Parallel Algorithms

Reduction

Next: parallel reduction
Parallel Algorithms

Reduction

Next: parallel reduction

Reduction has a natural parallelisation using a tree
Next: parallel reduction

Reduction has a natural parallelisation using a tree

Reducing a list of values using summation (read bottom up)
Parallel Algorithms

Reduction

Next: parallel reduction

Reduction has a natural parallelisation using a tree

Reducing a list of values using maximum
Parallel Algorithms

Reduction

This takes $O(\log n)$ steps to reduce $n$ values, using $O(n)$ processors.
This takes $O(\log n)$ steps to reduce $n$ values, using $O(n)$ processors.

Sequential time: $n - 1$ operations, giving speedup

$$S = O(n/ \log n)$$ using $O(n)$ processors.
This takes $O(\log n)$ steps to reduce $n$ values, using $O(n)$ processors.

Sequential time: $n - 1$ operations, giving speedup

$$S = O(n / \log n)$$ using $O(n)$ processors

This is not much less than $n$, as $\log n$ grows only slowly with $n$. 
Parallel Algorithms

Reduction

Efficiency

\[ E = O\left(\frac{1}{\log n}\right) \]

which slowly drops as \( n \) increases
For $p$ processors, divide the data into $p$ chunks of size $n/p$
Parallel Algorithms
Reduction

For $p$ processors, divide the data into $p$ chunks of size $n/p$

Time to reduce a chunk (sequential): $O(n/p)$
Time to reduce the chunks: $O(\log p)$
Parallel Algorithms

Reduction

For \( p \) processors, divide the data into \( p \) chunks of size \( n/p \)

Time to reduce a chunk (sequential): \( O(n/p) \)
Time to reduce the chunks: \( O(\log p) \)

Total

\[
O \left( \frac{n}{p} + \log p \right)
\]
Parallel Algorithms

Reduction

Speedup

\[ S_p = \frac{n}{n/p + \log p} = \frac{p}{1 + (p \log p)/n} \]

which approaches \( p \) as \( n \) gets large
Parallel Algorithms

Reduction

Speedup

\[ S_p = \frac{n}{n/p + \log p} = \frac{p}{1 + (p \log p)/n} \]

which approaches \( p \) as \( n \) gets large

Likewise, the efficiency approaches 1 for large \( n \)
Parallel Algorithms
Reduction

Speedup

\[ S_p = \frac{n}{n/p + \log p} = \frac{p}{1 + (p \log p)/n} \]

which approaches \( p \) as \( n \) gets large

Likewise, the efficiency approaches 1 for large \( n \)

Similar to previous examples, if you allow yourself an indefinite number of processors, the speedup will be greater, but at a high cost, i.e., low efficiency
Parallel Algorithms

Reduction

Speedup

\[ S_p = \frac{n}{n/p + \log p} = \frac{p}{1 + (p \log p)/n} \]

which approaches \( p \) as \( n \) gets large

Likewise, the efficiency approaches 1 for large \( n \)

Similar to previous examples, if you allow yourself an indefinite number of processors, the speedup will be greater, but at a high cost, i.e., low efficiency

For a fixed number of processors, you get a fixed bound on the speedup, but you will be using the hardware very efficiently as the dataset get large
There are a couple of issues, however

In real implementations we need to worry about the cost of data movement between processors. Probably small for a shared memory system, but it can easily be much larger than the cost of the reduction operation on other systems. 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.
There are a couple of issues, however

In real implementations we need to worry about the cost of data movement between processors
There are a couple of issues, however

In real implementations we need to worry about the cost of data movement between processors

Probably small for a shared memory system, but it can easily be much larger than the cost of the reduction operation on other systems
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

Probably small for a shared memory system, but it can easily be much larger than the cost of the reduction operation on other systems

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

Probably small for a shared memory system, but it can easily be much larger than the cost of the reduction operation on other systems

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

Or a very costly reduction operation
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

Probably small for a shared memory system, but it can easily be much larger than the cost of the reduction operation on other systems

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
The other issue is about reduction in general, not just in parallel. Reduction relies on the associativity of the reduction operation.
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 –
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?
And a tree reduction will give

\[
\begin{array}{c}
0 \\
- \\
-1 & -1 \\
1 & 2 & 3 & 4
\end{array}
\]
And a tree reduction will give

```
0
\_\_\_\_
-1 -1
\_\_\_\_
1 2 3 4
```

Or something else entirely depending on where the data ended up in the tree
And a tree reduction will give

```
         0
        /\  
       /  \ 
  -1   -1
  /    /  
1    2  3  4
```

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
Parallel Algorithms

Reduction

And a tree reduction will give

```
0
/  \
/    \
-1    -1
/  \
/   \
1    2
```

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

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:

\(((\text{array}[0] \text{ op } \text{array}[1]) \text{ op } \text{array}[2]) \text{ op } \ldots\)
Parallel Algorithms

Reduction

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

\[(\text{array}[0] \, \text{op} \, \text{array}[1]) \, \text{op} \, \text{array}[2]) \, \text{op} \, \ldots\]

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