Parallel Programming

ForkJoin Framework & Task Parallel Algorithms

This Lecture: Java’s ForkJoin Framework

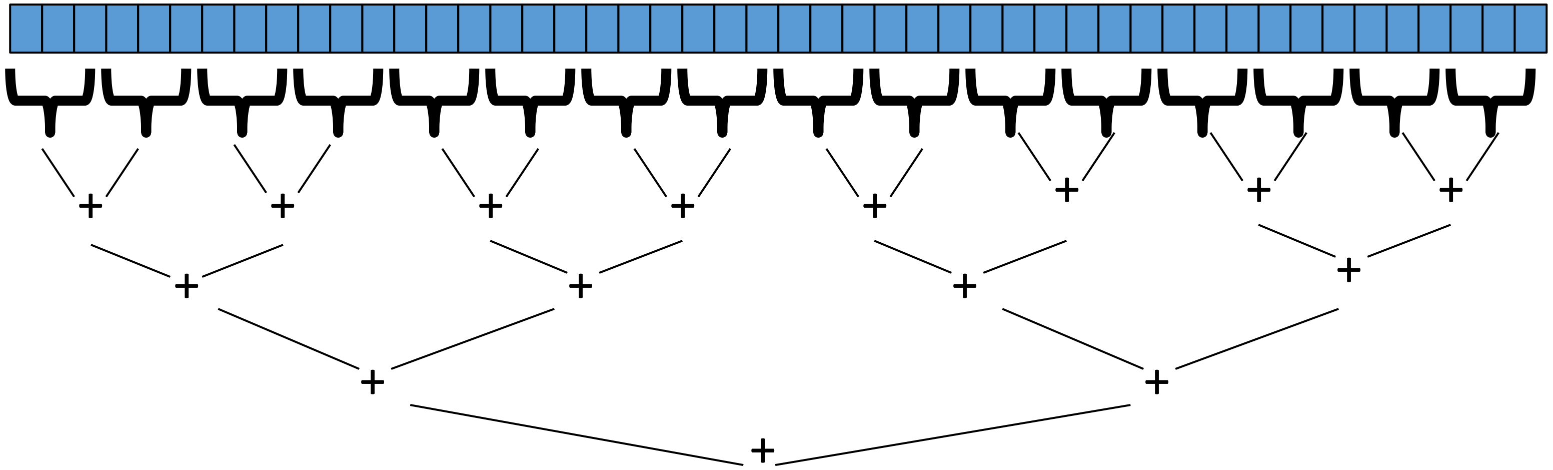
We need a new framework that supports Divide and Conquer style parallelism

That is, when a task is waiting, it is suspended and other tasks are allowed to run!

Motivation

We said:

- Adding numbers recursively is straightforward to implement using divide-and-conquer
- Parallelism for the recursive calls



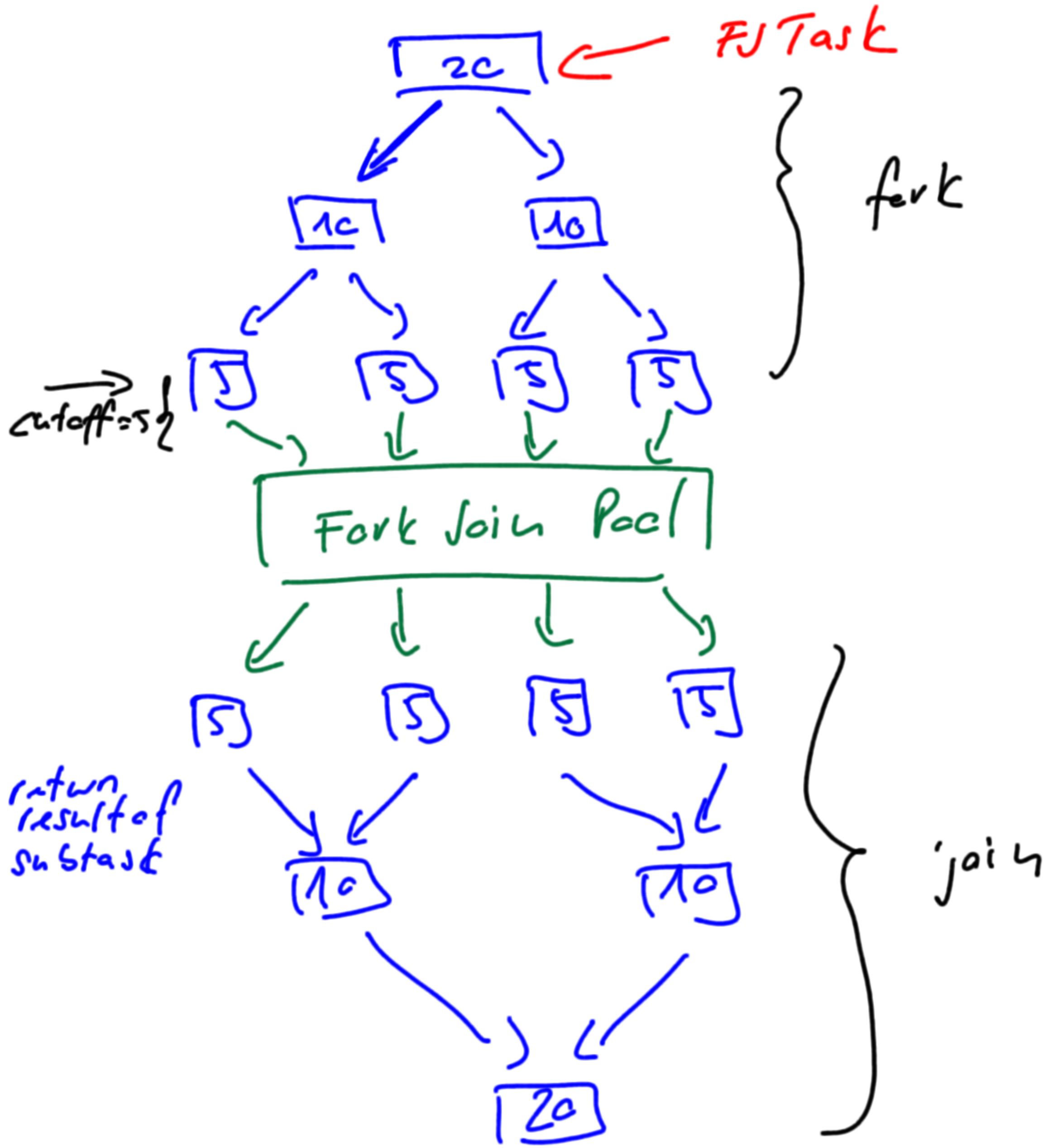
That library, finally

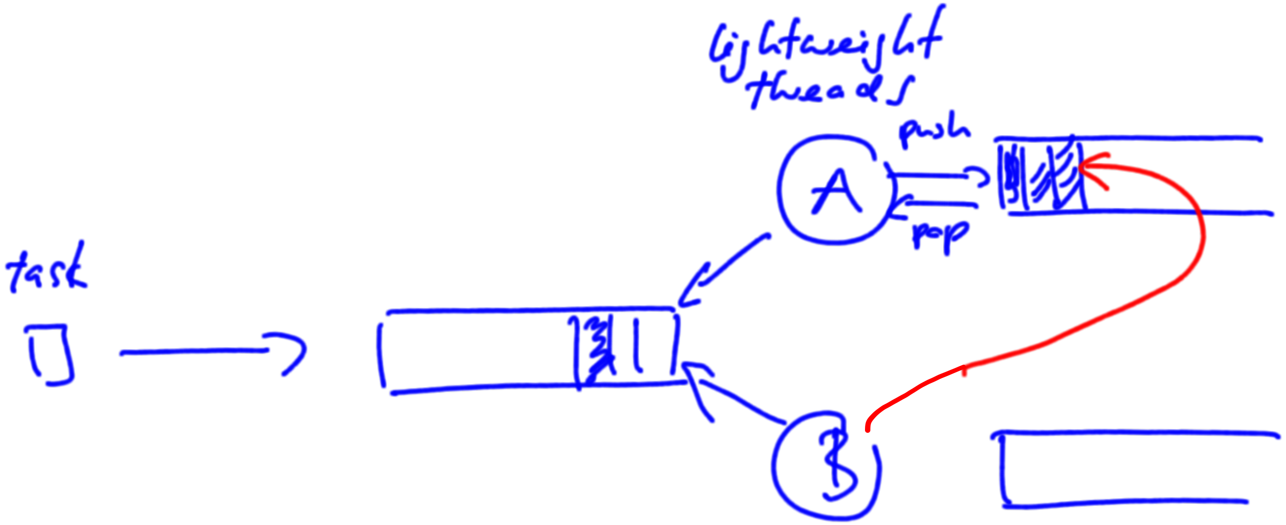
The ForkJoin Framework is designed to meet the needs of divide-and-conquer fork-join parallelism

- In the Java standard libraries

Lecture will focus on pragmatics/logistics

- Similar libraries available for other languages
  - C/C++: Cilk (inventors), Intel's Thread Building Blocks
  - C#: Task Parallel Library
  - …
- Library's implementation is also fascinating


FJTask
20
10
10
5
5
5
5
fork
cutoff=5
Fork Join Pool
5
5
5
5
return result of subtask
10
10
20
join


lightweight
threads
task
A
B
push
pop

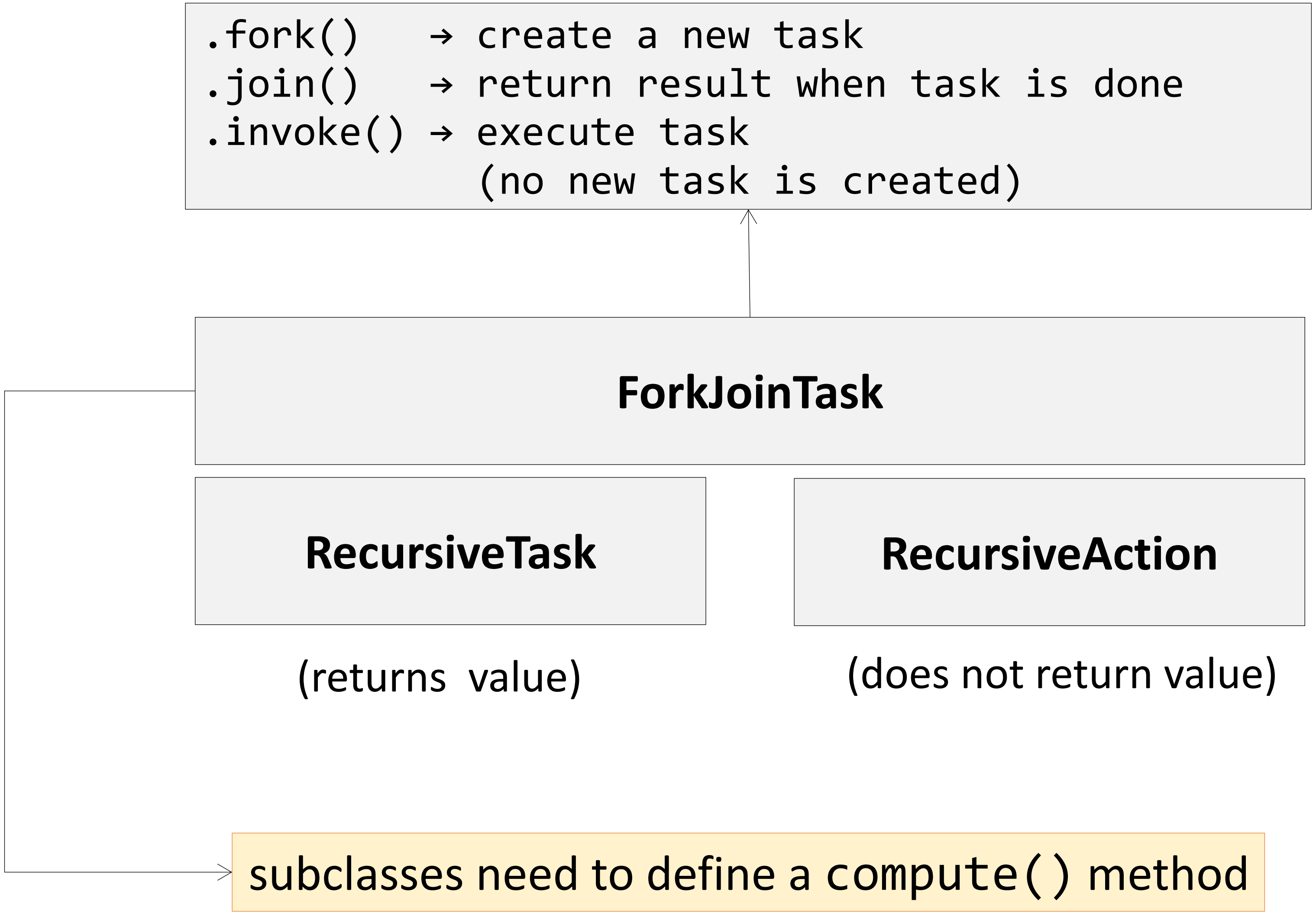
Different terms, same basic idea

To use the ForkJoin Framework:

A little standard set-up code (e.g., create a **ForkJoinPool**)

Don’t subclass **Thread**	Do subclass **RecursiveTask<V>**
Don’t override **run**	Do override **compute**
Do not use an **ans** field	Do return a **V** from **compute**
Don’t call **start**	Do call **fork**
Don’t just call **join**	Do call **join** which returns answer
Don’t call **run** to hand-optimize	Do call **compute** to hand-optimize
Don’t have a topmost call to **run**	Do create a pool and call **invoke**

Tasks in Fork/Join Framework



```
.fork()   → create a new task
.join()   → return result when task is done
.invoke() → execute task
            (no new task is created)
```

ForkJoinTask

RecursiveTask

(returns value)

RecursiveAction

(does not return value)

subclasses need to define a `compute()` method

Tasks in Fork/Join Framework – Usual Procedure

`t.fork()` → spawn a new task

… → possibly do other things

`res = t.join()` → wait for task result

`t.invoke()` → execute the task computation without spawning a task (in-place)

Recursive sum with ForkJoin (use)

```
class Globals {
    static ForkJoinPool fjPool = new ForkJoinPool();
}
```

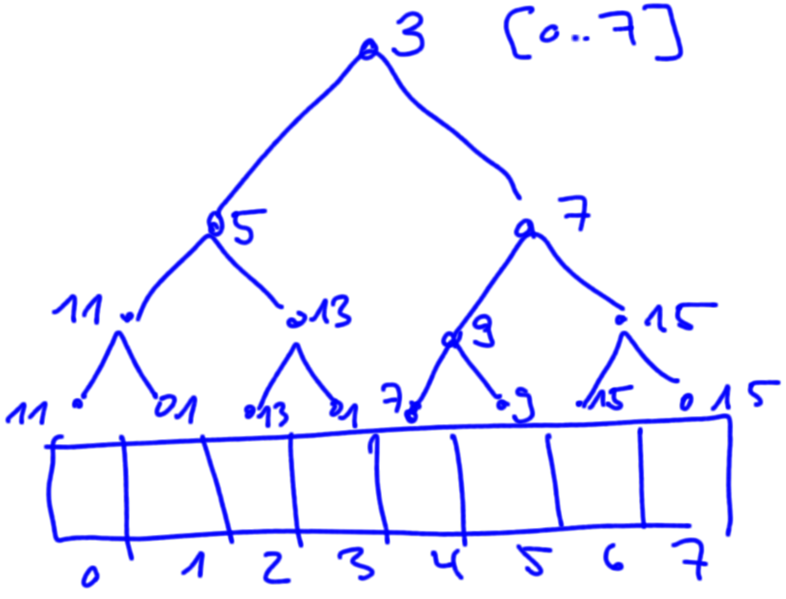
Default # of processors

```
static long sumArray(int[] array) {
    return Globals.fjPool.invoke(new SumForkJoin(array,0,array.length));
}
```

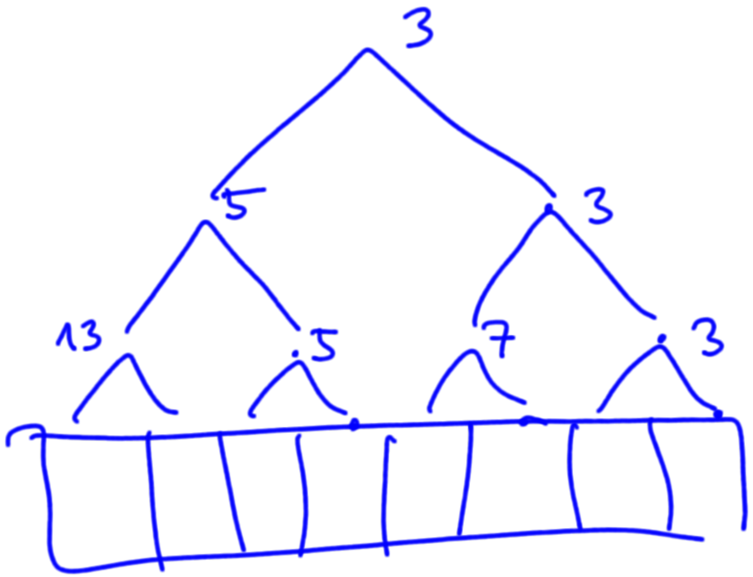
ForkJoinPool:

- constructor creates a number of threads equal to the number of available processors
- .invoke() submits task and waits until it is completed
- .submit() submits task (receives a Future)

Code example: PP-L11-01ForkJoinRecursiveSum



3
[0..7]
5
7
11
13
9
15
11
01
13
01
7
9
15
0
15
0
1
2
3
4
5
6
7



3
5
3
13
5
7
3

Recursive sum with ForkJoin (RecursiveTask Class)

```java
class SumForkJoin extends RecursiveTask<Long> {
    int low;
    int high;
    int[] array;

    SumForkJoin(int[] arr, int lo, int hi) {
        array = arr;
        low   = lo;
        high  = hi;
    }

    protected Long compute() { /*…*/ }
```

Recursive sum with ForkJoin (compute)

```
protected Long compute() {
        if(high - low <= 1)
            return array[high];
        else {
            int mid = low + (high - low) / 2;
            SumForkJoin left  = new SumForkJoin(array, low, mid);
            SumForkJoin right = new SumForkJoin(array, mid, high);
            left.fork();
            right.fork();
            return left.join() + right.join();
        }
    }
```

Results

If you try the code you will observe that it performs poorly.

The reasons have to do with the ForkJoin framework implementation and how it was retrofitted to Java

- this is not a problem inherent to the model
- in other languages this is not a problem

Situation has improved with Java 8
Performance can still be an issue

Fixes/Work-Around – cont’d

```
protected Long compute() {
       if(high - low <= SEQUENTIAL_THRESHOLD) {
           long sum = 0;
           for(int i=low; i < high; ++i)
               sum += array[i];
           return sum;
        } else {
           int mid = low + (high - low) / 2;
           SumForkJoin left  = new SumForkJoin(array, low, mid);
           SumForkJoin right = new SumForkJoin(array, mid, high);
           left.fork();
           long rightAns = right.compute();
           long leftAns  = left.join();
           return leftAns + rightAns;
        }
    }
```

Make sure each task has ‘enough’ to do!

Getting good results in practice

Sequential threshold

- Library documentation recommends doing approximately 100-5000 basic operations in each “piece” of your algorithm

Library needs to “warm up”

- May see slow results before the Java virtual machine re-optimizes the library internals
- Put your computations in a loop to see the “long-term benefit”

Wait until your computer has more processors ☺

- Seriously, overhead may dominate at 4 processors, but parallel programming is likely to become much more important

Beware memory-hierarchy issues

- E.g. locality.

Enough Adding-Up Numbers!

Done:

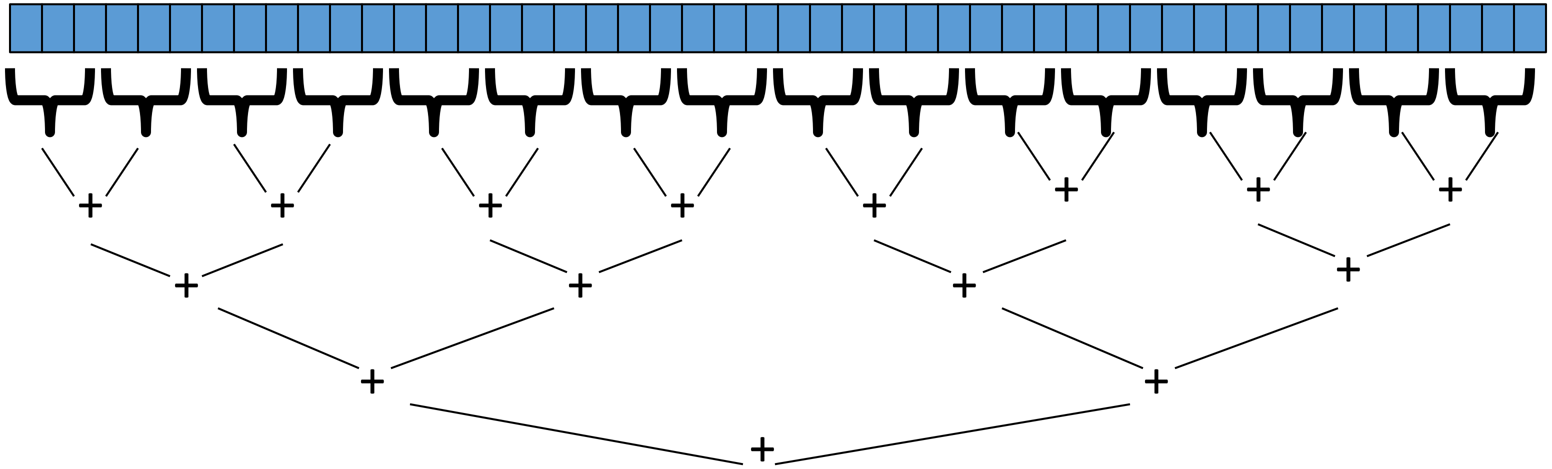
- How to use **`fork`** and **`join`** to write a parallel algorithm
- Why using divide-and-conquer with lots of small tasks is best
    - combines results in parallel
- Some Java and ForkJoin Framework specifics

Now:

- More examples of simple parallel programs
- Arrays & balanced trees support parallelism better than linked lists
- Asymptotic analysis for fork-join parallelism

What else looks like this?

We saw how summing an array went from $O(\boldsymbol{n})$ sequential to $O(\mathtt{log}\ n)$ parallel



Anything that can use results from two halves and merge them in $O(1)$ time has the same property…

Examples

Maximum or minimum element

Is there an element satisfying some property (e.g., is there a 17)?

Left-most element satisfying some property (e.g., first 17)

- What should the recursive tasks return?
- How should we merge the results?

Counts, for example, number of strings that start with a vowel

- This is just summing with a different base case
- Many problems are!

Reductions

Computations of this form are called reductions (or reduces?)

Produce single answer from collection via an associative operator

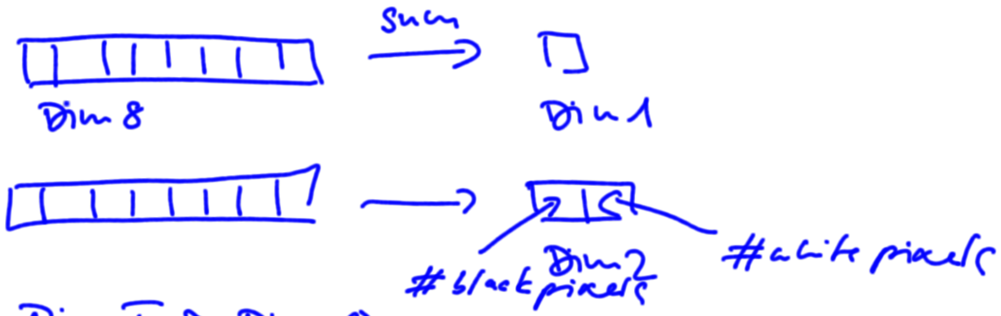
- Examples: max, count, leftmost, rightmost, sum, product, …
- Non-examples: median, subtraction, exponentiation

(Recursive) results don't have to be single numbers or strings. They can be arrays or objects with multiple fields.

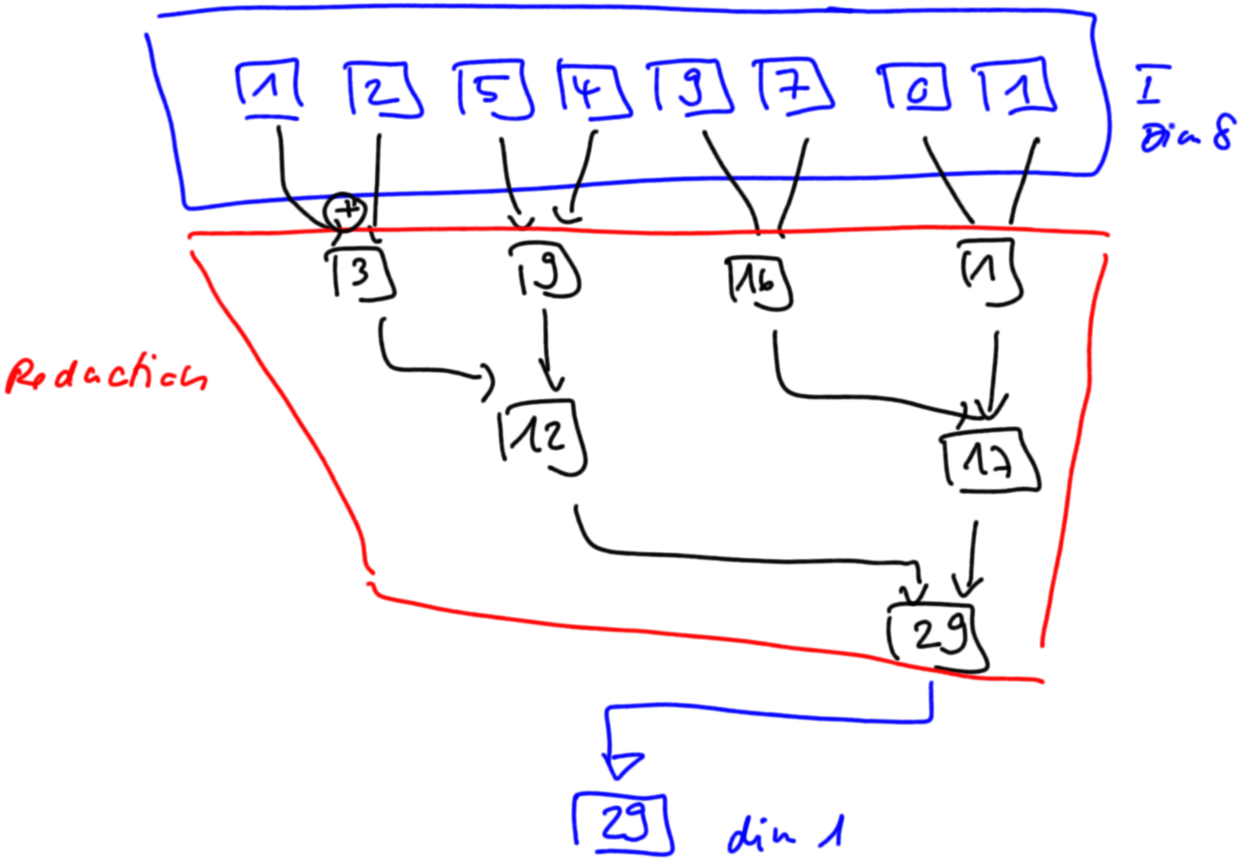
- Example: Histogram of test results is a variant of sum

But some things are inherently sequential

- How we process **`arr[i]`** may depend entirely on the result of processing **`arr[i-1]`**


sum
Dim 8
Dim 1
2
6
Dim 2
#black pixels
#white pixels

Dim I > Dim O


1
2
5
4
9
7
0
1
I
Dim 8
+
3
9
16
1
Reduction
12
17
29
29
dim 1

non-associative: $(1-2)-3 \mathrel{!=} 1-(2-3)$

$$-4 \mathrel{!=} 2$$

combine different results in one result:

same input → same output

→ algo deterministic

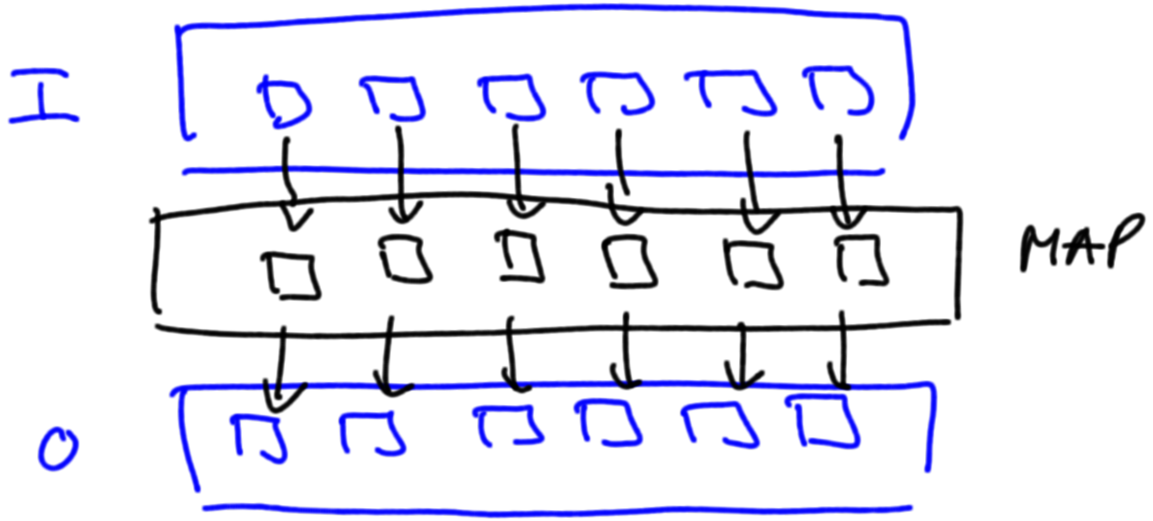
Even easier: Maps (Data Parallelism)

A map operates on each element of a collection independently to create a new collection of the same size

- No combining results
- For arrays, this is so trivial some hardware has direct support

Canonical example: Vector addition

```
int[] vector_add(int[] arr1, int[] arr2){
  assert (arr1.length == arr2.length);
  result = new int[arr1.length];
  FORALL(i=0; i < arr1.length; i++) {
    result[i] = arr1[i] + arr2[i];
  }
  return result;
}
```



$Dim_I = Dim_O$

Code example: PP-L11-02MapsArraySummation

Maps in ForkJoin Framework (compute)

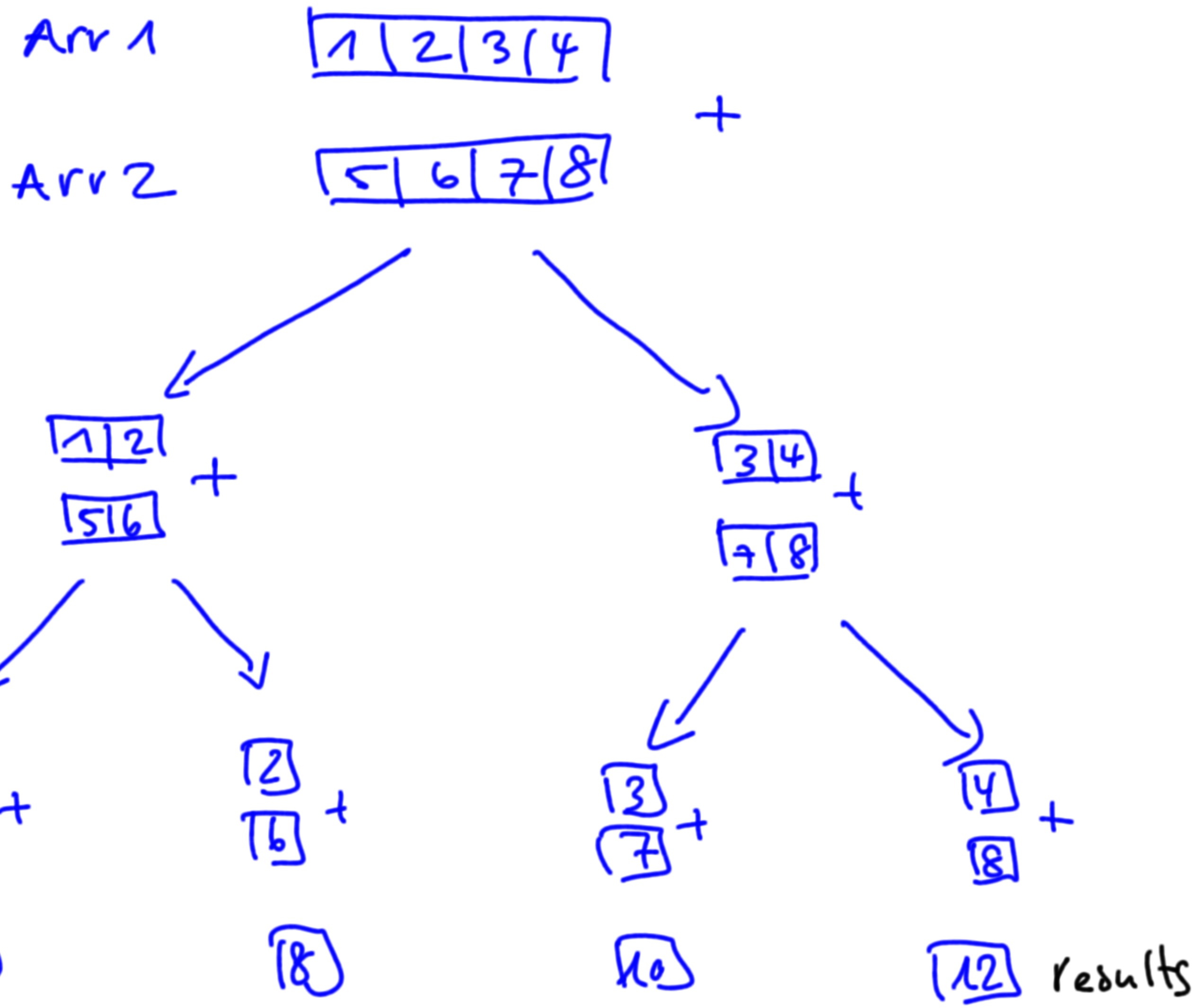
```
protected void compute(){
if(hi-lo < SEQUENTIAL_CUTOFF) {
    for(int i=lo; i < hi; i++)
        res[i] = arr1[i] + arr2[i]; // 2 inputs, 1 output
    } else {
    int mid = (hi+lo)/2;
    VecAdd left = new VecAdd(lo,mid,res,arr1,arr2);
    VecAdd right= new VecAdd(mid,hi,res,arr1,arr2);
    left.fork();
    right.compute();
    left.join();
    }
}
```

Maps in ForkJoin Framework (use)

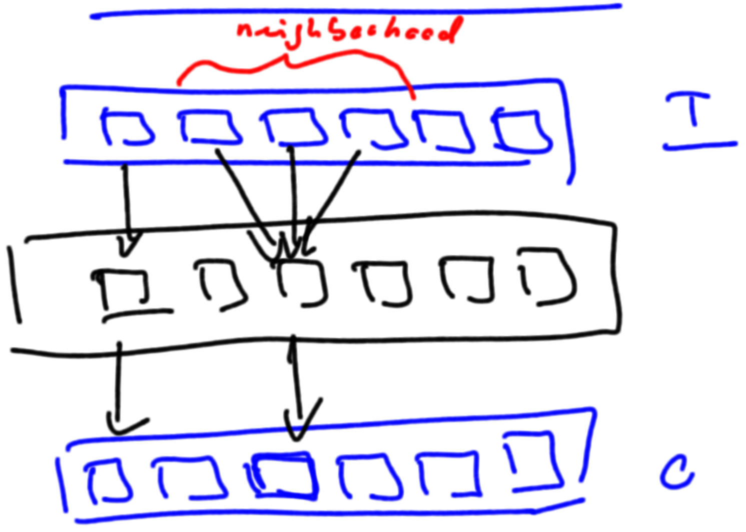
```
class Globals {
    static ForkJoinPool fjPool = new ForkJoinPool();
}

static int[] add(int[] arr1, int[] arr2){
  assert (arr1.length == arr2.length);
  int[] ans = new int[arr1.length];
  Globals.fjPool.invoke(new VecAdd(0,ans.length,ans,arr1,arr2));
  return ans;
}
```

MAP: Vector addition

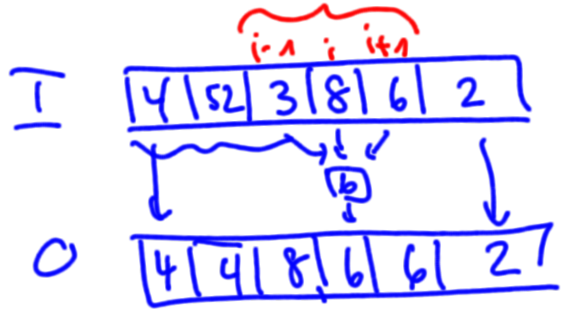


STENCIL PATTERN

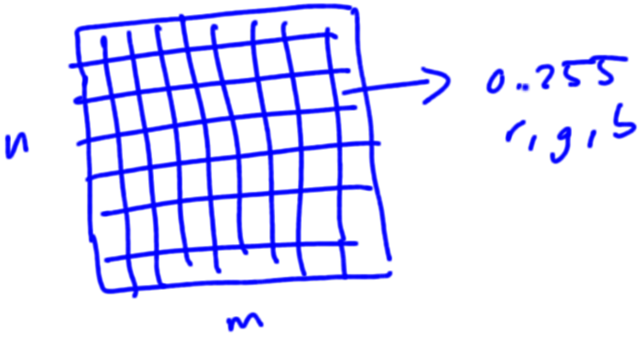


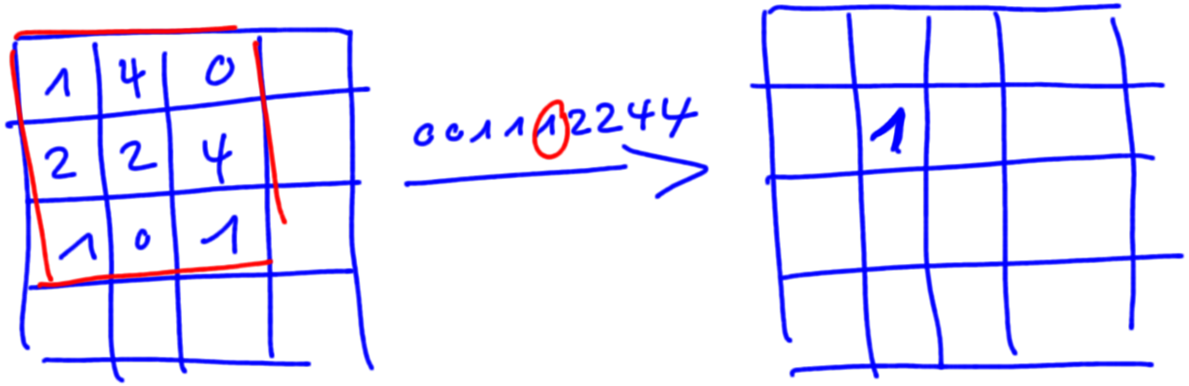
1D MAP: $O(i) = f(I(i))$

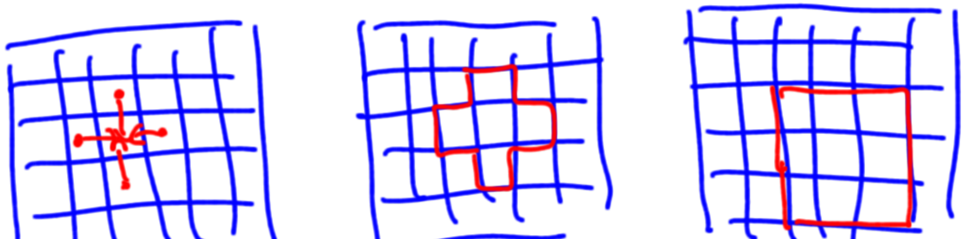
1D STENCIL: $O(i) = f(I(i-1), I(i), I(i+1))$



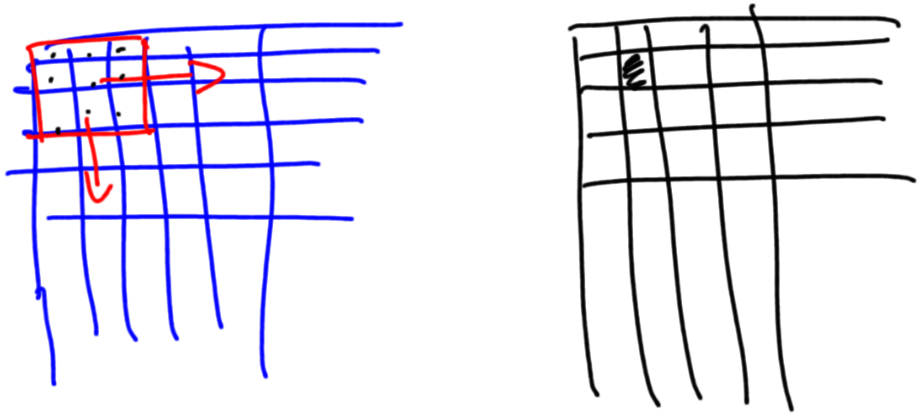
$Dim_I = Dim_O$




$$\begin{array}{|c|c|c|}\hline 1 & 4 & 0 \\ \hline 2 & 2 & 4 \\ \hline 1 & 0 & 1 \\ \hline \end{array} \xrightarrow{0\,0\,1\,1\,\boxed{1}\,2\,2\,4\,4} \begin{array}{|c|c|c|}\hline & & \\ \hline & 1 & \\ \hline & & \\ \hline \end{array}$$


4 point stencil 5 point 9 point

Smoothing kernel: $\frac{1}{9}\begin{bmatrix}1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1\end{bmatrix}$



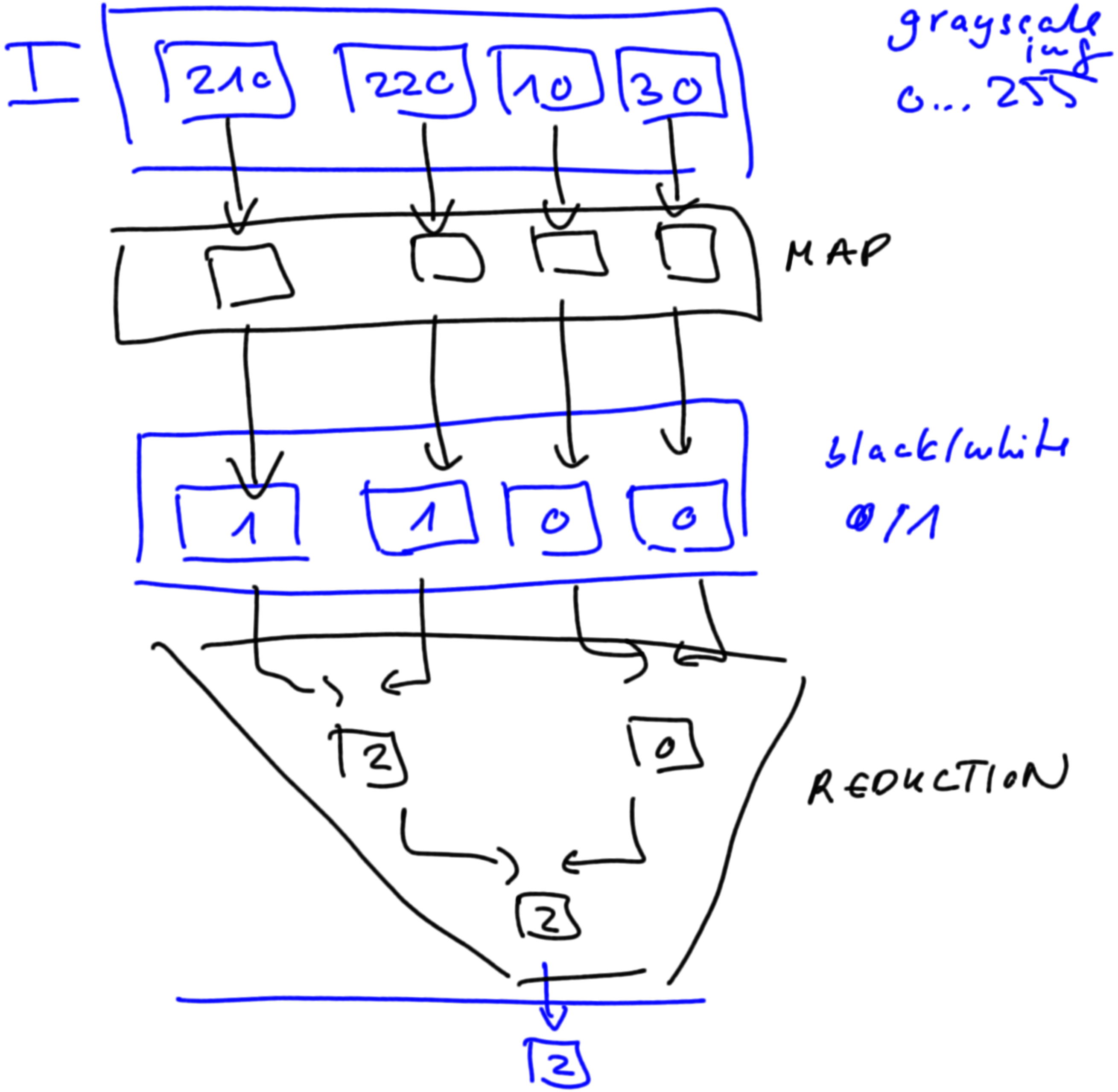
$$O(i,j) = f(I(i,j), I(i-1,j), I(i+1,j), \ldots)$$

Gaussian kernel

Sobel filter: $\begin{bmatrix}-1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1\end{bmatrix}$ $\begin{bmatrix}+1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1\end{bmatrix}$

Gx Gy

$$|G| = \sqrt{Gx^2 + Gy^2}$$


I
210
220
10
30
grayscale img
0...255
MAP
1
1
0
0
black/white
0/1
REDUCTION
2
0
2
2

Maps and reductions

Maps and reductions: the “work horses” of parallel programming

- By far the two most important and common patterns
- Learn to recognize when an algorithm can be written in terms of maps and reductions
- Use maps and reductions to describe (parallel) algorithms
- Programming them becomes “trivial” with a little practice
    - Exactly like sequential for-loops seem second-nature

Digression: MapReduce on clusters

You may have heard of Google’s “map/reduce”

- Or the open-source version Hadoop

Idea: Perform maps/reduces on data using many machines

- The system takes care of distributing the data and managing fault tolerance
- You just write code to map one element and reduce elements to a combined result

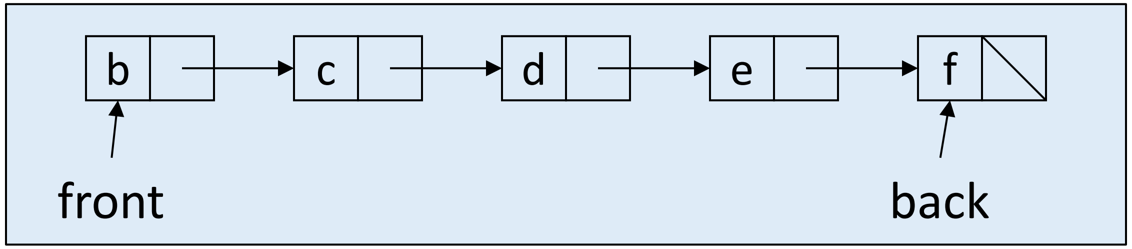
Separates how to do recursive divide-and-conquer from what computation to perform

- Old idea in higher-order functional programming transferred to large-scale distributed computing

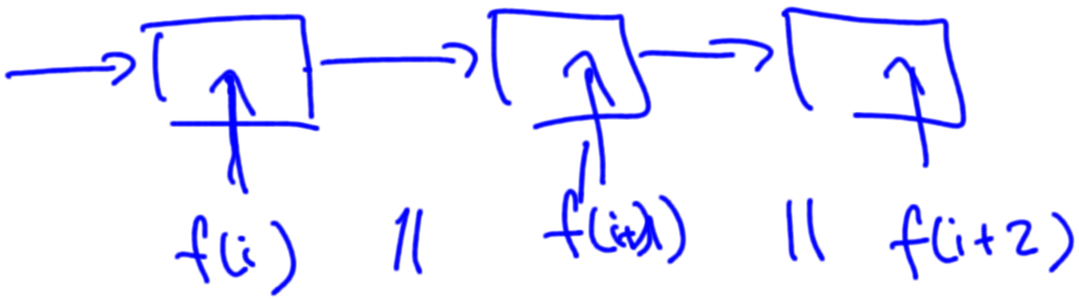
Linked lists

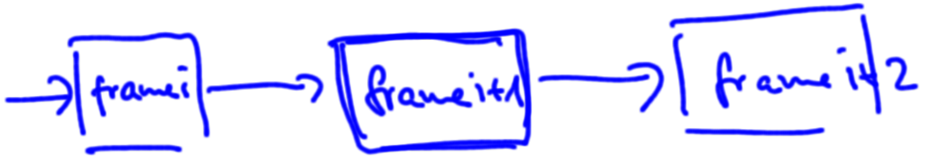
Can you parallelize maps or reduces over linked lists?

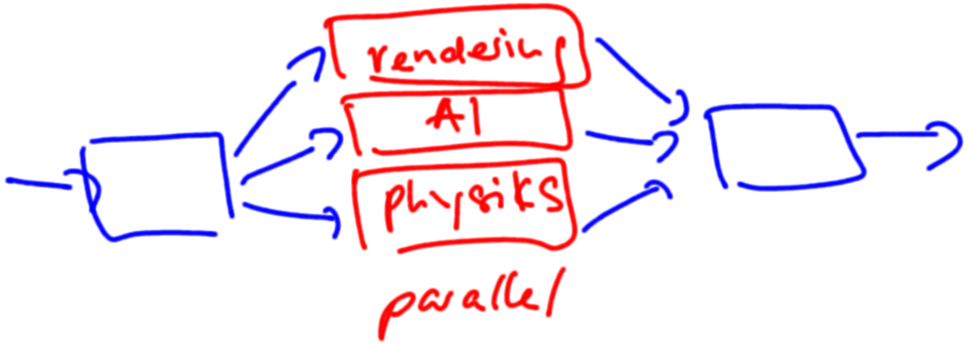
- Example: Increment all elements of a linked list
- Example: Sum all elements of a linked list
- Parallelism still beneficial for expensive per-element operations



- Once again, data structures matter!
- For parallelism, balanced trees generally better than lists so that we can get to all the data exponentially faster *O*(`log` *n*) vs. *O*(*n*)
  - Trees have the same flexibility as lists compared to arrays


f(i)
||
f(i+1)
||
f(i+2)


frame i
frame i+1
frame i+2


rendering
AI
physiks
parallel

Analyzing algorithms

Like all algorithms, parallel algorithms should be correct and efficient

For our algorithms so far, correctness is “obvious” so we’ll focus on efficiency

- Want asymptotic bounds
- Want to analyze the algorithm without regard to a specific number of processors
- The key “magic” of the ForkJoin Framework is getting expected run-time performance asymptotically optimal for the available number of processors
  - So we can analyze algorithms assuming this guarantee

Work and Span [Recap]

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

Two key measures of run-time (recap):

Work: How long it would take 1 processor = $\mathbf{T_1}$

- Just “sequentialize” the recursive forking

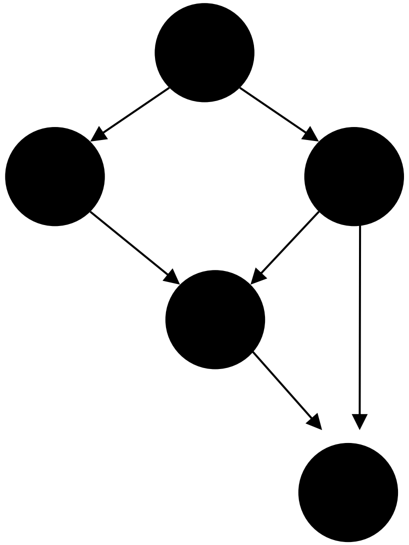
Span: How long it would take infinity processors = $\mathbf{T_\infty}$

- The longest dependence-chain
- Example: $O(\texttt{log}\ n)$ for summing an array
- Also called “critical path length” or “computational depth”

The DAG

A program execution using **`fork`** and **`join`** can be seen as a DAG

- Nodes: Pieces of work
- Edges: Source must finish before destination starts

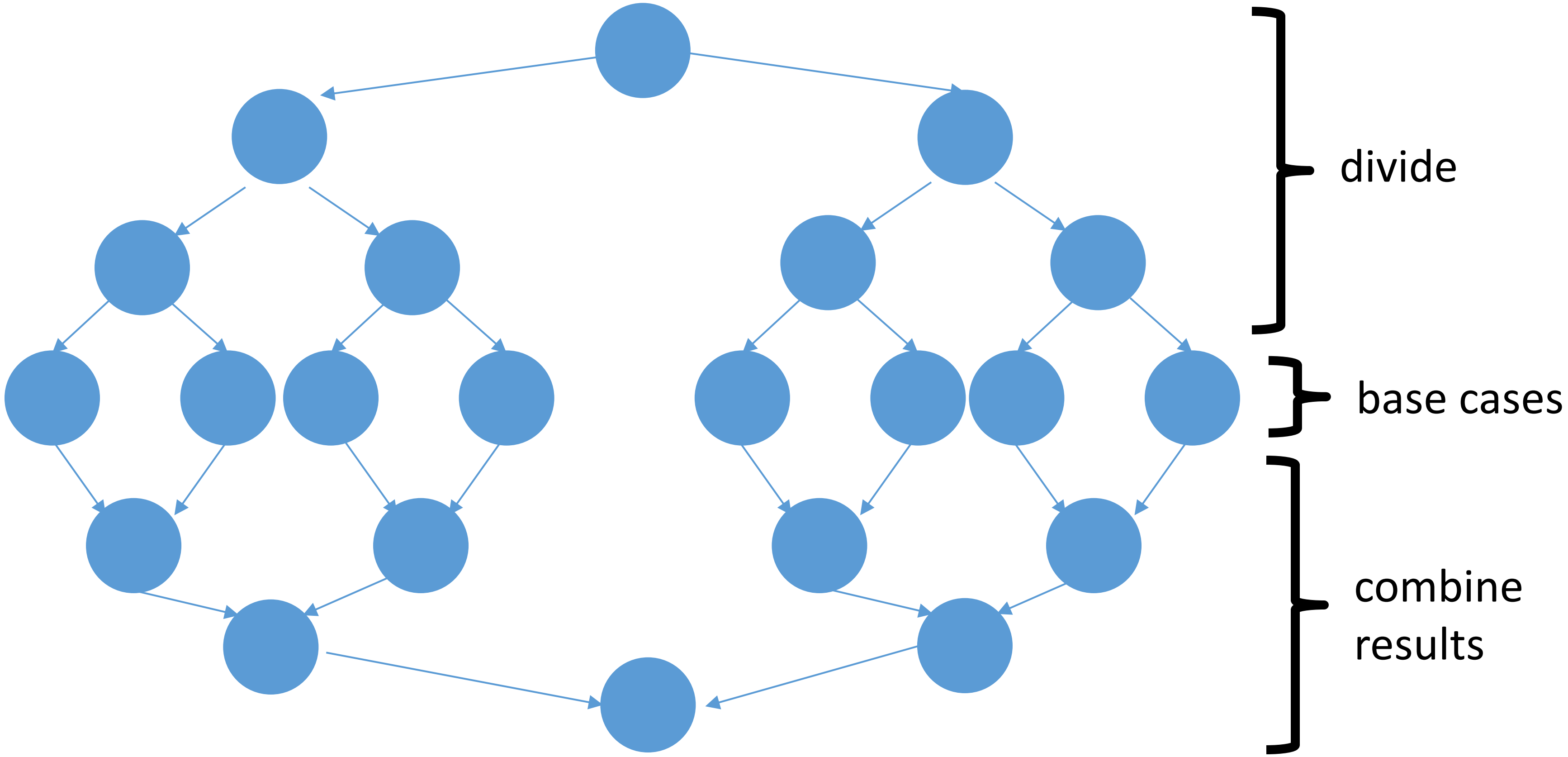


- A `fork` “ends a node” and makes two outgoing edges
  - New thread
  - Continuation of current thread
- A `join` “ends a node” and makes a node with two incoming edges
  - Task just ended
  - Last node of thread joined on

Our simple examples

`fork` and `join` are very flexible, but divide-and-conquer maps and reductions use them in a very basic way:

- A tree on top of an upside-down tree



Connecting to performance

Recall: $\mathbf{T_P}$ = running time if there are **P** processors available

Work = $\mathbf{T_1}$ = sum of run-time of all nodes in the DAG

- That lonely processor does everything
- *O*(*n*) for simple maps and reductions

Span = $\mathbf{T_\infty}$ = sum of run-time of all nodes on the most-expensive path in the DAG

- Note: costs are on the nodes not the edges
- Our infinite army can do everything that is ready to be done, but still has to wait for earlier results
- *O*(**`log`** *n*) for simple maps and reductions

Recap: Definitions

A couple more terms:

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 won't help
- What that point is depends on the span

Designing parallel algorithms is about decreasing span without increasing work too much

Optimal T_P: Thanks ForkJoin library!

So we know $\mathbf{T_1}$ and $\mathbf{T_\infty}$ but we want $\mathbf{T_P}$ (e.g., for **p**=4)

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**

The ForkJoin Framework gives an *expected-time guarantee* of asymptotically optimal!

- Expected time because it flips coins when ***scheduling***
- How? For an advanced course (few need to know)
- Guarantee requires a few assumptions about your code…

Division of responsibility

Our job as ForkJoin Framework users:

- Pick a good algorithm, write a program
- When run, program creates a DAG of things to do
- *Make all the nodes a small-ish and approximately equal amount of work*

The framework-writer's job:

- Assign work to available processors to avoid idling
  - Let framework-user ignore all scheduling issues
- Keep constant factors low
- Give the expected-time optimal guarantee assuming framework-user did his/her job

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

Examples

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

In the algorithms seen so far (e.g., sum an array):

- $\mathbf{T_1} = O(n)$
- $\mathbf{T_\infty} = O(\texttt{log } n)$
- So expect (ignoring overheads): $\mathbf{T_P = \mathit{O}(\mathit{n}/P + \texttt{log } \mathit{n})}$

Suppose instead:

- $\mathbf{T_1} = O(n^2)$
- $\mathbf{T_\infty} = O(n)$
- So expect (ignoring overheads): $\mathbf{T_P = \mathit{O}(\mathit{n}^2/P + \mathit{n})}$

Amdahl’s Law (mostly bad news)

So far: analyze parallel programs in terms of work and span

In practice, typically have parts of programs that parallelize well…

- Such as maps/reductions over arrays and trees

…and parts that don’t parallelize at all

- Such as reading a linked list, getting input, doing computations where each needs the previous step, etc.

All is not lost

Amdahl's Law is a bummer!

- Unparallelized parts become a bottleneck very quickly
- But it doesn't 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

The prefix-sum problem

Given `int[] input`,

produce `int[] output` where:

`output[i]` = `input[0]`+`input[1]`+…+`input[i]`

Sequential prefix-sum

```
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: $O(\log n)$

Example

How to compute the prefix-sum in parallel?

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

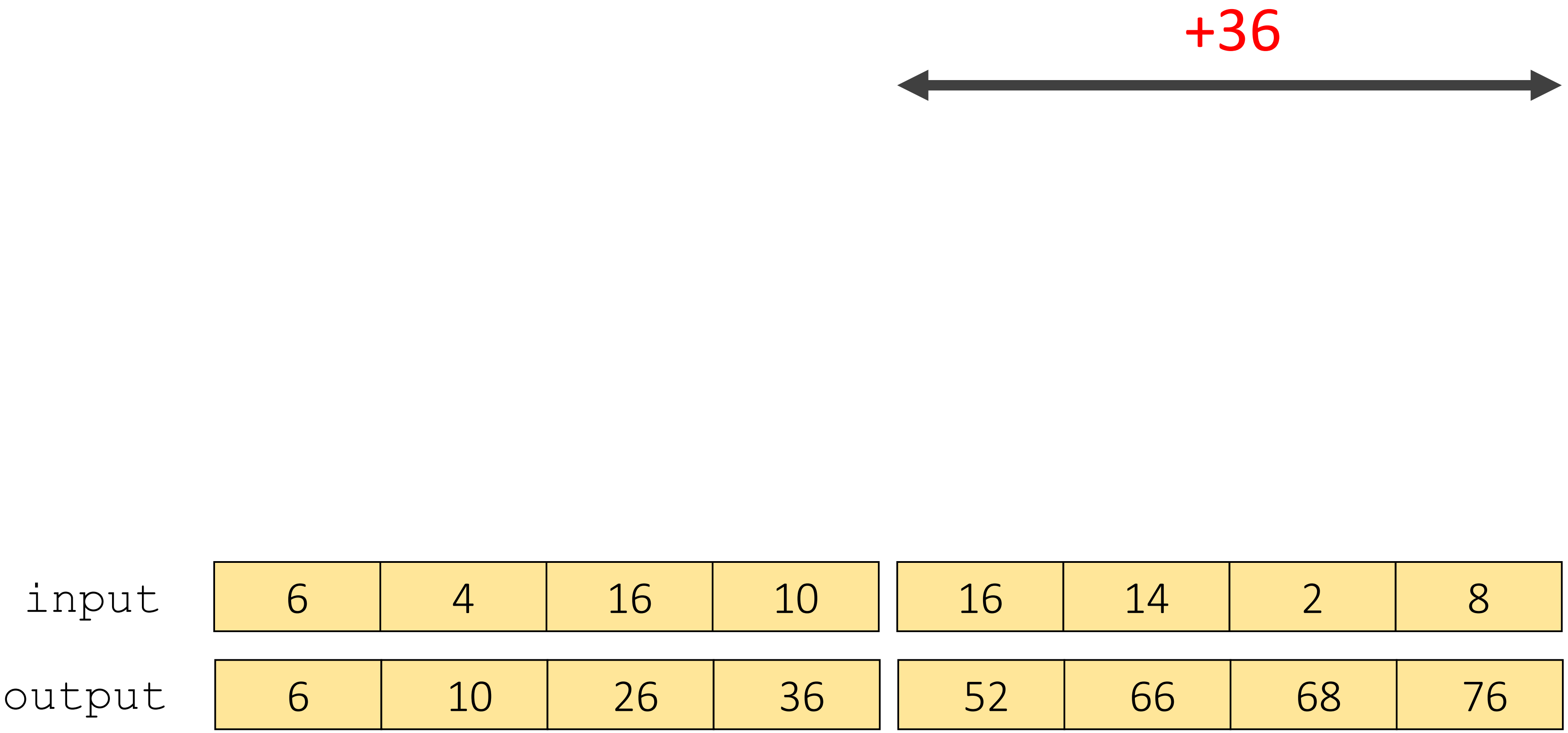
Example

input	6	4	16	10	16	14	2	8
output								

Example

`input`	6	4	16	10	16	14	2	8
`output`	6	10	26	36	16	30	32	40

Example



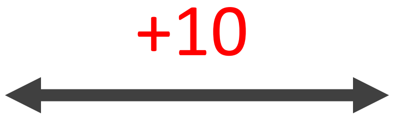
+36

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

Example

`input`	6	4	16	10	16	14	2	8
`output`	6	10	16	26	16	30	2	10

Example



input	6	4	16	10	16	14	2	8
output	6	10	26	36	16	30	2	10

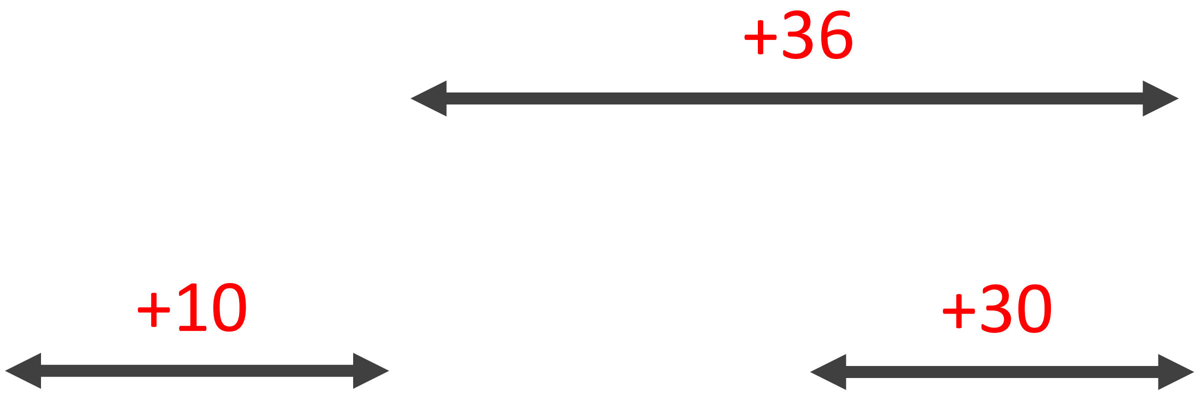
Example

+10

+30

input	6	4	16	10	16	14	2	8
output	6	10	26	36	16	30	32	40

Example



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

Parallel prefix-sum

The parallel-prefix algorithm does two passes

- Each pass has $O(n)$ work and $O(\mathtt{log}\ n)$ span
- So in total there is $O(n)$ work and $O(\mathtt{log}\ n)$ span
- So like with array summing, parallelism is $n/\mathtt{log}\ n$

First pass builds a tree bottom-up: the “up” pass

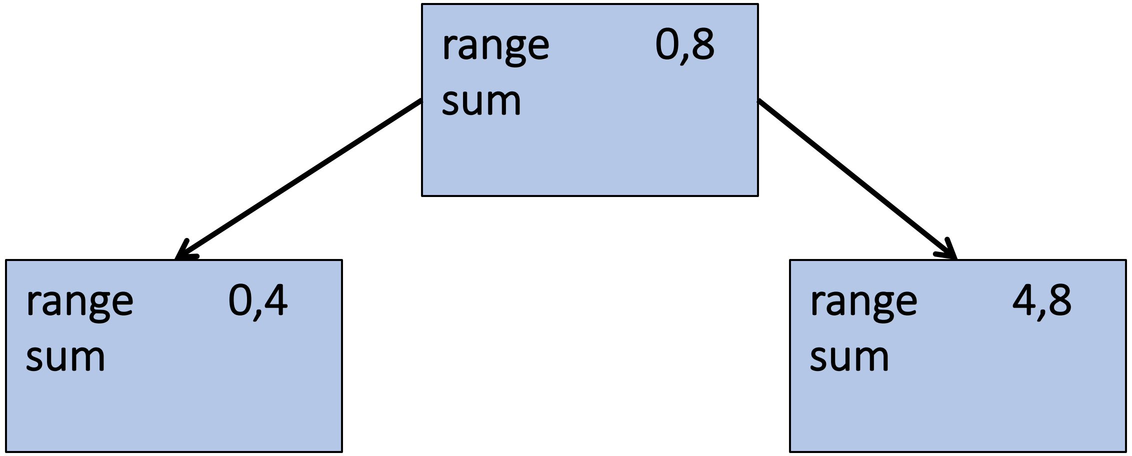
Second pass traverses the tree top-down: the “down” pass

Example

range 0,8
sum

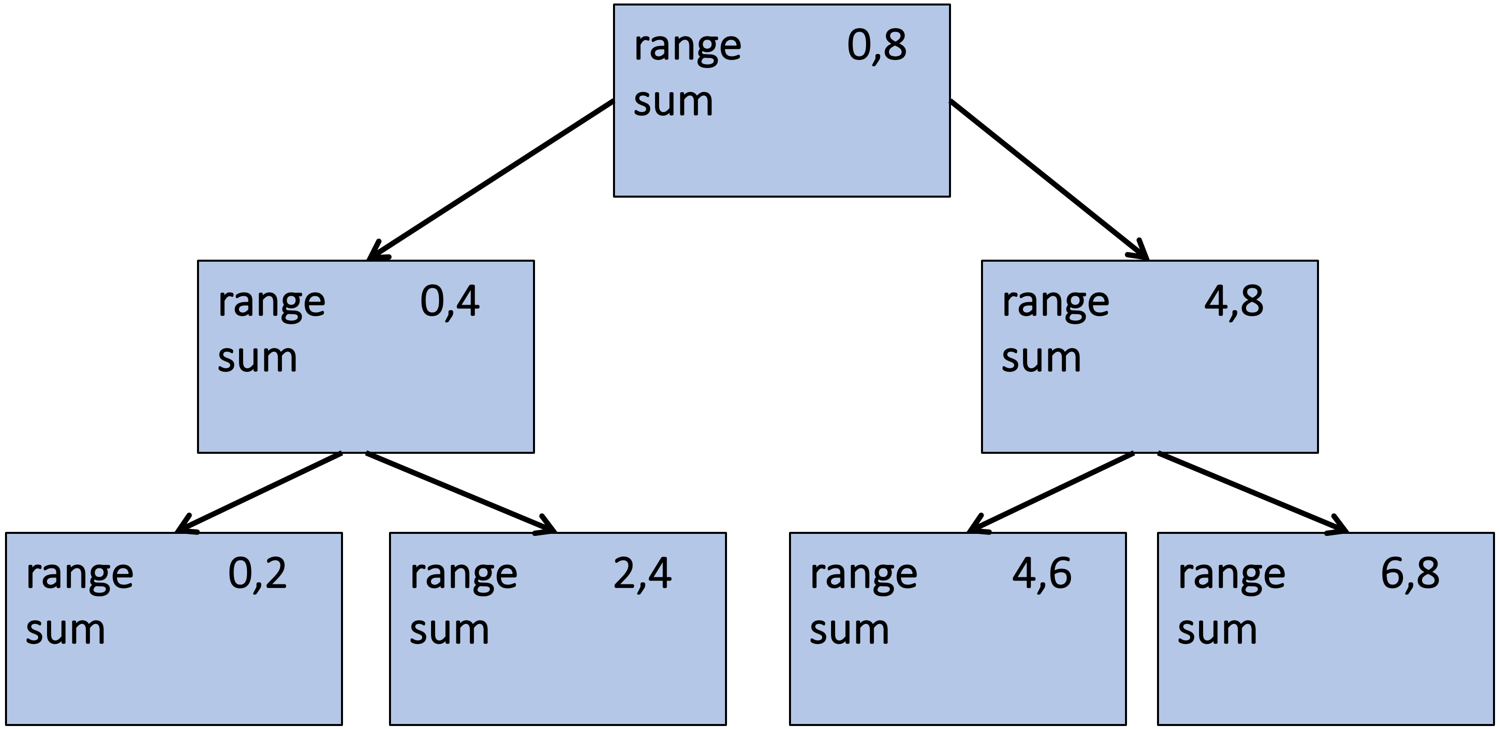
`input`	6	4	16	10	16	14	2	8
`output`								

Example



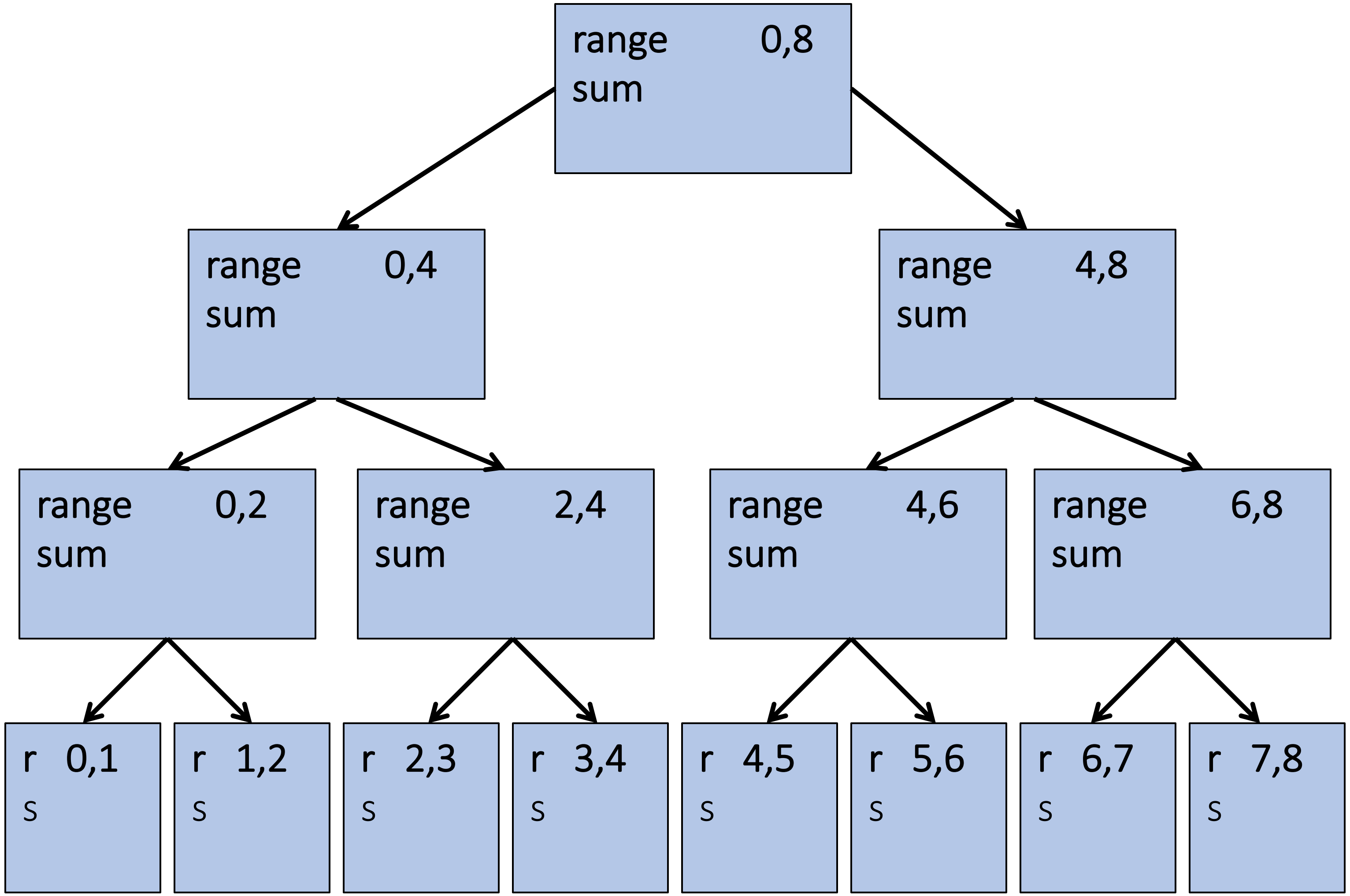
input	6	4	16	10	16	14	2	8
output								

Example



input	6	4	16	10	16	14	2	8
output								

Example



range 0,8
sum

range 0,4
sum

range 4,8
sum

range 0,2
sum

range 2,4
sum

range 4,6
sum

range 6,8
sum

r 0,1
s

r 1,2
s

r 2,3
s

r 3,4
s

r 4,5
s

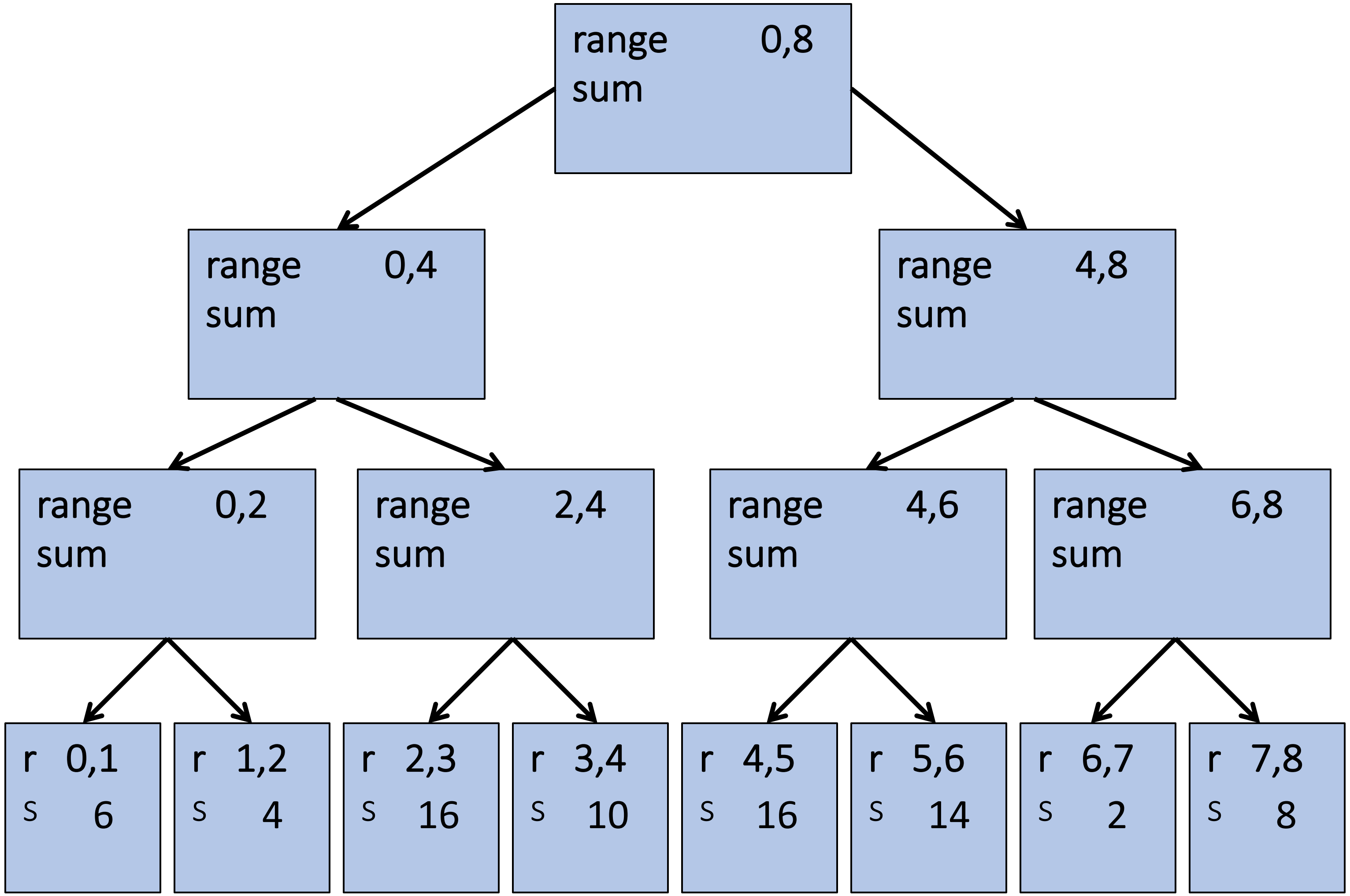
r 5,6
s

r 6,7
s

r 7,8
s

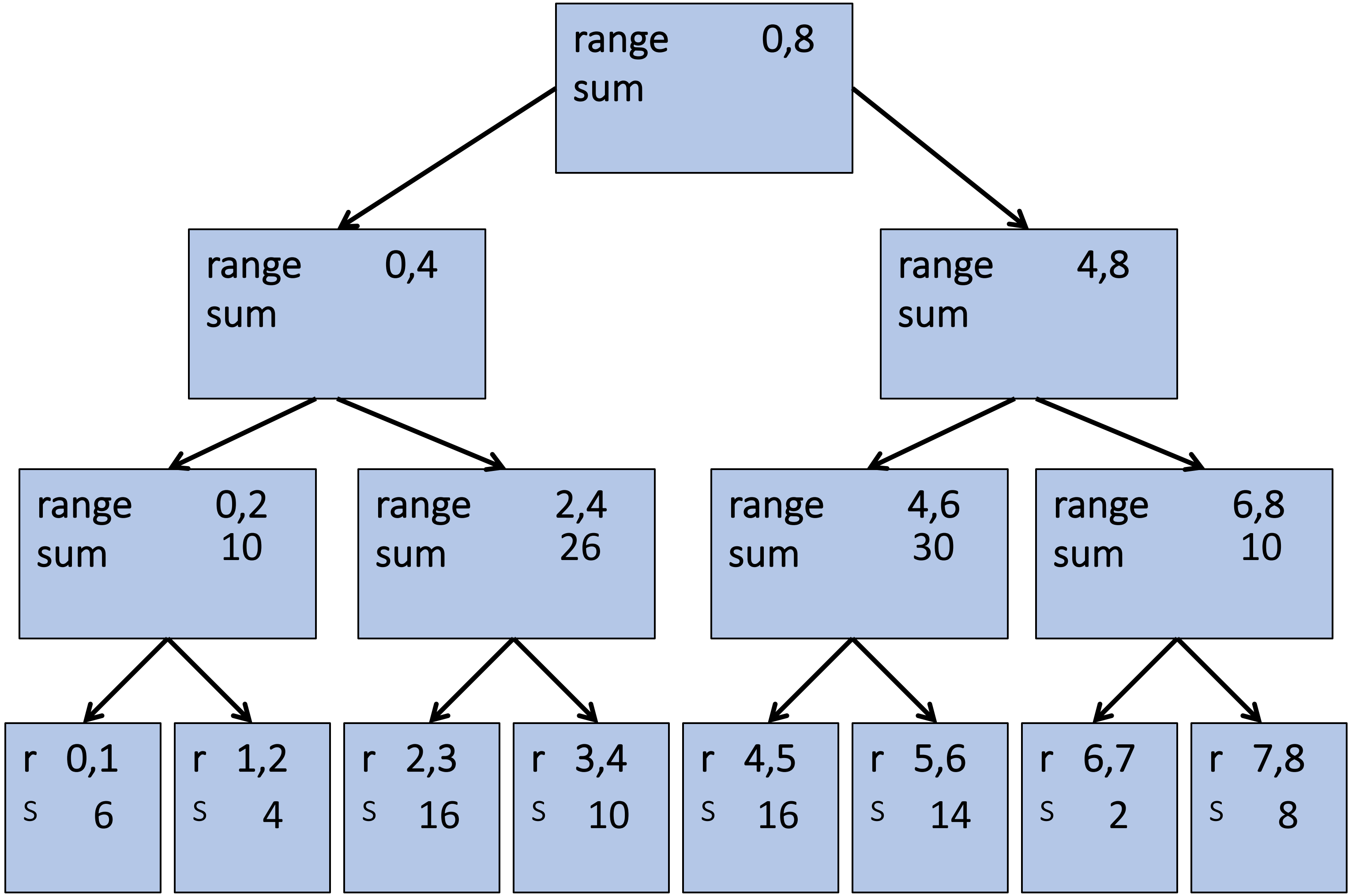
input	6	4	16	10	16	14	2	8
output								

Example



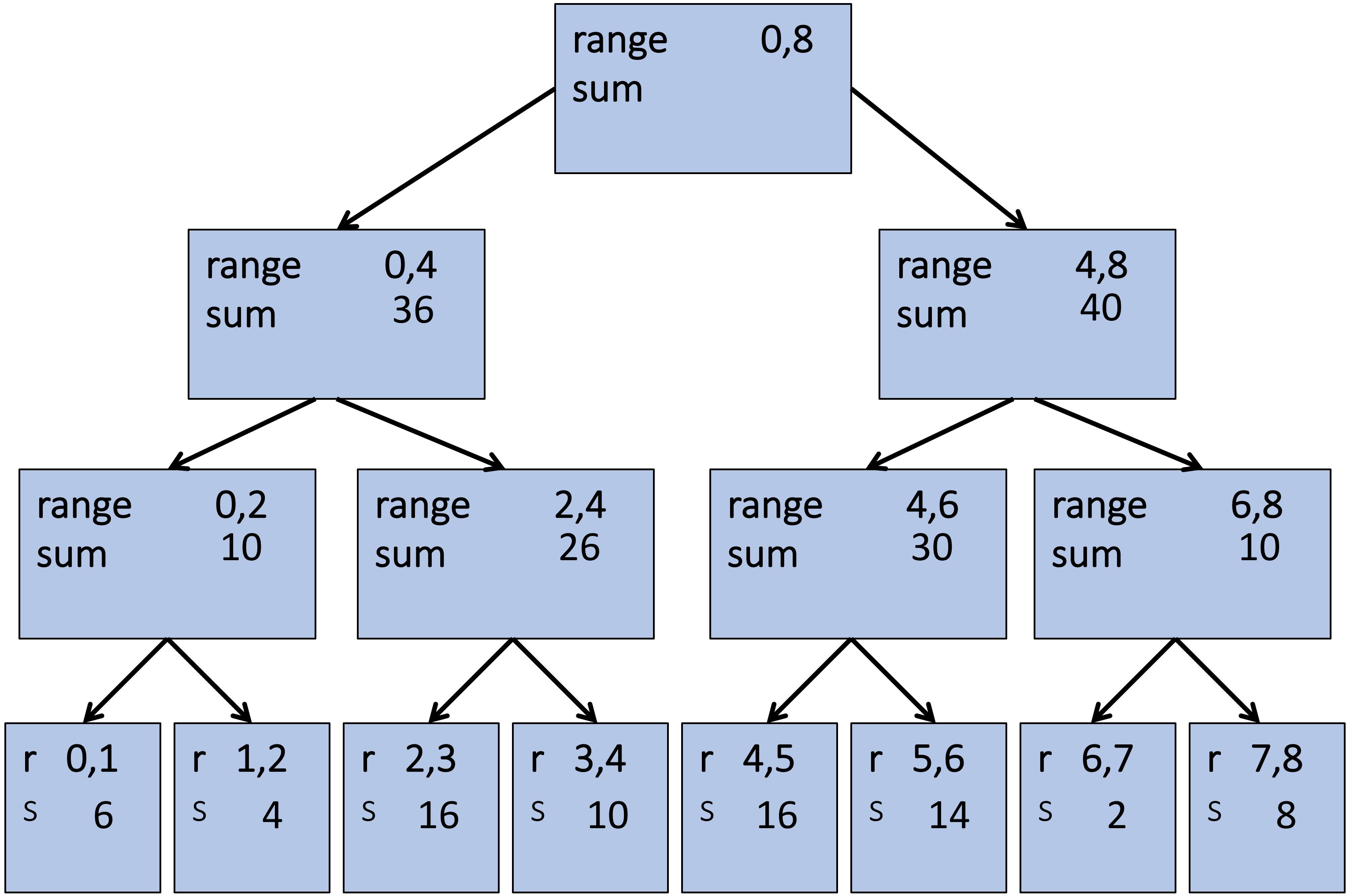
input	6	4	16	10	16	14	2	8
output								

Example



input	6	4	16	10	16	14	2	8
output								

Example



range 0,8
sum

range 0,4
sum 36

range 4,8
sum 40

range 0,2
sum 10

range 2,4
sum 26

range 4,6
sum 30

range 6,8
sum 10

r 0,1
s 6

r 1,2
s 4

r 2,3
s 16

r 3,4
s 10

r 4,5
s 16

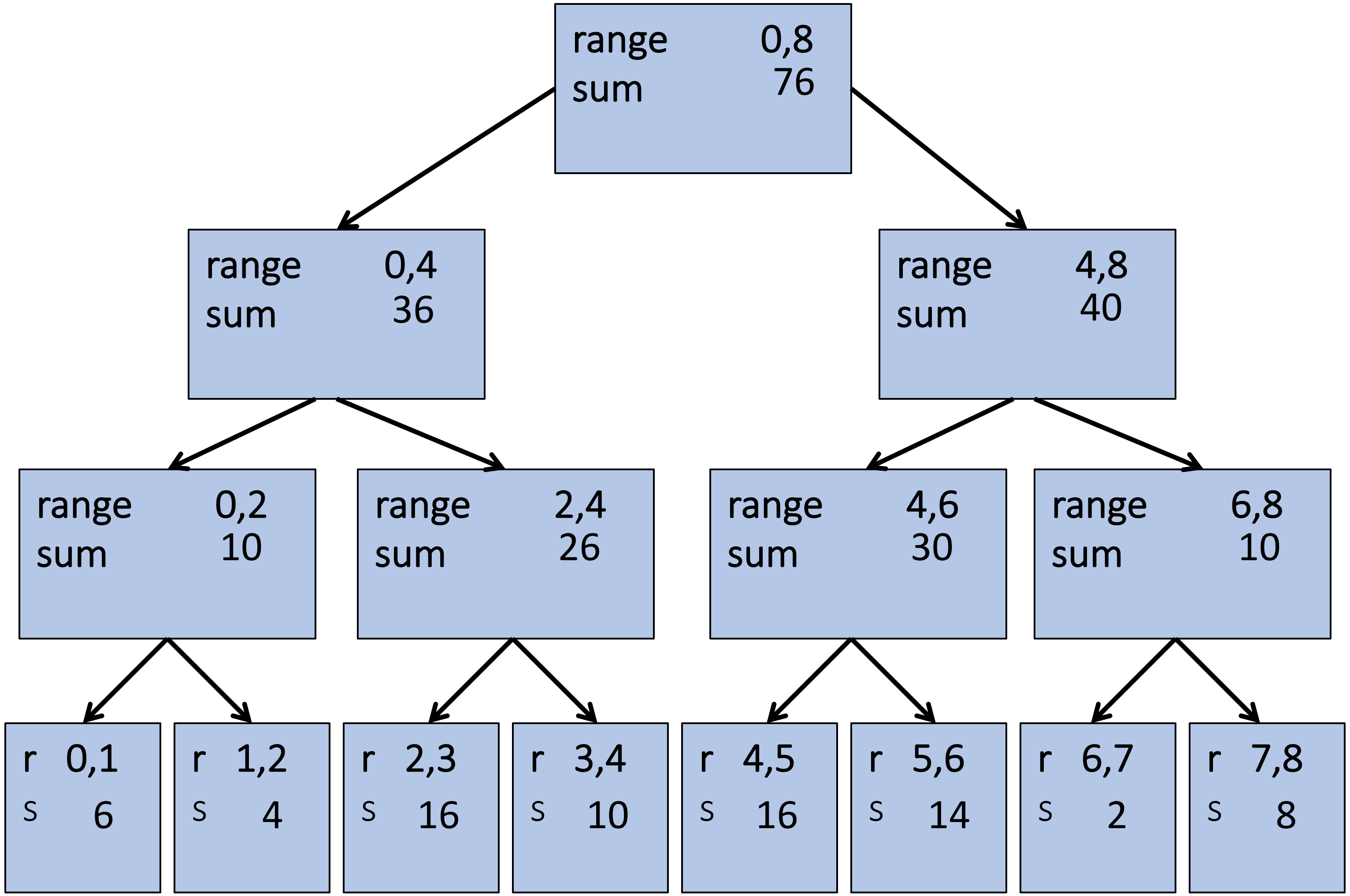
r 5,6
s 14

r 6,7
s 2

r 7,8
s 8

input	6	4	16	10	16	14	2	8
output								

Example



range 0,8
sum 76

range 0,4
sum 36

range 4,8
sum 40

range 0,2
sum 10

range 2,4
sum 26

range 4,6
sum 30

range 6,8
sum 10

r 0,1
s 6

r 1,2
s 4

r 2,3
s 16

r 3,4
s 10

r 4,5
s 16

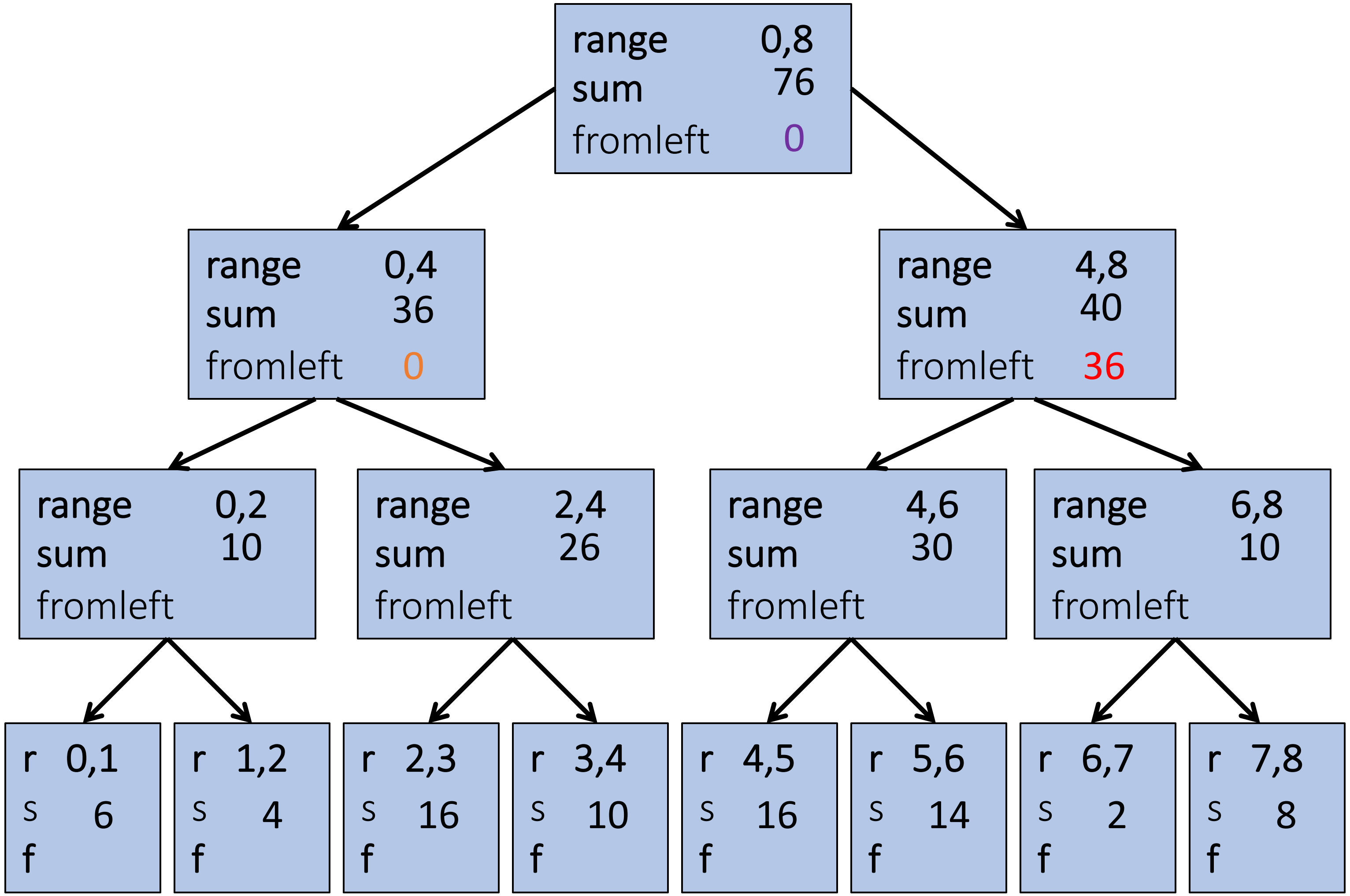
r 5,6
s 14

r 6,7
s 2

r 7,8
s 8

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

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

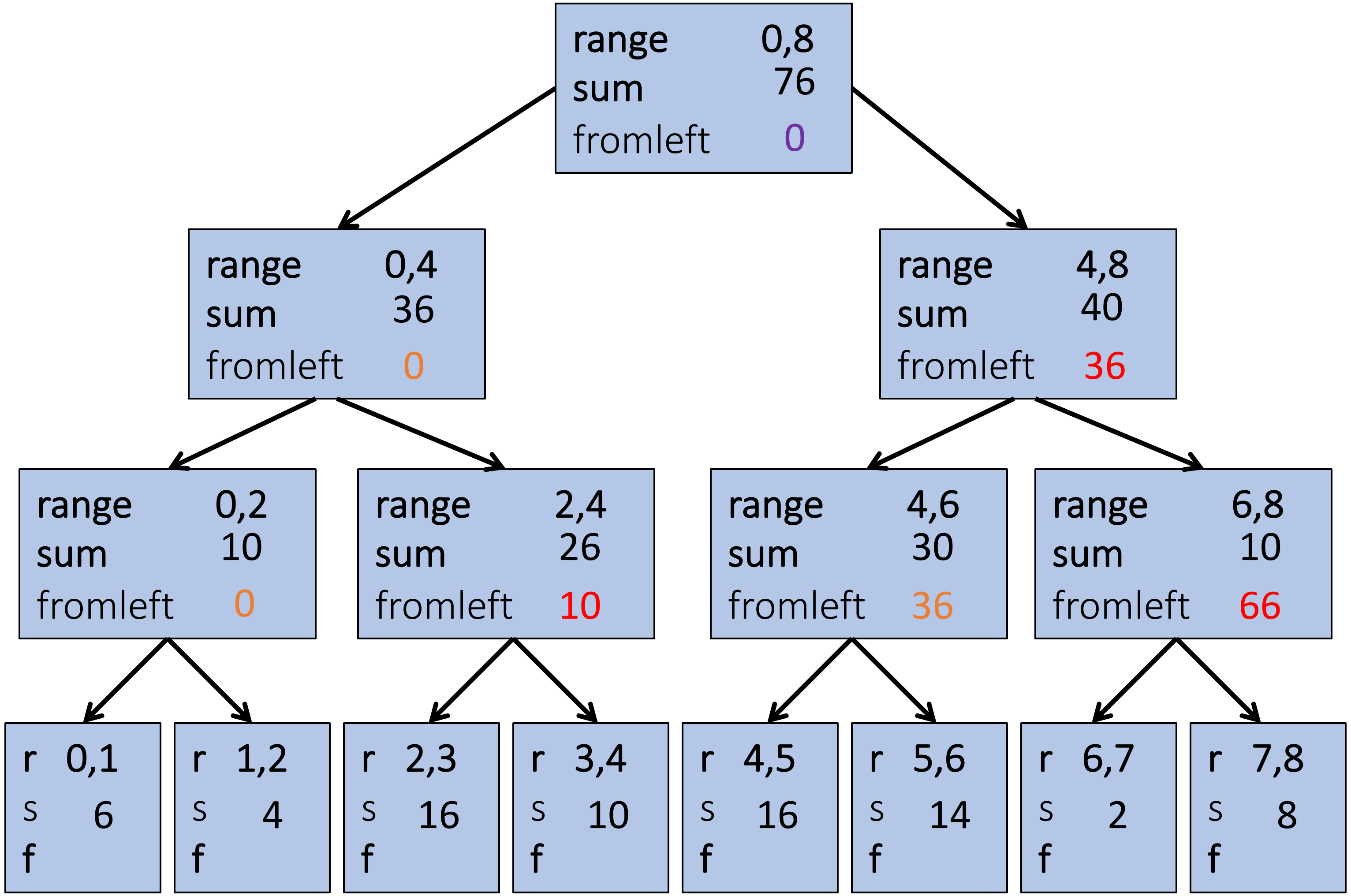
r 5,6
s 14
f

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

r 1,2
s 4
f

r 2,3
s 16
f

r 3,4
s 10
f

r 4,5
s 16
f

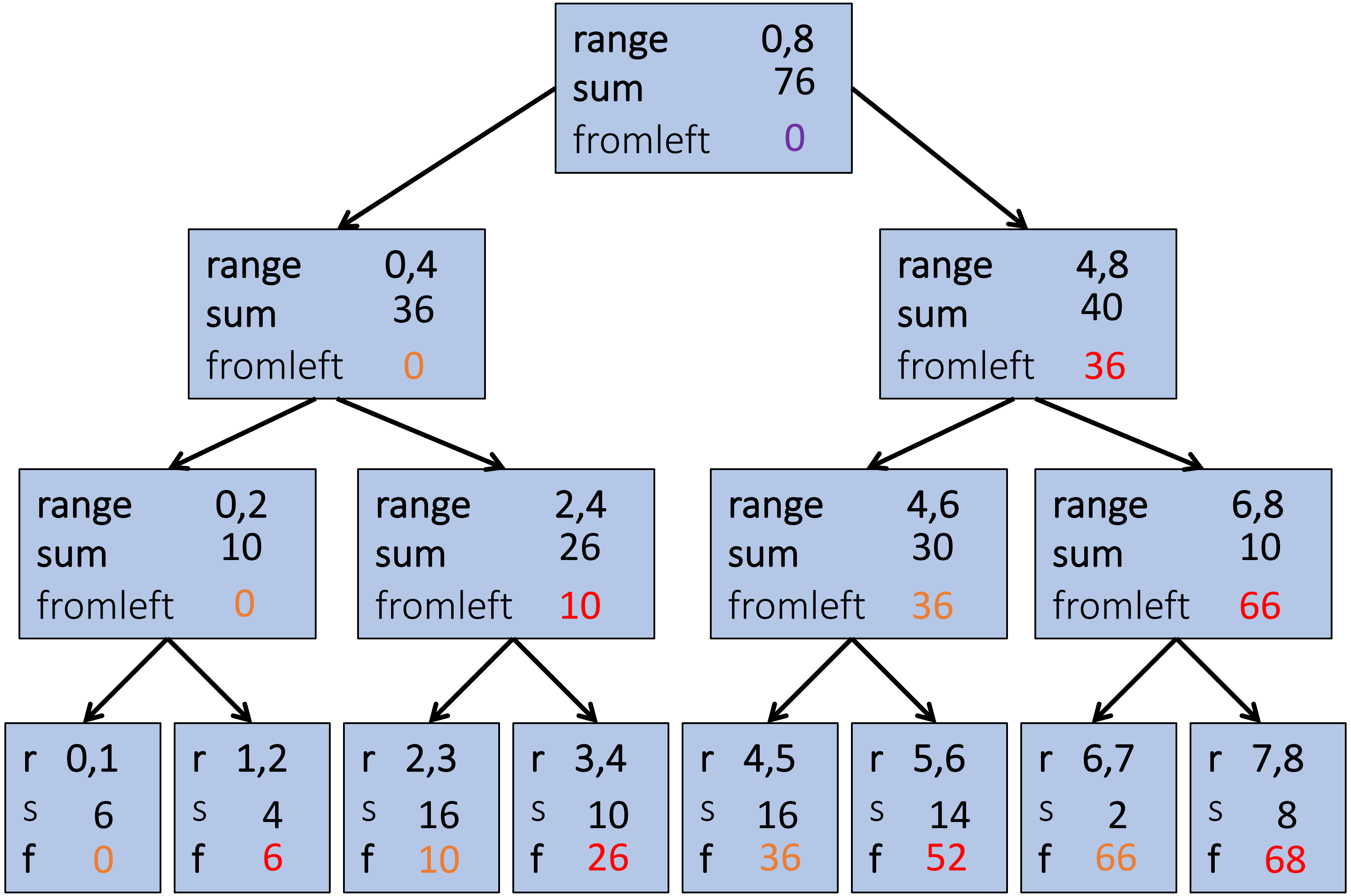
r 5,6
s 14
f

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

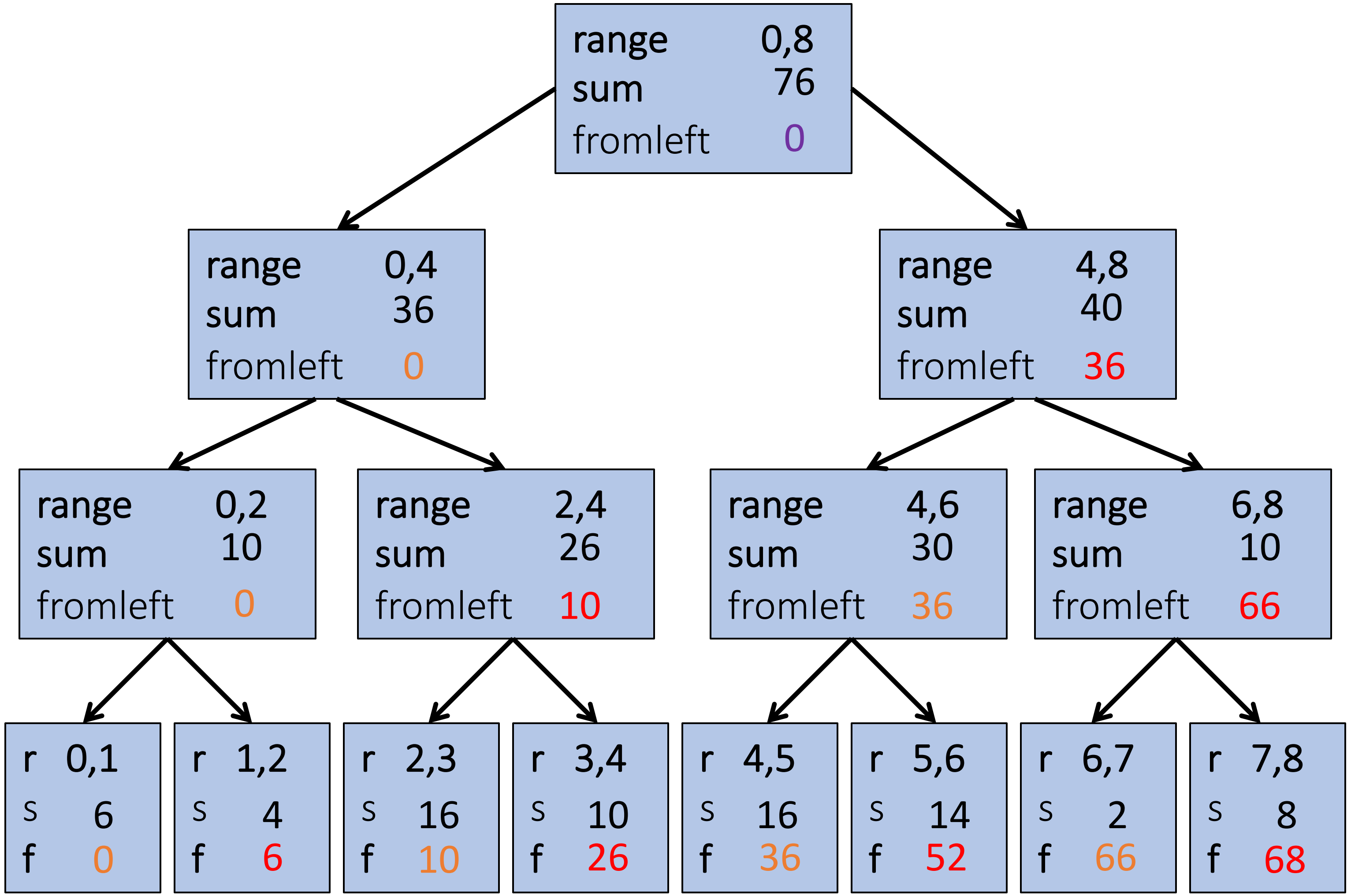
r 5,6
s 14
f 52

r 6,7
s 2
f 66

r 7,8
s 8
f 68

input	6	4	16	10	16	14	2	8
output								

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 all the range-sums

- Tree built bottom-up in parallel

Analysis: *O*(*n*) work, *O*(**`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`** (as 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]
```

Parallel prefix, generalized

Just as sum-array was the simplest example of a common pattern, prefix-sum illustrates a pattern that arises 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]

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 prefix to the rescue

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]
```

Parallel prefix to the rescue

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]
```

Parallel prefix to the rescue

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 (with condition) 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

- Just using 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

- 2 or 3 passes, but 3 is a constant

Parallelized packs will help us parallelize quicksort…

Quicksort review

Recall Quicksort was sequential, in-place, expected time $O(n \log n)$

	Best / expected case *work*
1. Pick a pivot element	O(1)
2. Partition all the data into:	O(n)
A. The elements less than the pivot	
B. The pivot	
C. The elements greater than the pivot	
3. Recursively sort A and C	2T(n/2)

How should we parallelize this?

Quicksort

	Best / expected case *work*
1. Pick a pivot element	O(1)
2. Partition all the data into:	O(n)
A. The elements less than the pivot	
B. The pivot	
C. The elements greater than the pivot	
3. Recursively sort A and C	2T(n/2)

Easy: Do the two recursive calls in parallel

- Work: unchanged of course $O(n \log n)$
- Span: now $T(n) = O(n) + 1T(n/2) = O(n)$
- So parallelism (i.e., work / span) is $O(\log n)$

Doing better

$O($`log` $n)$ speed-up with an infinite number of processors is okay, but a bit underwhelming

- Sort 10^9 elements 30 times faster

Google searches strongly suggest quicksort cannot do better because the partition cannot be parallelized

- The Internet has been known to be wrong ☺
- But we need auxiliary storage (no longer in place)
- In practice, constant factors may make it not worth it, but remember Amdahl's Law

Already have everything we need to parallelize the partition…

Parallel partition (not in place)

Partition all the data into:

A. The elements less than the pivot
B. The pivot
C. The elements greater than the pivot

This is just two packs!

- We know a pack is $O(n)$ work, $O(\mathtt{log}\ n)$ span
- Pack elements less than pivot into left side of **`aux`** array
- Pack elements greater than pivot into right size of **`aux`** array
- Put pivot between them and recursively sort
- With a little more cleverness, can do both packs at once but no effect on asymptotic complexity

With $O(\mathtt{log}\ n)$ span for partition, the total span for quicksort is

$T(n) = O(\mathtt{log}\ n) + 1T(n/2) = O(\mathtt{log}^2\ n)$

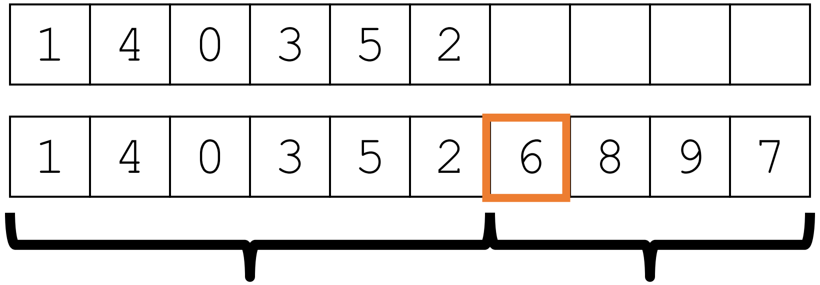
Example

Step 1: pick pivot as median of three

8	1	4	9	**0**	3	5	2	7	**6**

Steps 2a and 2c (combinable): pack less than, then pack greater than into a second array

- Fancy parallel prefix to pull this off <u>not shown</u>



1	4	0	3	5	2				
1	4	0	3	5	2	6	8	9	7

Step 3: Two recursive sorts in parallel

- Can sort back into original array (like in mergesort)

Summary

Task Parallelism is simple yet powerful parallel programming paradigm

Based on the idea of divide and conquer (parallel equivalent to recursion)

Can be done by ‘hand’ or using dedicated frameworks

Java’s implementation is the Fork/Join framework

Getting good performance is not always easy -> but scales well with additional parallelism