op array[2], ...)`

The partial reductions, usually left associated

Parallel Algorithms

Prefix Scan

This can also be done in $O(\log n)$ steps (on n processors)

Parallel Algorithms

Prefix Scan

This can also be done in $O(\log n)$ steps (on n processors)

Even though it seems you need to compute $1 + 2$ before computing $1 + 2 + 3$ before computing $1 + 2 + 3 + 4$, thus serialising the whole thing

This can also be done in $O(\log n)$ steps (on n processors)

Even though it seems you need to compute $1 + 2$ before computing $1 + 2 + 3$ before computing $1 + 2 + 3 + 4$, thus serialising the whole thing

But this is sequential thinking!

This can also be done in $O(\log n)$ steps (on n processors)

Even though it seems you need to compute $1 + 2$ before computing $1 + 2 + 3$ before computing $1 + 2 + 3 + 4$, thus serialising the whole thing

But this is sequential thinking!

For example, you can compute $3 + 4$ at the same time as $1 + 2$; and then $(1 + 2) + 3$ in parallel with $(1 + 2) + (3 + 4)$

Parallel Algorithms

Prefix Scan

This can also be done in $O(\log n)$ steps (on n processors)

Even though it seems you need to compute $1 + 2$ before computing $1 + 2 + 3$ before computing $1 + 2 + 3 + 4$, thus serialising the whole thing

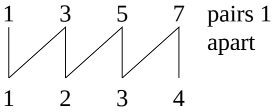
But this is sequential thinking!

For example, you can compute $3 + 4$ at the same time as $1 + 2$; and then $(1 + 2) + 3$ in parallel with $(1 + 2) + (3 + 4)$

We can proceed in a tree-like sequence of combination of pairs of values

Parallel Algorithms

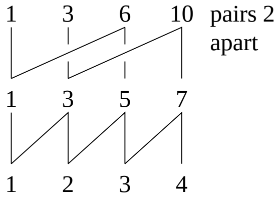
Prefix Scan



Prefix Scan 1 apart

Parallel Algorithms

Prefix Scan



Prefix Scan 2 apart

First step is to sum `array[i] = array[i] + array[i-1]` in parallel

Parallel Algorithms

Prefix Scan

First step is to sum `array[i] = array[i] + array[i-1]` in parallel

Then double the distances:
`array[i] = array[i] + array[i-2]`

Parallel Algorithms

Prefix Scan

First step is to sum `array[i] = array[i] + array[i-1]` in parallel

Then double the distances:
`array[i] = array[i] + array[i-2]`

Then double the distances:
`array[i] = array[i] + array[i-4]`

Parallel Algorithms

Prefix Scan

First step is to sum `array[i] = array[i] + array[i-1]` in parallel

Then double the distances:
`array[i] = array[i] + array[i-2]`

Then double the distances:
`array[i] = array[i] + array[i-4]`

And so on, for $\log n$ steps on $O(n)$ processors: this gives us all the prefix sums, including the total reduction as the last element

When limited to p processors we can produce a scan in time

$$O\left(\frac{n}{p} + \log p\right)$$

Parallel Algorithms

Prefix Scan

When limited to p processors we can produce a scan in time

$$O\left(\frac{n}{p} + \log p\right)$$

Scan has the same issues as reduce, namely data travel and associativity

Scan appears to give us more answers than reduce for the same amount of work!

Scan appears to give us more answers than reduce for the same amount of work!

It's not: for a start, reduce uses at most $n/2$ processors, while scan uses up to $n - 1$

But more importantly, reduce halves the number of active processors in each step, while scan uses more processors more of the time. It uses $n - 2^r$ active processors in step r, so it *ends* with about $n/2$ active processors

Parallel Algorithms

Prefix Scan

But more importantly, reduce halves the number of active processors in each step, while scan uses more processors more of the time. It uses $n - 2^r$ active processors in step r, so it *ends* with about $n/2$ active processors

They both complete in the same amount of time so they have the same speedup, but scan is more efficient

But more importantly, reduce halves the number of active processors in each step, while scan uses more processors more of the time. It uses $n - 2^r$ active processors in step r, so it *ends* with about $n/2$ active processors

They both complete in the same amount of time so they have the same speedup, but scan is more efficient

Meaning scan uses more hardware more of the time (and therefore takes more energy)

But more importantly, reduce halves the number of active processors in each step, while scan uses more processors more of the time. It uses $n - 2^r$ active processors in step r, so it *ends* with about $n/2$ active processors

They both complete in the same amount of time so they have the same speedup, but scan is more efficient

Meaning scan uses more hardware more of the time (and therefore takes more energy)

We can see that reduce has quite a lot of slack in parallel!

Note that both scan and reduce work well on a SIMD architecture

Note that both scan and reduce work well on a SIMD architecture

They work on distributed memory, too, but we have to watch the cost of the messaging

Parallel Algorithms

Prefix Scan

Note that both scan and reduce work well on a SIMD architecture

They work on distributed memory, too, but we have to watch the cost of the messaging

MPI includes several scan operations including `MPI_MAX`, `MPI_MIN`, `MP_SUM`, `MPI_PROD`, `MPI_LAND` (logical AND), `MPI_LOR` (logical OR)
amongst others

Parallel Algorithms

Prefix Scan

Note that both scan and reduce work well on a SIMD architecture

They work on distributed memory, too, but we have to watch the cost of the messaging

MPI includes several scan operations including `MPI_MAX`, `MPI_MIN`, `MP_SUM`, `MPI_PROD`, `MPI_LAND` (logical AND), `MPI_LOR` (logical OR)
amongst others

Exercise Write a parallel prefix scan in OpenMP

Exercise In fact there is a better, work efficient, more complicated algorithm that only needs $n/2$ processors. Look it up

The Fast Fourier Transform (FFT) is one of the basic algorithms in CS, known by everybody who knows anything about CS

Parallel Algorithms

FFT

The Fast Fourier Transform (FFT) is one of the basic algorithms in CS, known by everybody who knows anything about CS

The Discrete Fourier Transform (DFT) takes a sequence of n (complex) numbers and returns a sequence of n numbers

The Fast Fourier Transform (FFT) is one of the basic algorithms in CS, known by everybody who knows anything about CS

The Discrete Fourier Transform (DFT) takes a sequence of n (complex) numbers and returns a sequence of n numbers

If the input numbers represent a signal, the DFT values represent the constituent frequencies of that signal

Parallel Algorithms

FFT

The Fast Fourier Transform (FFT) is one of the basic algorithms in CS, known by everybody who knows anything about CS

The Discrete Fourier Transform (DFT) takes a sequence of n (complex) numbers and returns a sequence of n numbers

If the input numbers represent a signal, the DFT values represent the constituent frequencies of that signal

$$y_k = \sum_{j=0}^{n-1} x_j e^{-2\pi ijk/n}, \text{ for } 0 \leq k < n$$

The n values x_i are input; the n values y_i are output

Parallel Algorithms

FFT

This has two obvious elements of parallelism:

Parallel Algorithms

FFT

This has two obvious elements of parallelism:

- each y_k can be computed independently, for a n-way parallelism

Parallel Algorithms

FFT

This has two obvious elements of parallelism:

- each y_k can be computed independently, for a n-way parallelism
- each summation can be done as a tree, for a $\log n$-way parallelism

Parallel Algorithms
FFT

This has two obvious elements of parallelism:

- each y_k can be computed independently, for a n-way parallelism
- each summation can be done as a tree, for a $\log n$-way parallelism
- taking total time $O(\log n)$ on $O(n^2)$ processors

Parallel Algorithms

FFT

This has two obvious elements of parallelism:

- each y_k can be computed independently, for a n-way parallelism
- each summation can be done as a tree, for a $\log n$-way parallelism
- taking total time $O(\log n)$ on $O(n^2)$ processors

But, instead let us look at a sequential divide and conquer version

This sum can be computed as presented: summing n values for each of n values y_k, thus taking time $O(n^2)$

This sum can be computed as presented: summing n values for each of n values y_k, thus taking time $O(n^2)$

However, if n is even, then we get a nice recursive presentation by splitting the sum into evens and odds

Parallel Algorithms

FFT

$$
\begin{aligned}
y_k &= \sum_{j=0}^{n-1} x_j e^{-2\pi ijk/n} \\
&= \sum_{j=0}^{n/2-1} x_{2j} e^{-2\pi i(2j)k/n} + \sum_{j=0}^{n/2-1} x_{2j+1} e^{-2\pi i(2j+1)k/n} \\
&= \sum_{j=0}^{n/2-1} x_{2j} e^{-2\pi ijk/(n/2)} + e^{-2\pi ik/n} \sum_{j=0}^{n/2-1} x_{2j+1} e^{-2\pi ijk/(n/2)}
\end{aligned}
$$

Decomposition of Fourier Transform

Parallel Algorithms

FFT

$$
\begin{aligned}
y_k &= \sum_{j=0}^{n-1} x_j e^{-2\pi ijk/n} \\
&= \sum_{j=0}^{n/2-1} x_{2j} e^{-2\pi i(2j)k/n} + \sum_{j=0}^{n/2-1} x_{2j+1} e^{-2\pi i(2j+1)k/n} \\
&= \sum_{j=0}^{n/2-1} x_{2j} e^{-2\pi ijk/(n/2)} + e^{-2\pi ik/n} \sum_{j=0}^{n/2-1} x_{2j+1} e^{-2\pi ijk/(n/2)}
\end{aligned}
$$

Decomposition of Fourier Transform

This is just two half-size DFTs

For n a power of 2 we can repeat recursively, leading to the *Fast Fourier Transform*, a way to implement the DFT

For n a power of 2 we can repeat recursively, leading to the *Fast Fourier Transform*, a way to implement the DFT

In fact, the FFT is an unwinding of the recursion into an iteration that runs slightly faster, but is harder to understand

Parallel Algorithms

FFT

For n a power of 2 we can repeat recursively, leading to the *Fast Fourier Transform*, a way to implement the DFT

In fact, the FFT is an unwinding of the recursion into an iteration that runs slightly faster, but is harder to understand

The FFT takes sequential time $O(n \log n)$, which is a huge improvement over $O(n^2)$; e.g., for $n = 1{,}000{,}000$, this is about 20,000,000 against 1,000,000,000,000

For n a power of 2 we can repeat recursively, leading to the *Fast Fourier Transform*, a way to implement the DFT

In fact, the FFT is an unwinding of the recursion into an iteration that runs slightly faster, but is harder to understand

The FFT takes sequential time $O(n \log n)$, which is a huge improvement over $O(n^2)$; e.g., for $n = 1{,}000{,}000$, this is about 20,000,000 against 1,000,000,000,000

But, for our purposes, we can see this as a simple divide and conquer, thus easily parallelisable

The parallelisation of the FFT works in a way very similar to what we have seen before and has complexity $O(\log n)$ on $O(n)$ processors, and $O(\log p + (n/p)\log(n/p))$ on p processors

Parallel Algorithms

FFT

The parallelisation of the FFT works in a way very similar to what we have seen before and has complexity $O(\log n)$ on $O(n)$ processors, and $O(\log p + (n/p)\log(n/p))$ on p processors

As the FFT is such an important algorithm, much has been written about it and its parallel variants, in particular matching it to the various kinds of hardware (SIMD, pipeline, shared memory, etc.)

Parallel Algorithms

And So On

There are very many other parallel algorithms: just think of the large literature on sequential algorithms that exists

Parallel Algorithms

And So On

There are very many other parallel algorithms: just think of the large literature on sequential algorithms that exists

We have just looked at a couple, but everything that you have done in the past sequentially will probably have a parallel counterpart

Parallel Algorithms

And So On

There are very many other parallel algorithms: just think of the large literature on sequential algorithms that exists

We have just looked at a couple, but everything that you have done in the past sequentially will probably have a parallel counterpart

Some algorithms will map best to shared memory, some distributed, some SIMD, and so on

Parallel Algorithms

And So On

There are very many other parallel algorithms: just think of the large literature on sequential algorithms that exists

We have just looked at a couple, but everything that you have done in the past sequentially will probably have a parallel counterpart

Some algorithms will map best to shared memory, some distributed, some SIMD, and so on

Some will be sensitive to the topology of the architecture (full connect, torus, etc.), others work well regardless

Parallel Algorithms

And So On

There are very many other parallel algorithms: just think of the large literature on sequential algorithms that exists

We have just looked at a couple, but everything that you have done in the past sequentially will probably have a parallel counterpart

Some algorithms will map best to shared memory, some distributed, some SIMD, and so on

Some will be sensitive to the topology of the architecture (full connect, torus, etc.), others work well regardless

Still more will not work well in parallel at all

Parallel Algorithms

And So On

There are very many other parallel algorithms: just think of the large literature on sequential algorithms that exists

We have just looked at a couple, but everything that you have done in the past sequentially will probably have a parallel counterpart

Some algorithms will map best to shared memory, some distributed, some SIMD, and so on

Some will be sensitive to the topology of the architecture (full connect, torus, etc.), others work well regardless

Still more will not work well in parallel at all

Exercise Look some up!