

Parallel Algorithms

Yihan Sun

Why parallelism? Everyone wants performance!



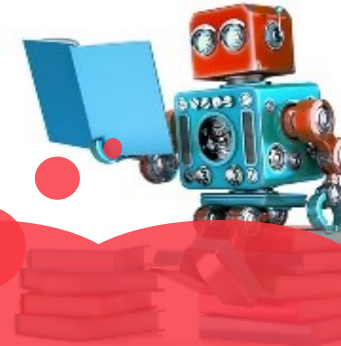
Database /
Data warehouses



Data mining /
Data science



Machine learning /
Artificial intelligence

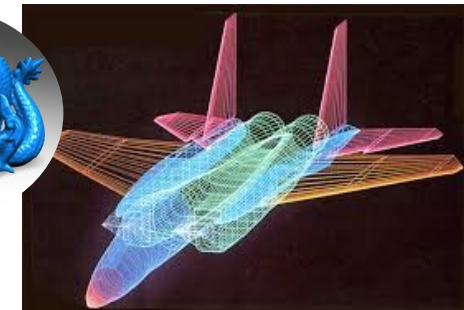


Get Faster!

Many, many
others



Computer graphics /
computational geometry

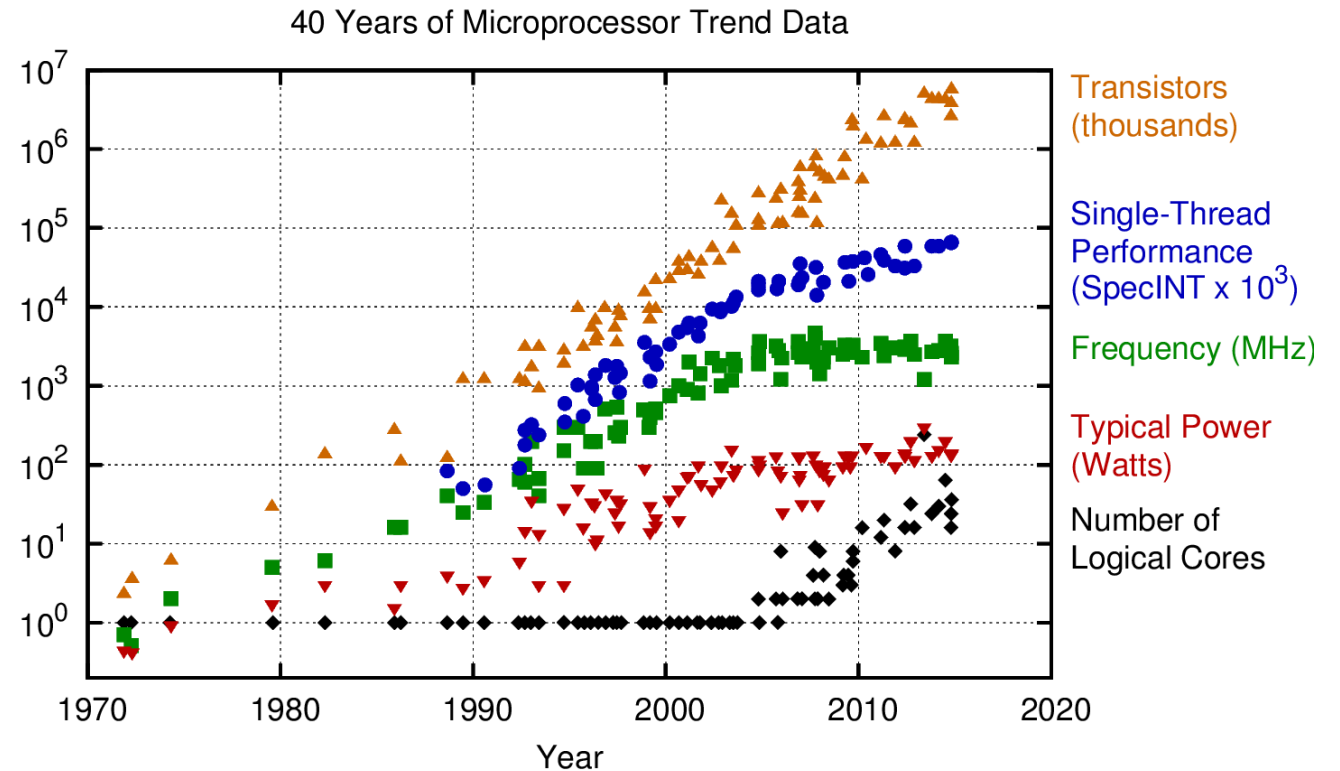


Computational
biology



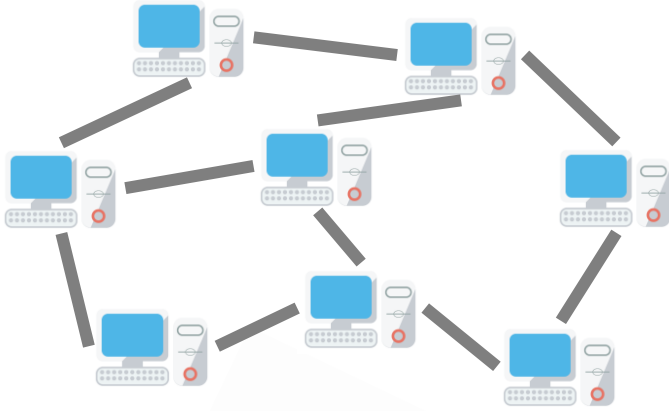
Get Faster!

- Nowadays it's very hard to further improve single core performance ...
- If you still want your code run faster, use **parallelism**!
- Now, your laptop, desktop, smart phone, ... are all multi-core!
- We need to learn how to design parallel algorithms and write parallel code!



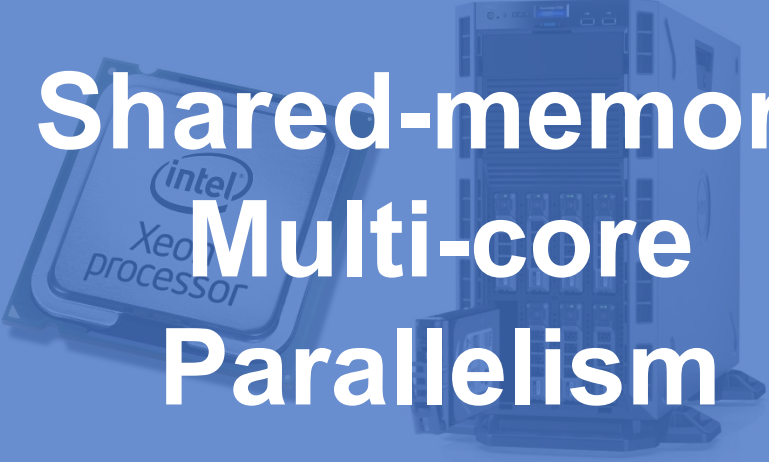
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Ways to Make Code Faster: Parallelism



Shared-memory
Multi-core
Parallelism

What you will learn in this lecture

A blue rounded rectangle containing a faint background image of an Intel Xeon processor and a server rack. Overlaid on this image is the text "Shared-memory Multi-core Parallelism" in white.

Shared-memory Multi-core Parallelism

Multiple processors **collaborate** to get a task done

Example: parallel sorting

- Consider sorting 10^9 64-bit keys
- How long will `std::sort` take using one core?
 - A. 0.1s
 - B. 1s
 - C. 5s
 - D. 10s
 - E. 50s
 - F. 100s
 - G. 500s

Example: parallel sorting

- Consider sorting 10^9 64-bit keys
- How long will a parallel sorting algorithm take on a machine with 96 cores?
 - A. 0.1s
 - B. 1s
 - C. 5s
 - D. 10s
 - E. 50s
 - F. 100s
 - G. 500s

Parallel machines



4 cores, 8 hyperthreading
Usually \$700-\$1500



- ❖ 96-cores, 192 hyper-threading
- ❖ 1.5TB of main memory
- ❖ Cost: about 30k USD, mostly due to memory

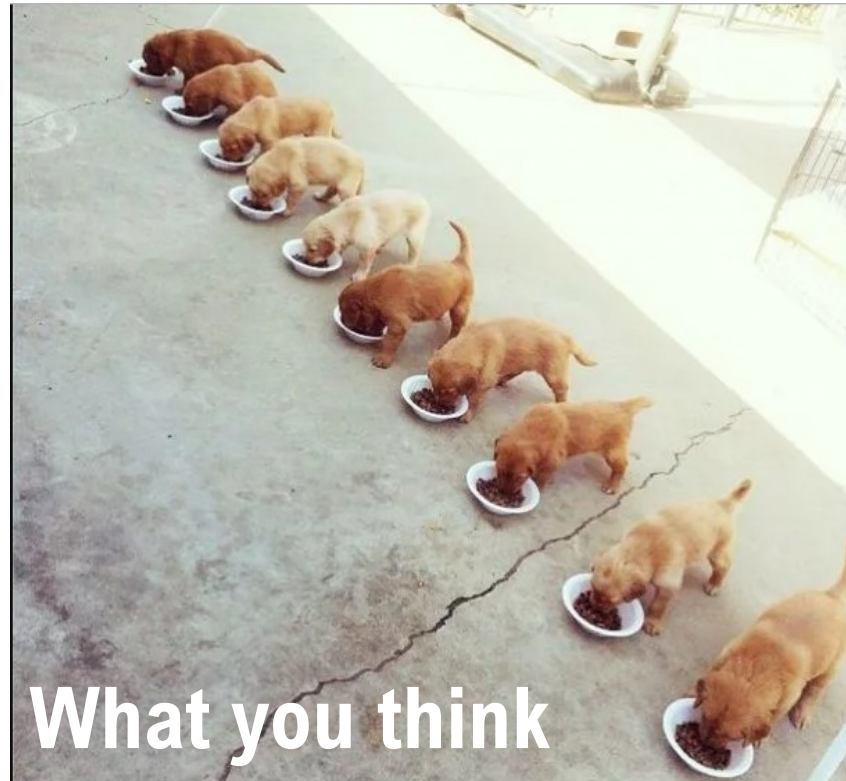


AWS: 448 hyper-threads and 24TB of memory

0.01 to ~6 dollars per hour

**We need to consider parallelism
in algorithm design!**

Multi-core Programming: What you think and what's in practice



(Pictures from 9gag.com)

Multi-core Programming

- **We need to learn theory:**
 - Making performance predictable
- **Not let this to happen →**



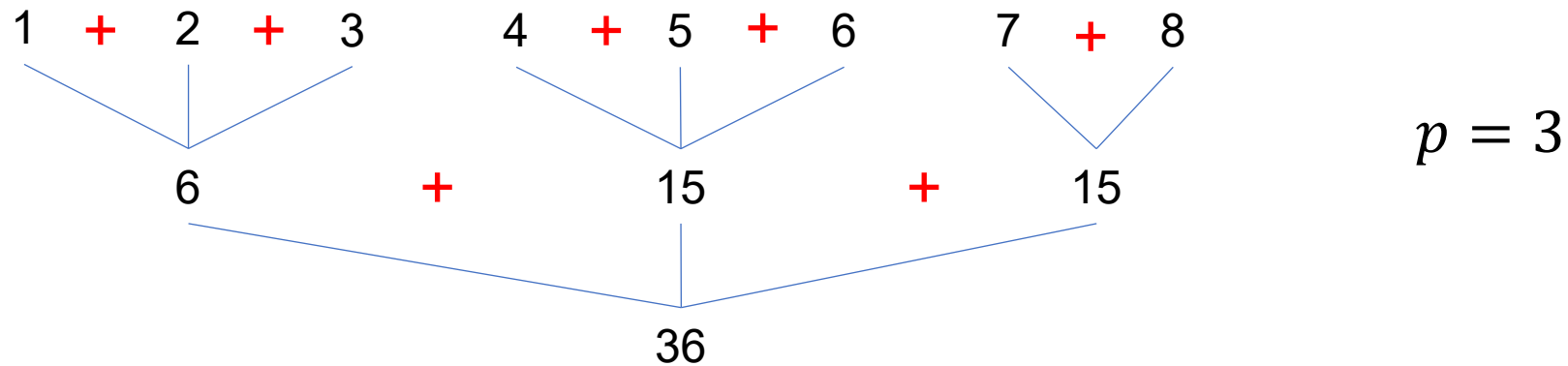
Parallel algorithms

- We'll see some fundamental knowledge about parallel algorithm design
- Implementation is not required in CS141
- See more details in CLRS Section 27
- Or take the course CS214 (parallel algorithms) / CS142 (algorithm engineering) in Winter :-D

Warm-up: reduce
Compute the sum of values in an array

Warm-up: the reduce algorithm

- Compute the sum of all values in an array (using p processors)



```
Sum(A, n) {  
  int B[p];  
  for processor i (i=0..p-1) {  
    for (j=i*n/p to i*n/p+n/p) B[i] += A[j];  
  }  
  sync all processors;  
  for (j = 0 to p) ret += B[i];  
  return ret; }
```

What if:

We do not know the number of available processors?

The number of processors is $O(n)$?

Problems

- **Do we know the number of processors p ahead of time?**
 - It can even change during the algorithm!
- **Dealing with system-level issues is error-prone**
 - makes parallel programming notoriously hard 😞
- **Locks and deadlocks?**
 - Parallelism \neq Using locks
 - Parallelism can be entirely lock-free!

Is there an easier way for parallel algorithm/programming?

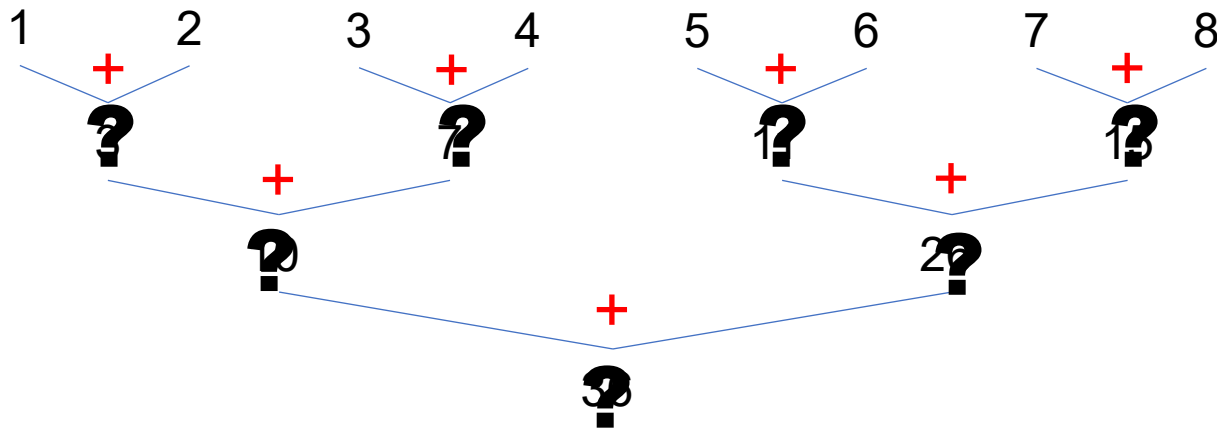
Dynamic Multi-threading (task-parallel) + Scheduler

Dynamic Multi-threading

- Let's not worry about which thread will do what, but only do it in a high level, a more abstract way
- The semantics we need is “let this task run in parallel”, or “let the following tasks run in parallel”
- We don't want to worry about more details! Is it possible?
- Greatly simplifies programming and theoretical analysis

Reduce using divide-and-conquer

- Compute the sum (reduce) of all values in an array



```
reduce(A, n) {  
    if (n == 1) return A[0];  
    In parallel:  
        L = reduce(A, n/2);  
        R = reduce(A + n/2, n-n/2);  
    return L+R;  
}
```

Binary fork-join model

- All computation start from a thread
- A thread can:
 - Do normal operations as in RAM model (arithmetic operations, memory access)
 - **Fork**: have two threads running in parallel, can be done in a **nested** manner
 - **Join**: synchronize with previous forked thread
 - Each fork has a corresponding join
- No concurrent write to the same memory location

```
reduce(A, n) {  
    if (n == 1) return A[0];  
    In parallel:  
        L = reduce(A, n/2);  
        R = reduce(A + n/2, n-n/2);  
    return L+R;  
}
```

```
reduce(A, n) {  
    if (n == 1) return A[0];  
    Fork:  
        L = reduce(A, n/2);  
        R = reduce(A + n/2, n-n/2);  
    Join  
    return L+R; }
```

Binary fork-join model

- Simple for theoretical analysis
- Simple for programming – almost exactly the code!
- Most of the state-of-the-art parallel languages support similar semantics and they know what to do with “fork” and “join”

```
reduce(A, n) {  
    if (n == 1) return A[0];  
    Fork:  
        L = reduce(A, n/2);  
        R = reduce(A + n/2, n-n/2);  
    Join  
    return L+R; }
```

```
1  #include <iostream>  
2  #include <cstdio>  
3  #include <stdlib.h>  
4  #include <cilk/cilk.h>  
5  #include <cilk/cilk_api.h>  
6  using namespace std;  
7  
8  int reduce(int* A, int n) {  
9      if (n == 1) return A[0];  
10     int L, R;  
11     L = cilk_spawn reduce(A, n/2);  
12     R = reduce(A+n/2, n-n/2);  
13     cilk_sync;  
14     return L+R;  
15 }  
16  
17 int main() {  
18     int n = atoi(argv[1]);  
19     int* A = new int[n];  
20     cilk_for (int i = 0; i < n; i++) A[i] = i;  
21     cout << reduce(A, n) << endl;  
22  
23     return 0;  
24 }
```

Who will execute the algorithm in the right way?

- **Scheduler!**
- **Your scheduler will figure out how to execute all parallel tasks**
 - For tasks that can be done in parallel, the scheduler will try to make them parallel as much as possible (if we have enough threads)
 - For tasks that cannot be done in parallel, they will be execute one after the other

```
reduce(A, n) {  
    if (n == 1) return A[0];  
    In parallel:  
        L = reduce(A, n/2);  
        R = reduce(A + n/2, n-n/2);  
    return L+R;  
}
```

How to evaluate the running time (time complexity) of a parallel algorithm

(without knowing how many processors can be used)

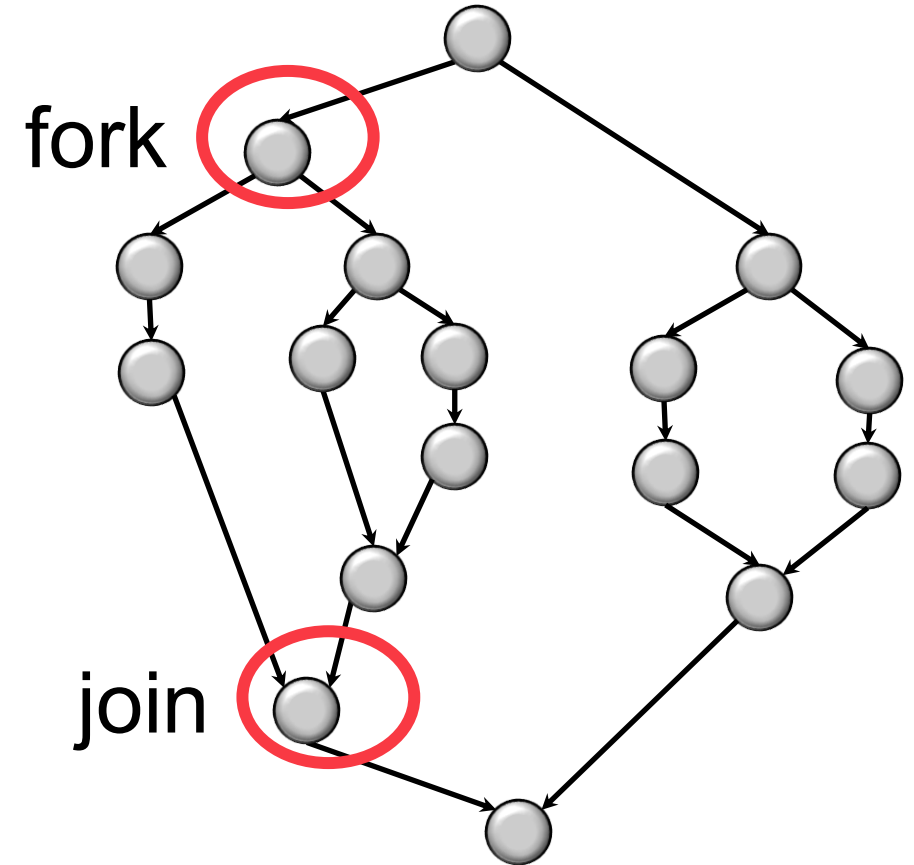
Binary fork-join model

- All computation start from a thread
- A thread can:
 - Do normal operations as in RAM model (arithmetic operations, memory access)
 - **Fork**: have two threads running in parallel, can be done in a **nested** manner
 - **Join**: synchronize with previous forked thread
 - Each fork has a corresponding join

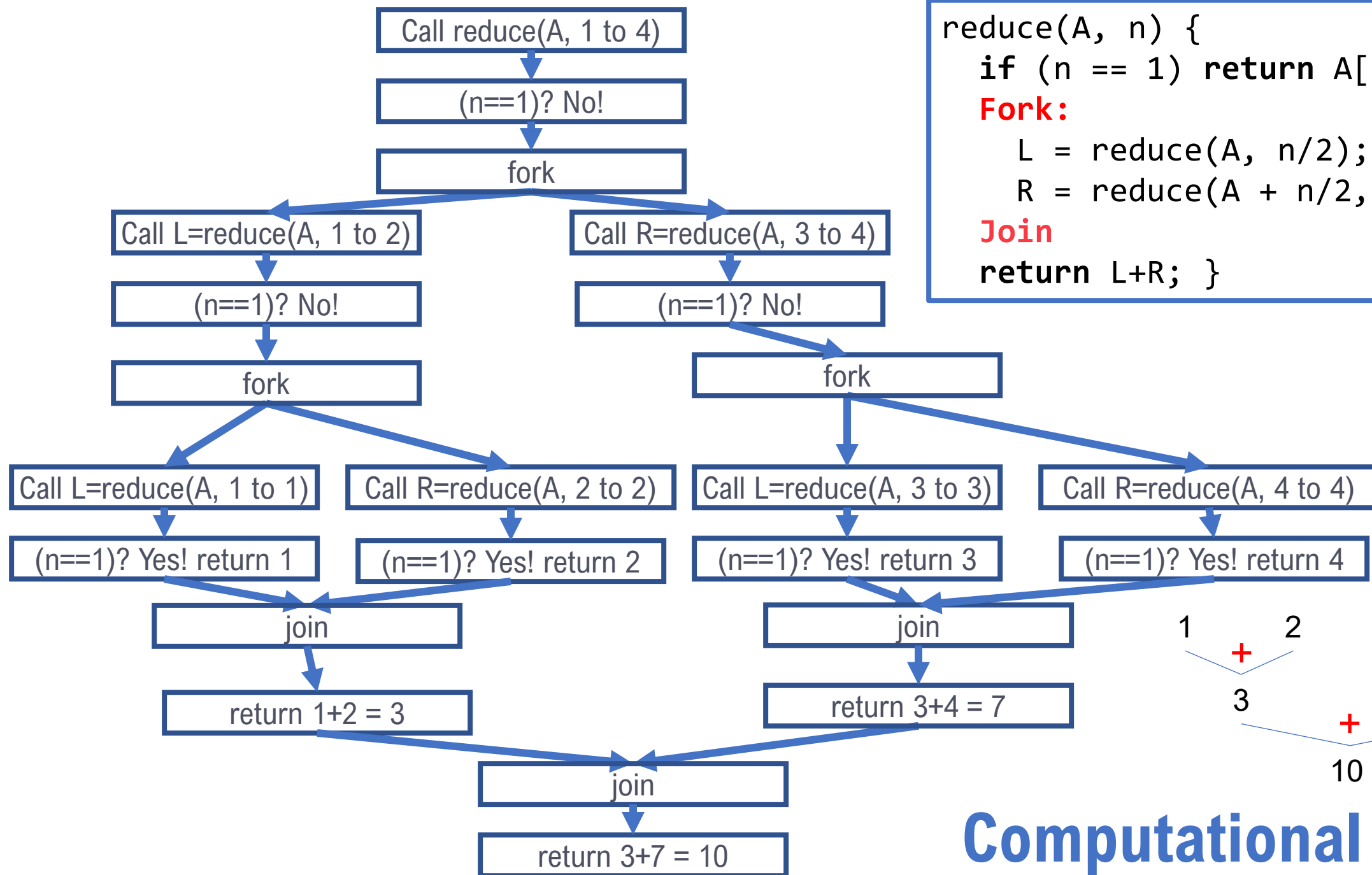
```
reduce(A, n) {  
    if (n == 1) return A[0];  
    Fork:  
        L = reduce(A, n/2);  
        R = reduce(A + n/2, n-n/2);  
    Join  
    return L+R; }
```

The “control flow” of a parallel algorithm

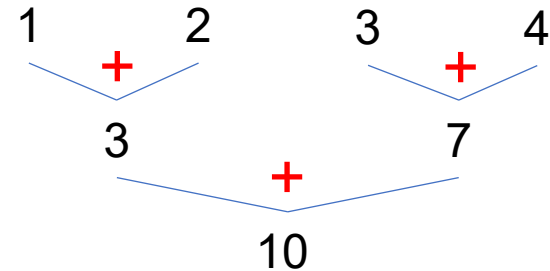
- It looks like a DAG (directed acyclic graph)
 - Each vertex is an operation/instruction
 - $A \rightarrow B$ means that B can be performed only when A has been finished



- It shows the dependency of operations in the algorithm



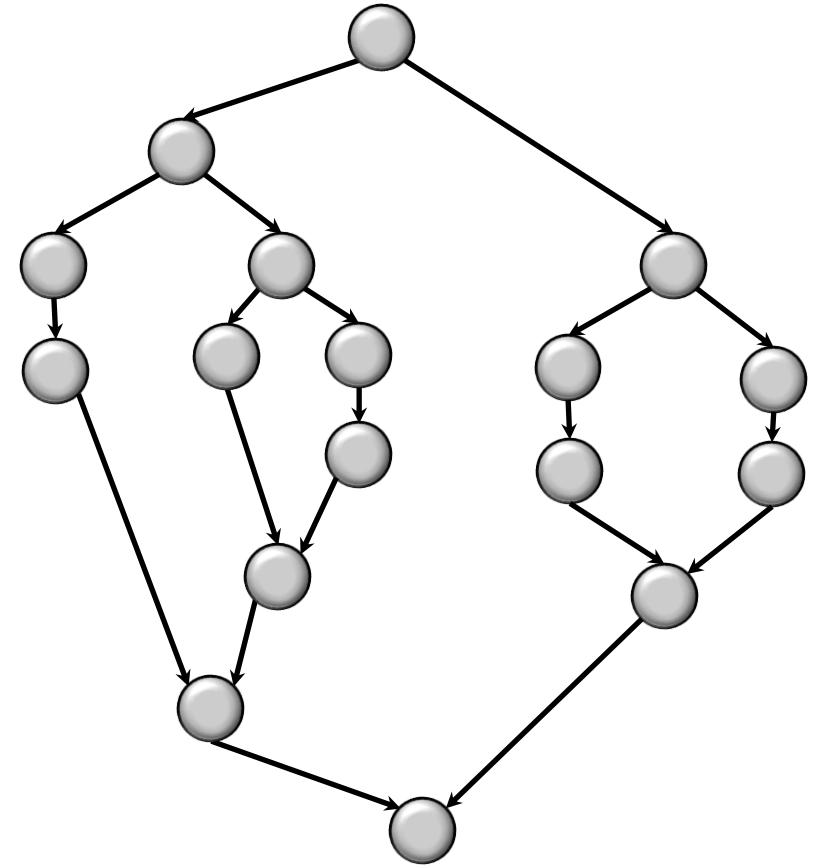
```
reduce(A, n) {  
  if (n == 1) return A[0];  
  Fork:  
    L = reduce(A, n/2);  
    R = reduce(A + n/2, n-n/2);  
  Join  
  return L+R; }
```



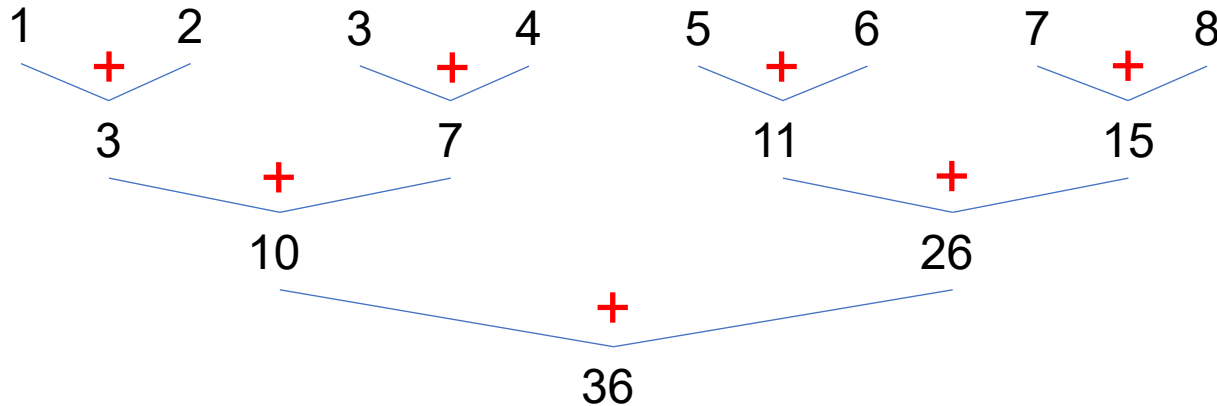
Computational DAG

Cost model: work-span

- **In the computation DAG**
 - Each vertex is an operation/instruction
 - $A \rightarrow B$ means that B can be performed only when A has been finished
- **Work:** The total number of operations in the algorithm
- **Span (depth):** The longest dependency chain



Cost model: work-span



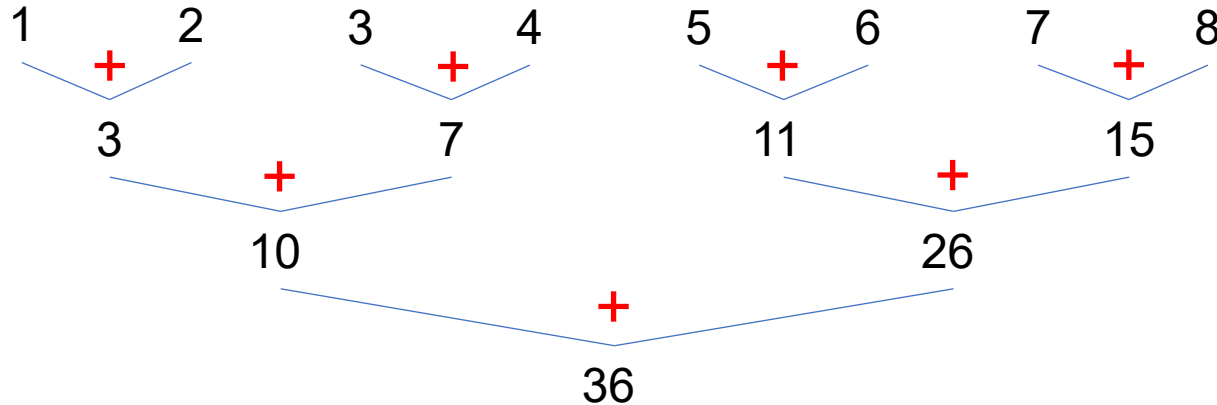
```
reduce(A, n) {  
    if (n == 1) return A[0];  
    In parallel:  
        L = reduce(A, n/2);  
        R = reduce(A + n/2, n-n/2);  
    return L+R;  
}
```

- **Work: The total number of operations in the algorithm**

Work: $O(n)$

- Sequential running time when the algorithm runs on **one processor**
- Work-efficiency: the work is (asymptotically) no more than the best (optimal) sequential algorithm
- Goal: make the parallel algorithm efficient when a small number of processor are available

Cost model: work-span



```
reduce(A, n) {  
    if (n == 1) return A[0];  
    In parallel:  
        L = reduce(A, n/2);  
        R = reduce(A + n/2, n-n/2);  
    return L+R;  
}
```

- **Span (depth): The longest dependency chain**

Span: $O(\log n)$

- Total time required if there are **infinite number of processors**
- Our goal is usually to make span polylogarithmic
- Goal: make the parallel algorithm faster and faster when more and more processors are available - scalability

Compute work and span

- When we see a fork-join:

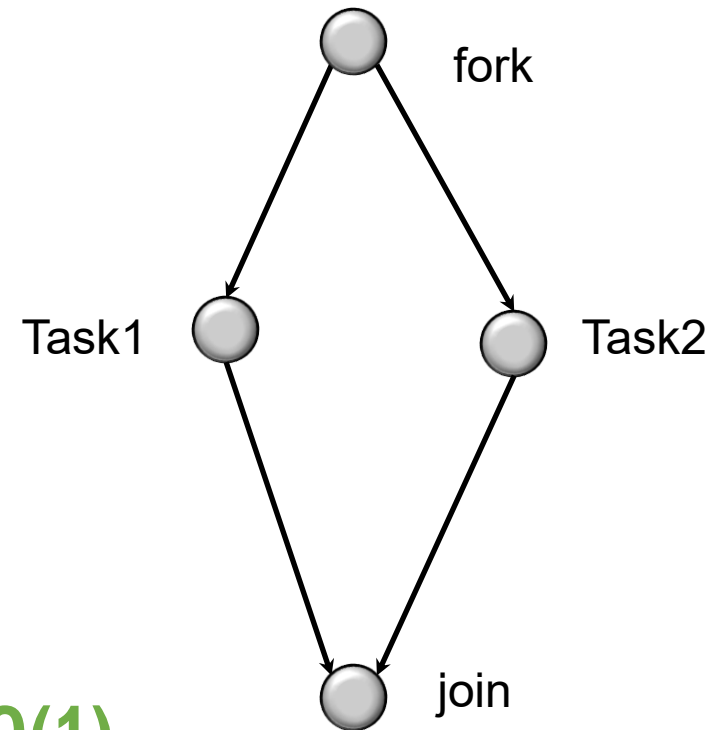
- **Fork**

- Task1
 - Task2

- **Join**

- Work = **work of Task1 + work of Task2 + $O(1)$**

- Span = **$\max(\text{span of Task1}, \text{span of Task2}) + O(1)$**

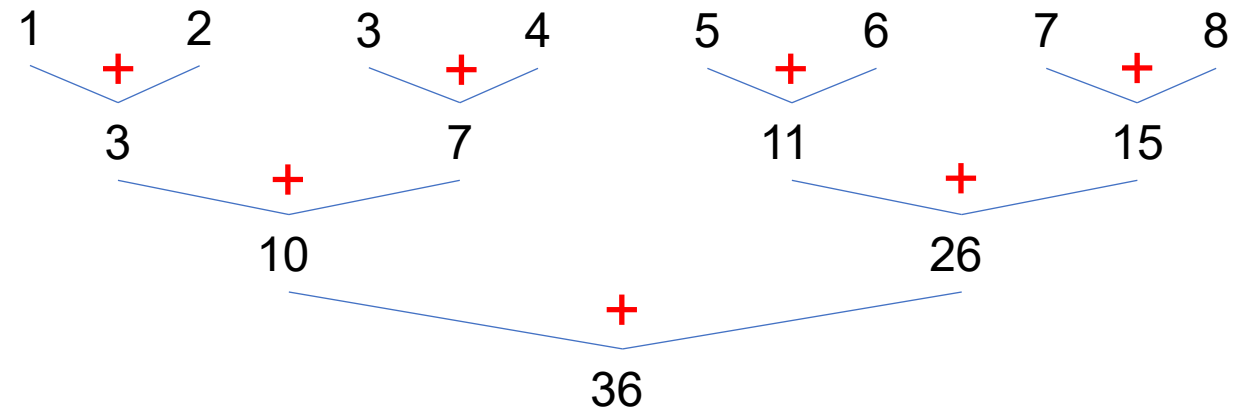


Compute work and span

- $W(n) = 2W\left(\frac{n}{2}\right) + \Theta(1)$
- $\Rightarrow W(n) = \Theta(n)$

```
reduce(A, n) {  
  if (n == 1) return A[0];  
  Fork:  
    L = reduce(A, n/2);  
    R = reduce(A + n/2, n-n/2);  
  Join  
  return L+R; }
```

- $S(n) = \Theta(1) + \max\left(S\left(\frac{n}{2}\right), S\left(\frac{n}{2}\right)\right)$
- $S(n) = \Theta(1) + S\left(\frac{n}{2}\right)$
- $\Rightarrow S(n) = \Theta(\log n)$



Parallel For Loop

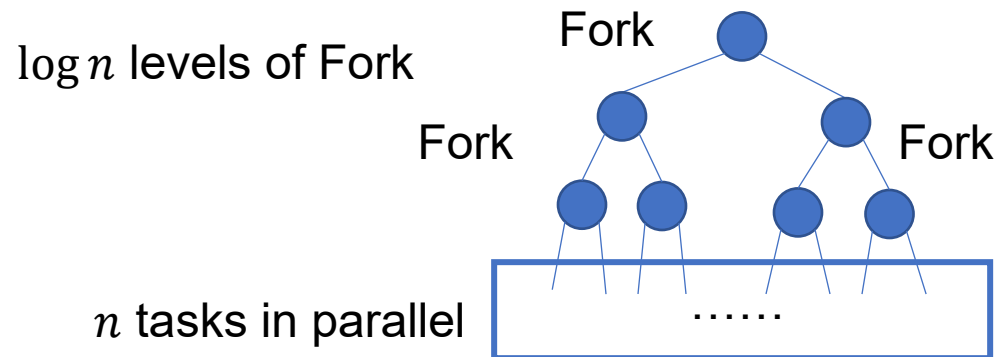
- Commonly used semantics in parallel programming

```
parallel_for i=1 to n {  
    A[i] = i;  
}
```

```
parallel_for i=1 to n {  
    s = s + A[i];  
}
```

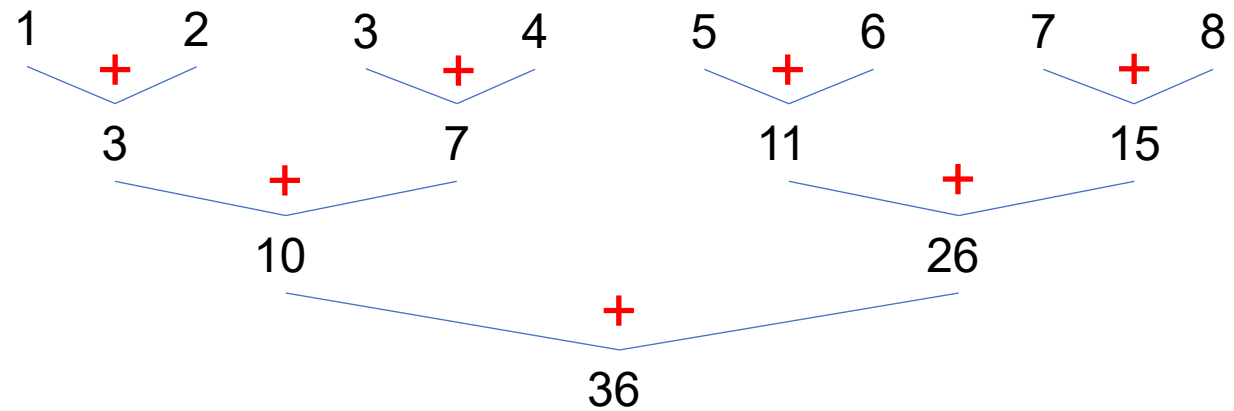


- How can we implement a parallel for-loop using fork-join?
- Incurs $O(n)$ work and $O(\log n)$ span



Another way to implement reduce

- $W(n) = \Theta(n) + W\left(\frac{n}{2}\right)$
- $\Rightarrow W(n) = \Theta(n)$



- $S(n) = \Theta(\log n) + S\left(\frac{n}{2}\right)$
- $\Rightarrow S(n) = \Theta(\log^2 n)$

```
reduce(A, n) {  
  if (n == 1) return A[0];  
  if (n is odd) n=n+1;  
  parallel_for i=1 to n/2  
    B[i]=A[2i]+A[2i+1];  
  return reduce(B, n/2); }
```

Needs $\log n$ rounds
to fork out tasks!

How do work and span relate to the real execution and running time?

Work, span and the actual running time

- An algorithm with work W and span S can be scheduled using p threads to finish in

$$O\left(\frac{W}{p} + S\right)$$

- time

Summary

- **Parallel algorithms**

- Some theoretical results/tools, help you reason your parallel code/performance

- **Dynamic multi-threading**

- Keep things simple – only focus on high-level parallelism and dependency
- The actual execution will be done by a scheduler

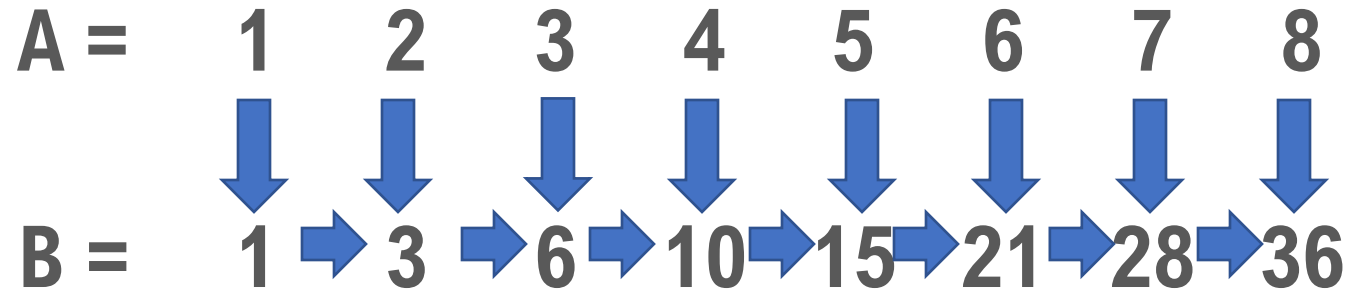
- **Fork-join**

- Fork: create a new thread working on a task in parallel
- Join: synchronous previously forked threads

- **Work-span model**

- A parallel algorithm/computation can be viewed as a DAG
- Work: the total number of operations. Running time using 1 processor
- Span (depth): the longest dependency chain. Running time using an unlimited number of processors

Next class



- Parallel algorithm for prefix sum
- Parallel algorithm for partition
- Parallel quicksort