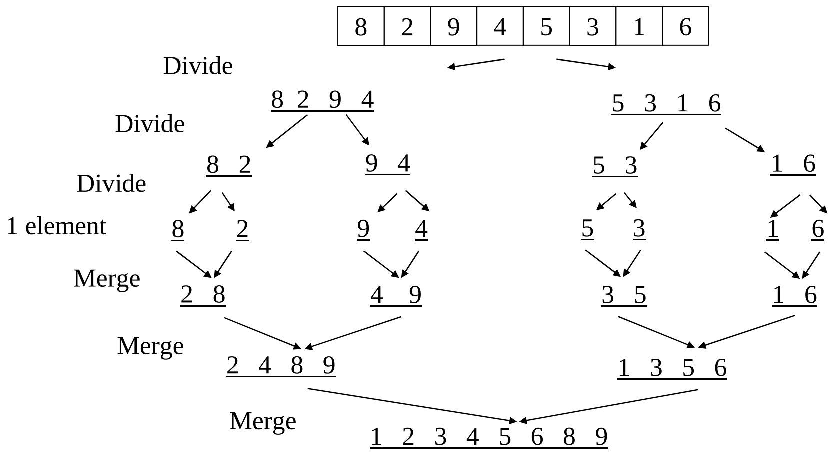
Mergesort example: Merge as we return from recursive calls



We need another array in which to do each merging step; merge results into there, then copy back to original array

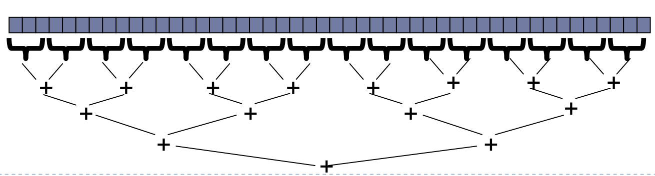
Dijkstra's Algorithm Overview

- Given a weighted graph and a vertex in the graph (call it A), find the shortest path from A to each other vertex
  - Cost of path defined as sum of weights of edges
  - Negative edges not allowed
- The algorithm:
  - Create a table like this:
  - Init A's cost to 0, others infinity (or just '??')
  - While there are unknown vertices:
    - Select unknown vertex w/ lowest cost (A initially)
    - Mark it as known
    - Update cost and path to all uknown vertices adjacent to that vertex

vertex	known?	cost	path
A		0	
B		??	
C		??	
D		??	

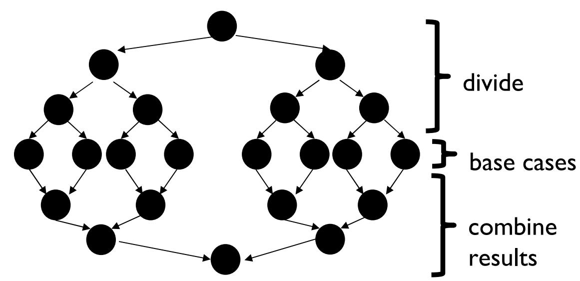
Parallelism Overview

- We say it takes time $\mathbf{T_P}$ to complete a task with P processors
- Adding together an array of n elements would take O(n) time, when done sequentially (that is, P=1)
  - Called the **work**; $\mathbf{T_1}$
- If we have 'enough' processors, we can do it much faster; O(logn) time
  - Called the **span**; $\mathbf{T_\infty}$



Considering Parallel Run-time

Our **`fork`** and **`join`** frequently look like this:



- Each node takes O(1) time
- Even the base cases, as they are at the cut-off
- Sequentially, we can do this in O(n) time; O(1) for each node, ~3n nodes, if there were no cut-off (linear # on base case row, halved each row up/down)
- Carrying this out in (perfect) parallel will take the time of the longest branch; ~2logn, if we halve each time

Some Parallelism Definitions

- ***Speed-up*** on **P** processors: $\mathbf{T_1 / T_P}$
- We often assume perfect linear speed-up
  - That is, $\mathbf{T_1 / T_P} = P$; w/ 2x processors, it's twice as fast
  - 'Perfect linear speed-up '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

The ForkJoin Framework Expected Performance

If you write your program well, you can get the following expected performance:

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

- T_1/P for the overall work split between P processors
  - P=4? Each processor takes 1/4 of the total work
- $O(T_\infty)$ for merging results
  - Even if $P=\infty$, then we still need to do $O(T_\infty)$ to merge results
- *What does it mean??*
- We can get decent benefit for adding more processors; effectively linear speed-up at first (expected)
- With a large # of processors, we're still bounded by $\mathbf{T_\infty}$; that term becomes dominant

Amdahl's Law

Let the ***work*** (time to run on 1 processor) be 1 unit time

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

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

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

Amdahl's Law: The overall ***speedup*** with **P** processors is:

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

And the ***parallelism*** (infinite processors) is:

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

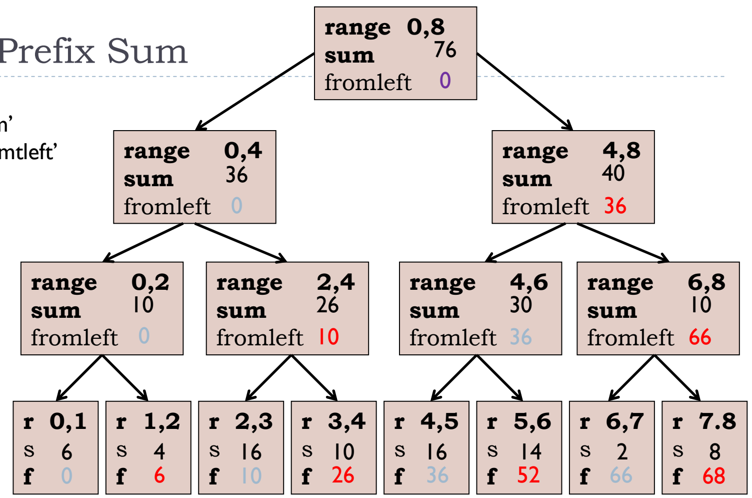
Parallel Prefix Sum

- Given an array of numbers, compute an array of their running sums in *O(logn)* span
- Requires 2 passes (each a parallel traversal)
  - First is to gather information
  - Second figures out output

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

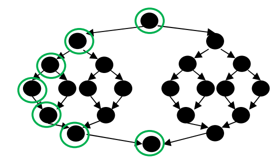
Parallel Prefix Sum

2 passes:
1. Compute 'sum'
2. Compute 'fromleft'



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

Parallel Quicksort



2 optimizations:
1. Do the two recursive calls in parallel
   - Now recurrence takes the form:
     $O(n) + 1T(n/2)$
     So $O(n)$ span
2. Parallelize the partitioning step
   - Partitioning normally $O(n)$ time
   - Recall that we can use Parallel Prefix Sum to 'filter' with $O(\log n)$ span
   - Partitioning can be done with 2 filters, so $O(\log n)$ span for each partitioning step

These two parallel optimizations bring parallel quicksort to a span of $O(\log^2 n)$

Race Conditions

A race condition occurs when the computation result depends on scheduling (how threads are interleaved)
- If T1 and T2 happened to get scheduled in a certain way, things go wrong
- We, as programmers, cannot control scheduling of threads; result is that we need to write programs that work independent of scheduling

Race conditions are bugs that exist only due to concurrency
- No interleaved scheduling with 1 thread

Typically, problem is that some *intermediate state* can be seen by another thread; screws up other thread
- Consider a 'partial' insert in a linked list; say, a new node has been added to the end, but 'back' and 'count' haven't been updated

Data Races

- A ***data race*** is a specific type of ***race condition*** that can happen in 2 ways:
  - Two different threads can ***potentially*** write a variable at the same time
  - One thread can ***potentially*** write a variable while another reads the variable
  - Simultaneous reads are fine; not a data race, and nothing bad would happen
  - 'Potentially' is important; we say the code itself has a data race – it is independent of an actual execution
- Data races are bad, but we can still have a race condition, and bad behavior, when no data races are present

Readers/writer locks

0 ≤ writers ≤ 1 &&
0 ≤ readers &&
writers*readers==0

A new synchronization ADT: The readers/writer lock

- Idea: Allow any number of readers OR one writer
- This allows more concurrent access (multiple readers)
- A lock's states fall into three categories:
  - "not held"
  - "held for writing" by one thread
  - "held for reading" by *one or more* threads

- `new:` make a new lock, initially "not held"
- `acquire_write:` block if currently "held for reading" or "held for writing", else make "held for writing"
- `release_write:` make "not held"
- `acquire_read:` block if currently "held for writing", else make/keep "held for reading" and increment *readers count*
- `release_read:` decrement readers count, if 0, make "not held"

Deadlock

- As illustrated by the 'The Dining Philosophers' problem

•A deadlock occurs when there are threads **T1**, …, **Tn** such that:

•Each is waiting for a lock held by the next

•**Tn** is waiting for a resource held by **T1**

•In other words, there is a cycle of waiting

```
class BankAccount {
  …
  synchronized void withdraw(int amt) {…}
  synchronized void deposit(int amt) {…}
  synchronized void transferTo(int amt,BankAccount a){
    this.withdraw(amt);
    a.deposit(amt);
  }
}
```

Consider simultaneous transfers from account x to account y, and y to x