CSE332: Data Abstractions

Lecture 17: Parallel Analysis and Parallel Prefix

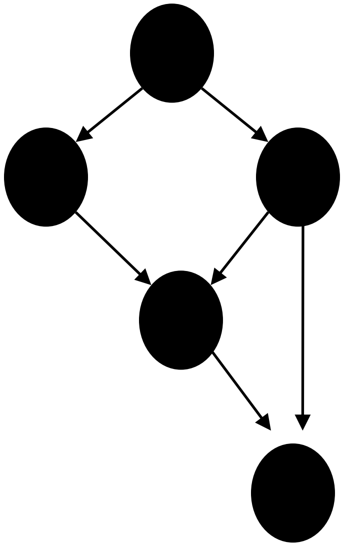
James Fogarty

Winter 2012

Including slides developed in part by
Ruth Anderson, James Fogarty, Dan Grossman

Work and Span in the DAG

- **`fork`** and **`join`** execution can be seen as a DAG
  - Nodes: Pieces of work
  - Edges: Source must finish before destination starts



- A **`fork`** “ends a node” and makes two outgoing edges
  - New thread
  - Continuation of current thread
- A **`join`** “ends a node” and makes a node with two incoming edges
  - Node just ended
  - Last node of thread joined on

Work and Span

Run-time costs are on the nodes, not on the edges

- Work: $\mathbf{T_1}$ = sum of all nodes in the DAG
  - One processor has to do all the work
  - Any topological sort is a legal execution

- Span: $\mathbf{T_\infty}$ = sum of all nodes on **most-expensive** path in the DAG
  - Can do everything that is ready, but still must wait for results
  - If all nodes are roughly equal cost, this is the **longest** path
  - Example: $O($`log` $n)$ for summing an array
    - Notice having > $n/2$ processors is no additional help

Let $\mathbf{T_P}$ be the running time if there are **P** processors available

More Definitions

- Speed-up on **P** processors: $\mathbf{T_1 / T_P}$
- If speed-up is **P** as we vary **P**, we call it perfect linear speed-up
  - Perfect linear speed-up means doubling **P** halves running time
  - Usually our goal; hard to get in practice
- Parallelism is the maximum possible speed-up: $\mathbf{T_1 / T_\infty}$
  - At some point, adding processors will not help
  - What that point is depends on the span

Parallel algorithms are about decreasing span without increasing work

… or at least not increasing work too much …

Optimal T_P

- So we know $\mathbf{T_1}$ and $\mathbf{T_\infty}$ but actually care about $\mathbf{T_P}$ (e.g., **P**=4)

- Ignoring memory-hierarchy issues, $\mathbf{T_P}$ cannot beat
  - $\mathbf{T_1 / P}$ why not?
  - $\mathbf{T_\infty}$ why not?

- So an *asymptotically* optimal execution would be:

$$\mathbf{T_P = \mathit{O}((T_1 / P) + T_\infty)}$$

First term dominates for small **P**, second for large **P**

Division of Responsibility

- Our job as users of a ForkJoin Framework:
  - Pick a good algorithm, write a program
  - When run, it creates a DAG of things to do
  - Make all nodes small-ish and approximately equal work

- The job of the framework developer:
  - Assign work to available processors to avoid idling
  - Keep constant factors low
  - Give the expected-time optimal guarantee

$$\mathbf{T_P = \mathit{O}((T_1 / P) + T_\infty)}$$

assuming framework-user did their job

  - We will not study how the framework does this

What That Means: Mostly Good News

The fork-join framework guarantee:

$$T_P = O((T_1 / P) + T_\infty)$$

- No implementation can beat $O(T_\infty)$ by more than a constant factor
- No implementation on **P** processors can beat $O(T_1 / P)$
- So the framework on average gets within a constant factor of the best you can do, assuming framework user did their part correctly

You can focus on your algorithm, data structures, and cut-offs

Do not worry about number of processors and scheduling

- Analyze running time given T_1, T_∞, and **P**

Examples

$$T_P = O((T_1 / P) + T_\infty)$$

- In the algorithms seen so far (e.g., sum an array):
  - $T_1 = O(n)$
  - $T_\infty = O(\log n)$
  - So expect (ignoring overheads): $T_P = O(n/P + \log n)$
- Suppose instead:
  - $T_1 = O(n^2)$
  - $T_\infty = O(n)$
  - So expect (ignoring overheads): $T_P = O(n^2/P + n)$

Amdahl's Law: Mostly Bad News

- We have analyzed a parallel program in terms of work and span

- In practice, it is common that your program has:

  a) parts that **parallelize well:**
  - Such as maps/reduces over arrays and trees

  b) …and parts that **don't parallelize at all:**
  - Such as reading a linked list, getting input, or just doing computations where each step needs the results of previous step

- These **unparallelized** parts can turn out to be a big bottleneck

Amdahl's Law: Mostly Bad News

Let the ***work*** be 1 unit time

Let **S** be the portion of the execution that cannot be parallelized

Then: $\mathbf{T_1 = S + (1\text{-}S) = 1}$

Suppose we get perfect linear speedup *on the parallel portion*

Then: $\mathbf{T_P = S + (1\text{-}S)/P}$

So the overall speedup with **P** processors (this is Amdahl's Law):

$$\mathbf{T_1 / T_P = 1 / (S + (1\text{-}S)/P)}$$

And the parallelism is (with infinite processors):

$$\mathbf{T_1 / T_\infty = 1 / S}$$

Amdahl’s Law Example

Suppose:

$T_1 = S + (1-S) = 1$ (aka total program execution time)
$T_1 = 1/3 + 2/3 = 1$
$T_1 = 33 \text{ sec} + 67 \text{ sec} = 100 \text{ sec}$

Time on P processors: **$T_P = S + (1-S)/P$**

So:

$T_P = 33 \text{ sec} + (67 \text{ sec})/P$
$T_3 = 33 \text{ sec} + (67 \text{ sec})/3$
$T_3 = 33 \text{ sec} + 22.33 \text{ sec} = 55.33 \text{ sec}$

Why Such Bad News?

$$T_1 / T_P = 1 / (S + (1-S)/P) \qquad T_1 / T_\infty = 1 / S$$

- Suppose 33% of a program is sequential
  - Then a billion processors will not give a speedup over 3
- Suppose you miss the good old days where you could get 100x speedup by just waiting about 12 years
- Now suppose in 12 years, clock speed is the same but you get 256 processors instead of 1
- For 256 processors to get at least 100x speedup, we need

$$100 \leq 1 / (\mathbf{S} + (1-\mathbf{S})/256)$$

Which means $\mathbf{S} \leq .0061$ (i.e., 99.4% perfectly parallelizable)

Plots You Need to See

1. Assume 256 processors
   - x-axis: sequential portion **S**, ranging from .01 to .25
   - y-axis: speedup $\mathbf{T_1 / T_P}$ (will go down as **S** increases)

2. Assume **S** = .01 or .1 or .25 (three separate lines)
   - x-axis: number of processors **P**, ranging from 2 to 32
   - y-axis: speedup $\mathbf{T_1 / T_P}$ (will go up as **P** increases)

Do this as a homework problem!

- More practice with a spreadsheet or graphing program
- Compare against your intuition
- A picture is worth 1000 words, especially if you made it

All is Not Lost

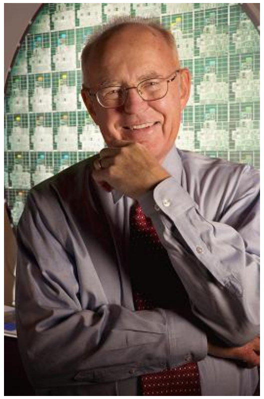
Amdahl’s Law is a harsh reality

  - But it does not mean additional processors are worthless

- We can find new parallel algorithms
  - Some things that seem sequential are actually parallelizable

- We can change the problem or do new things
  - Video games use tons of parallel processors
    - They are not rendering 10-year-old graphics faster
    - They are rendering better monsters, better scenery

Moore and Amdahl



- Moore's "Law" is an observation about
  the progress of the semiconductor industry
  - Transistor density doubles roughly every 18 months
- Amdahl's Law is a mathematical theorem
  - Implies diminishing returns of adding more processors
- Both are incredibly important in designing computer systems

Moving Forward

Done:

- Simple ways to use parallelism for counting, summing, finding
- Analysis of running time and implications of Amdahl’s Law

Now:

- Clever ways to parallelize more than is intuitively possible
- Parallel prefix:
  - This “key trick” typically underlies surprising parallelization
  - Enables other things like packs
- Parallel sorting: mergesort and quicksort (though not in place)
  - Easy to get a little parallelism
  - With cleverness can get a lot of parallelism

The Prefix-Sum Problem

Given **`int[] input`**, produce **`int[] output`** where:

`output[i]` is the sum of **`input[0]+input[1]+…+input[i]`**

Sequential can be a CS1 exam problem:

```
int[] prefix_sum(int[] input){
  int[] output = new int[input.length];
  output[0] = input[0];
  for(int i=1; i < input.length; i++)
    output[i] = output[i-1]+input[i];
  return output;
}
```

Does not seem parallelizable

- Work: $O(n)$, Span: $O(n)$

This *algorithm* is sequential, but a *different algorithm* has

- Work: O(n), Span: O(**`log`** n)

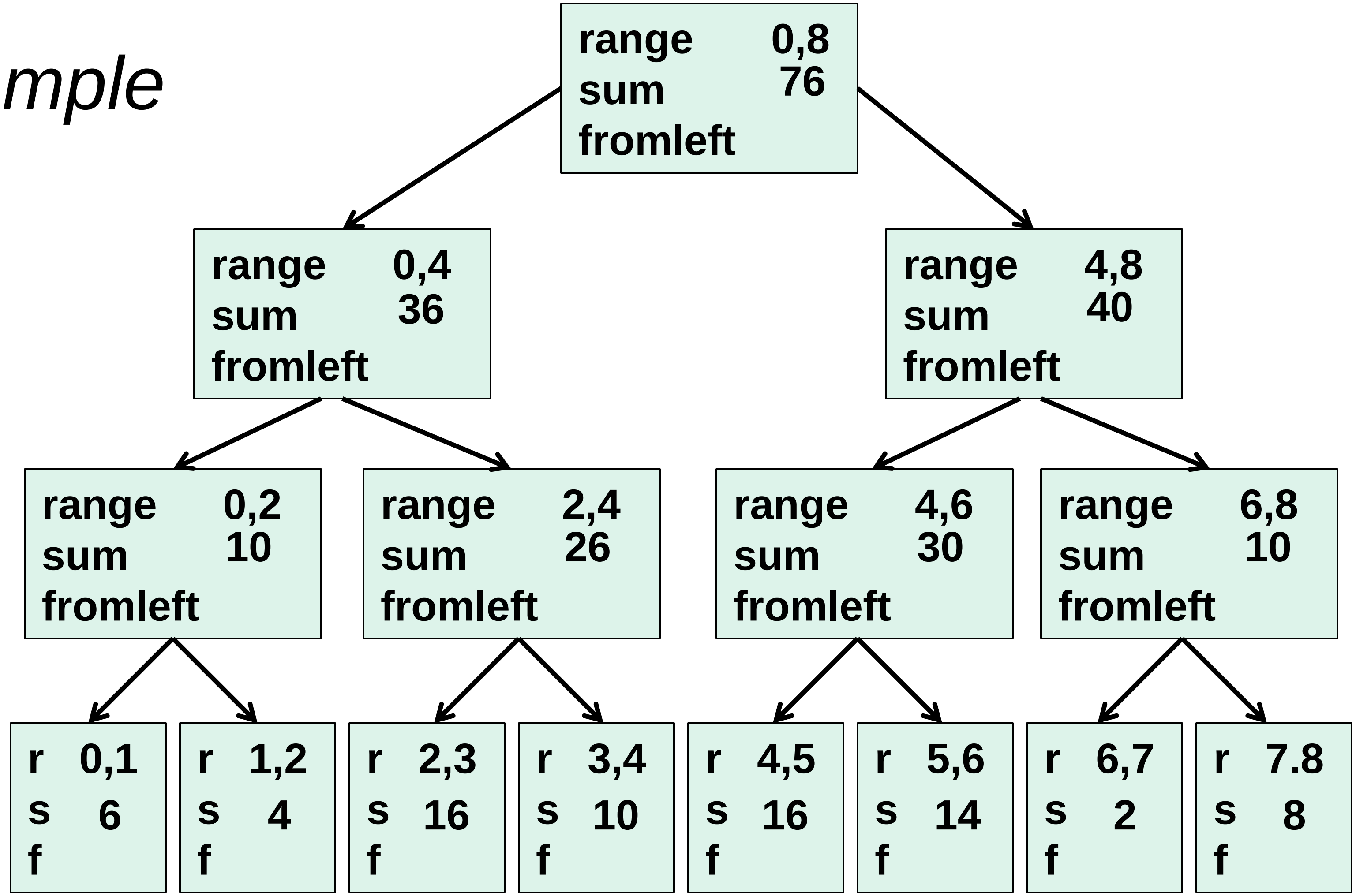
Parallel Prefix-Sum

- The parallel-prefix algorithm does two passes
  - Each pass has $O(n)$ work and $O(\texttt{log}\ n)$ span
  - So in total there is $O(n)$ work and $O(\texttt{log}\ n)$ span
  - So just like with array summing, parallelism is n / $\texttt{log}\ n$
  - An exponential speedup
- The first pass builds a tree bottom-up: the “up” pass
- The second pass traverses the tree top-down: the “down” pass

Historical note:

Original algorithm due to
R. Ladner and M. Fischer at the
University of Washington in 1977

Example



range 0,8
sum 76
fromleft

range 0,4
sum 36
fromleft

range 4,8
sum 40
fromleft

range 0,2
sum 10
fromleft

range 2,4
sum 26
fromleft

range 4,6
sum 30
fromleft

range 6,8
sum 10
fromleft

r 0,1
s 6
f

r 1,2
s 4
f

r 2,3
s 16
f

r 3,4
s 10
f

r 4,5
s 16
f

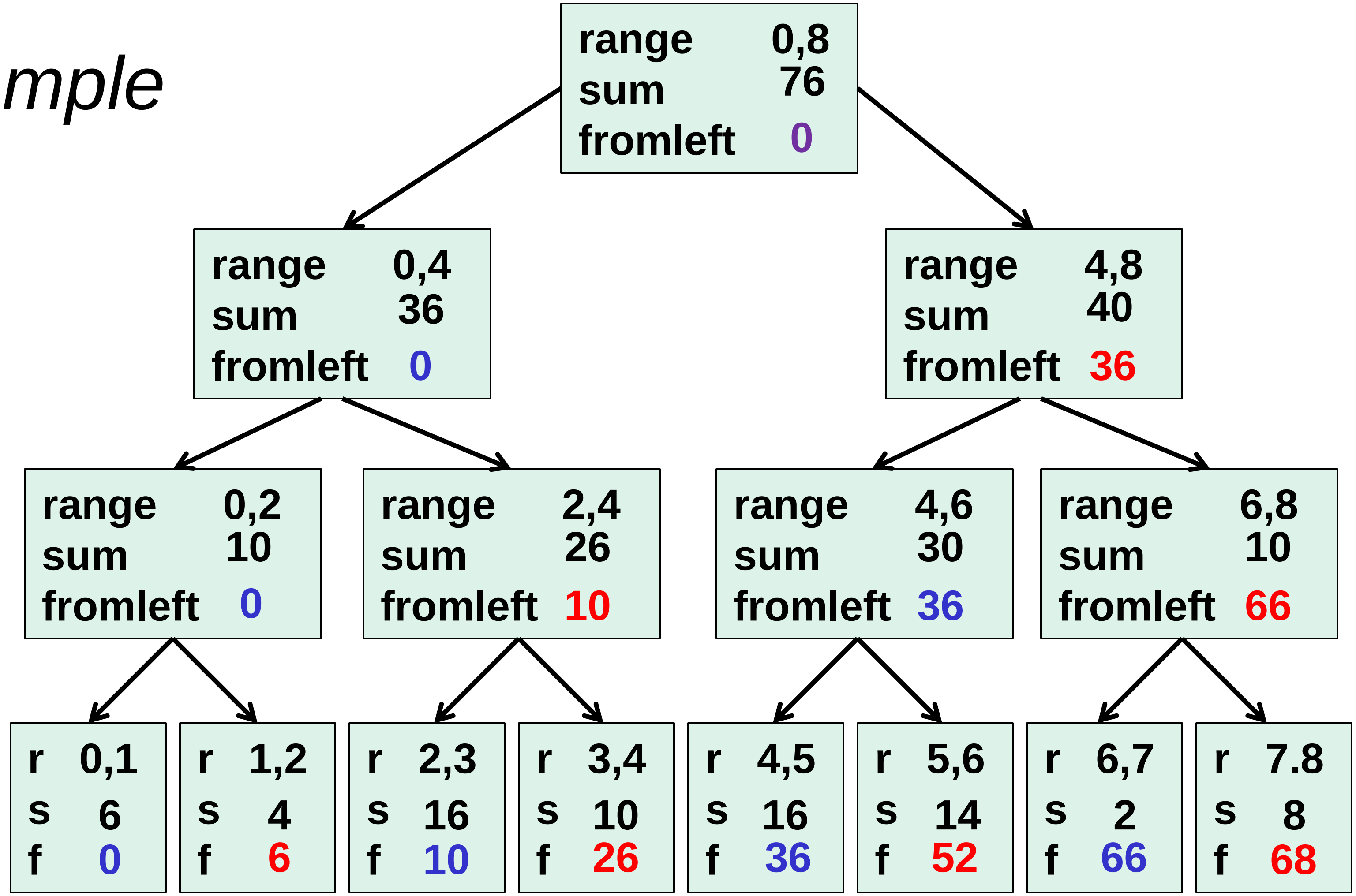
r 5,6
s 14
f

r 6,7
s 2
f

r 7.8
s 8
f

`input`	6	4	16	10	16	14	2	8
`output`								

Example



range 0,8
sum 76
fromleft 0

range 0,4
sum 36
fromleft 0

range 4,8
sum 40
fromleft 36

range 0,2
sum 10
fromleft 0

range 2,4
sum 26
fromleft 10

range 4,6
sum 30
fromleft 36

range 6,8
sum 10
fromleft 66

r 0,1
s 6
f 0

r 1,2
s 4
f 6

r 2,3
s 16
f 10

r 3,4
s 10
f 26

r 4,5
s 16
f 36

r 5,6
s 14
f 52

r 6,7
s 2
f 66

r 7.8
s 8
f 68

`input`	6	4	16	10	16	14	2	8
`output`	6	10	26	36	52	66	68	76

The Algorithm: Part 1

1. Up: Build a binary tree where
   - Root has sum of the range [**`x,y`**)
   - If a node has sum of [**`lo,hi`**) and **`hi>lo`**,
     - Left child has sum of [**`lo,middle`**)
     - Right child has sum of [**`middle,hi`**)
     - A leaf has sum of [**`i,i+1`**) i.e., **`input[i]`**

This is an easy fork-join computation:
combine results by actually building a binary tree with the range-sums

- Tree built bottom-up in parallel
- Could be more clever in an array, as we were with heaps

Analysis: $O(n)$ work, $O(\mathbf{log}\ n)$ span

The Algorithm: Part 2

2. Down: Pass down a value **`fromLeft`**
   - Root given a **`fromLeft`** of **`0`**
   - Node takes its **`fromLeft`** value and
     - Passes its left child
       - the same **`fromLeft`**
     - Passes its right child
       - its **`fromLeft`** plus its left child's **`sum`** (stored in part 1)
   - At the leaf for array position **`i`**,
     `output[i]=fromLeft+input[i]`

This is an easy fork-join computation:
traverse the tree built in step 1 and produce no result

- Leaves assign to **`output`**
- Invariant: **`fromLeft`** is sum of elements left of the node's range

Analysis: $O(n)$ work, $O(\log n)$ span

Sequential Cut-Off

Adding a sequential cut-off is easy as always:

- Up:

  just a sum, have leaf node hold the sum of a range

- Down:

```
 output[lo] = fromLeft + input[lo];
for(i=lo+1; i < hi; i++)
  output[i] = output[i-1] + input[i]
```

Generalizing Parallel Prefix

Just as sum-array was the simplest example of a common pattern, prefix-sum illustrates a pattern that can be used in many problems

- Minimum, maximum of all elements to the left of `i`
- Is there an element to the left of `i` satisfying some property?
- Count of elements to the left of `i` satisfying some property
  - This last one is perfect for an efficient parallel pack
  - Perfect for building on top of the “parallel prefix trick”

Pack

[Non-standard terminology, **`filter`** does not emphasize stability]

Given an array **`input`**,
produce an array **`output`**
containing only elements such that **`f(elt)`** is **`true`**

Example:
```
input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
f: is elt > 10
output [17, 11, 13, 19, 24]
```

Parallelizable
- Finding elements for the output is easy
- But getting them in the right place seems hard

Parallel Map, Parallel Prefix, Parallel Map

1. Parallel map to compute a bit-vector for true elements

```
input  [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
bits   [1,  0, 0, 0,  1, 0,  1,  1, 0,  1]
```

2. Parallel-prefix sum on the bit-vector

```
bitsum [1,  1, 1, 1,  2, 2,  3,  4, 4,  5]
```

3. Parallel map to produce the output

```
output [17, 11, 13, 19, 24]
```

```
output = new array of size bitsum[n-1]
FORALL(i=0; i < input.length; i++){
  if(bits[i]==1)
    output[bitsum[i]-1] = input[i];
}
```

Pack Comments

- First two steps can be combined into one pass
  - Use a different base case for the prefix sum
  - No effect on asymptotic complexity

- Can also combine third step into the down pass of the prefix sum
  - Again no effect on asymptotic complexity

- Analysis: $O(n)$ work, $O(\texttt{log}\ n)$ span
  - Multiple passes, but this is a constant

- Parallelized packs will help us parallelize quicksort