CPSC 221
Basic Algorithms and Data Structures

A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency, Part 2

Analysis of Fork-Join Parallel Programs

Steve Wolfman, based on work by Dan Grossman
(with minor tweaks by Hassan Khosravi)

Learning Goals

- Define work—the time it would take one processor to complete a parallelizable computation; span—the time it would take an infinite number of processors to complete the same computation; and Amdahl's Law—which relates the speedup in a program to the proportion of the program that is parallelizable.

- Use work, span, and Amdahl's Law to analyse the speedup available for a particular approach to parallelizing a computation.

- Judge appropriate contexts for and apply the parallel map, parallel reduce, and parallel prefix computation patterns.

Outline

Done:

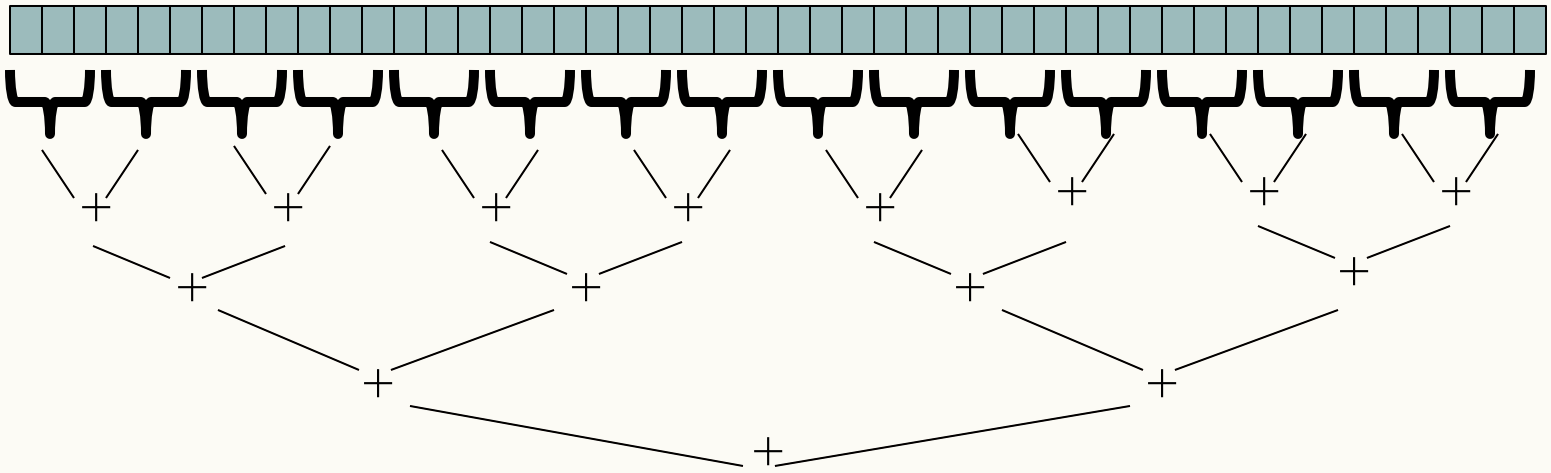
- How to use **`fork`** and **`join`** to write a parallel algorithm
- Why using divide-and-conquer with lots of small tasks is best
  - Combines results in parallel
- Some C++11 and OpenMP specifics
  - More pragmatics (e.g., installation) in separate notes

Now:

- More examples of simple parallel programs
- Other data structures that support parallelism (or not)
- Asymptotic analysis for fork-join parallelism
- Amdahl's Law

Easier Visualization for the Analysis

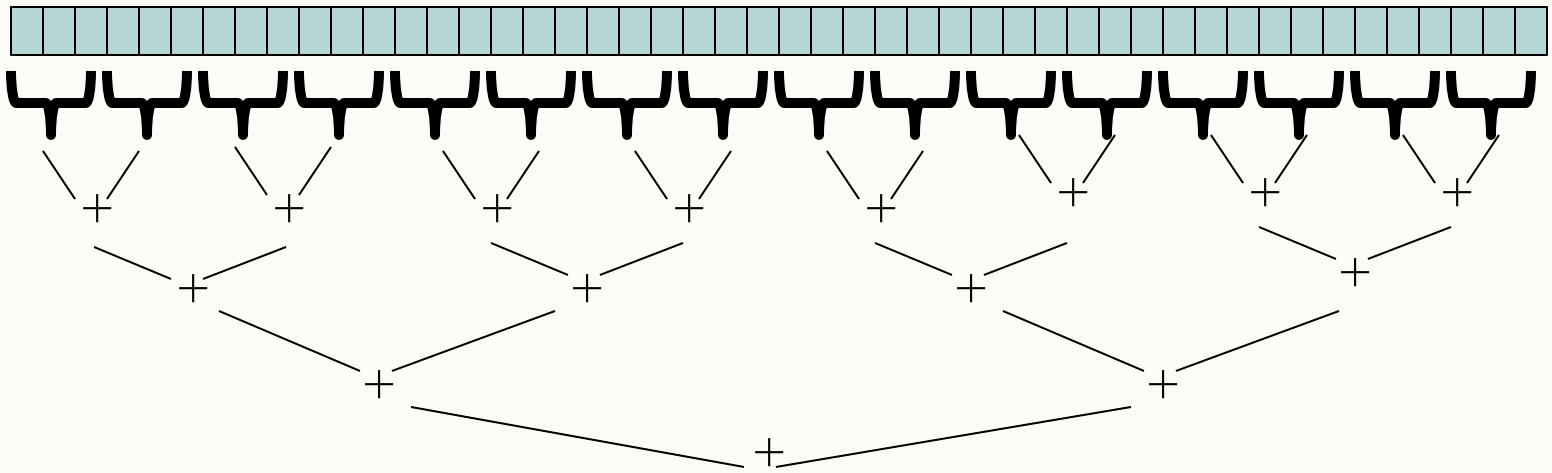
- It's Asymptotic Analysis Time!
  - How long does dividing up/recombining the work take with infinite number of processors? Um…?



Time $\in \Theta(\lg n)$ with an infinite number of processors.
Exponentially faster than our $\Theta(n)$ solution! Yay!

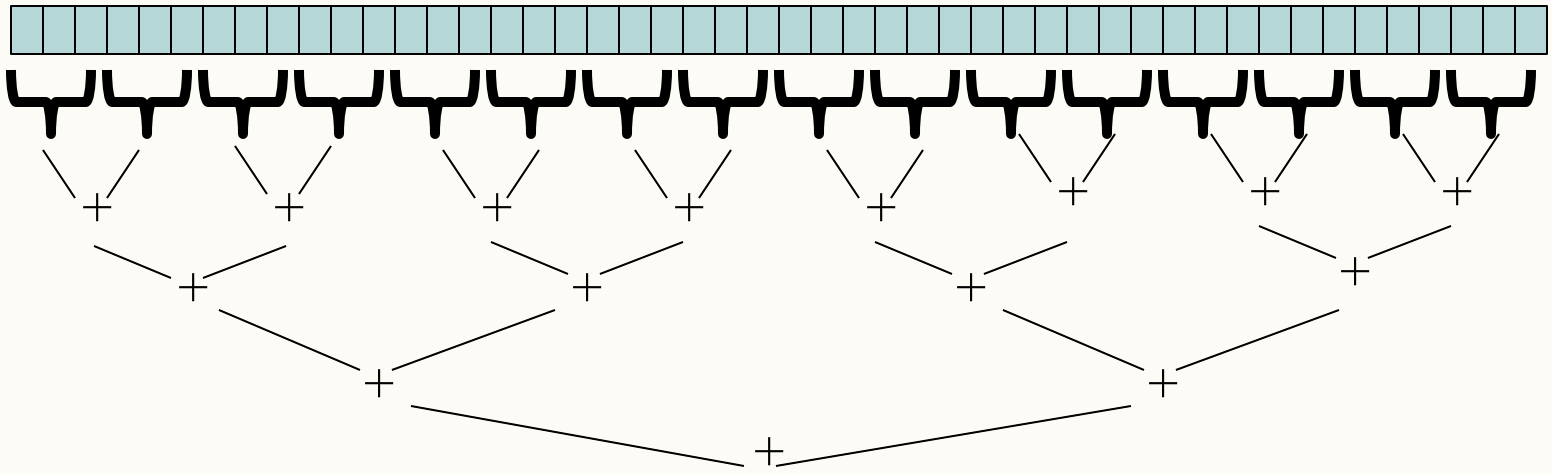
"Exponential speed-up" using Divide-and-Conquer

- Counting matches (lecture) and summing (reading) went from O(n) sequential to O(`log` n) parallel (assuming **lots** of processors!)
  - An exponential speed-up (or more like: the sequential version represents an exponential **slow-down**)



- Many other operations can also use this structure…

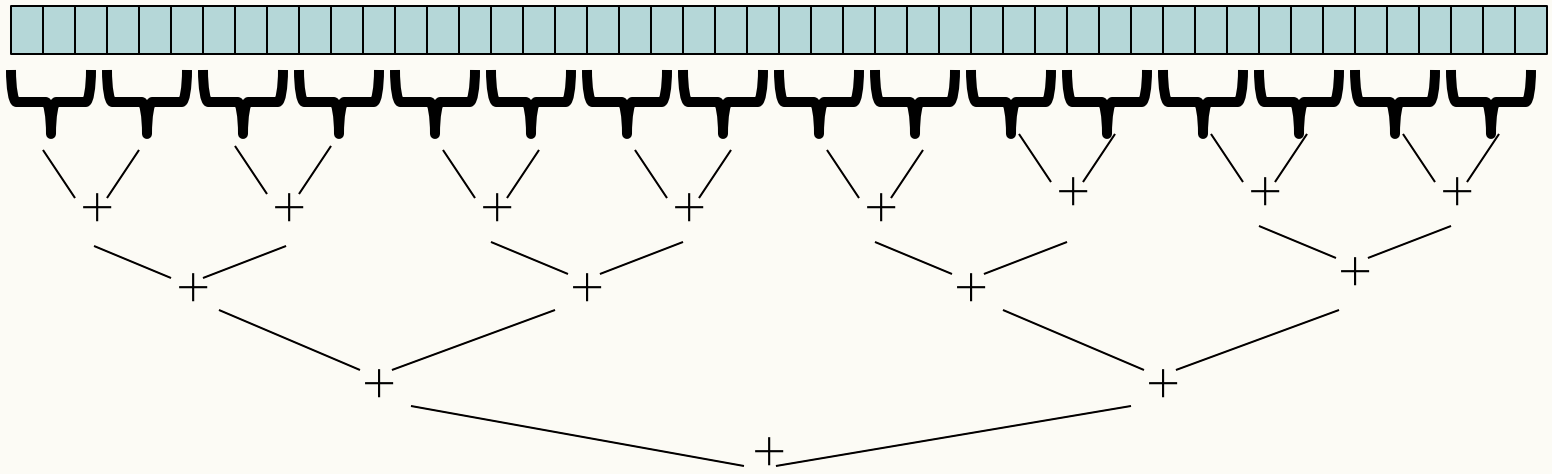
Other Operations?



What an example of something else we can put at the "+" marks?

- count elements that satisfy some property
- max or min
- concatenation
- Find the left-most array index that has an element that satisfies some property

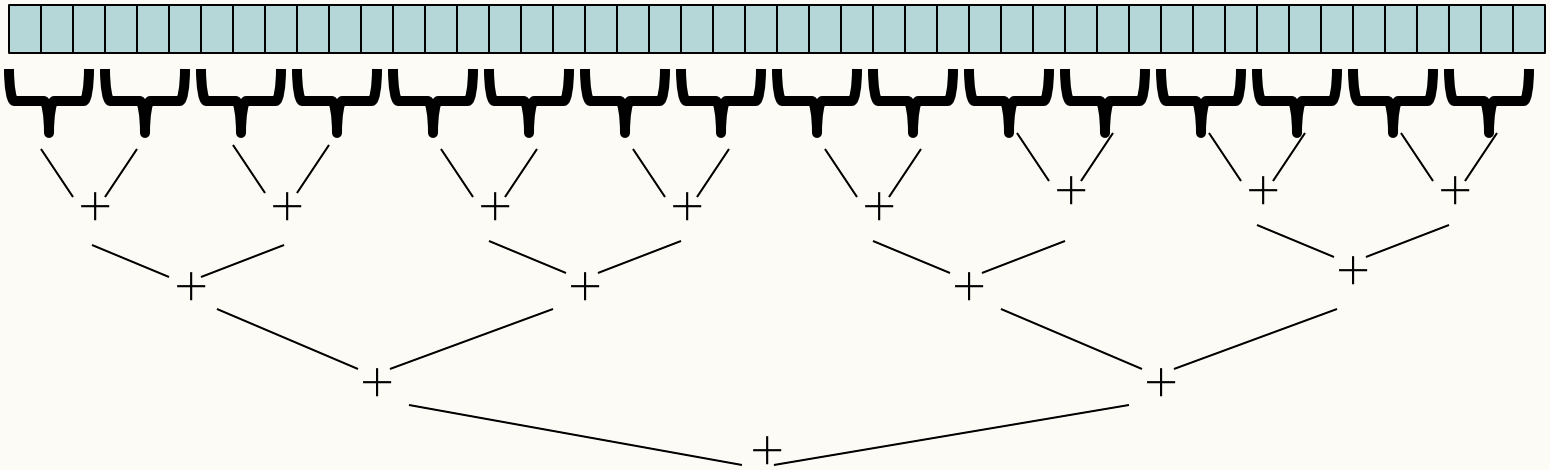
What else looks like this?



What's an example of something we cannot put there

- Subtraction: ((5-3)-2) <> (5-(3-2))
- Exponentiation: 2^{3^4} <> $(2^3)^4$
  - 2^{81} <> 2^{12}

What else looks like this?



Note: The "single" answer can be a list or other collection.

What are the basic requirements for the reduction operator?

The operator has be associative

CPSC 221 Administrative Notes

- Programming project #1 handin trouble
  - Brian has an office hour 3:30-4:40 DLC
  - There will be a 15% penalty, but if your files were stored on ugrad servers, we can remark them.

- Programming project #2 due
  - Apr Tue, 07 Apr @ 21.00
  - TA office hours during the long weekend
  - Friday Lynsey: 12:00 – 2:00
  - Saturday Kyle 11:00 – 12:00
  - Sunday Kyle 11:00 – 12:00

CPSC 221 Administrative Notes

- Lab 10 Parallelism Mar 26 – Apr 2
  - Some changes to the code since Friday
  - Marking Apr 7 – Apr 10 (Also doing Concept Inventory). Doing the Concept inventory is worth 1 lab point (0.33% course grade).

- PeerWise Call #5 due today (5pm)
  - The deadline for contributing to your "Answer Score" and "Reputation score" is Monday April 20.

So… Where Were We?

- We talked about
  - Parallelism
  - Concurrency
- Problem: Count Matches of a Target
  - Race conditions
  - Out of scope variables
- Fork/Join Parallelism
- Divide-and-Conquer Parallelism

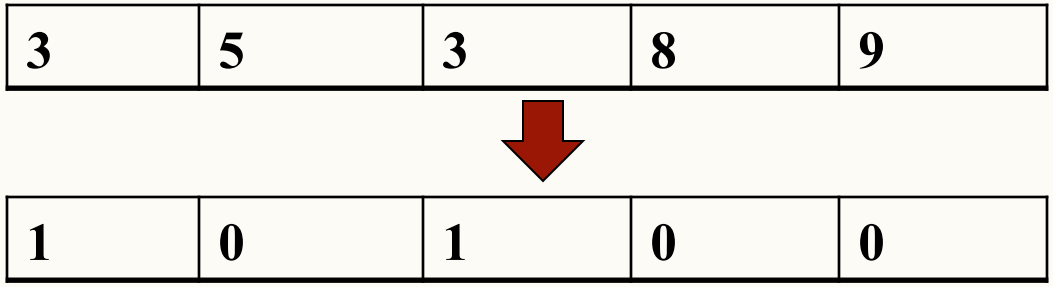
Reduction

- 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

Even easier: Maps (Data Parallelism)

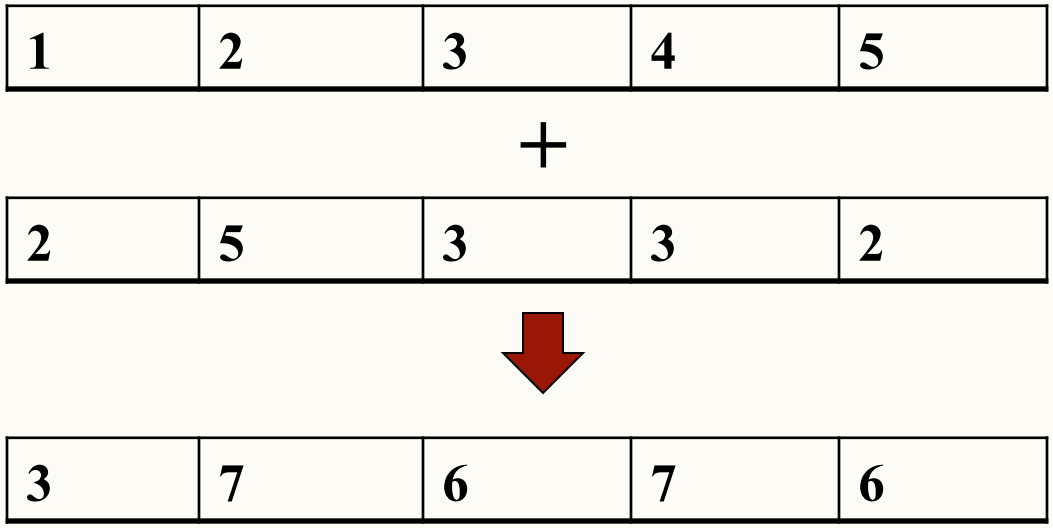
- A map operates on each element of a collection independently to create a new collection of the same size
  - No combining results
  - For arrays, this is so trivial some hardware has direct support
- **One we already did**: counting matches becomes mapping "number → 1 if it matches, else 0" and then reducing with +

```
void equals_map(int result[], int array[], int len, int target) {
  FORALL(i=0; i < len; i++) {
    result[i] = (array[i] == target) ? 1 : 0;
  }
}
```



Another Map Example: Vector Addition

```
void vector_add(int result[], int arr1[], int arr2[], int len) {
  FORALL(i=0; i < len; i++) {
    result[i] = arr1[i] + arr2[i];
  }
}
```



Maps in OpenMP (w/explicit Divide & Conquer)

```
void vector_add(int result[], int arr1[], int arr2[],
int lo, int hi)
{
  const int SEQUENTIAL_CUTOFF = 1000;
  if (hi – lo <= SEQUENTIAL_CUTOFF) {
    for (int i = lo; i < hi; i++)
      result[i] = arr1[i] + arr2[i];
    return;
  }

#pragma omp task untied
  {
    vector_add(result, arr1, arr2, lo, lo + (hi–lo)/2);
  }

  vector_add(result, arr1, arr2, lo + (hi–lo)/2, hi);
#pragma omp taskwait}
```

Maps and reductions

- These are by far the two most important and common patterns.

- Learn to recognize when an algorithm can be written in terms of maps and reductions! They make parallel programming simple…

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

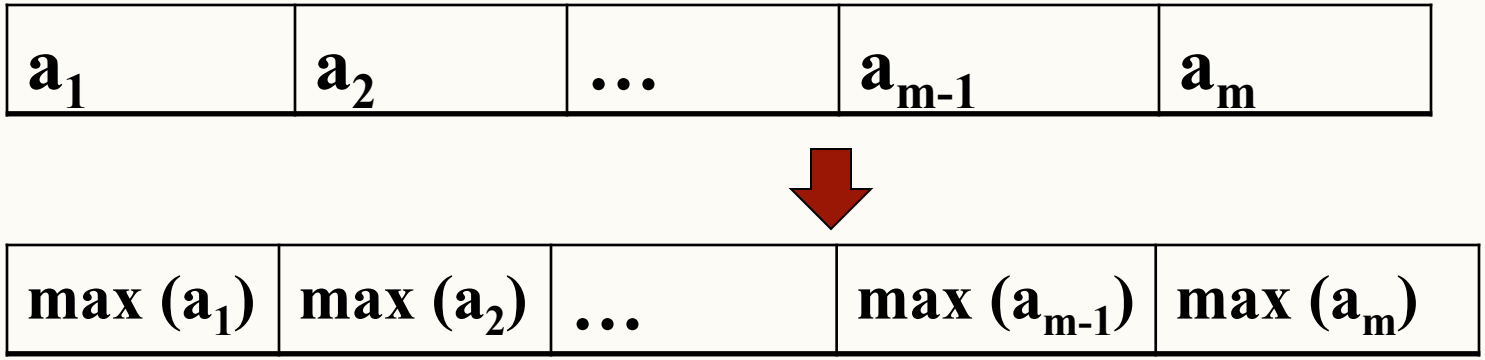
- system distributes the data and manages fault tolerance
- your code just operates on one element (map) or combines two elements (reduce)
- old functional programming idea → big data/distributed computing

What is specifically possible in a Hadoop "map/reduce" is more general than the examples we've so far seen.

Exercise: find largest

Given an array of positive integers, find the largest number.

How is this a map and/or reduce?

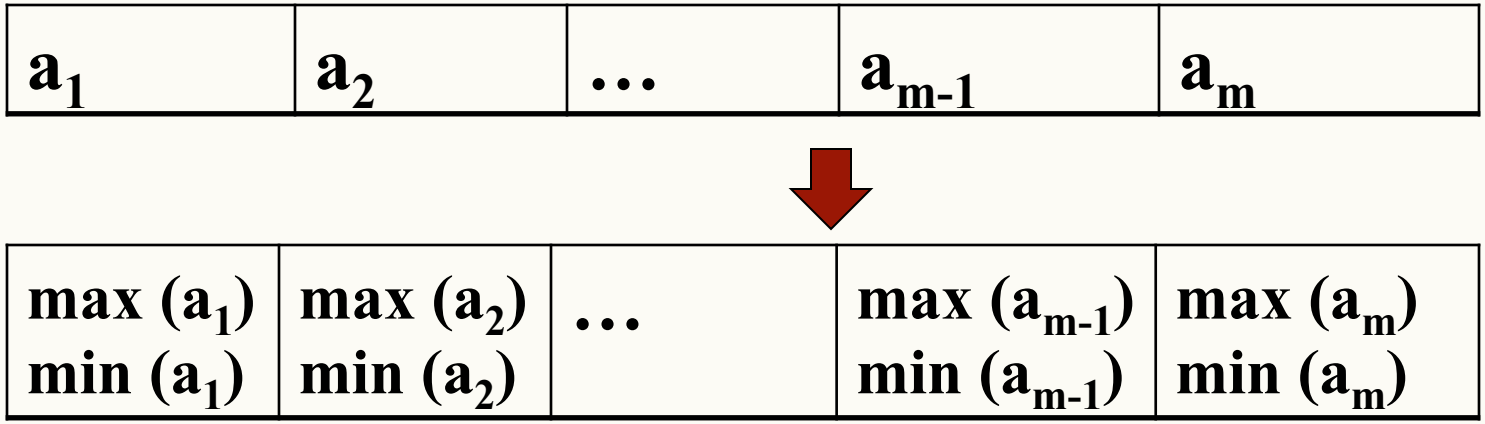


Reduce: max

Exercise: find largest **AND** smallest

Given an array of positive integers, find the largest and the smallest number.

How is this a map and/or reduce?



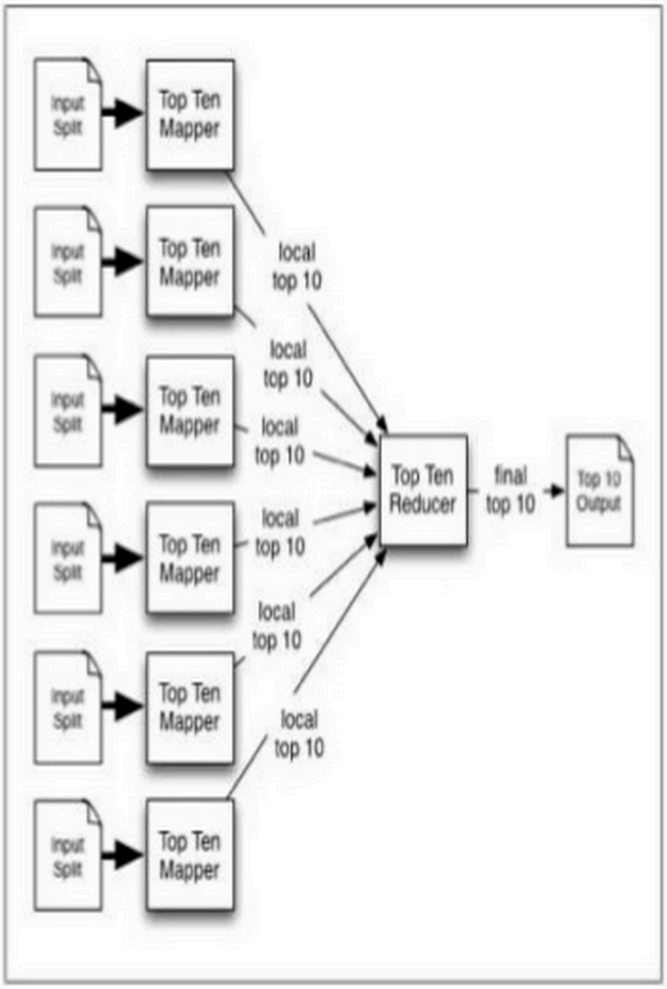
Reduce: max, and min

Exercise: find the K largest numbers

Given an array of positive integers. return the k largest in the list.

Map: Same as max

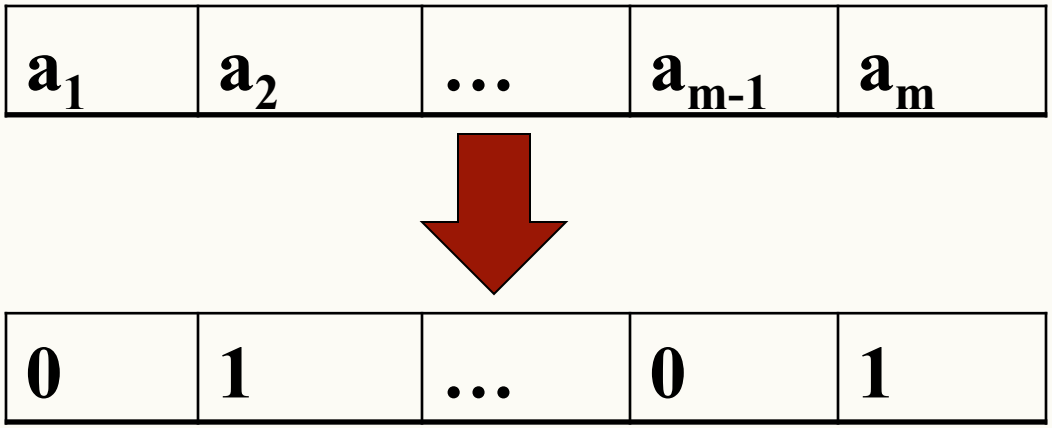
Reduce: Find k max values



Exercise: count prime numbers

Given an array of positive integers, count the number of prime numbers.

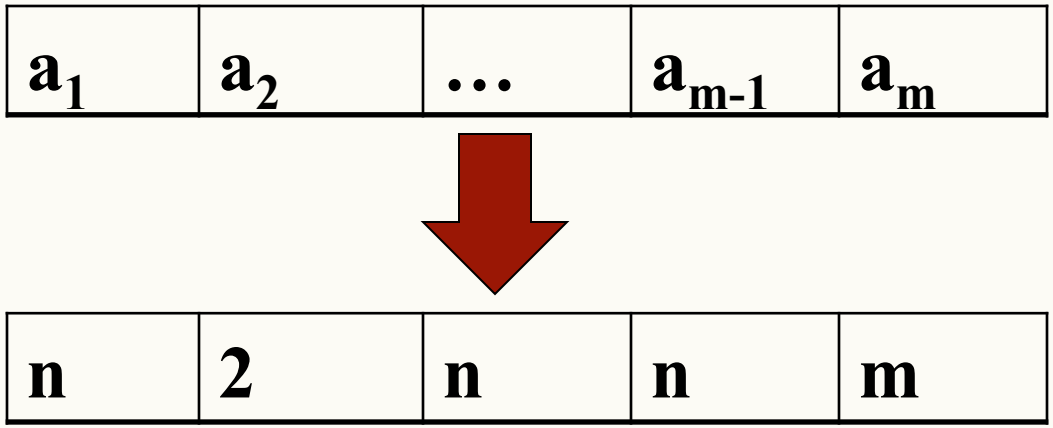
Map: call is-prime on array and produce array2. for each element write 1 if it is prime, and 0 otherwise



Reduce: + on array2

Exercise: find first substring match

Given an extremely long string (DNA sequence?) find the index of the first occurrence of a short substring



Reduce: Find min

Outline

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 C++11 and OpenMP specifics
  - More pragmatics (e.g., installation) in separate notes

Now:

- More examples of simple parallel programs
- Other data structures that support parallelism (or not)
- Asymptotic analysis for fork-join parallelism
- Amdahl's Law

Trees

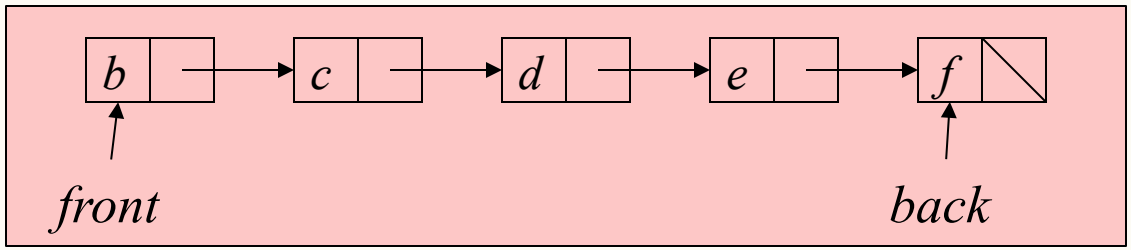
- Maps and reductions work just fine on balanced trees
  - Divide-and-conquer each child rather than array subranges
  - Correct for unbalanced trees, but won't get much speed-up

- Certain problems will not run faster in parallel
  - Searching for an element
- Some problems run faster
  - Summing the elements of a balanced binary tree

- How to do the sequential cut-off?
  - Store number-of-descendants at each node (easy to maintain)
  - Or could approximate it with, e.g., AVL-tree height

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. $O(n)$

Outline

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 C++11 and OpenMP specifics
  - More pragmatics (e.g., installation) in separate notes

Now:

- More examples of simple parallel programs
- Other data structures that support parallelism (or not)
- Asymptotic analysis for fork-join parallelism
- Amdahl's Law

Analyzing Parallel Algorithms

- Like all algorithms, parallel algorithms should be:
  - Correct
  - 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

CPSC 221 Administrative Notes

- Marking lab 10 :Apr 7 – Apr 10
- Written Assignment #2 is marked

Statistics

Count	228
Minimum Value	9.00
Maximum Value	40.00
Range	31.00
Average	33.85
Median	36.00
Standard Deviation	6.00
Variance	35.96

CPSC 221 Administrative Notes

- Marking lab 10 :Apr 7 – Apr 10
- Written Assignment #2 is marked
- Programming project is due tonight!
- Here is what I've been doing on PeerWise
  - Final call for Piazza question will be out tonight

Evaluations

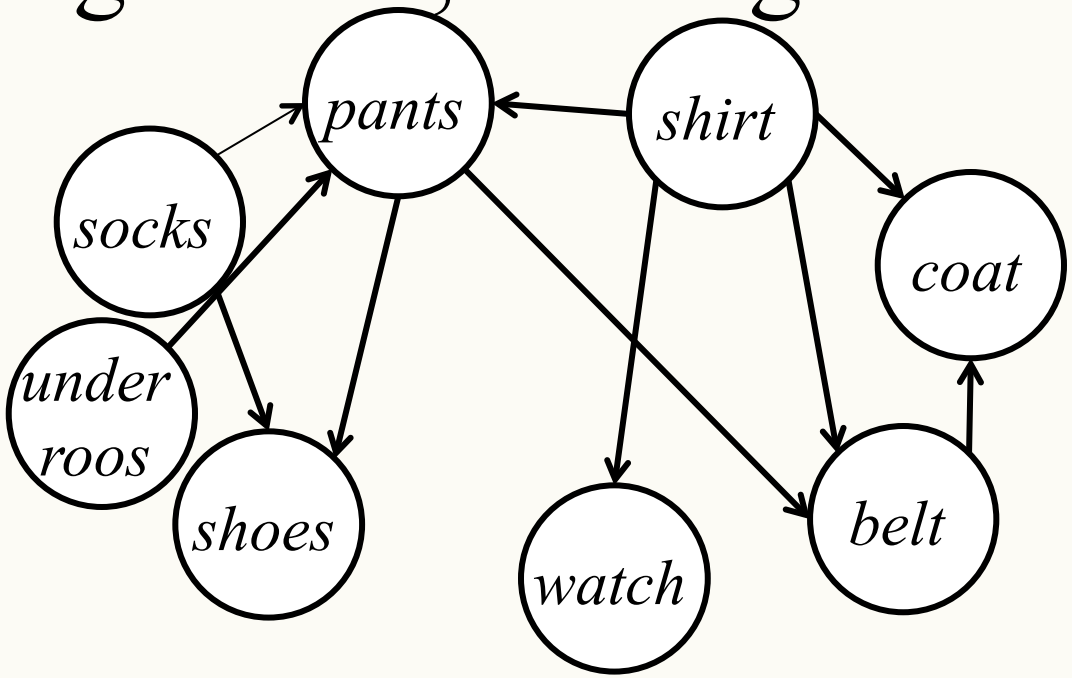
- TA Evaluation
  - Please only evaluate TAs that you know and worked with in some capacity.

- Instructor Evaluation
  - We'll spend some time on Thursday on this.

So… Where Were We?

- We've talked about
  - Parallelism and Concurrency
  - Fork/Join Parallelism
  - Divide-and-Conquer Parallelism
  - Map & Reduce
  - Using parallelism in other data structures such as Trees and Linked list
  - And Finally we talked about me getting dressed!

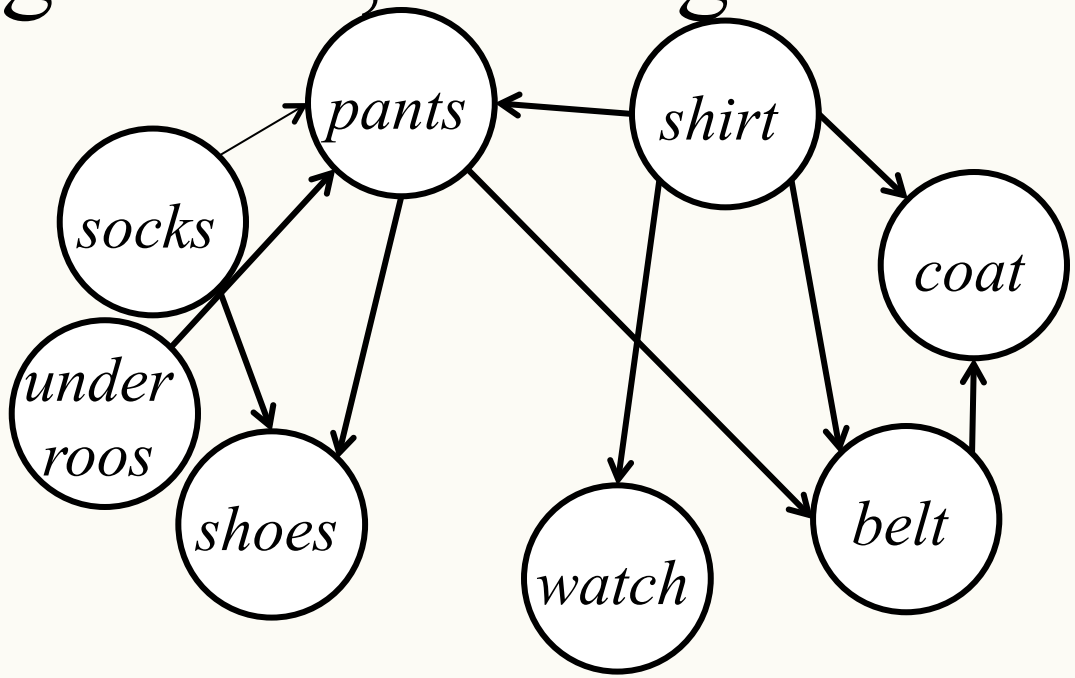
Digression, Getting Dressed



- Here's a graph representation for parallelism.
  - Nodes: (small) tasks that are potentially executable in parallel
  - Edges: dependencies (the target of the arrow depends on its source)

(Note: costs are on nodes, not edges.)

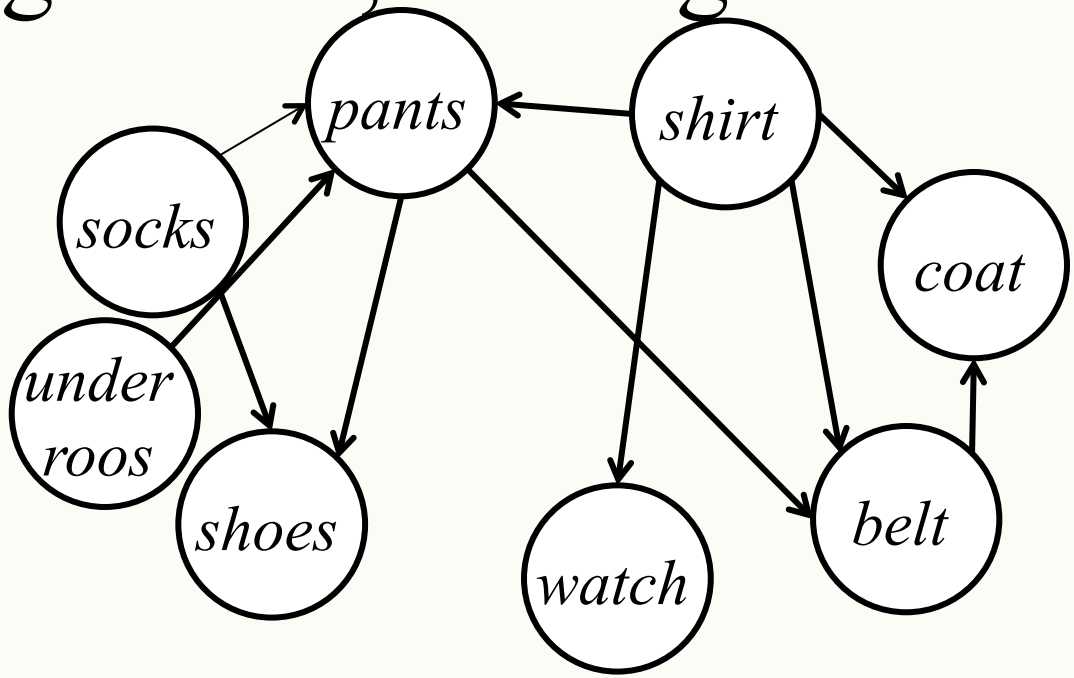
Digression, Getting Dressed (1)



- Assume it takes me 5 seconds to put on each item, and I cannot put on more than one item at a time. How long does it take me to get dressed?

A: 20 *B: 25* *C:30* *D:35* *E :40*

Digression, Getting Dressed (1)

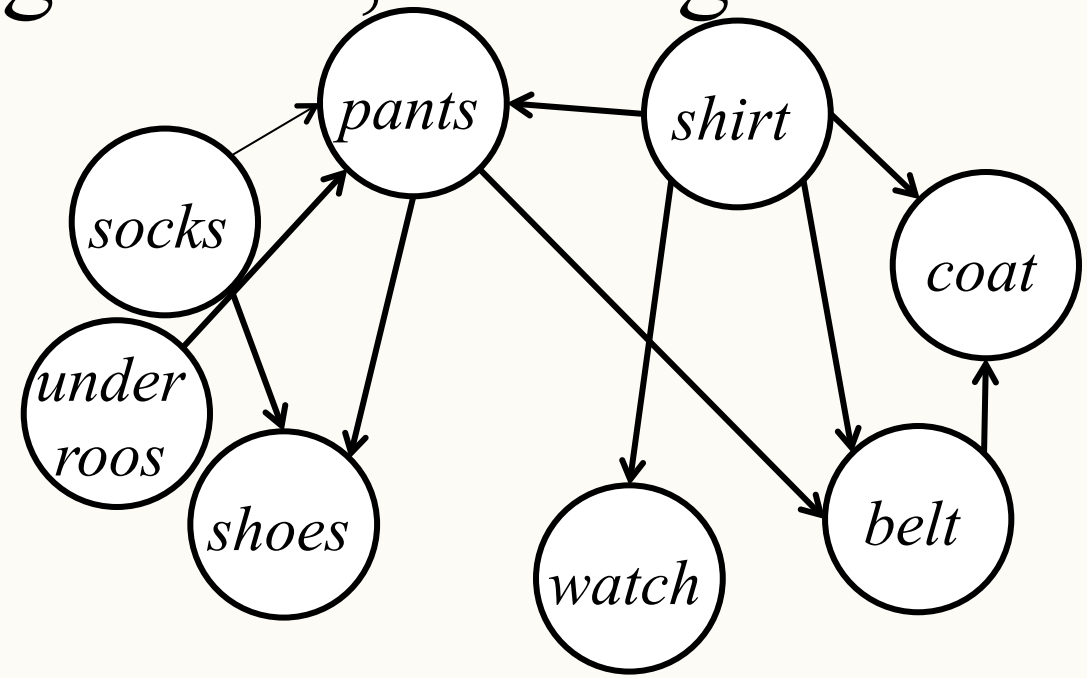


under roos | shirt | socks | pants | watch | belt | shoes | coat

40 Seconds

(Note: costs are on nodes, not edges.)

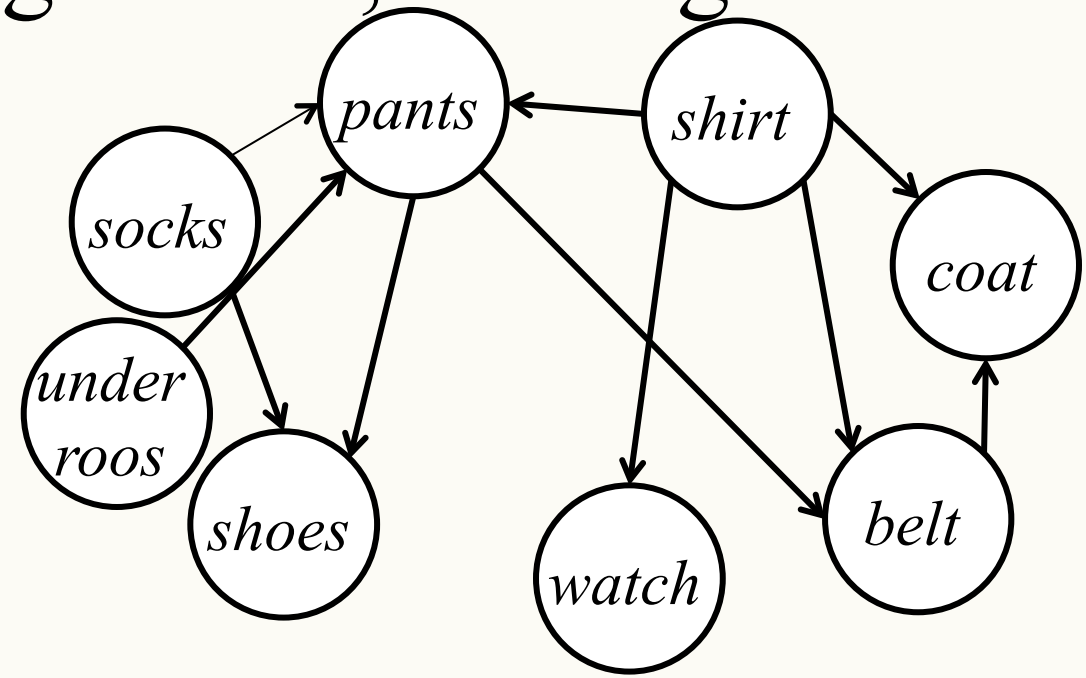
Digression, Getting Dressed (∞)



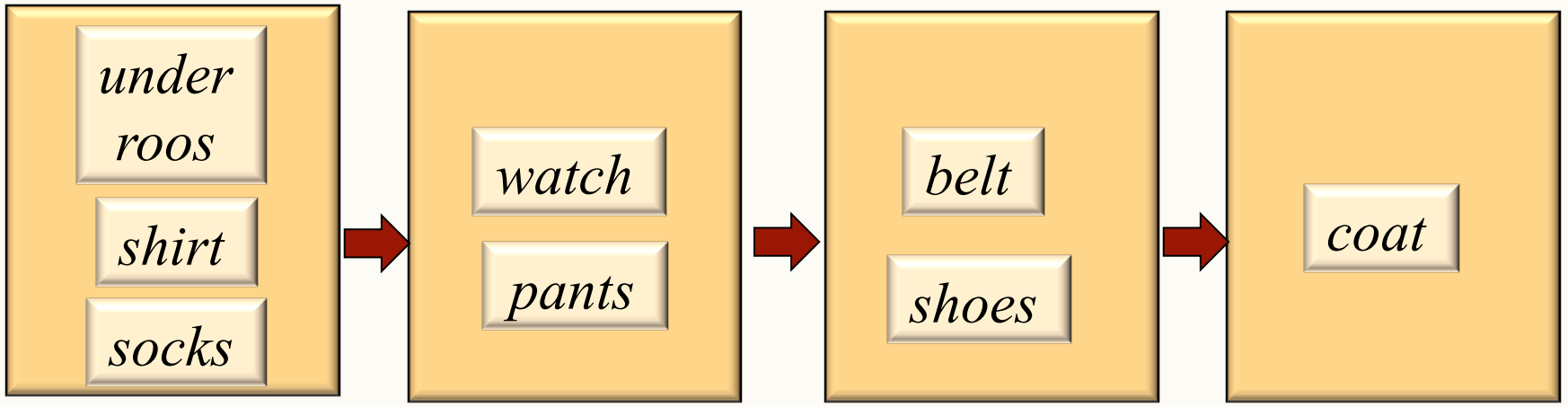
- Assume it takes my robotic wardrobe 5 seconds to put me into each item, and it can put on up to 20 items at a time. How long does it take me to get dressed?

A: 20 *B: 25* *C:30* *D:35* *E :40*

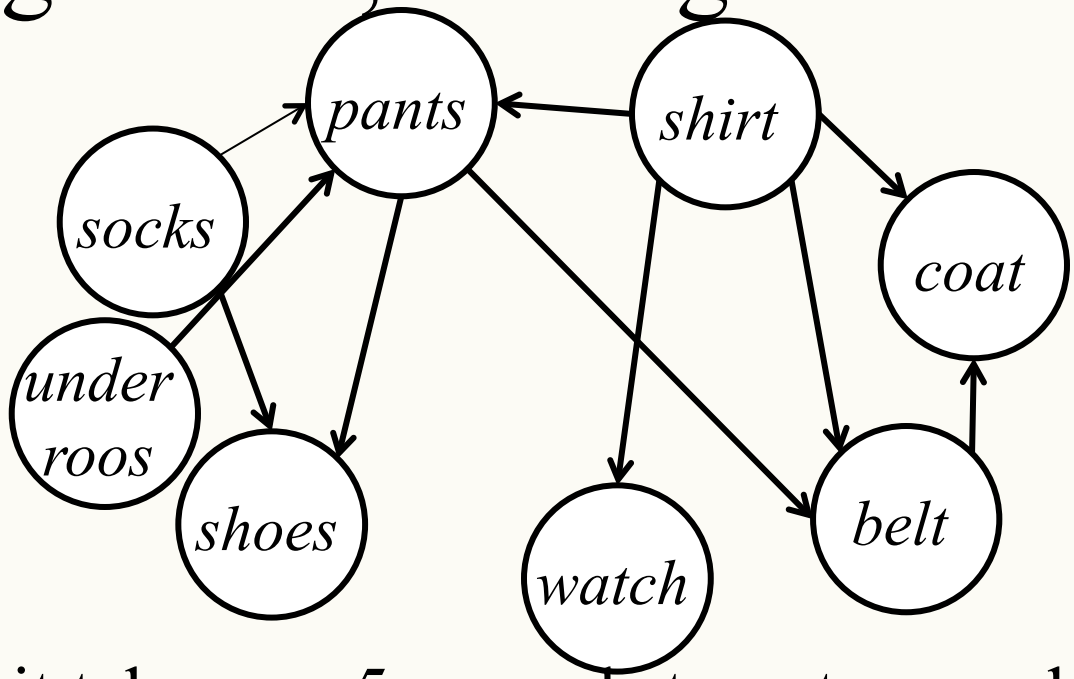
Digression, Getting Dressed (∞)



20 Seconds



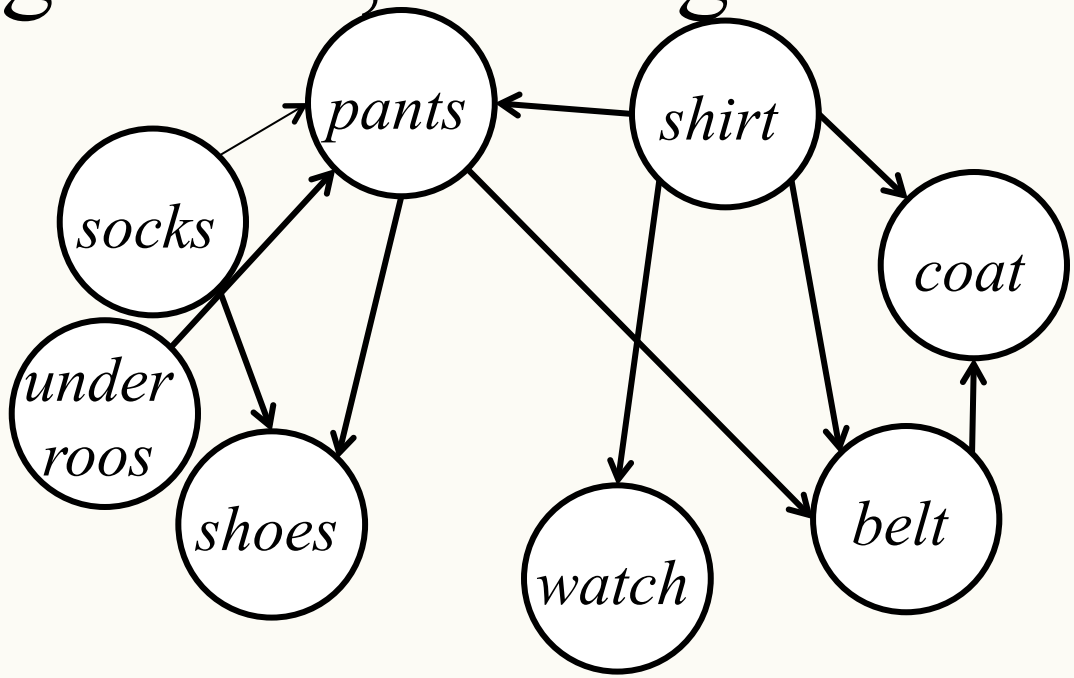
Digression, Getting Dressed (2)



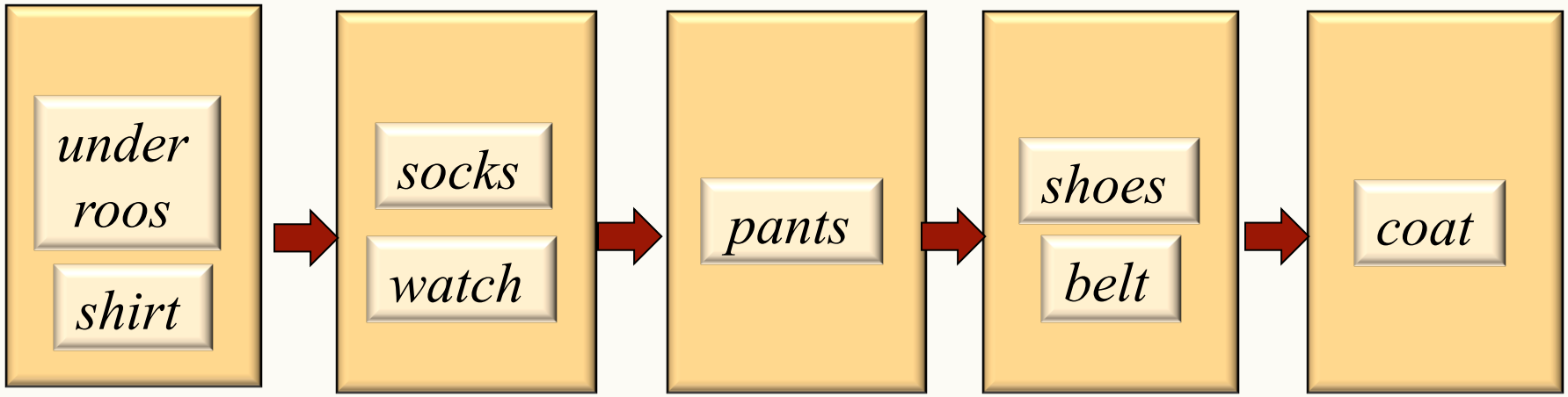
- Assume it takes me 5 seconds to put on each item, and I can use my two hands to put on 2 items at a time. (I am exceedingly ambidextrous.) How long does it take me to get dressed?

A: 20 *B: 25* *C:30* *D:35* *E :40*

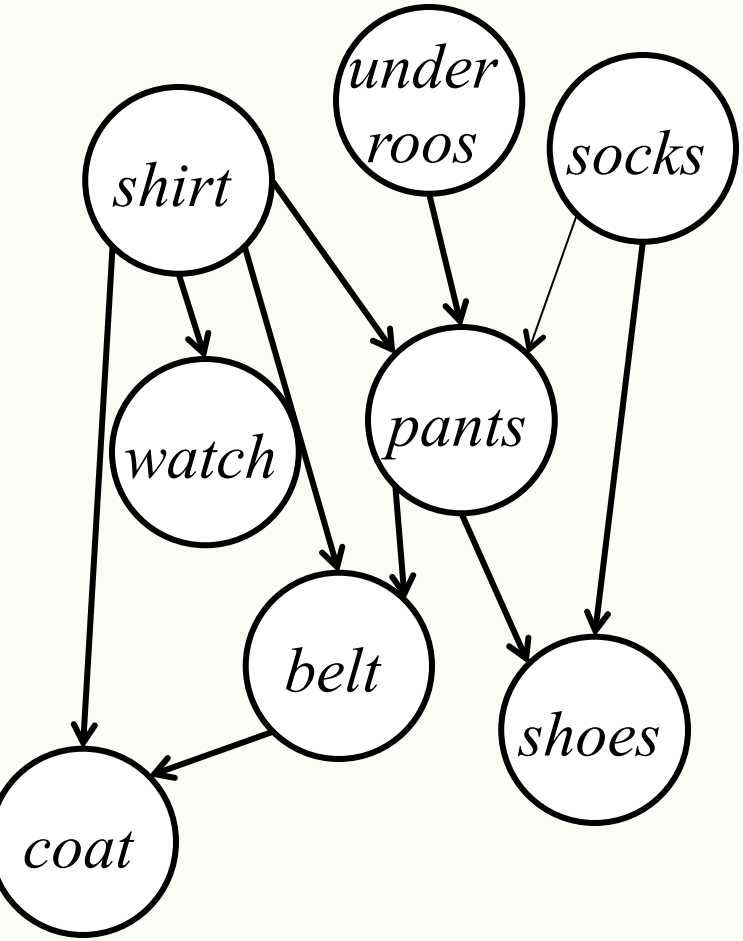
Digression, Getting Dressed (2)



25 Seconds

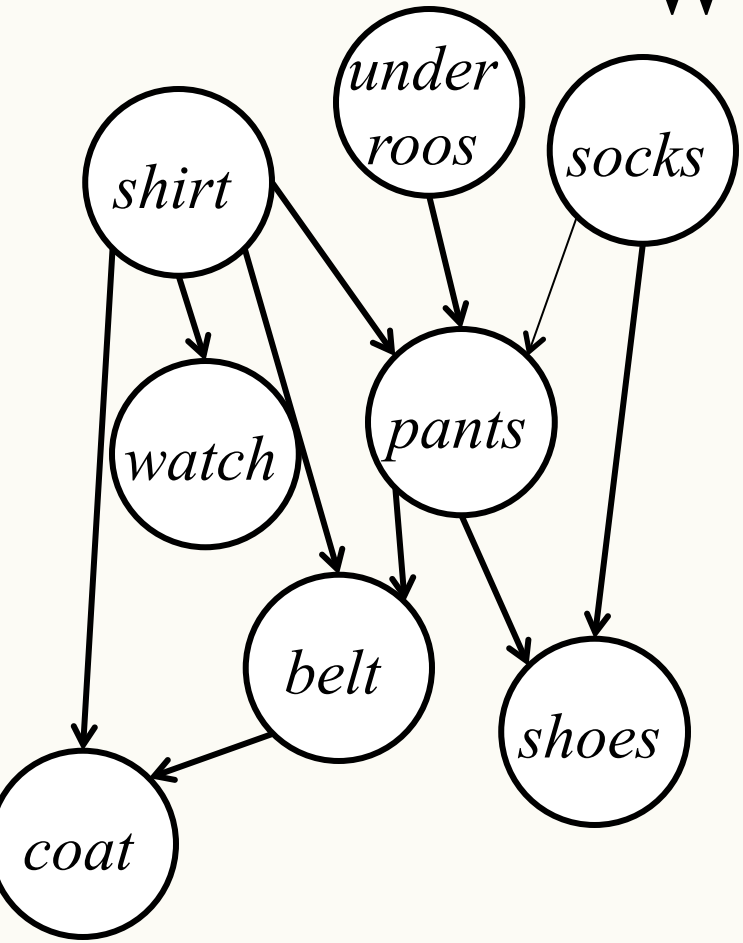


Un-Digression, Getting Dressed:



- Nodes are pieces of work the program performs. Each node will be a constant, i.e., O(1), amount of work that is performed sequentially.
- Edges represent that the source node must complete before the target node begins. That is, there is a computational dependency along the edge.
- The graph needs to be a directed acyclic graph (DAG)
-

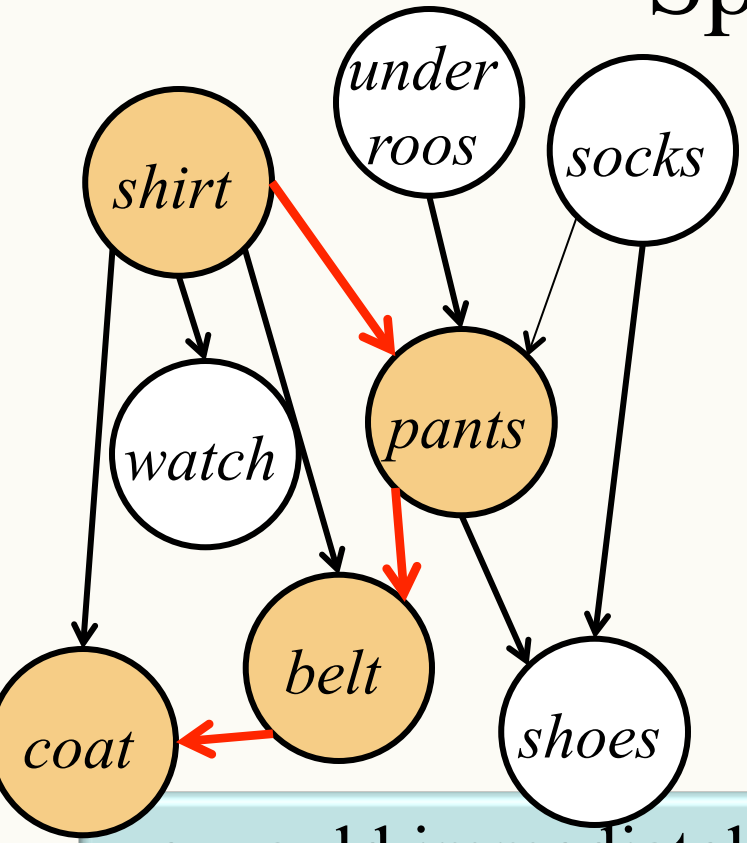
Un-Digression, Getting Dressed: "Work", AKA T_1



- T_1 is called the work. By definition, this is how long it takes to run on one processor.
- What mattered when I could put only one item on at a time? How do we count it?

T_1 is asymptotically just the number of nodes in the dag.

Un-Digression, Getting Dressed: "Span", AKA T_∞



- T_∞ is called the span, though other common terms are the critical path length or computational depth.
- What mattered when I could put on an infinite number of items on at a time? How do we count it?

we would immediately start every node as soon as its predecessors in the graph had finished. So it would be the length of the longest path in the DAG.

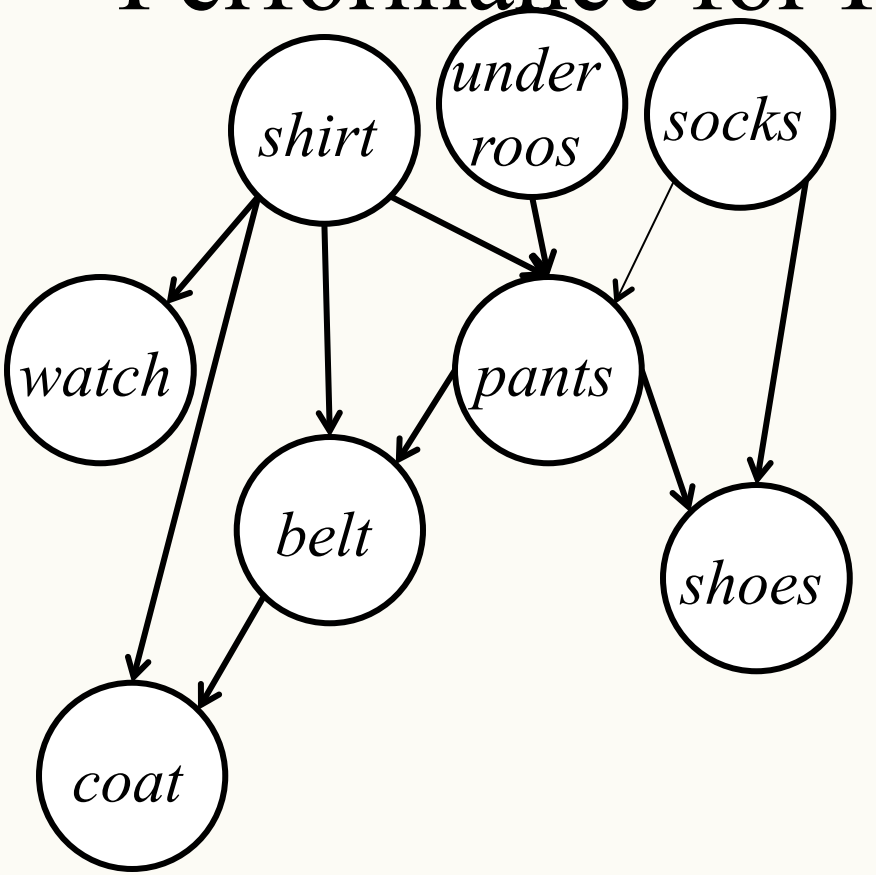
Work and Span

Two key measures of run-time:

- Work: How long it would take 1 processor = T_1
  - Just "sequentialize" the recursive forking

- Span: How long it would take infinity processors = T_∞
  - Example: $O(\texttt{log}\ n)$ for summing an array
    - Notice having > $n/2$ processors is no additional help

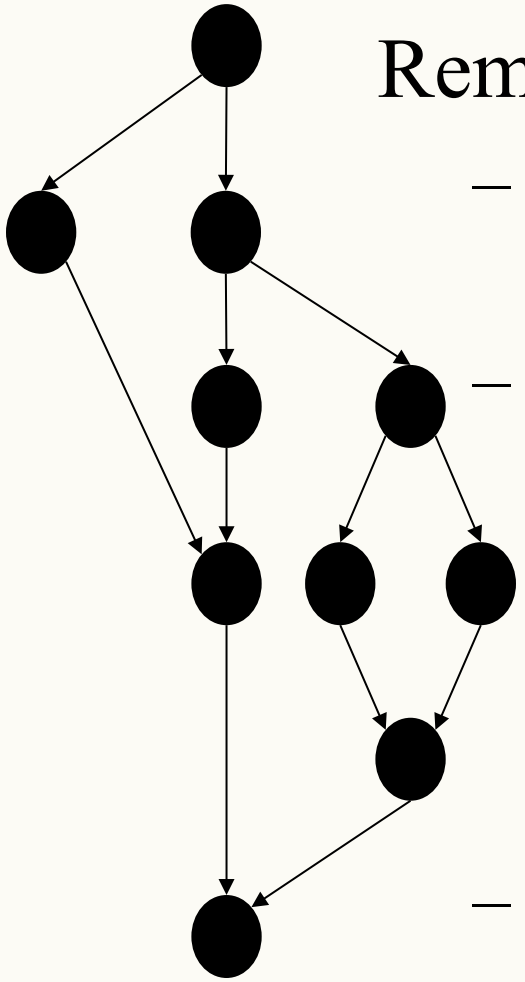
Un-Digression, Getting Dressed: Performance for P processors, AKA T_P



- T_P is the time a program takes to run if there are P processors available during its execution
- What mattered when I could put on 2 items on at a time? Was it as easy as work or span to calculate?

T_1 and T_∞ are easy, but we want to understand T_P in terms of P
We'll come back to this soon!

Analyzing Code, Not Clothes

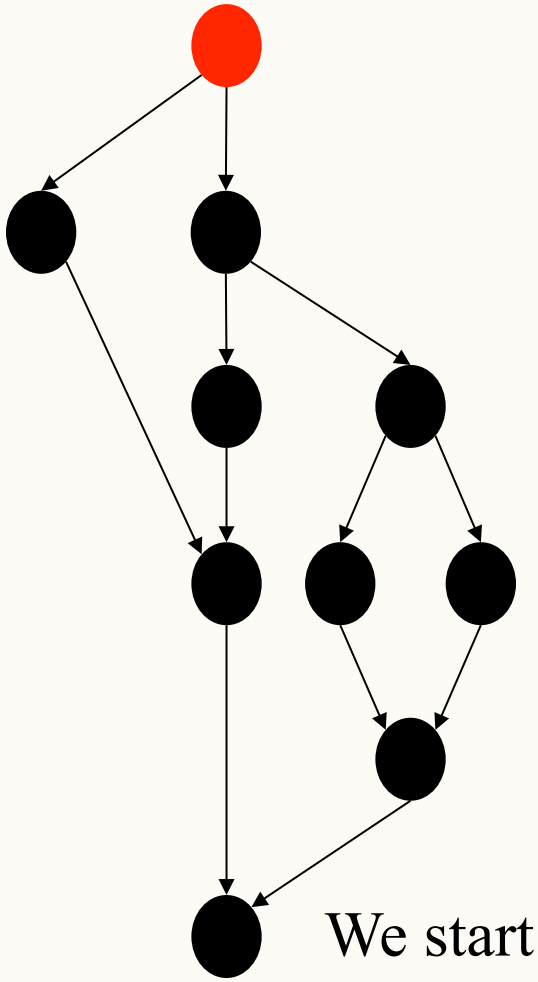


Reminder, in our DAG representation:

- Each node: one piece of *constant-sized* work
- Each edge: source must finish before destination starts

- What is T_∞ in this graph?

Where the DAG Comes From



pseudocode

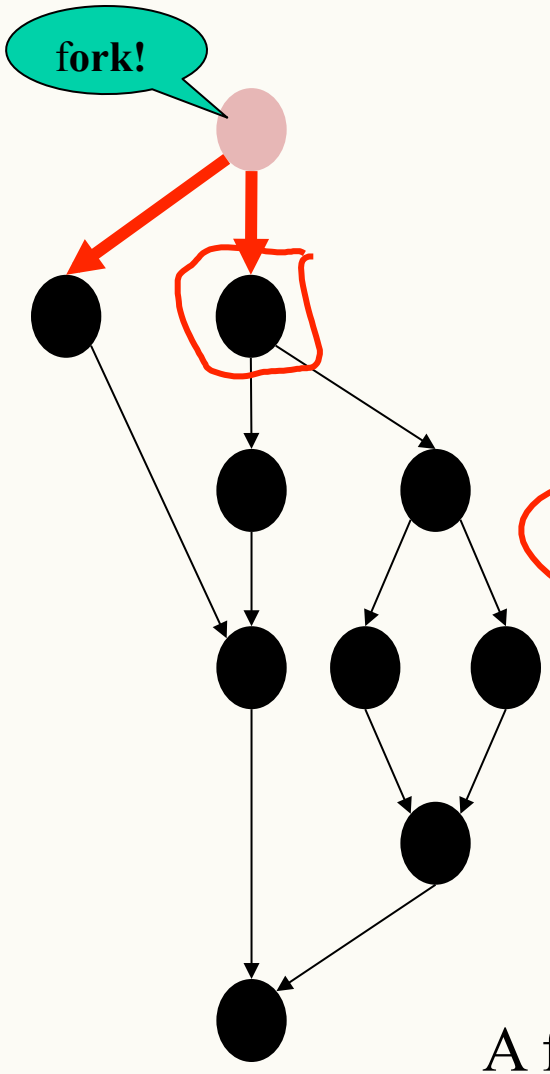
```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b


task1:
  O(1) work


task2:
  c = fork task1
  O(1) work
  join c
```

C++11

```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

We start with just one thread.
(Using C++11 not OpenMP syntax to make things cleaner.)

Where the DAG Comes From

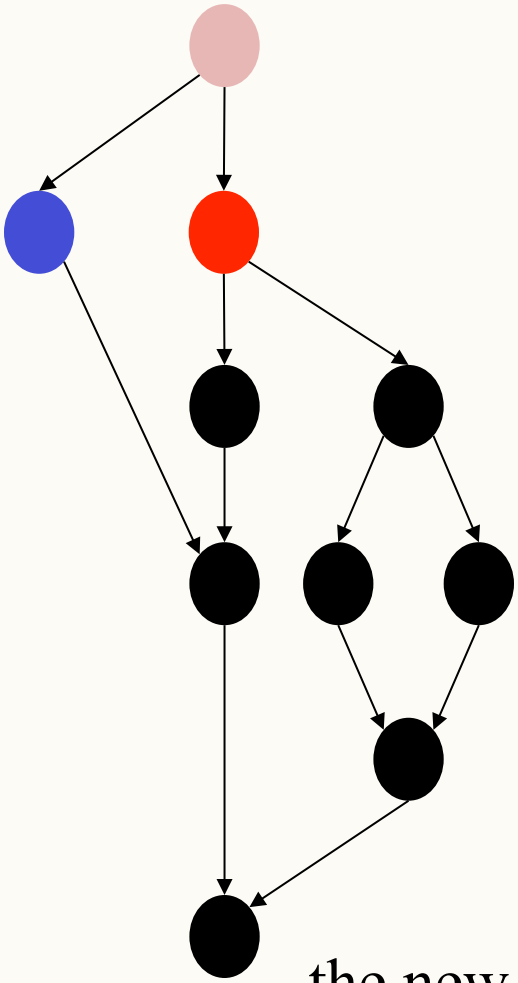



```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b


task1:
  O(1) work


task2:
  c = fork task1
  O(1) work
  join c
```

```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

A fork ends a node and generates two new ones…

Where the DAG Comes From

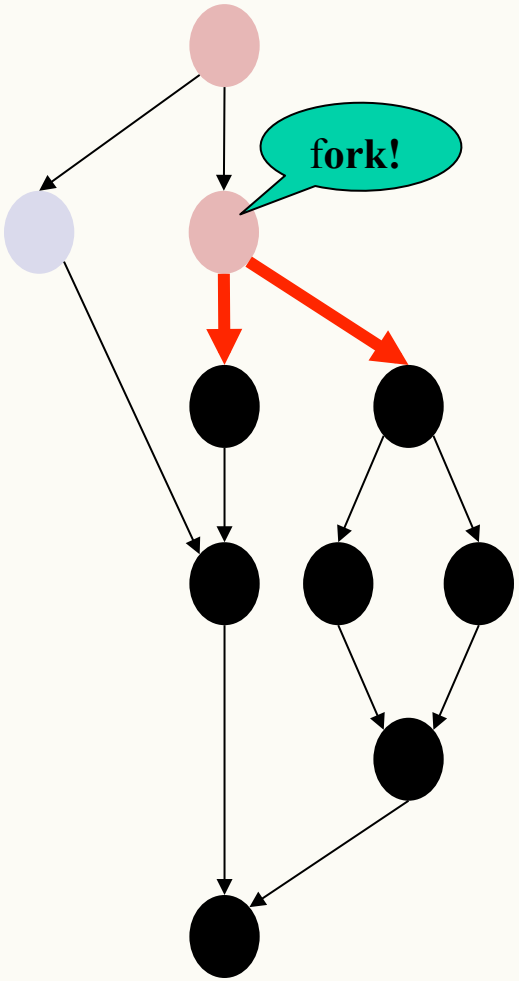


```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b


task1:
  O(1) work


task2:
  c = fork task1
  O(1) work
  join c
```

```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

…the new task/thread and the continuation of the current one.

Where the DAG Comes From

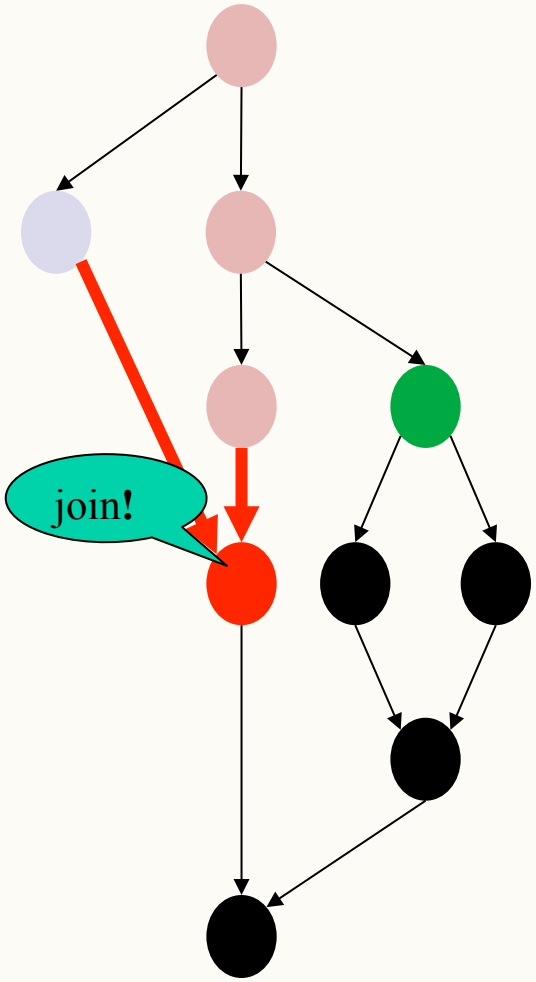



```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b


task1:
  O(1) work


task2:
  c = fork task1
  O(1) work
  join c
```

```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

Again, we fork off a task/thread.
Meanwhile, the left (blue) task finished.

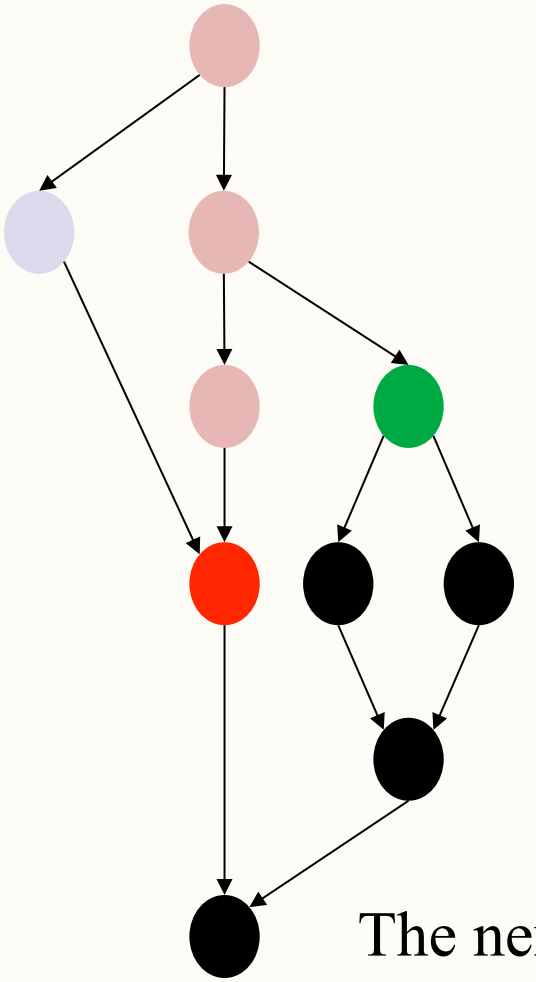
Where the DAG Comes From




```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b


task1:
  O(1) work


task2:
  c = fork task1
  O(1) work
  join c
```

```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

Where the DAG Comes From

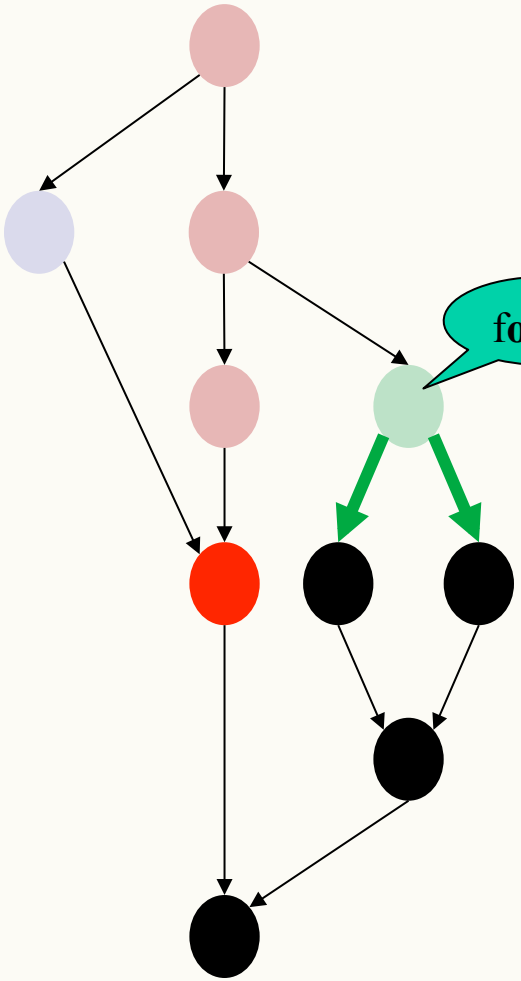


```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b


task1:
  O(1) work


task2:
  c = fork task1
  O(1) work
  join c
```

```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

The next join isn't ready to go yet. The task/thread it's joining isn't finished. So, it waits and so do we.

Where the DAG Comes From



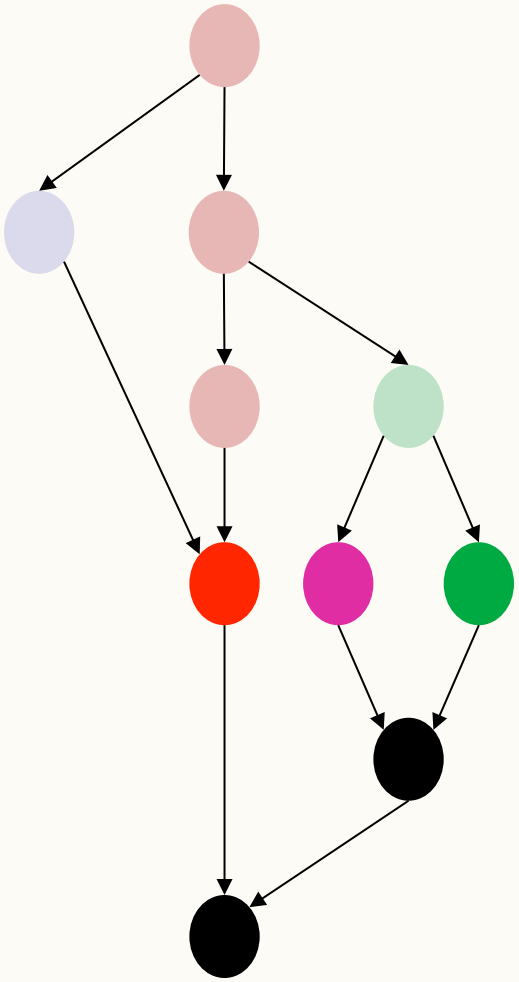

```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b


task1:
  O(1) work


task2:
  c = fork task1
  O(1) work
  join c
```

```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

Meanwhile, task2 also forks a task1.

(The DAG describes dynamic execution. We can run the same code many times!)

Where the DAG Comes From

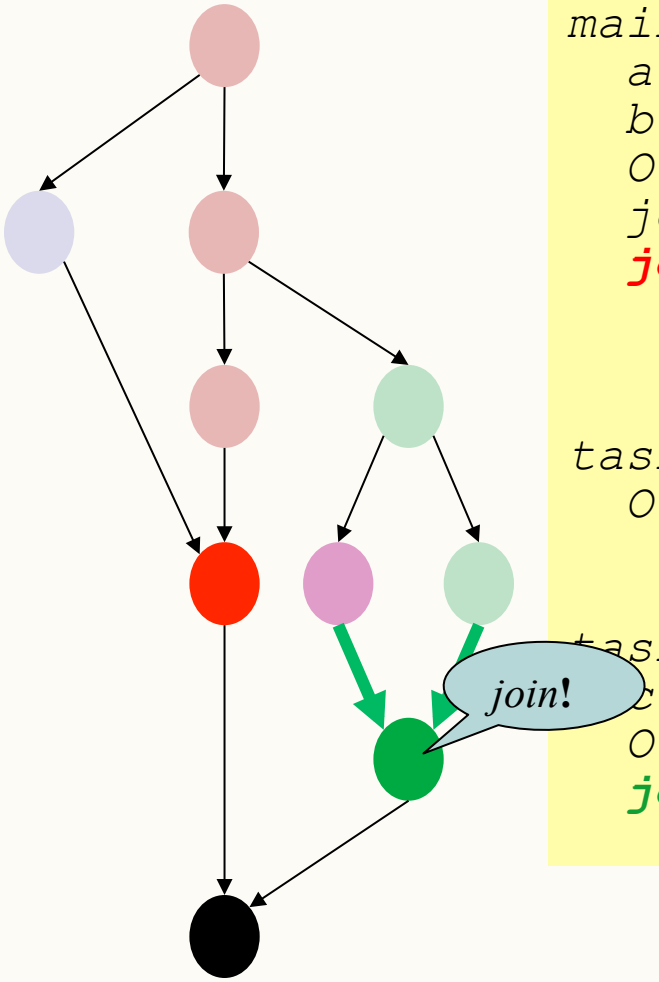


```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b


task1:
  O(1) work


task2:
  c = fork task1
  O(1) work
  join c
```

```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

task1 and task2 both chugging along.

Where the DAG Comes From

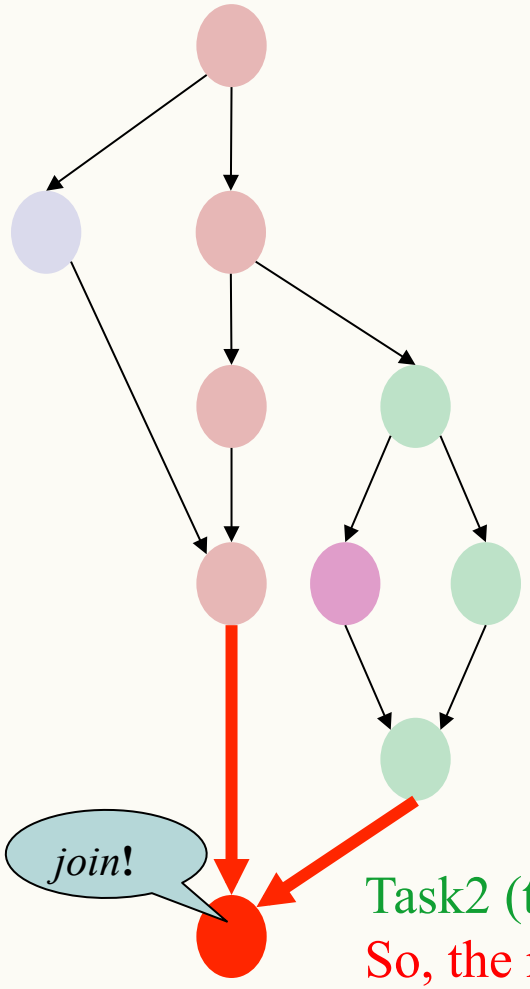



```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b


task1:
  O(1) work


task2:
  c = fork task1
  O(1) work
  join c
```

```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

task2 joins task1.

Where the DAG Comes From

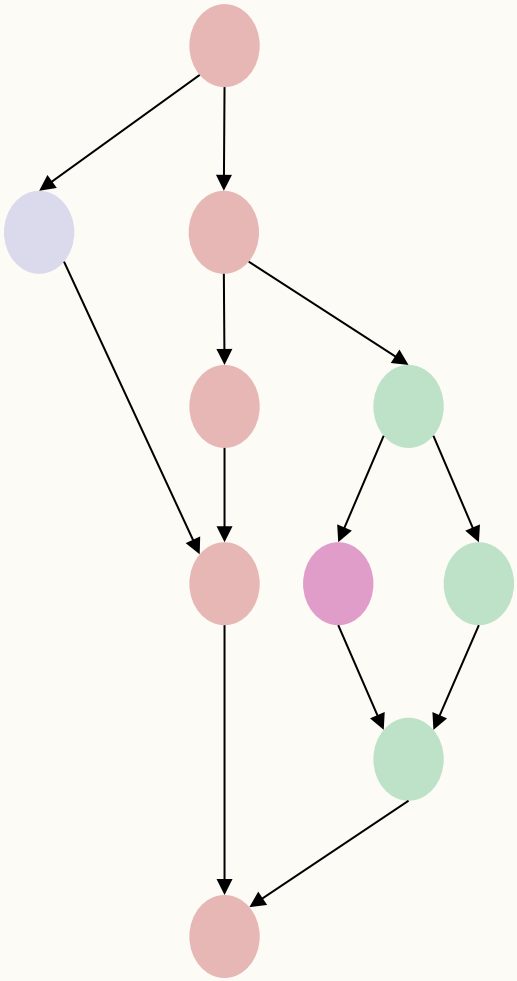



```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b


task1:
  O(1) work


task2:
  c = fork task1
  O(1) work
  join c
```

```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

Task2 (the right, green task) is finally done.
So, the main task joins with it.
(Arrow from the last node of the joining task and of the joined one.)

Where the DAG Comes From

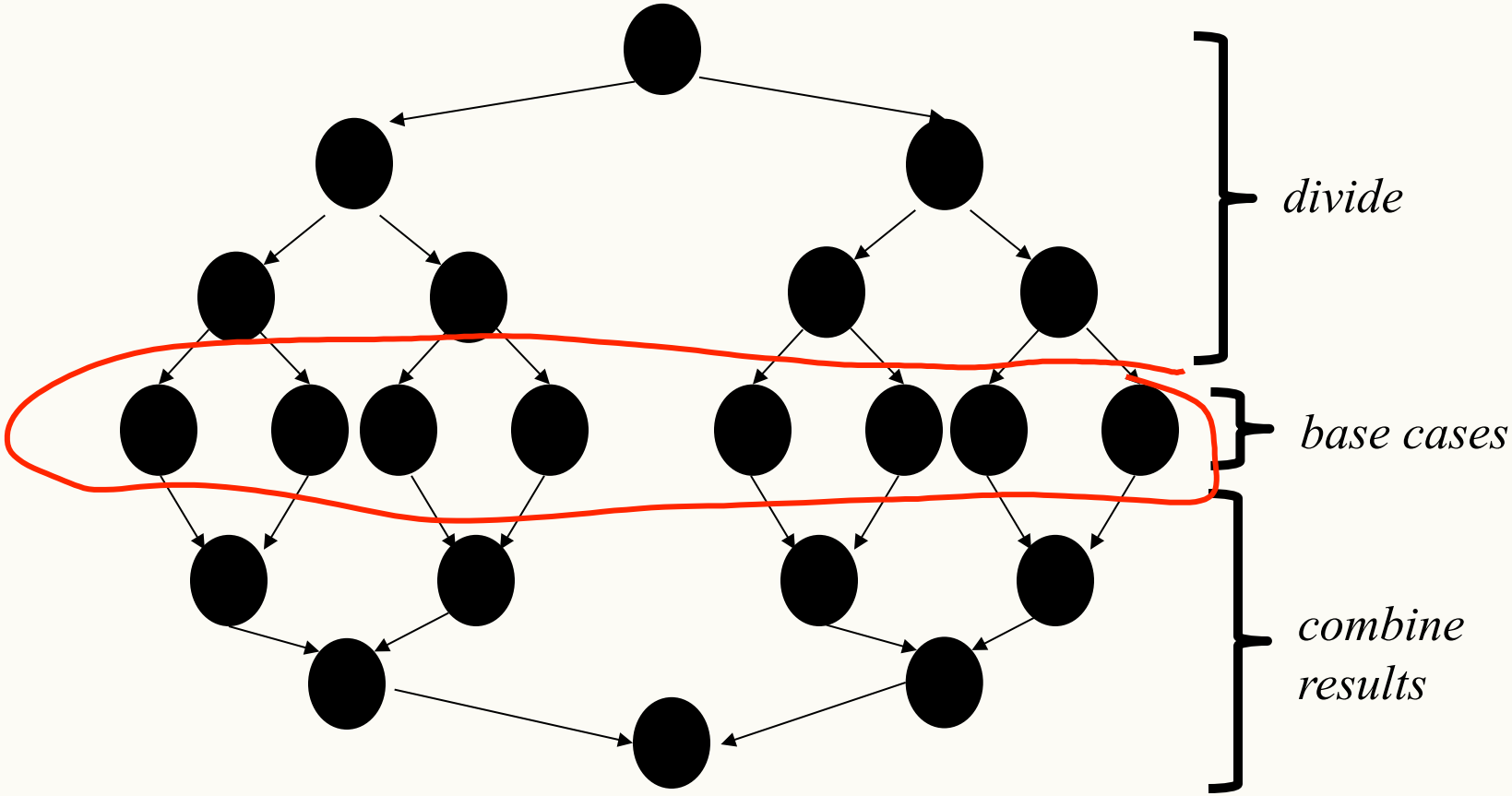


```
main:
  a = fork task1
  b = fork task2
  O(1) work
  join a
  join b


task1:
  O(1) work


task2:
  c = fork task1
  O(1) work
  join c
```

```
int main(..) {
  std::thread t1(&task1);
  std::thread t2(&task2);
  // O(1) work
  t1.join();
  t2.join();
  return 0;
}

void task1() {
  // O(1) work
}

void task2() {
  std::thread t(&task1);
  // O(1) work
  t.join();
}
```

Analyzing Real Code

- **`fork`/`join`** are very flexible, but divide-and-conquer maps and reductions (like **`count-matches`**) use them in a very basic way: **A tree on top of an upside-down tree**



More interesting DAGs?

- The DAGs are not always this simple

- Example:
  - Suppose combining two results might be expensive enough that we want to parallelize each one
  - Then each node in the inverted tree on the previous slide would itself expand into another set of nodes for that parallel computation

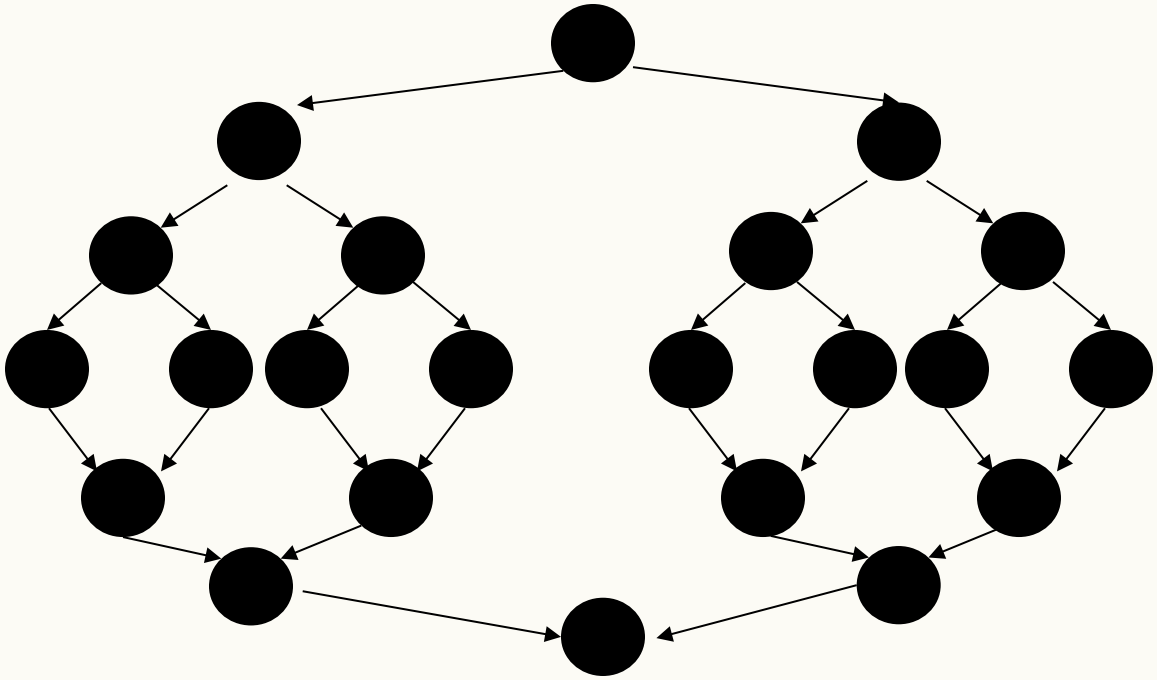
Map/Reduce DAG: Work and Span?

Asymptotically, what's the **work** in this DAG? $O(n)$

Asymptotically, what's the **span** in this DAG? $O(\lg n)$

Reasonable running with P processors?

$T_\infty < T_p < T_1 \rightarrow O(\lg n) < T_p < O(n)$



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
  - Any topological sort is a legal execution
  - $O(n)$ 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
  - $O(\mathbf{log}\ n)$ for simple maps and reductions

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

Parallel algorithms is about decreasing span without increasing work too muchs

Asymptotically Optimal T_P

Can $\mathbf{T_P}$ beat:

- $\mathbf{T_1 / P}$?

No, because otherwise we didn't do all the work!

- $\mathbf{T_\infty}$

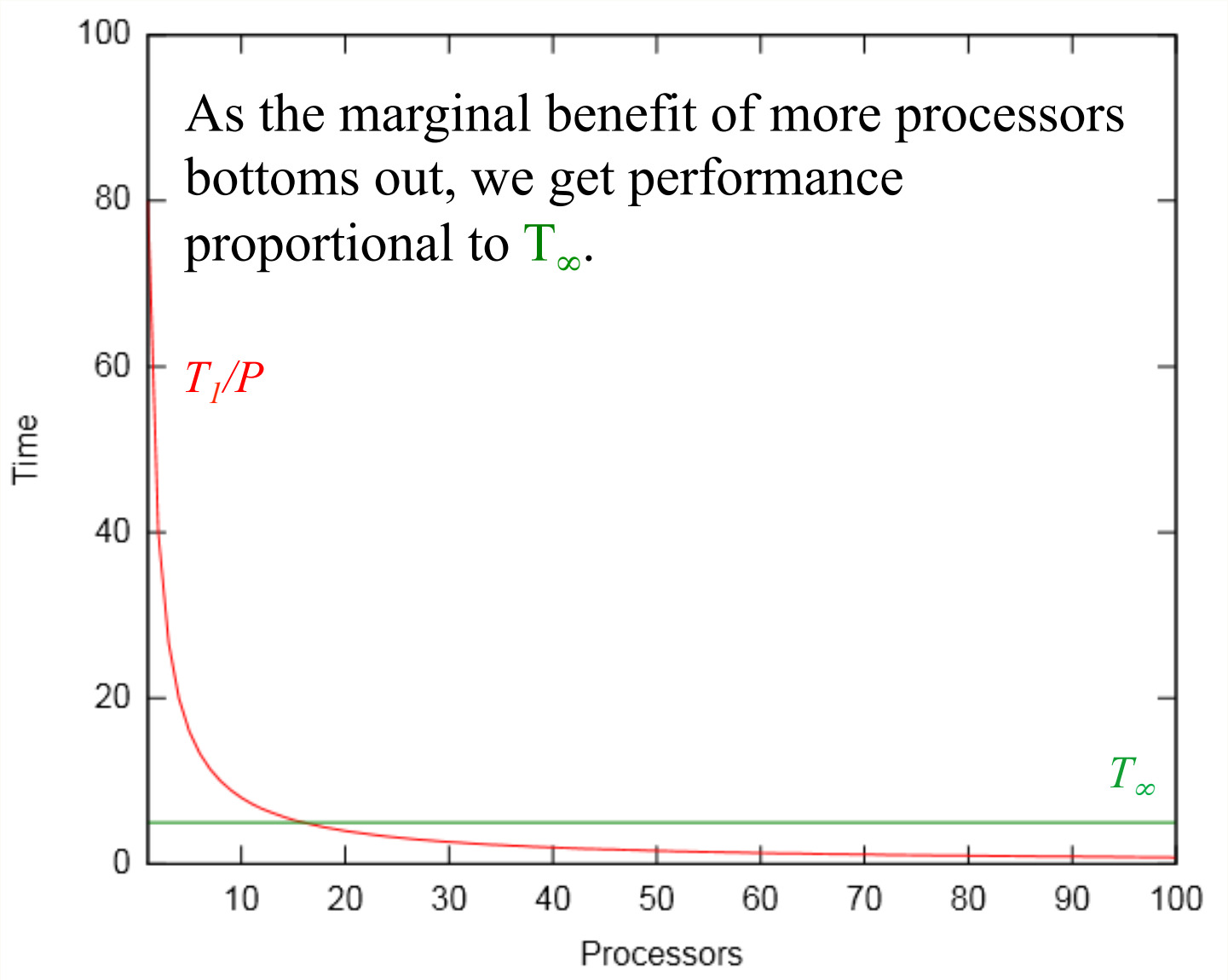
No, because we still don't have have ∞ processors!

So an **asymptotically** optimal execution would be:

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

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

Asymptotically Optimal T_P

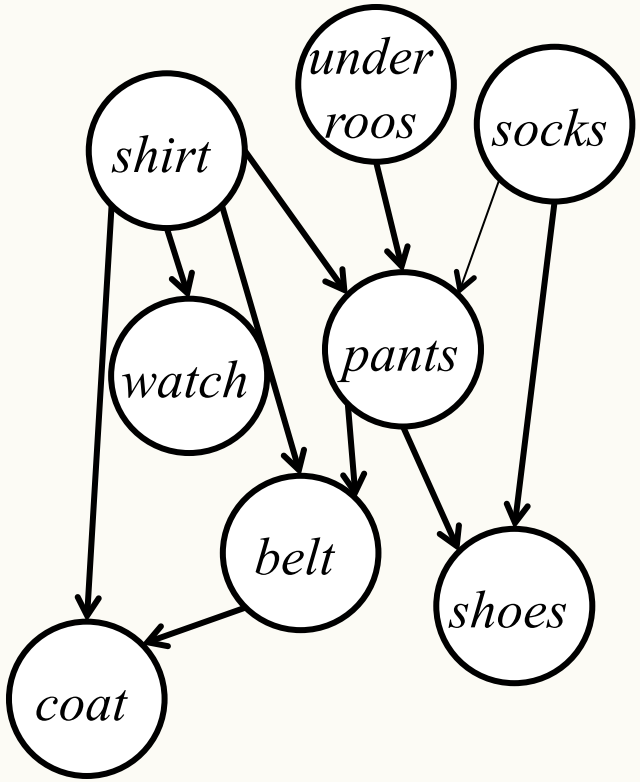


Getting an Asymptotically Optimal Bound

- Good OpenMP implementations guarantee expected bound of

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

  - Expected time because it flips coins when *scheduling*
    - I have two Processors and there are three tasks that I can start with. Coin flip to pick two of them
  - Guarantee requires a few assumptions about your code…



Division of responsibility

- Our job as OpenMP users:
  - Pick a good algorithm
  - Write a program. When run, it creates a DAG of things to do
  - Make all the nodes small-ish and (very) approximately equal amount of work

- The framework-implementer's job:
  - Assign work to available processors to avoid idling
  - Keep constant factors low
  - Give the expected-time optimal guarantee assuming framework-user did their job

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

Examples

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

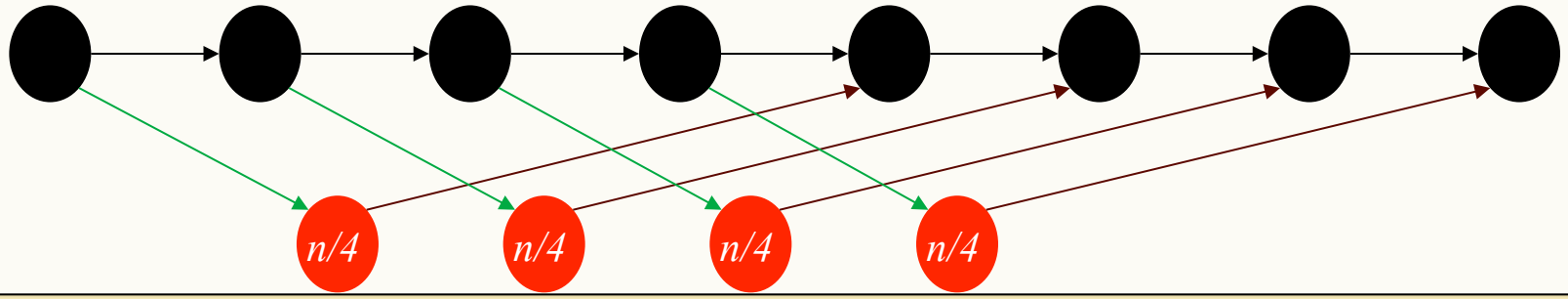
- In the algorithms seen so far (e.g., sum an array):
  - $\mathbf{T_1} = O(n)$
  - $\mathbf{T_\infty} = O(\mathbf{log}\ n)$
  - So expect (ignoring overheads): $\mathbf{T_P = O(n/P + log\ n)}$

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

Loop (**not** Divide-and-Conquer) DAG: Work/Span?

```
int divs = 4;  /* some number of divisions */

std::thread workers[divs];
int results[divs];
for (int d = 0; d < divs; d++)
// count matches in 1/divs sized part of the array
workers[d] = std::thread(&cm_helper_seql, ...);

int matches = 0;
for (int d = 0; d < divs; d++) {
  workers[d].join();
  matches += results[d];
}

return matches;
```

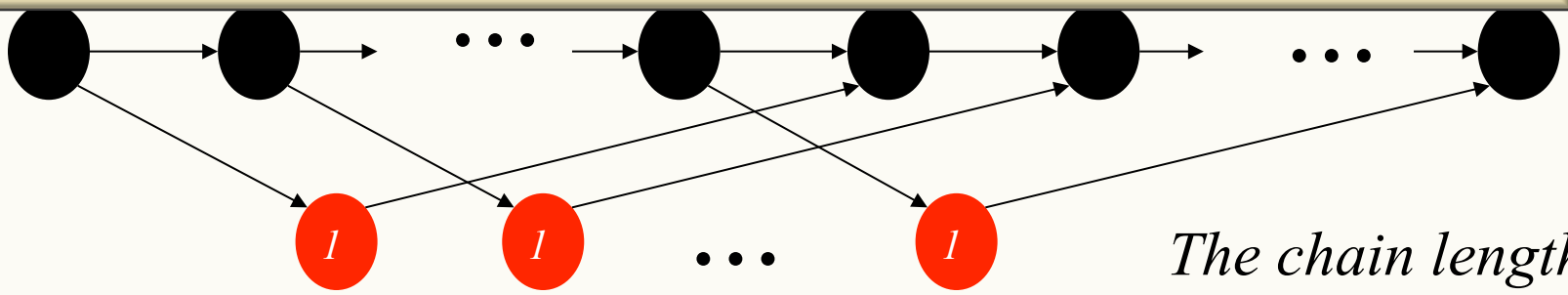
Black nodes take constant time.
Red nodes take non-constant time!



Loop (**not** Divide-and-Conquer) DAG: Work/Span?

```
int divs = n;  /* some number of divisions */

std::thread workers[divs];
int results[divs];
for (int d = 0; d < divs; d++)
// count matches in 1/divs sized part of the array
workers[d] = std::thread(&cm_helper_seql, ...);

int matches = 0;
for (int d = 0; d < divs; d++) {
  workers[d].join();
  matches += results[d];
}

return matches;
```

Black nodes take constant time.
Red nodes take constant time!

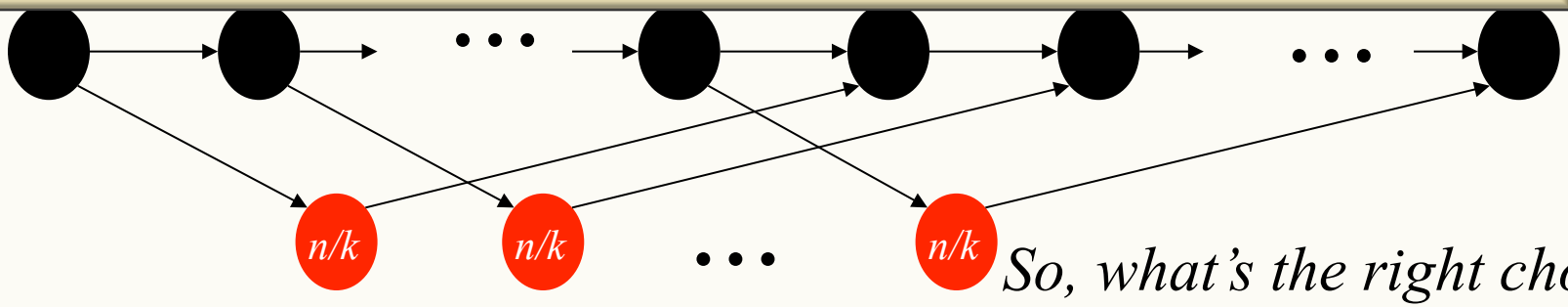


The chain length is O(n)

Loop (**not** Divide-and-Conquer) DAG: Work/Span?

```
int divs = k;  /* some number of divisions */

std::thread workers[divs];
int results[divs];
for (int d = 0; d < divs; d++)
// count matches in 1/divs sized part of the array
workers[d] = std::thread(&cm_helper_seql, ...);

int matches = 0;
for (int d = 0; d < divs; d++) {
  workers[d].join();
  matches += results[d];
}

return matches;
```

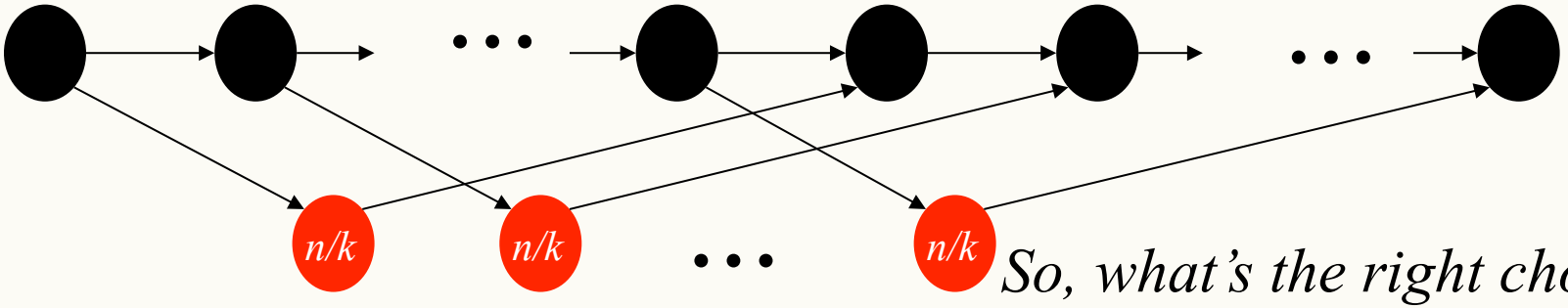
Black nodes take constant time.
Red nodes take non-constant time!



So, what's the right choice of k?

Loop (**not** Divide-and-Conquer) DAG: Work/Span?

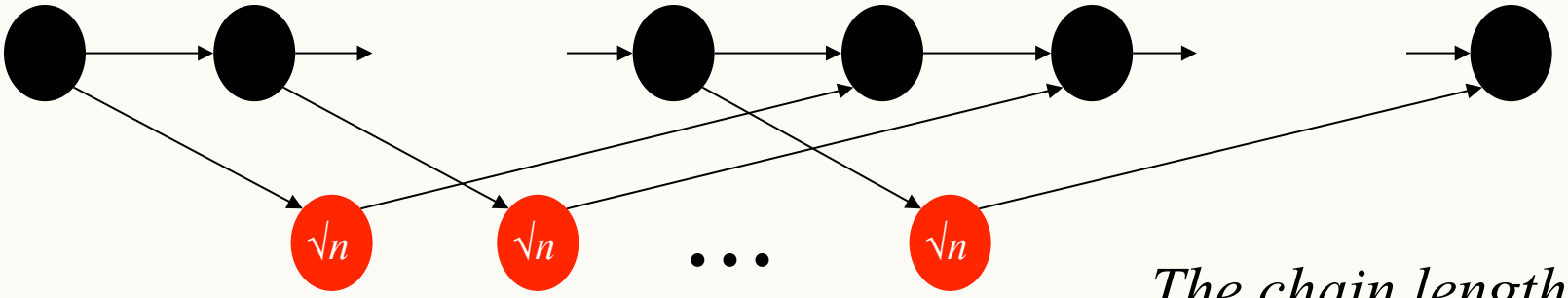
Black nodes take constant time.
Red nodes take non-constant time!



So, what's the right choice of k?

O(n/k + k)
When is n/k + k minimal?

$-n/k^2 + 1 = 0$
k =sqrt(n)



The chain length is O(√n)

Outline

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 C++11 and OpenMP specifics
  - More pragmatics (e.g., installation) in separate notes

Now:

- More examples of simple parallel programs
- Other data structures that support parallelism (or not)
- Asymptotic analysis for fork-join parallelism
- Amdahl's Law

Amdahl's Law (mostly bad news)

- Work/span is great, but real programs typically have:
  - parts that parallelize well like maps/reduces over arrays/trees
  - parts that don't parallelize at all like reading a linked list, getting input, doing computations where each needs the previous step, etc.

"Nine women can't make a baby in one month"

Amdahl's Law (mostly bad news)

- Let $\mathbf{T_1 = 1}$ (measured in weird but handy units)
- Let **S** be the portion of the execution that can't be parallelized

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

- Suppose we get perfect linear speedup on the parallel portion

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

- speedup with **P** processors is (Amdahl's Law):

$$\mathbf{T_1 / T_P}$$

- speedup with ∞ processors is (Amdahl's Law):

$$\mathbf{T_1 / T_\infty}$$

Clicker Question

speedup with **P** processors $T_1 / T_P = 1 / (S + (1-S)/P)$

speedup with ∞ processors $T_1 / T_\infty = 1 / S$

- Suppose 33% of a program is sequential
  - How much speed-up do you get from 2 processors?
  - A ~1.5
  - B ~2
  - C ~2.5
  - D ~3
  - E: none of the above

Clicker Question (Answer)

speedup with **P** processors $T_1 / T_P = 1 / (S + (1-S)/P)$

speedup with ∞ processors $T_1 / T_\infty = 1 / S$

- Suppose 33% of a program is sequential
  - How much speed-up do you get from 2 processors?
  - **A ~1.5**
  - B ~2
  - C ~2.5
  - D ~3
  - E: none of the above

$$\frac{1}{0.33 + \frac{0.66}{2}} = 1.51$$

Clicker Question

speedup with **P** processors $T_1 / T_P = 1 / (S + (1-S)/P)$

speedup with ∞ processors $T_1 / T_\infty = 1 / S$

- Suppose 33% of a program is sequential
  - How much speed-up do you get from 1,000,000 processors?
  - A ~1.5
  - B ~2
  - C ~2.5
  - D ~3
  - E: none of the above

Mostly Bad News

speedup with **P** processors $T_1 / T_P = 1 / (S + (1-S)/P)$

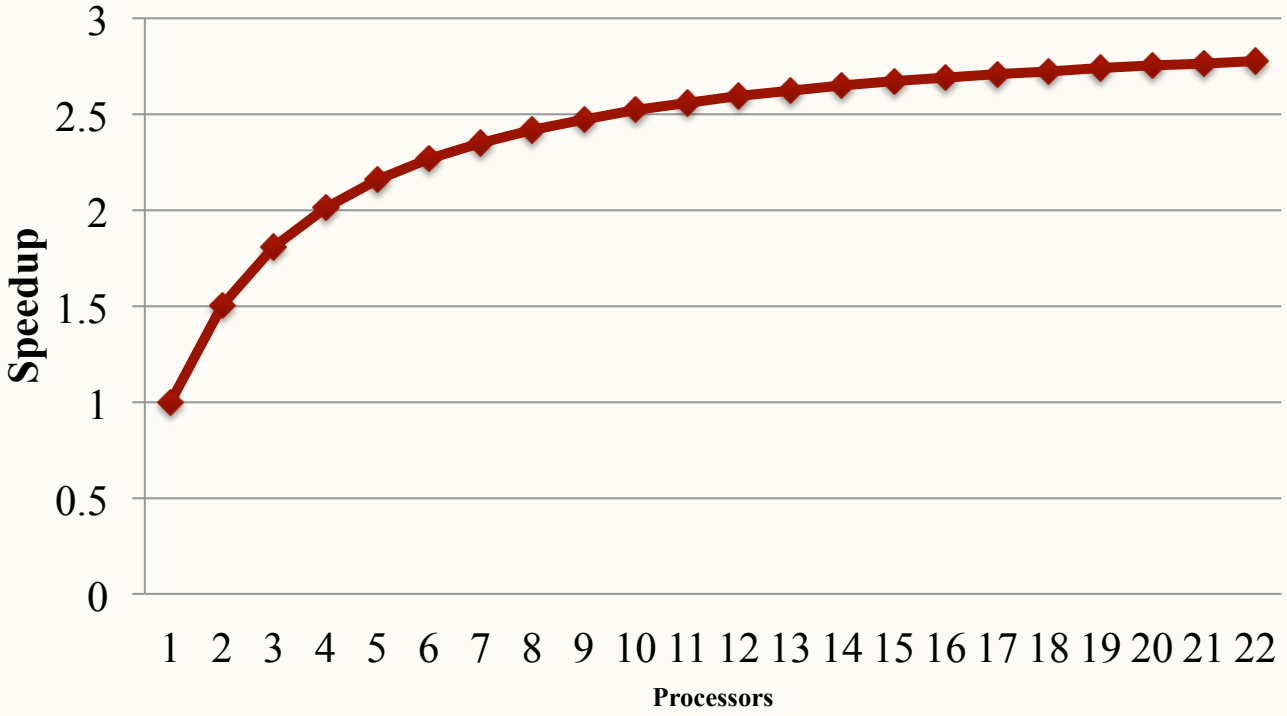
speedup with ∞ processors $T_1 / T_\infty = 1 / S$

- Suppose 33% of a program is sequential
  - How much speed-up do you get from 1,000,000 processors?
  - A ~1.5
  - B ~2
  - C ~2.5
  - **D ~3**
  - E: none of the above

$$\frac{1}{0.33 + \frac{0.66}{1000000}} \sim 3$$

Why Such Bad News?

- Suppose 33% of a program is sequential
  - How much speed-up do you get from more processors?



Why such bad news

speedup with **P** processors $T_1 / T_P = 1 / (S + (1-S)/P)$

speedup with ∞ processors $T_1 / T_\infty = 1 / S$

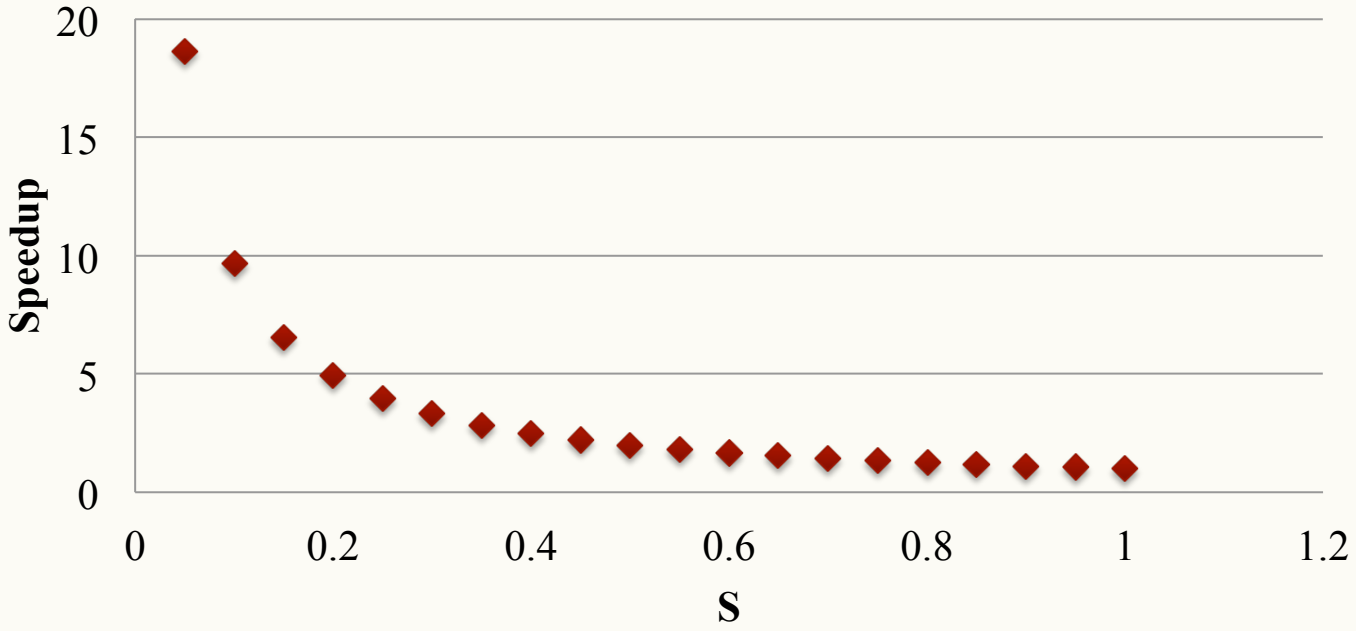
- Suppose you miss the good old days (1980-2005) where 12ish years was long enough to get 100x speedup
  - Now suppose in 12 years, clock speed is the same but you get 256 processors instead of 1
  - For 256 processors to get at least 100x speedup What do we need for **S**?

A: $S \leq 0.1$ B: $0.1 < S \leq 0.2$ C: $0.2 < S \leq 0.6$

D: $0.6 < S \leq 0.8$ E: $0.8 < S$

Why such bad news

- we need $100 \leq 1 / (S + (1-S)/256)$
- **You would need at most 0.61% of the program to be sequential, so S needs to be smaller than 0.0061. Answer: A**
  - with 256 processors how much speedup do you get?



All is not lost. Parallelism can still help!

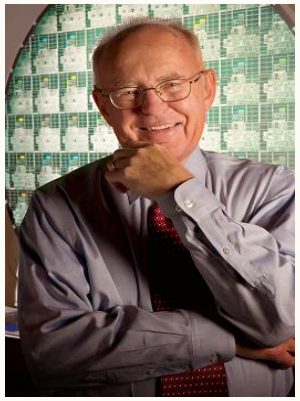
In our maps/reduces, the sequential part is ***O*(1)** and so becomes trivially small as ***n*** scales up. **(This is tremendously important!)**

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

We can change the problem we're solving or do new things

- Example: Video games use tons of parallel processors
  - They are not rendering 10-year-old graphics faster
  - They are rendering more beautiful(?) monsters

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
  - Diminishing returns of adding more processors
- Both are incredibly important in designing computer systems

CPSC 221 Administrative Notes

- Marking lab 10 :Apr 7 – Apr 10

- Written Assignment #2 is marked
  - If you have any questions or concerns attend office hours held by Cathy or Kyle

- Final call for Piazza question is out and is due Mon at 5pm.

CPSC 221 Administrative Notes

- Final exam Wed, Apr 22 at 12:00 SRC A
  - Open book (same as midterm) check course webpage

- PRACTICE Written HW #3 is available on the course website (Solutions will be released next week)

CPSC 221 Administrative Notes

- Office hours
- Apr 14 Tue Kyle (12-1)
- Apr 15 Wed Hassan(5-6)
- Apr 16 Thu Brian (1-3)
- Apr 17 Fri Kyle(11-1)
- Apr 18 Sat Lynsey (12 -2)
- Apr 19 Sun Justin (12 -2)
- Apr 20 Mon Benny (10-12)
- Apr 21 Tue Hassan(11-1) Kai Di(4 -6)

Evaluations

- Instructor Evaluation
  - We'll spend some time at the end of the lecture on this.

So… Where Were We?

- We've talked about
  - Parallelism and Concurrency
  - Fork/Join Parallelism
  - Divide-and-Conquer Parallelism
  - Map & Reduce
  - Using parallelism in other data structures such as Trees and Linked list
  - Work, Span, Asymptotic analysis T_p
  - Amdahl's Law

FANCIER FORK-JOIN ALGORITHMS: PREFIX, PACK, SORT

Motivation

- This section presents a few more sophisticated parallel algorithms to demonstrate:
  - sometimes problems that seem inherently sequential turn out to have efficient parallel algorithms.
  - we can use parallel-algorithm techniques as building blocks for other larger parallel algorithms.
  - we can use asymptotic complexity to help decide when one parallel algorithm is better than another.
- As is common when studying algorithms, we will focus on the algorithms instead of code.

The prefix-sum problem

- Given a list of integers as input, produce a list of integers as output where
- **`output[i]`** = **`input[0]+input[1]+…+input[i]`**

- `Example`

input	*42*	*3*	*4*	*7*	*1*	*10*
output	*42*	*45*	*49*	*56*	*57*	*67*

- It is not at all obvious that a good parallel algorithm exists.
- it seems we need output[i-1] to compute output[i].

The prefix-sum problem

- Given a list of integers as input, produce a list of integers as output where
- **`output[i]`** = **`input[0]+input[1]+…+input[i]`**

Sequential version is straightforward:

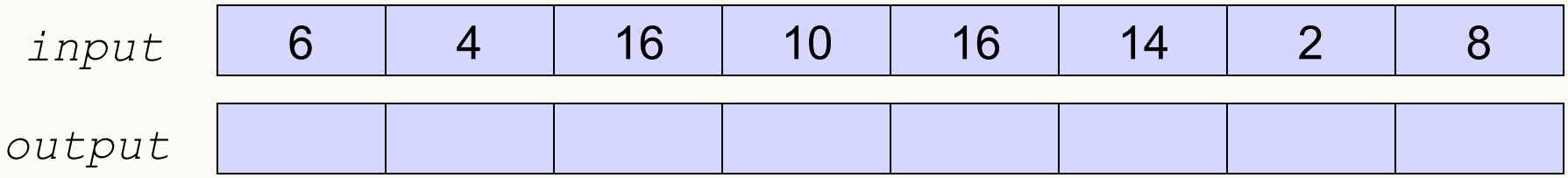
```
vector<int> prefix_sum(const vector<int>& input)
{
  vector<int> output(input.size());
  output[0] = input[0];
  for(int i=1; i < input.size(); i++)
    output[i] = output[i-1]+input[i];
  return output;
}
```

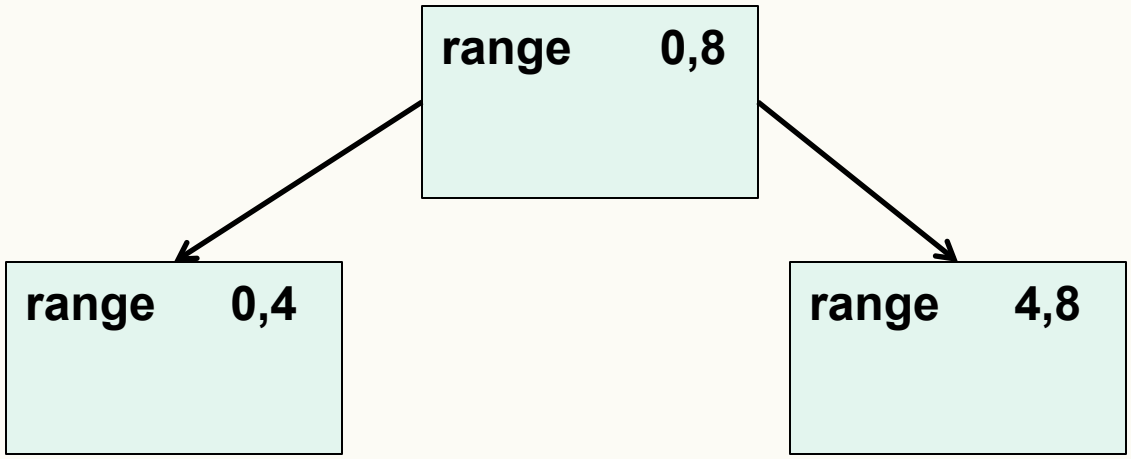
Parallel prefix-sum

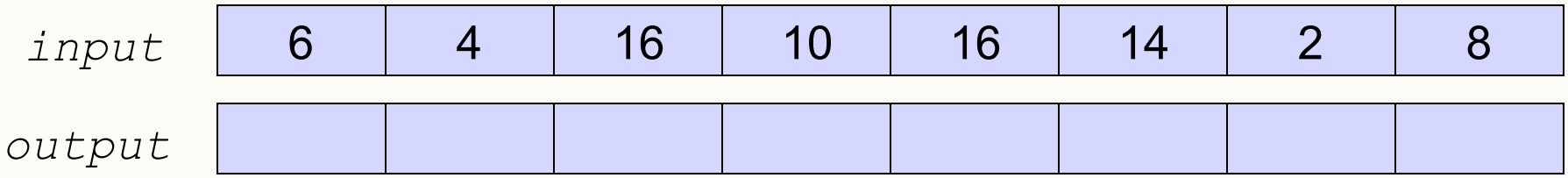
The parallel-prefix algorithm does two passes:

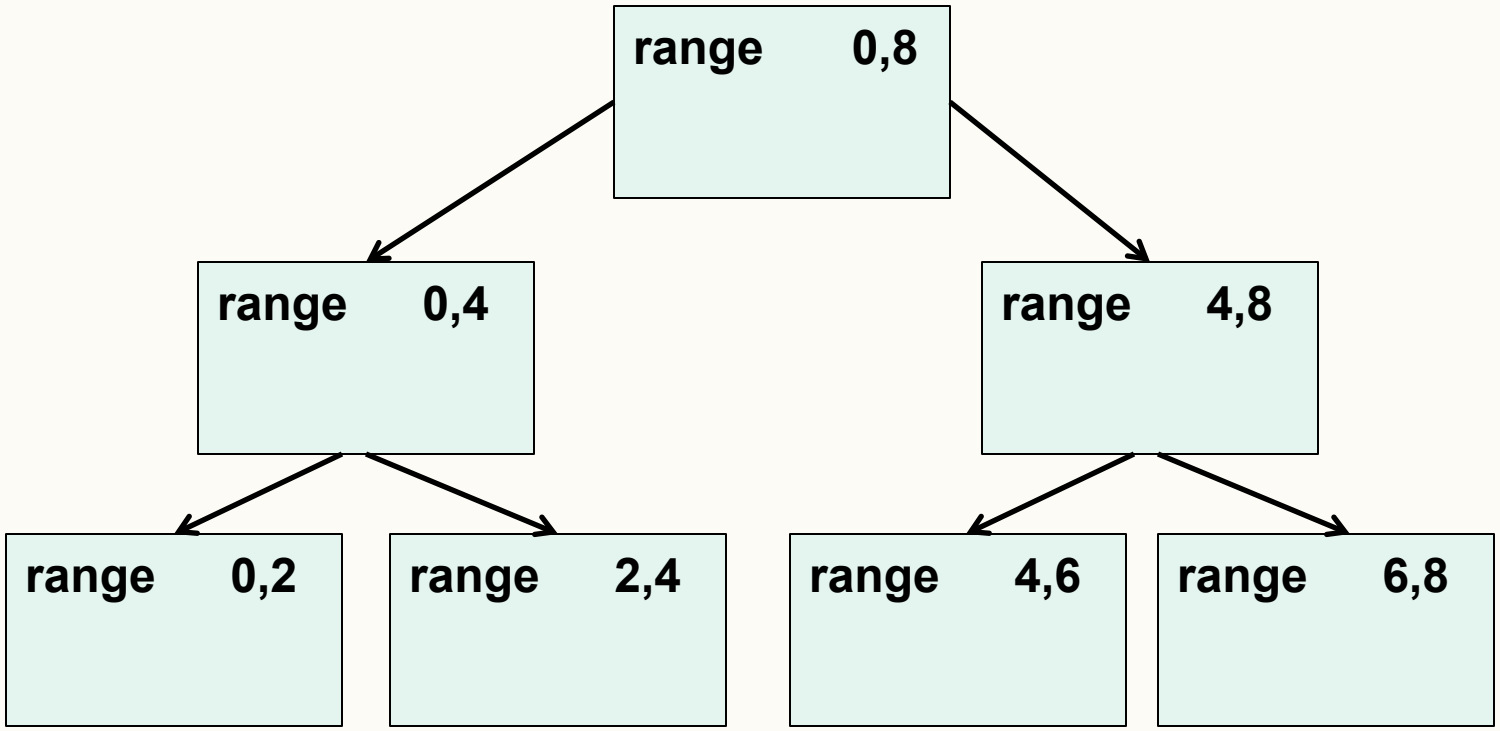
1. A parallel sum to build a binary tree:
   - Root has sum of the range [`0,n`)
   - An internal node with the sum of [`lo,hi`) has
     - Left child with sum of [`lo,middle`)
     - Right child with sum of [`middle,hi`)
   - A leaf has sum of [`i,i+1`), i.e., `input[i]` (or an appropriate larger region w/a cutoff)

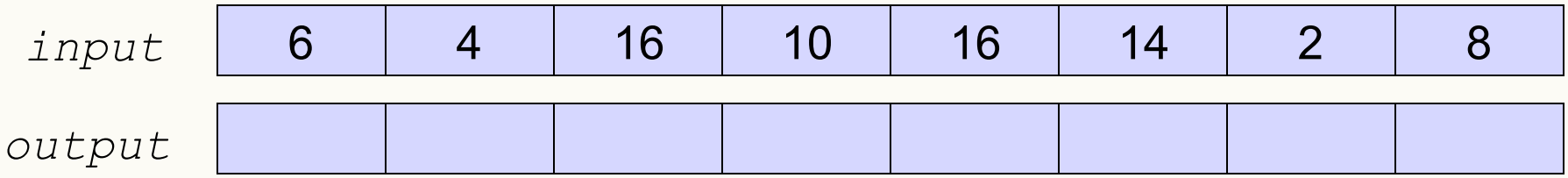
range	0,8

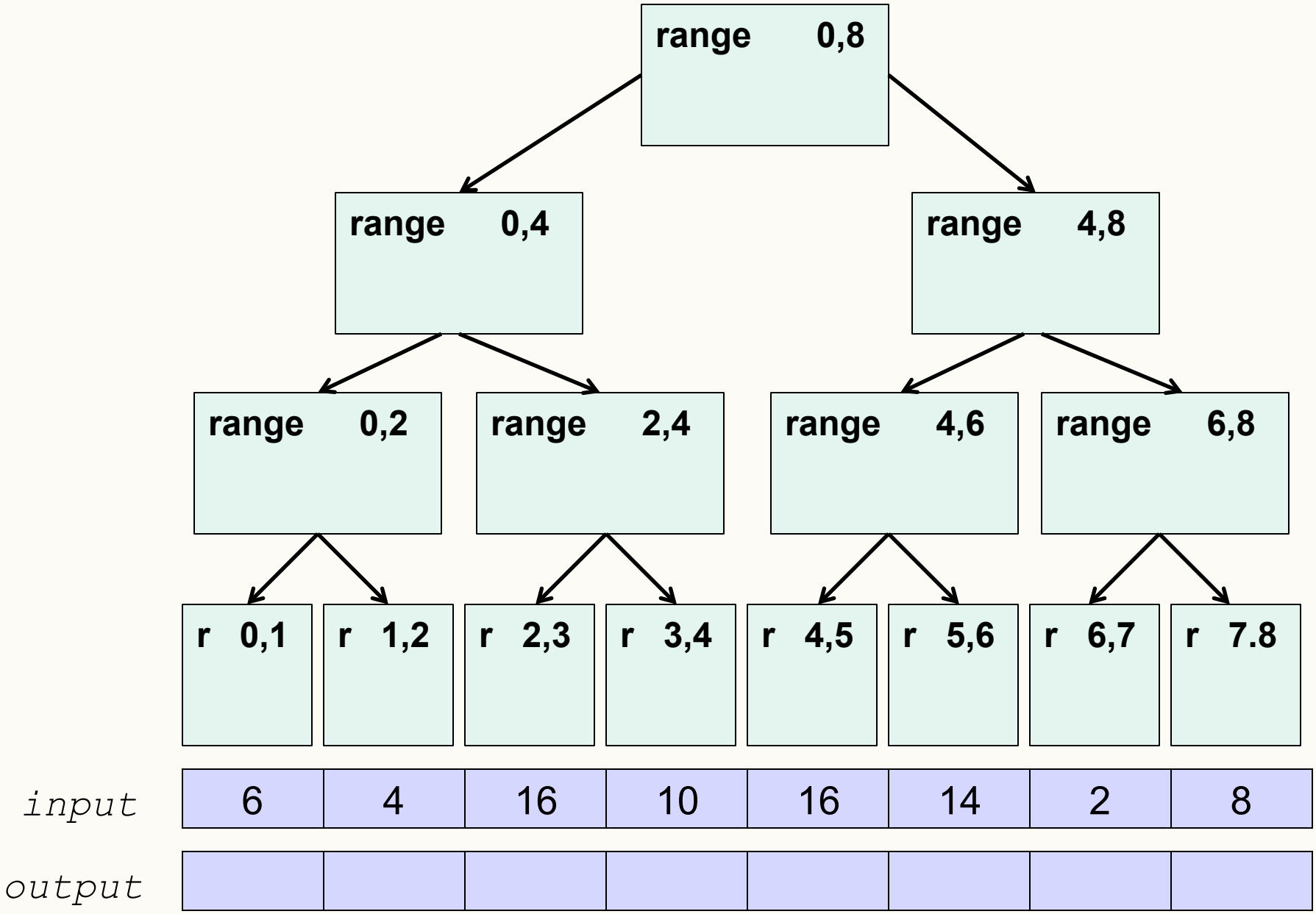


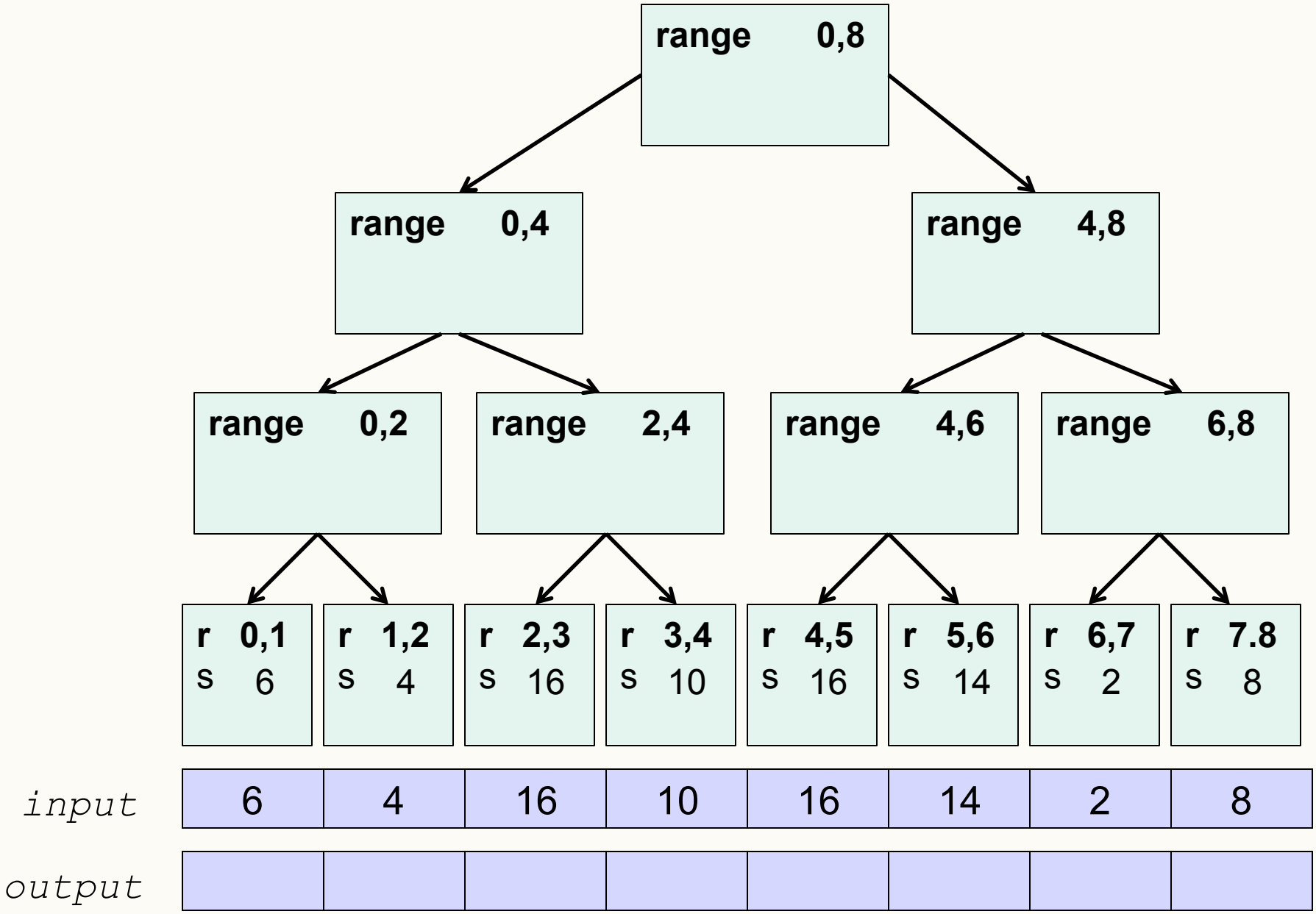

range 0,8
range 0,4
range 4,8

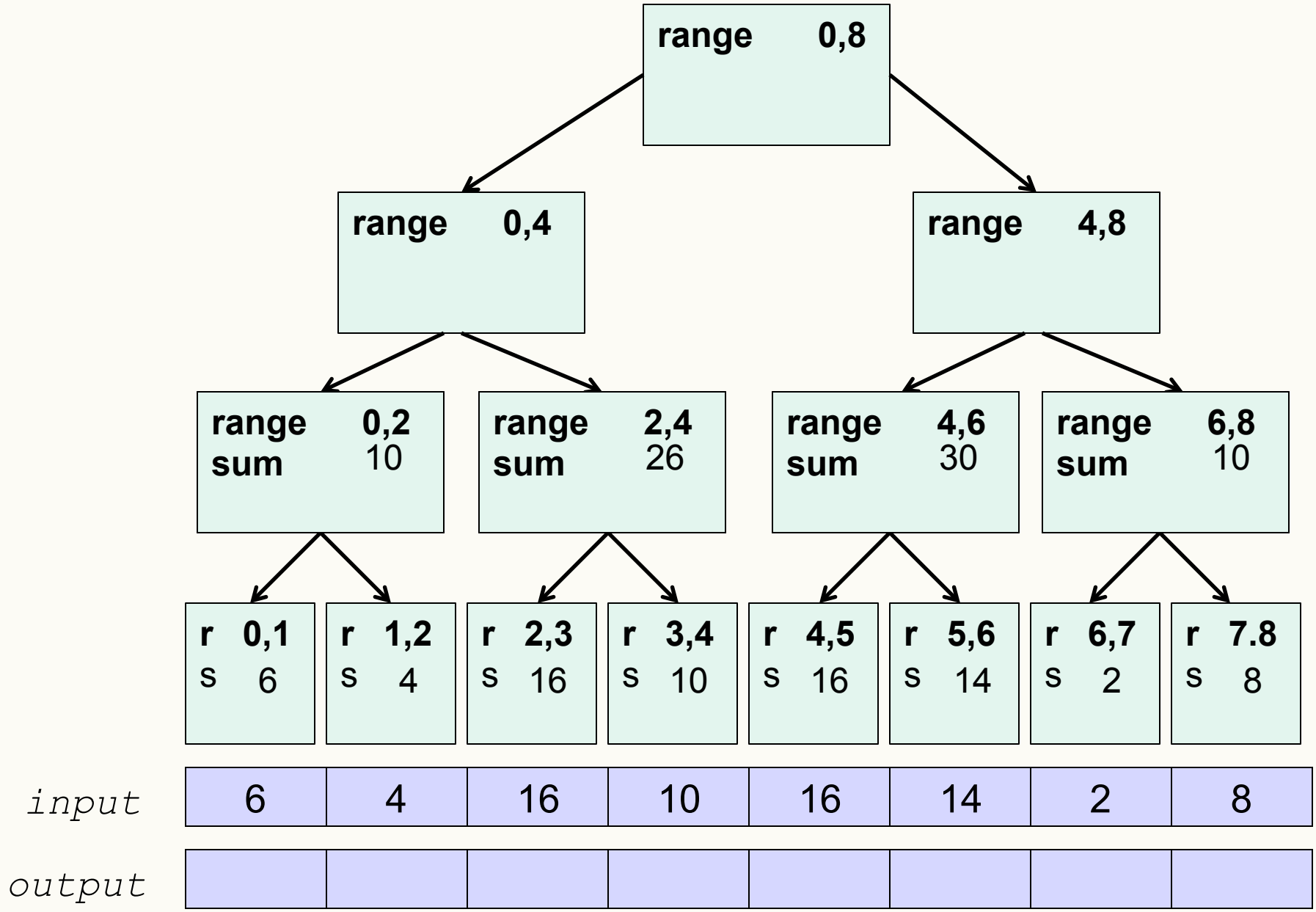

input
6
4
16
10
16
14
2
8
output

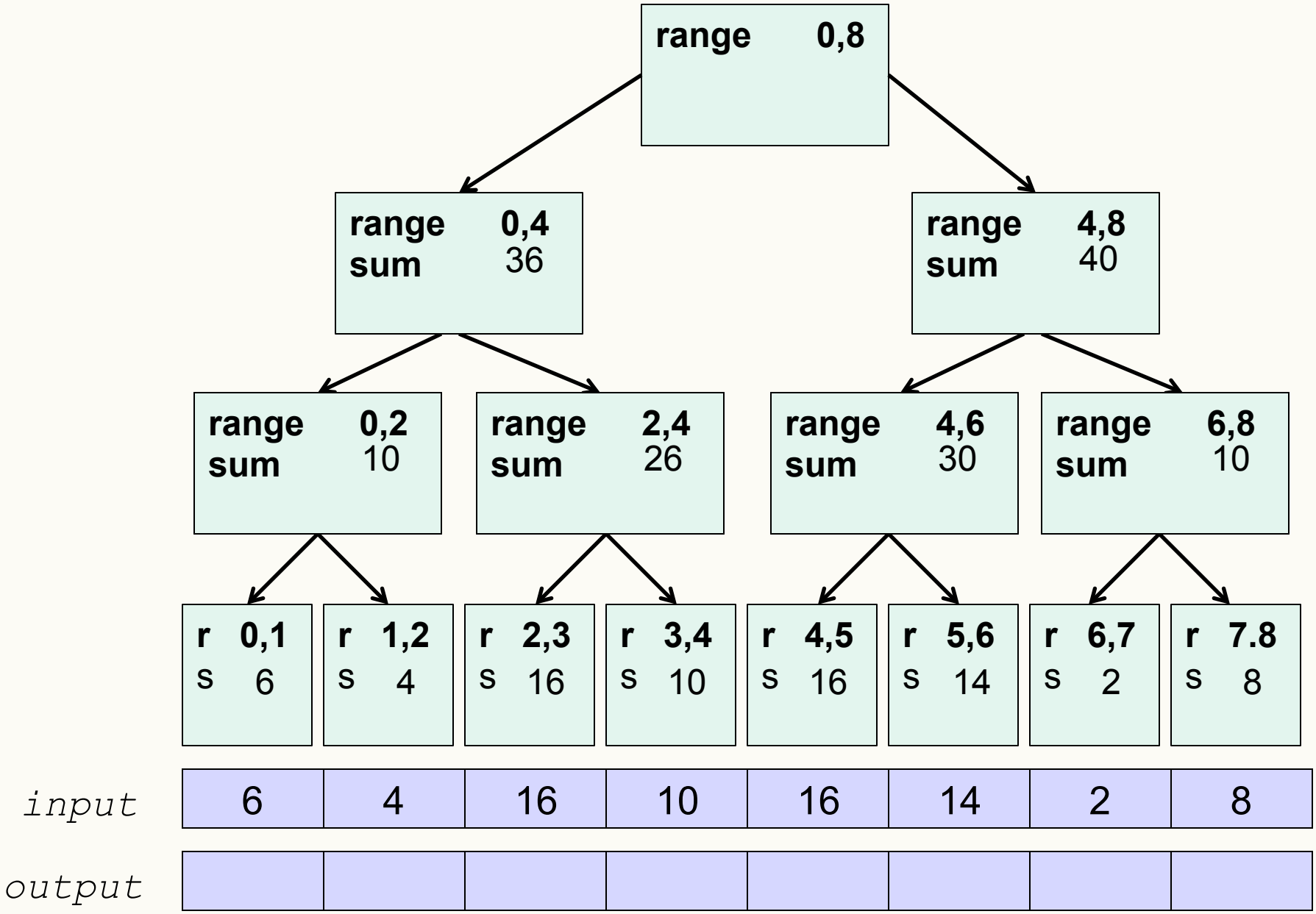

range 0,8
range 0,4
range 4,8
range 0,2
range 2,4
range 4,6
range 6,8

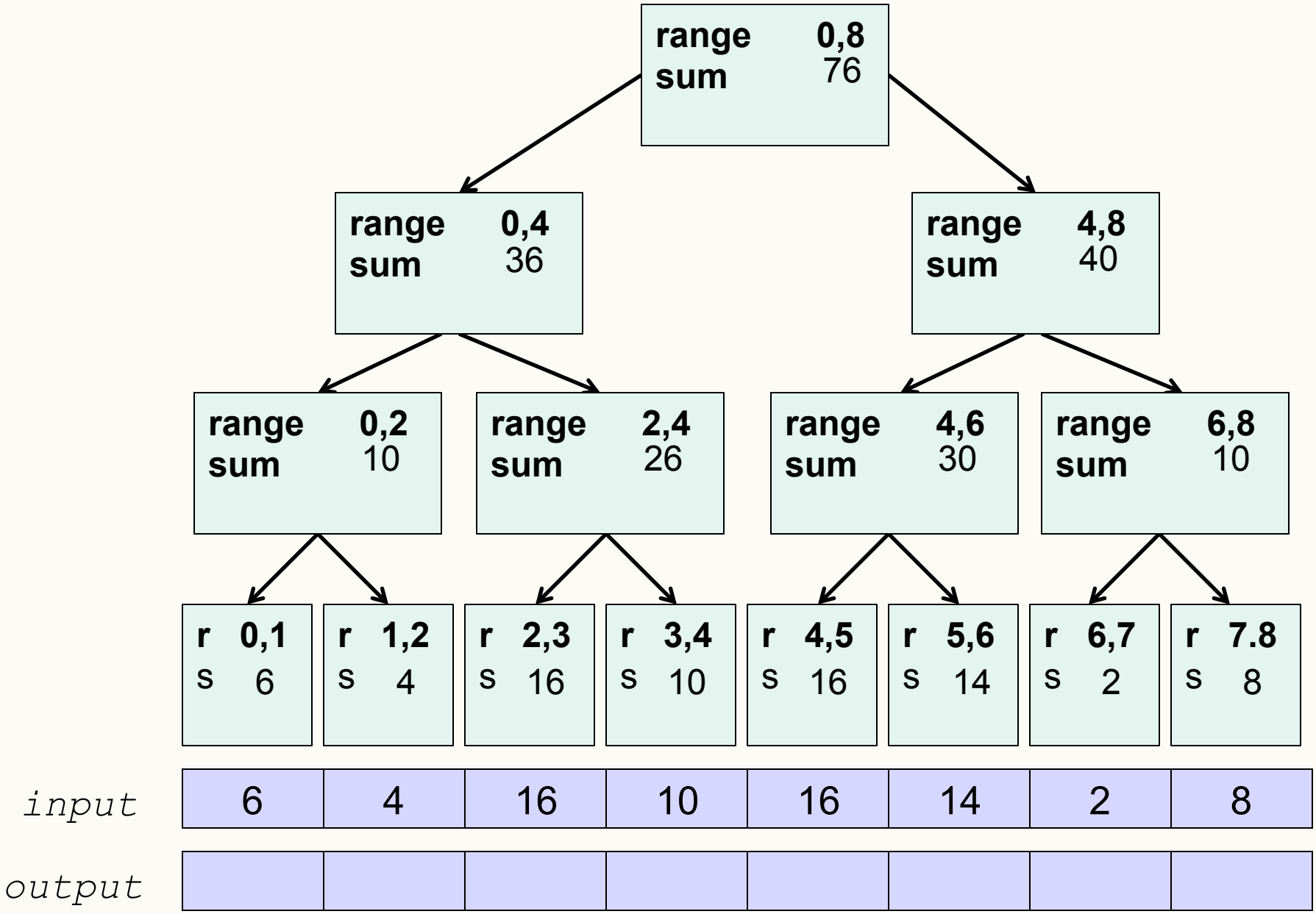

input
6
4
16
10
16
14
2
8
output


range 0,8
range 0,4
range 4,8
range 0,2
range 2,4
range 4,6
range 6,8
r 0,1
r 1,2
r 2,3
r 3,4
r 4,5
r 5,6
r 6,7
r 7.8
input
6
4
16
10
16
14
2
8
output


range 0,8
range 0,4
range 4,8
range 0,2
range 2,4
range 4,6
range 6,8
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


range 0,8
range 0,4
range 4,8
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


range 0,8
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


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

The algorithm, step 1

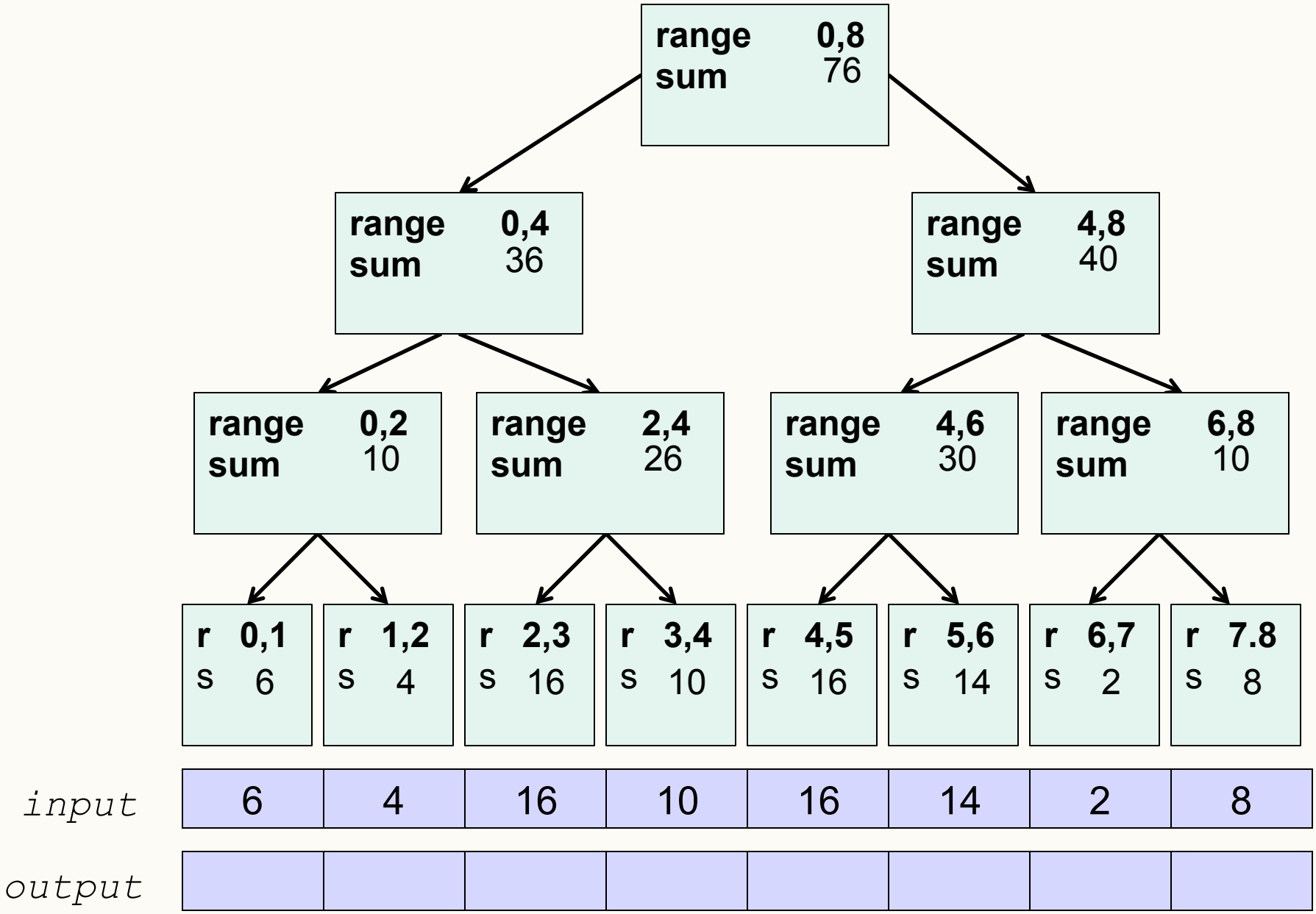
1. A parallel sum to build a binary tree:
   - Root has sum of the range [**`0,n`**)
   - An internal node with the sum of [**`lo,hi`**) has
     - Left child with sum of [**`lo,middle`**)
     - Right child with sum of [**`middle,hi`**)
   - A leaf has sum of [**`i,i+1`**), i.e., **`input[i]`**
     (or an appropriate larger region w/a cutoff)
   - **`Work O(n)`**
   - **`Span O(lg n)`**

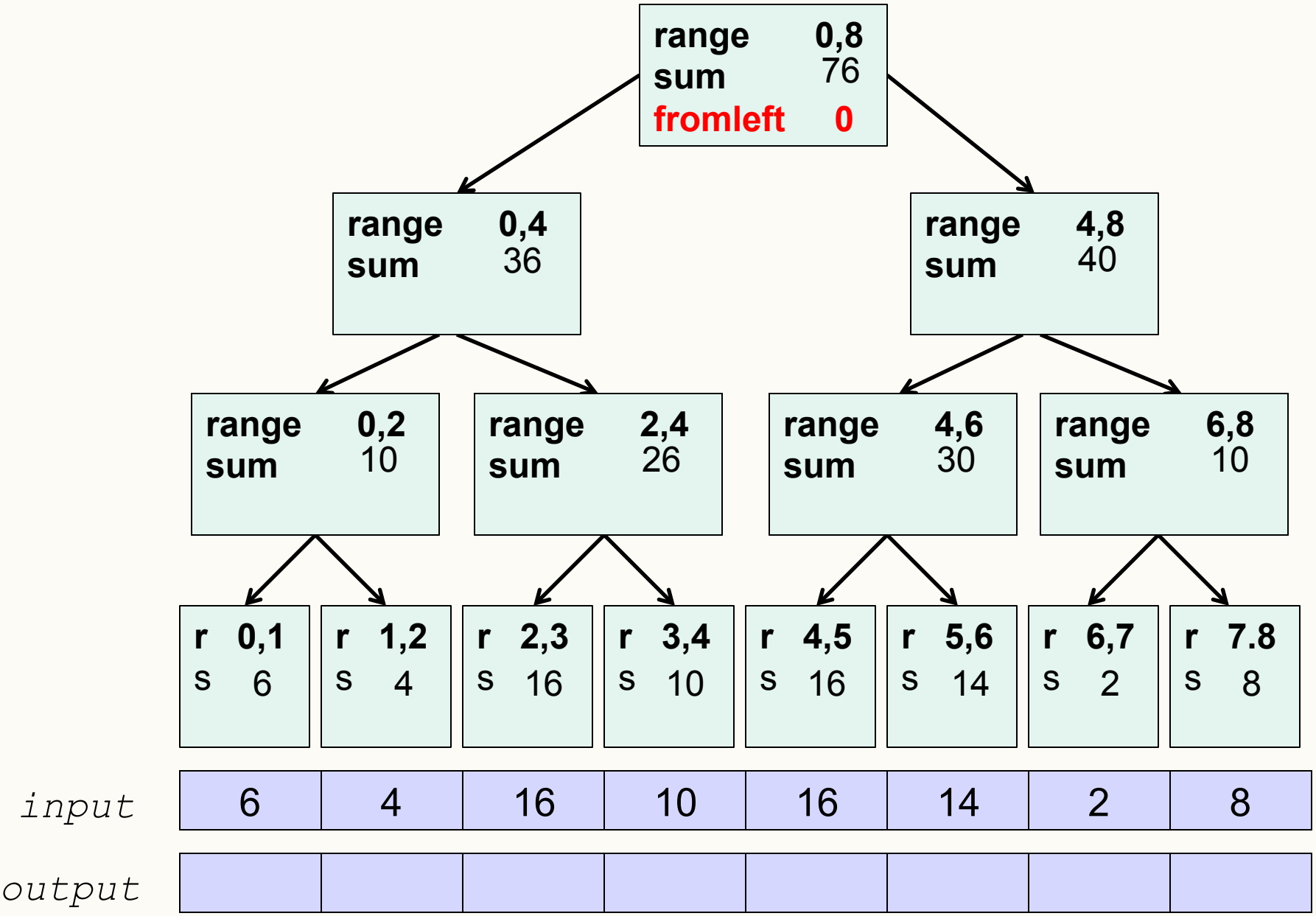
The algorithm, step 2

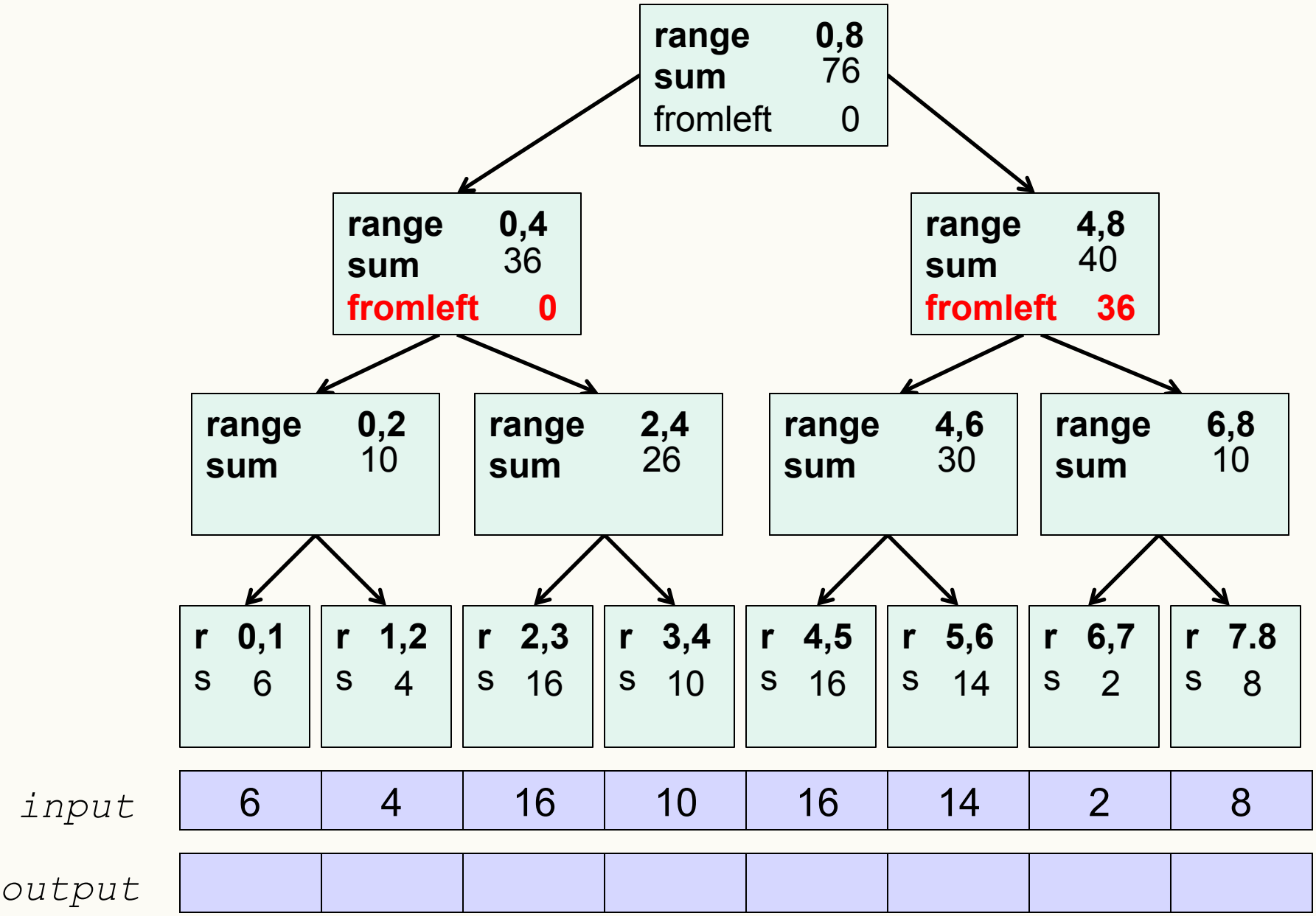
2. Parallel map, passing down a **`fromLeft`** parameter
   - Root gets a **`fromLeft`** of **`0`**
   - Internal nodes pass along:
     - to its left child the same **`fromLeft`**
     - to its right child **`fromLeft`** plus its left child's **`sum`**
   - At a leaf node for array position **`i`**, **`output[i]=fromLeft +input[i]`**

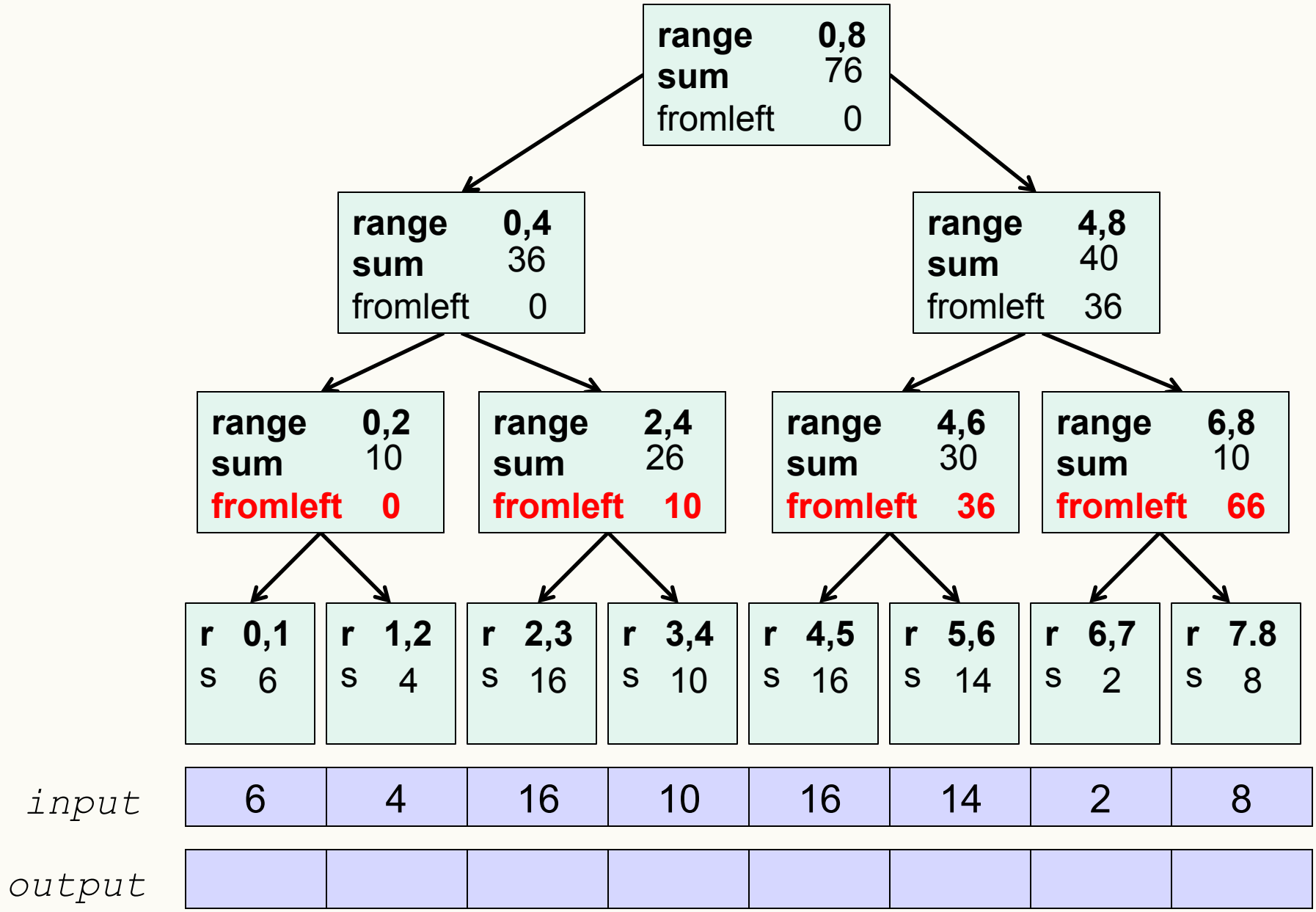
How? A map down the step 1 tree, leaving results in the output array.

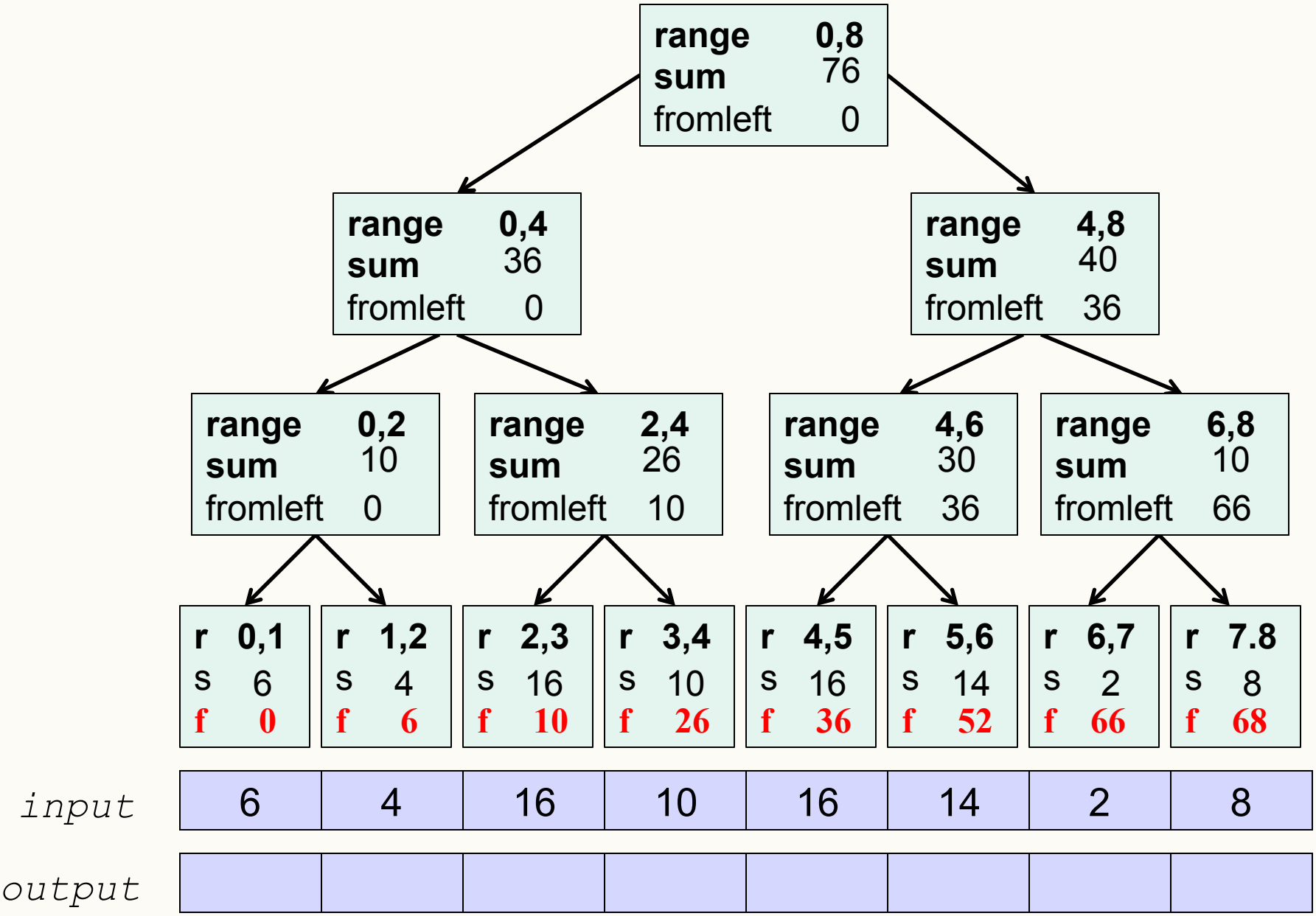
Notice the invariant: **`fromLeft`** is the sum of elements left of the node's rangE

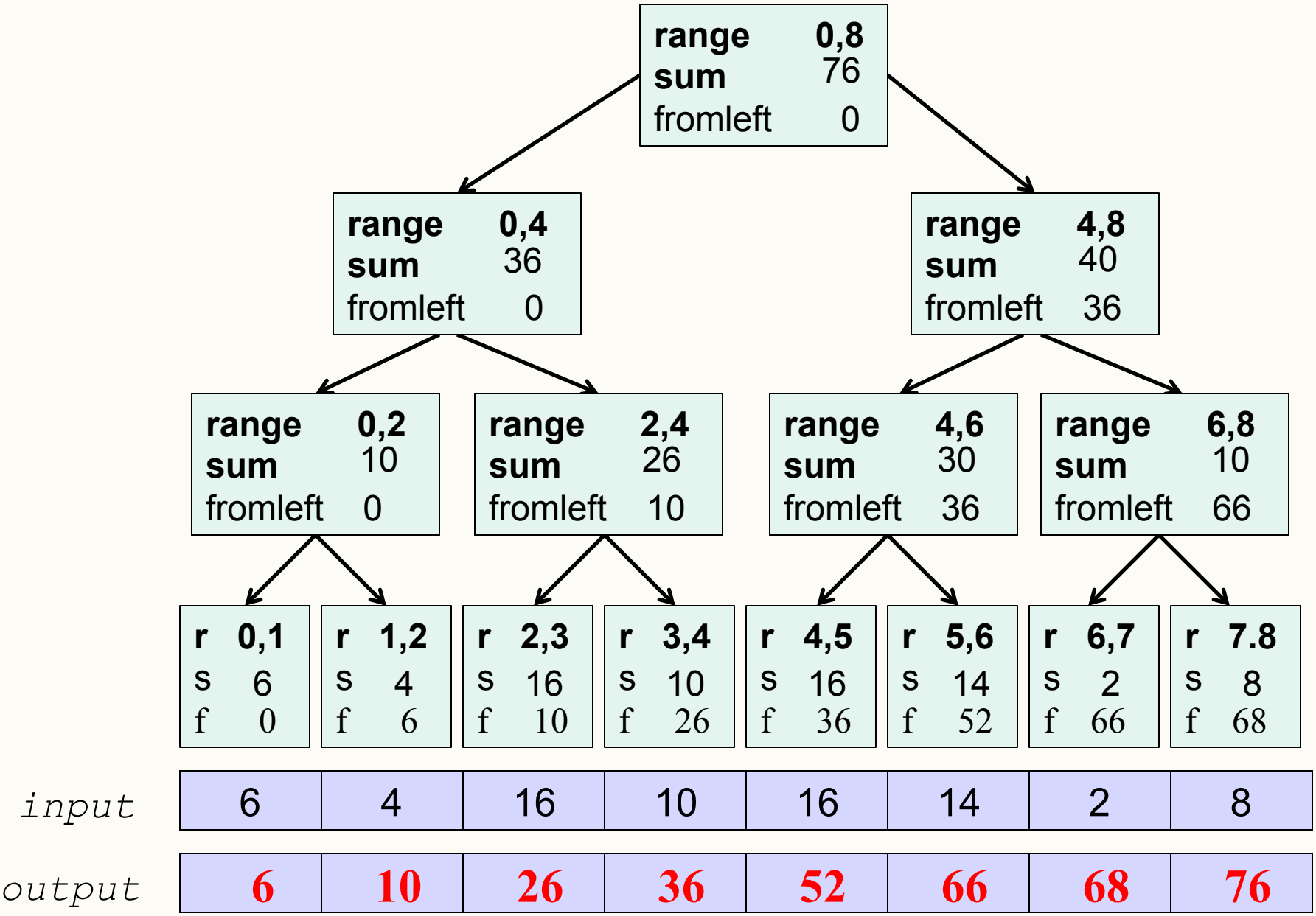

range 0,8
sum 76
range 0,4
sum 36
range 4,8
sum 40
range 0,2
sum 10
range 2,4
sum 26
range 4,6
sum 30
range 6,8
sum 10
r 0,1
s 6
r 1,2
s 4
r 2,3
s 16
r 3,4
s 10
r 4,5
s 16
r 5,6
s 14
r 6,7
s 2
r 7.8
s 8
input
6
4
16
10
16
14
2
8
output


range 0,8
sum 76
fromleft 0
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


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
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


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
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


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


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, step 2

2. Parallel map, passing down a **`fromLeft`** parameter
   - Root gets a **`fromLeft`** of **`0`**
   - Internal nodes pass along:
     - to its left child the same **`fromLeft`**
     - to its right child **`fromLeft`** plus its left child's **`sum`**
   - At a leaf node for array position **`i`**, **`output[i]=fromLeft +input[i]`**

- Work? O(n)
- Span? O(lg n)

Parallel prefix, generalized

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

Pack

- Given an array input, produce an array output containing only those elements of input that satisfy some property, and in the same order they appear in input.
- Example:
  - `input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]`
  - `Values greater than 10`
  - `output [17, 11, 13, 19, 24]`
- Notice the length of output is unknown in advance but never longer than input.

Parallel Prefix Sum 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 to produce the output

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

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

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, $O(\texttt{lg}\ n)$ span
  - 2 or 3 passes, but 3 is a constant

- Parallelized packs will help us parallelize quicksort…

Parallelizing Quicksort

- Recall quicksort was sequential, recursive, 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?

Parallelizing 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)

$T_\infty(n) = n + T(n/2)$ only doing of the recursive calls

$= n + n/2 + T(n/4)$

$= n/1 + n/2 + n/4 + n/8 + \ldots + 1$ assuming $n = 2^k$

$= n\,(1 + 1/2 + 1/4 + 1/n) \in \Theta(n)$

  - So parallelism (i.e., work / span) is O(`log` n)

Parallelizing 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)

O(**`log`** n) speed-up with an infinite number of processors is okay, but a bit underwhelming (Sort 10^9 elements 30 times faster)

Parallelizing Quicksort (Doing better)

- We need to split the work done in Partition

Partition all the data into: O(n)

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, $O(\log n)$ 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

Parallelizing Quicksort (Doing better)

- We need to split the work done in Partition

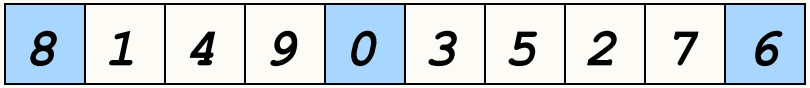
Partition all the data into: O(n)

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

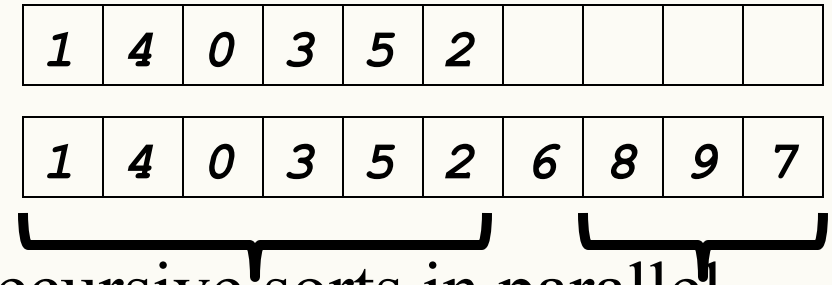
- This is just two packs!
  - We know a pack is $O(n)$ work, $O(\log n)$ span
  - Pack elements less than pivot into left side of **`aux`** array
  - Pack elements greater than pivot into right size of **`aux`** array
  - Put pivot between them and recursively sort
  - With a little more cleverness,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

Example

- Step 1: pick pivot as median of three



- Steps 2a and 2c (combinable): pack less than, then pack greater than into a second array
  - Fancy parallel prefix to pull this off not shown



- Step 3: Two recursive sorts in parallel
  - Can sort back into original array (like in mergesort)
  - Note that it uses O(n) extra space like mergesort too!

Parallelizing Quicksort (Doing better)

		Best / expected case work
1.	Pick a pivot element	O(1)
2.	Partition all the data into:	O(lg n)
3.	Recursively sort A and C	T(n/2)

- With $O(\texttt{lg}\ n)$ span for partition, the total best-case and expected-case span for quicksort is $T(n) = \lg n + T(n/2)$

$T(n) = \lg n + T(n/2)$

$= \lg n + \lg n - 1 + T(n/4)$

$= \lg n + \lg n - 1 + \lg n - 2 + \ldots + 1$

$= k + k - 1 + k - 2 + \ldots + 1$ (let $\lg n = k$)

$$= \sum_{i=1}^{k} i \in O(k^2) \in O(\lg^2 n)$$

Span: $O(lg^2\ n)$
So parallelism is O(n / `lg` n)
Sort 10^9 elements 10^8 times faster

Parallelizing mergesort

- Recall mergesort: sequential, not-in-place, worst-case $O(n \log n)$

1. Sort left half and right half	2T(n/2)
2. Merge results	O(n)

- Just like quicksort, doing the two recursive sorts in parallel changes the recurrence for the span to

$$T(n) = O(n) + 1T(n/2) = O(n)$$

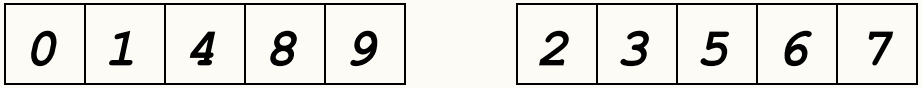
- Again, parallelism is $O(\log n)$

To do better, need to parallelize the merge

– The trick won't use parallel prefix this time

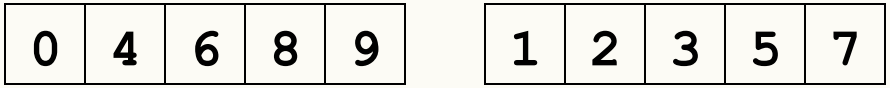
Parallelizing the merge

- Need to merge two sorted subarrays (may not have the same size)

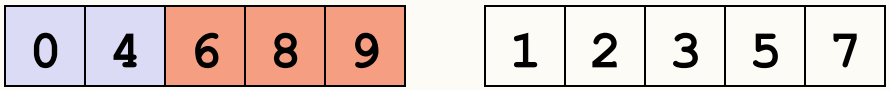


- Idea: Suppose the larger subarray has m elements. In parallel:
  - Merge the first m/2 elements of the larger half with the "appropriate" elements of the smaller half
  - Merge the second m/2 elements of the larger half with the rest of the smaller half

Parallelizing the merge

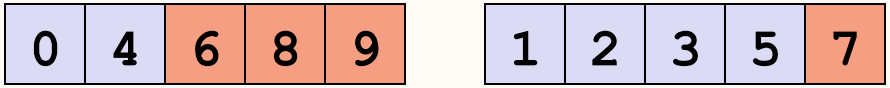


Parallelizing the merge



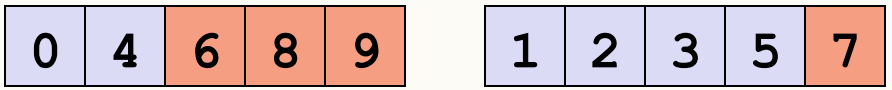
1. Get median of bigger half: $O(1)$ to compute middle index

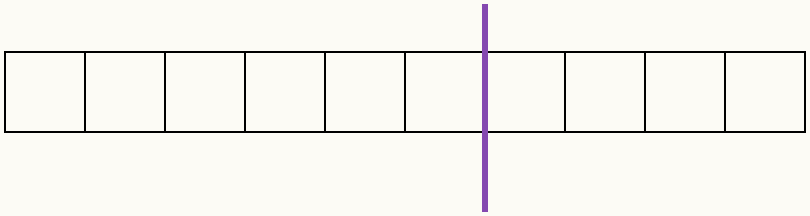
Parallelizing the merge



1. Get median of bigger half: $O(1)$ to compute middle index
2. Find how to split the smaller half at the same value as the left-half split: O(log n) to do binary search on the sorted small half

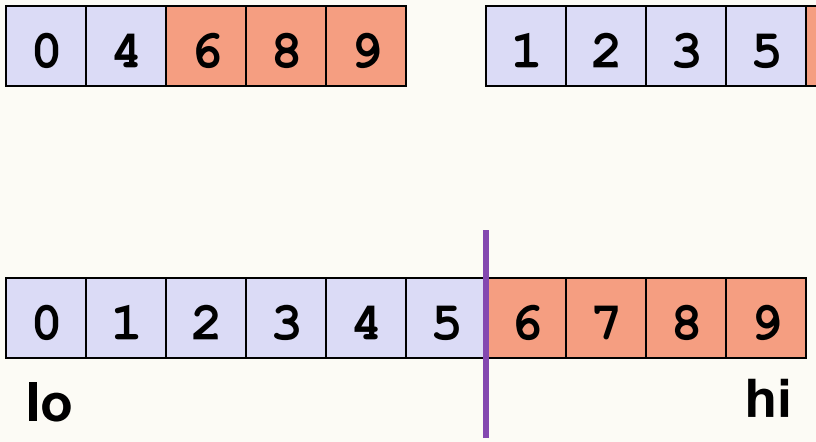
Parallelizing the merge





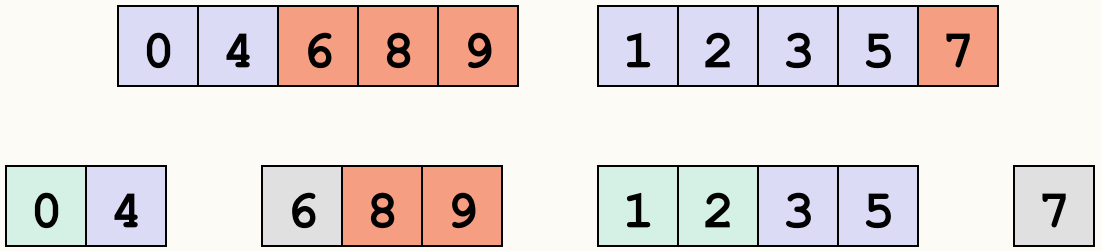
1. Get median of bigger half: $O(1)$ to compute middle index
2. Find how to split the smaller half at the same value as the left-half split: O(log n) to do binary search on the sorted small half
3. Size of two sub-merges conceptually splits output array: O(1)

Parallelizing the merge



1. Get median of bigger half: $O(1)$ to compute middle index
2. Find how to split the smaller half at the same value as the left-half split: O(log n) to do binary search on the sorted small half
3. Size of two sub-merges conceptually splits output array: O(1)
4. Do two submerges in parallel

The Recursion



- When we do each merge in parallel, we split the bigger one in half and use binary search to split the smaller one

Analysis

- Sequential recurrence for mergesort:

$$T(n) = 2T(n/2) + O(n) \text{ which is } O(n \log n)$$

- Doing the two recursive calls in parallel but a sequential merge:

Work: same as sequential

Span: $T(n)=1T(n/2)+O(n)$ which is $O(n)$

- Parallel merge makes work and span harder to compute
  - Each merge step does an extra $O(\log n)$ binary search to find how to split the smaller subarray
  - To merge n elements total, do two smaller merges of possibly different sizes

- worst-case split is $(1/4)n$ and $(3/4)n$
  - When subarrays same size and "smaller" splits "all" / "none"

Analysis

For just a parallel **merge** of n elements:

Work is $T(n) = T(3n/4) + T(n/4) + O(\lg n)$ which is $O(n)$

Span is $T(n) = T(3n/4) + O(\lg n)$, which is $O(\lg^2 n)$

- (neither bound is immediately obvious, but "trust me")

So for mergesort with parallel merge overall:

Work is $T(n) = 2T(n/2) + O(n)$, which is $O(n \lg n)$

Span is $T(n) = 1T(n/2) + O(\lg^2 n)$, which is $O(\lg^3 n)$

So parallelism (work / span) is $O(n / \lg^2 n)$

– Not quite as good as quicksort's $O(n / \lg n)$

• But worst-case guarantee

Sort 10^9 elements 10^7 times faster

Learning Goals (revisited)

- Define work—the time it would take one processor to complete a parallelizable computation; span—the time it would take an infinite number of processors to complete the same computation; and Amdahl's Law—which relates the speedup in a program to the proportion of the program that is parallelizable.

- Use work, span, and Amdahl's Law to analyse the speedup available for a particular approach to parallelizing a computation.

- Judge appropriate contexts for and apply the parallel map, parallel reduce, and parallel prefix computation patterns.