Parallel Algorithms

Yan Gu

Course announcement

- Problem-solving training 2 is available (due date: next Wednesday)
- Performance-engineering homework 1 is due 11:59 PM today
- You can check your candy count by submitting an empty file on gradescope

CS142 | Winter 2021 Entry Code: I

DESCRIPTION

Edit your course description on the Course Settings page.

THINGS TO DO

Finish grading Programming-solving Training 1.

ACTIVE ASSIGNMENTS	RELEASED	DUE (PST)	SUBMISSIONS	% GRADED	PUBLISHED	REGRAI
Programming-solving Training 1	JAN 04	MAR 01 AT 12:00AM	16	93%		ON
Candies	JAN 16	FEB 08 AT 11:00PM	0	0%		ON
Performance-engineering Homework 1	JAN 10	JAN 20 AT 11:59PM	1	0%		ON

Course announcement

- **Grading for problem-solving training 1 is available**
    - However, I still have some confusions
    - 23 students have registered for the course
    - 18 have solved at least one problem
    - Only 16 submitted the report
    - 23 students on ilearn, banner, and igrade
    - 18 has reserved lab machines
    - 25 students on piazza and gradescope

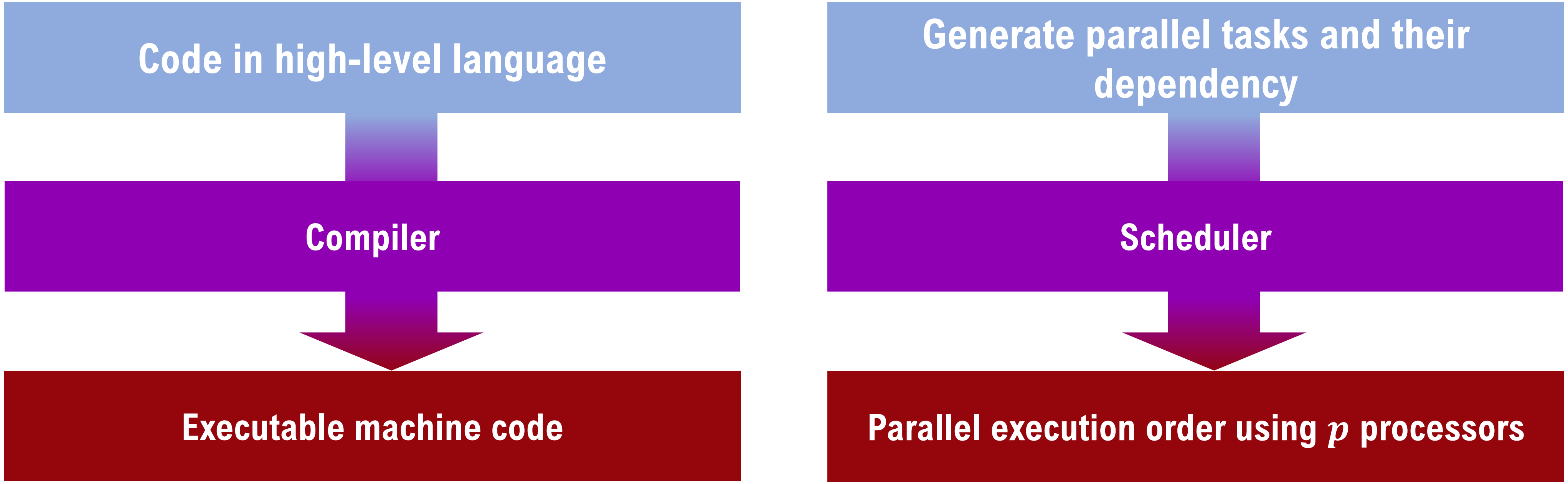
- **For those who solved problems after our first due, please resubmit an updated report and denote which are solved after the first due**

Parallel Algorithms

Yan Gu

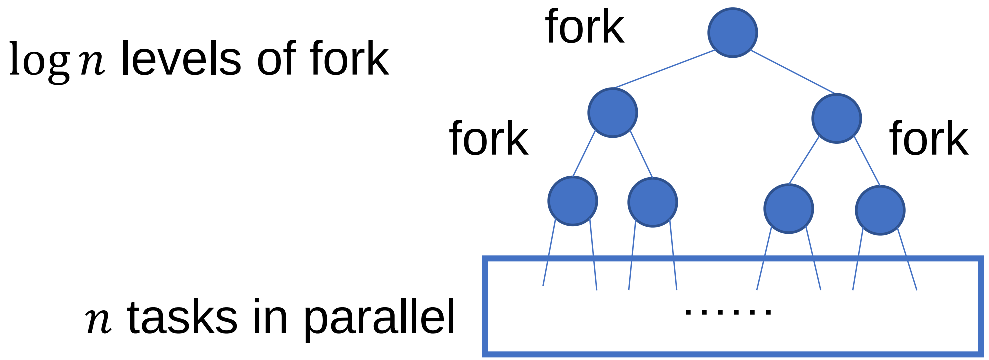
Scheduler

- Consider it as a complier. Programmers then only need to focus on high-level algorithm design



- We always assume an effective scheduler
- We design algorithms only focusing on generating parallel tasks

Binary Fork-Join Model



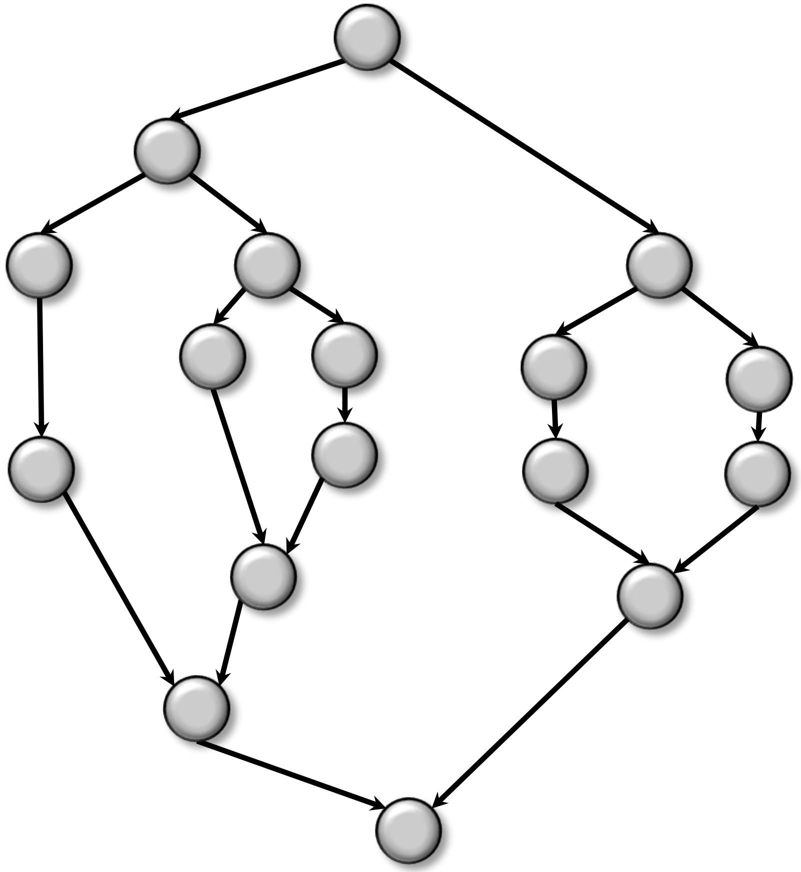
- **You write the code exactly the same as the sequential code, except that**
  - The "in parallel" instruction: fork two tasks (functions) and they can be run in parallel (but not necessarily run in parallel)
  - The "parallel for" instruction: all iterations in this for loop can be run in parallel

```
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;
}
```

```
copy(A, B, n) {
    parallel for (i=0; i<n; i++)
        B[i] = A[i];
}
```

Cost model: work-span

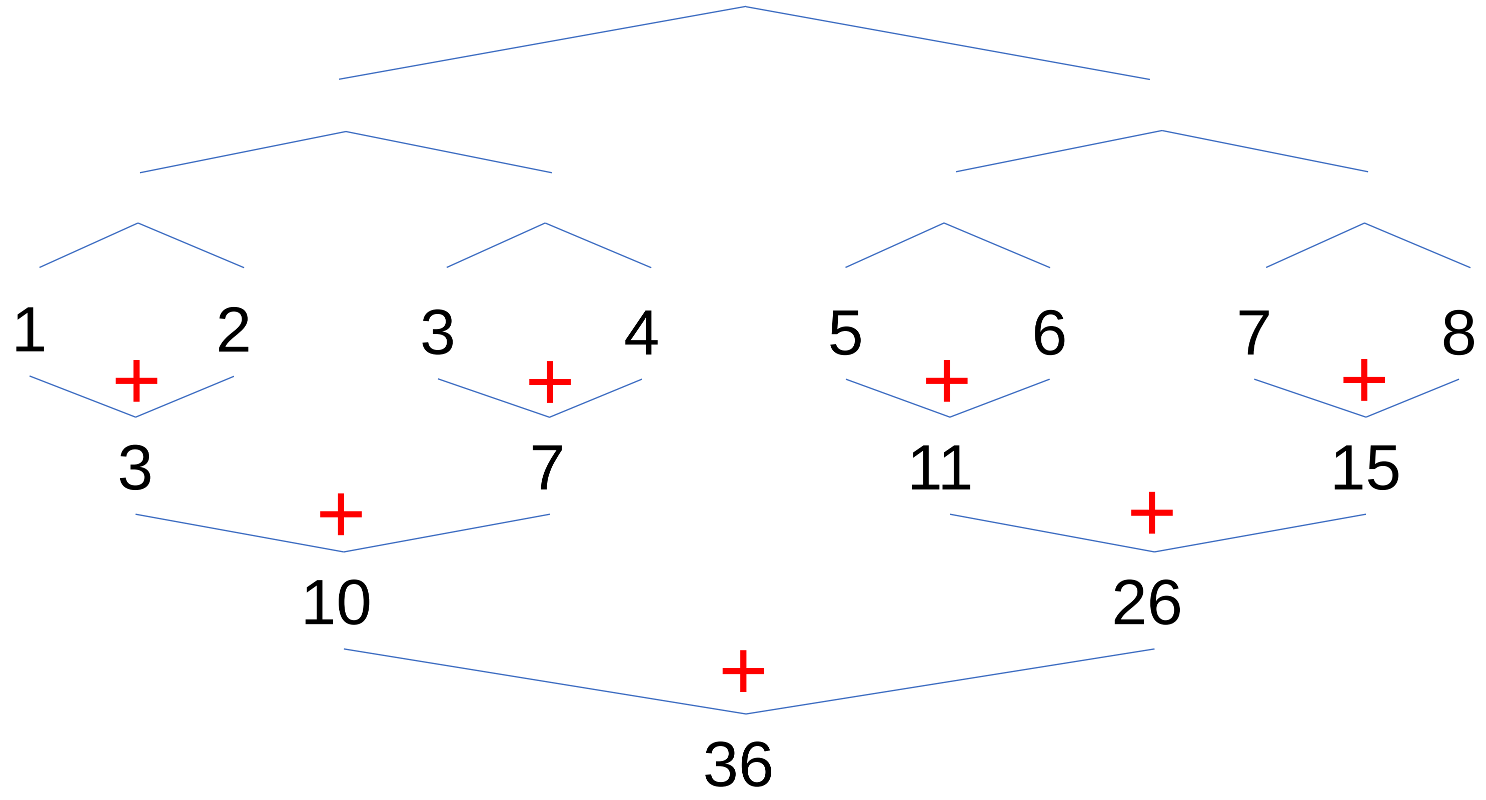
- **For all computations, draw a DAG**
  - A->B means that B can be performed only when A has been finished
- **Work: the total number of operations**
- **Span (depth): the longest length of chain**



- **It shows the dependency of operations in the algorithm**

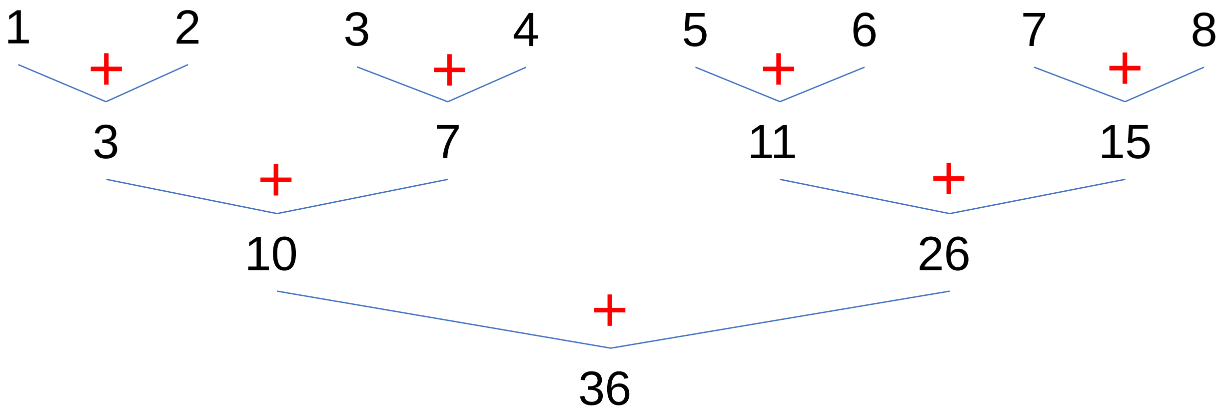
Computational DAG

```
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;
}
```



Cost model: work-span

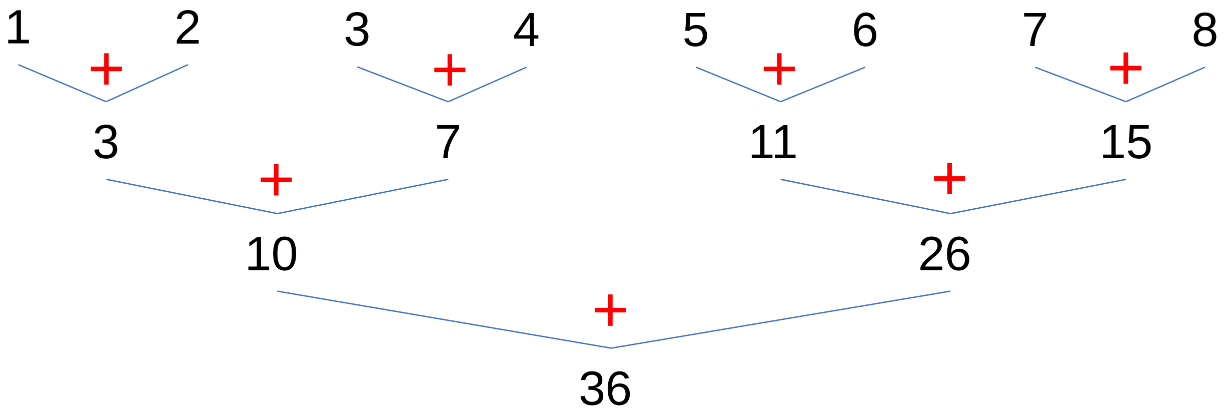
```
reduce(A, n) {
    if (n == 1) return A[0];
    In parallel:
        L = reduce(A, n/2);
        R = reduce(A + n/2, n-n/2);
    return L+R;
}
```



Work: $O(n)$

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



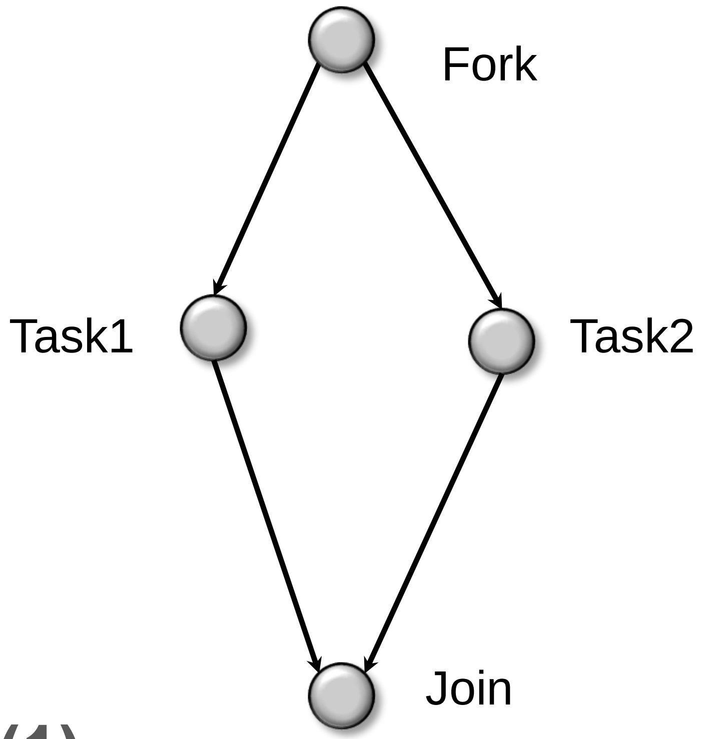

```
reduce(A, n) {
    if (n == 1) return A[0];
    In parallel:
        L = reduce(A, n/2);
        R = reduce(A + n/2, n-n/2);
    return L+R;
}
```

Span: $O(\log n)$

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

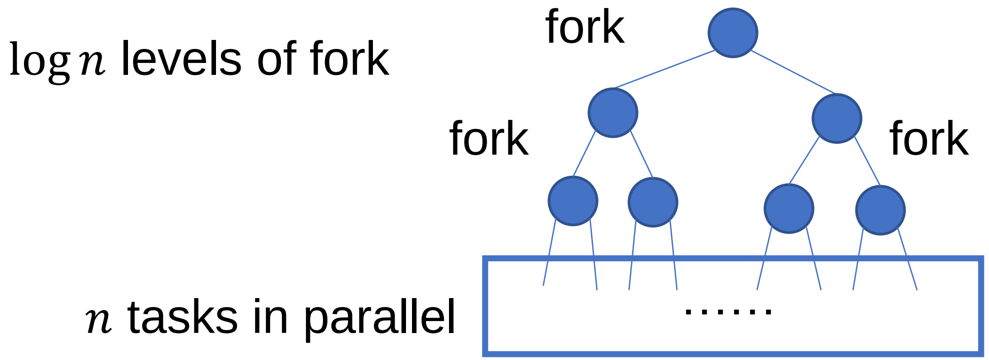
Compute work and span

- When we see a in-parallel (fork-join, spawn-sync):
    - in-parallel
    - Task1
    - Task2
- Work = work of Task1 + work of Task2+O(1)
- Span = max(span of Task1, span of Task2)+O(1)



Programming fork-join parallelism

Binary Fork-Join Model



- **You write the code exactly the same as the sequential code, except that**
  - The "in parallel" instruction: fork two tasks (functions) and they can be run in parallel (but not necessarily run in parallel)
  - The "parallel for" instruction: all iterations in this for loop can be run in parallel

```
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;
}
```

```
copy(A, B, n) {
    parallel for (i=0; i<n; i++)
        B[i] = A[i];
}
```

Fork-join parallelism

As long as you can design a parallel algorithm in fork-join, implementing them requires very little work on top of your sequential C++ code

- **Supported by many programming languages**

- **Cilk/cilk+ (silk – thread)**
  - Based on C++
  - Execute two tasks in parallel
    - do_thing_1 can be done in parallel in another thread
    - do_thing_2 will be done by the current thread
  - Parallel for-loop: execute n tasks in parallel
    - For cilk, it first forks two tasks, then four, then eight, … in O(log n) rounds

```
#include <cilk/cilk.h>
#include <cilk/cilk_api.h>
```

Fork

Join

```
cilk_spawn do_thing_1;
do_thing_2;
cilk_sync;
```

```
cilk_for (int i = 0; i < n; i++) {
  do_something;
}
```

Fork-join parallelism

- A lightweighted library: PBBS (Problem-based benchmark suite)
- Code available at: https://github.com/cmuparlay/pbbslib

```
#include “pbbslib/utilities.h”
```

You can also use cilk or openmp to compile your code

```
par_do([&] () {do_thing_1;},
       [&] () {do_thing_2;});
```

lambda expression
(must be function calls)

```
parallel_for (0, 100, [&] (int i) {Do_something});
```

Implementing parallel reduce in cilk

Pseudocode

```
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;
}
```

Code using Cilk

```
int reduce(int* A, int n) {
    if (n == 1) return A[0];
    int L, R;
    L = cilk_spawn reduce(A, n/2);
        R = reduce(A+n/2, n-n/2);
    cilk_sync;
    return L+R; }
```

It is still valid is running sequentially,
i.e., by one processor

Implementing parallel reduce in PBBS

```
#include “pbbslib/utilities.h”
```

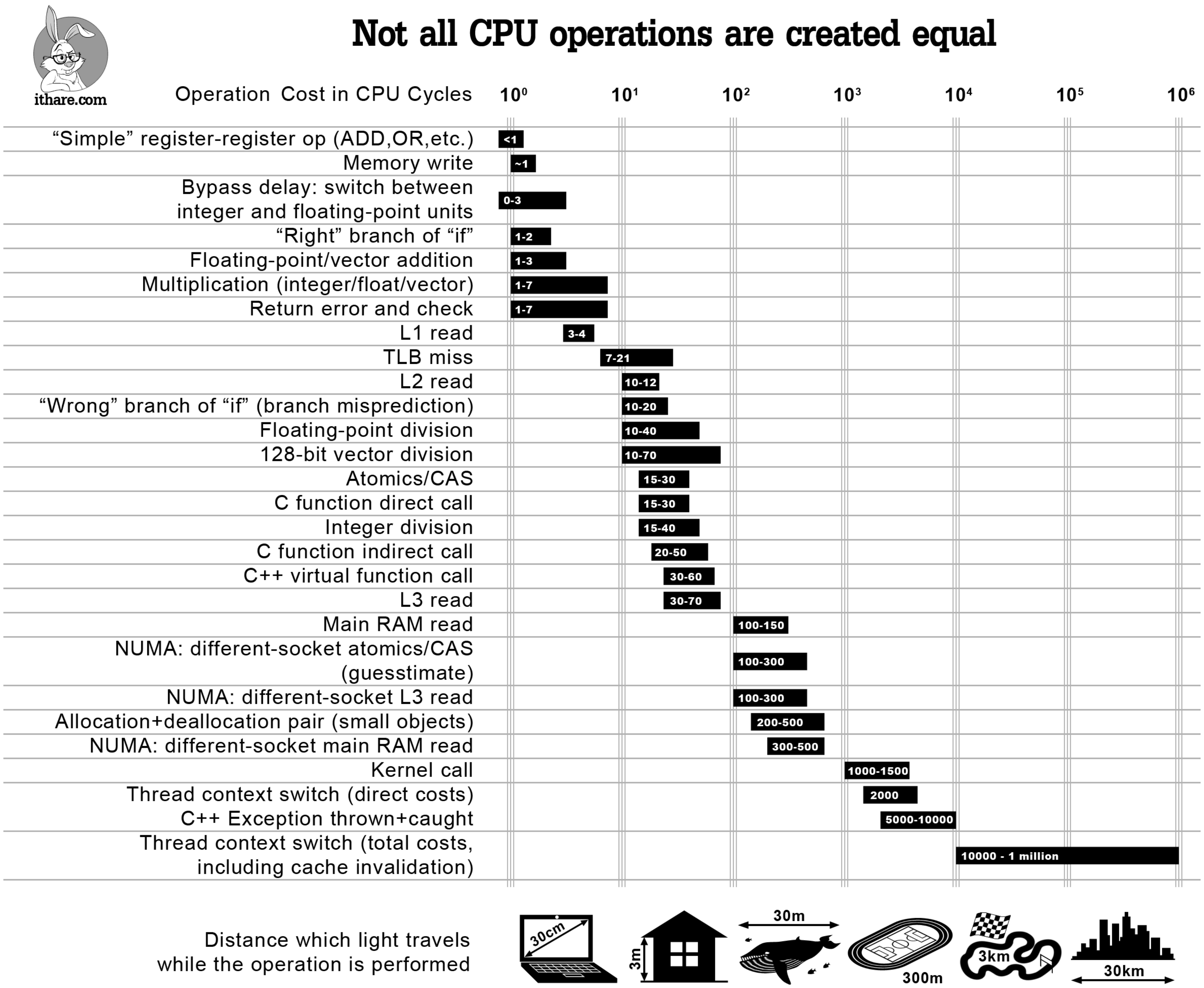
You can also use cilk or openmp to compile your code

```
void reduce(int* A, int n, int& ret) {
  if (n == 1) ret = A[0]; else {
    int L, R;
    par_do([&] () {reduce(A, n/2, L);},
           [&] () {reduce(A+n/2, n-n/2, R);});
    ret = L+R;
  }
}
```

lambda expression
(must be function calls)

```
parallel_for (0, 100, [&] (int i) {A[i] = i;});
```

Implementation trick 1: coarsening



- **A cilk-spawn is about 100 cycles**

Image from ithare.com: http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/

Coarsening

- Forks and Joins are costly – they are the overhead of using parallelism
- If each task is too small, the overhead will be significant
- Solution: let each parallel task get enough work to do!

```
int reduce(int* A, int n) {
    if (n == 1) return A[0];
    int L, R;
    L = cilk_spawn reduce(A, n/2);
    R = reduce(A+n/2, n-n/2);
    cilk_sync;
    return L+R; }
```

```
int reduce(int* A, int n) {
    if (n < threshold) {
      int ans = 0;
      for (int i = 0; i < n; i++)
        ans += A[i];
      return ans; }
    int L, R;
    L = cilk_spawn reduce(A, n/2);
    R = reduce(A+n/2, n-n/2);
    cilk_sync;
    return L+R; }
```

Computational Model

Time complexity

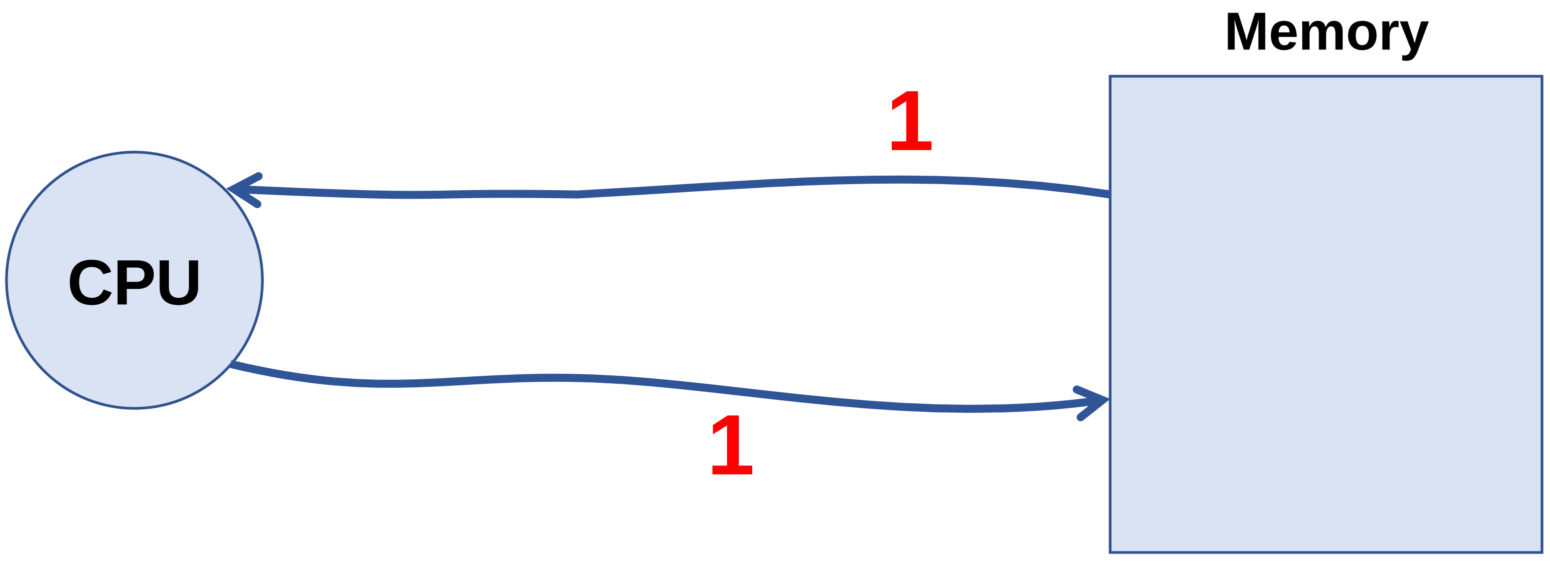
- **Mergesort, quicksort:** $O(n \log n)$
- **Insertion sort, bubble sort:** $O(n^2)$
- But what is time complexity?

What is an algorithm?

- An algorithm (/ˈælɡərɪðəm/) is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation (from Wikipedia)

Random-Access Machine (RAM)

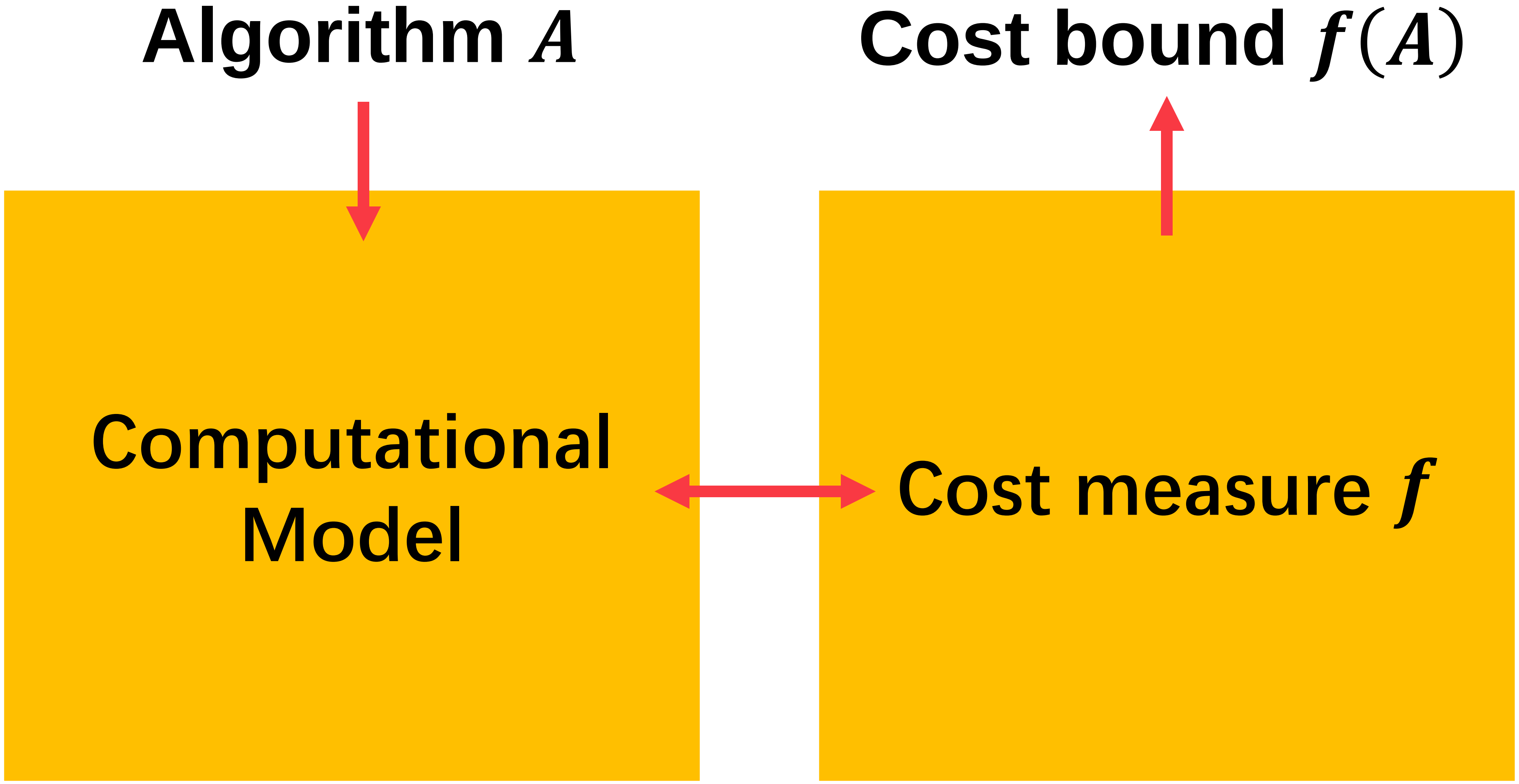
- **Unit cost for:**
  - Any instruction on $\Theta(\log n)$-bit words
  - Read/write a single memory location from an infinite memory
- **The cost measure: time complexity**



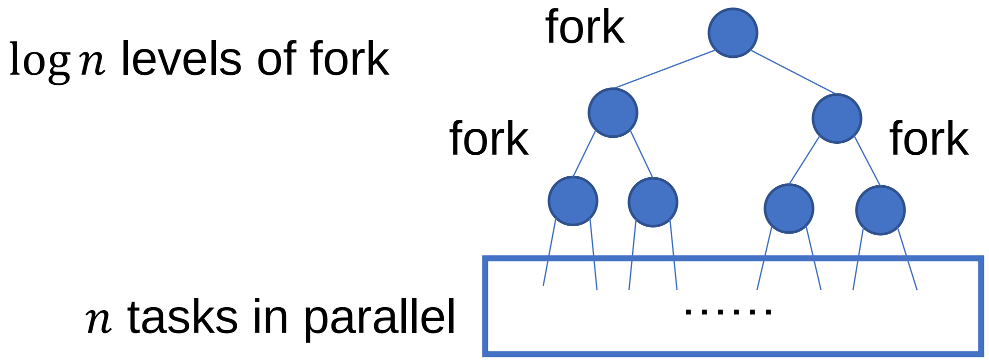
The equivalent programming model

```
INSERTION-SORT(A, n)
  for j = 2 to n
      key = A[j]
      // Insert A[j] into the sorted sequence A[1 .. j − 1].
      i = j − 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

cost	times
c_1	n
c_2	$n-1$
0	-
c_4	$n-1$
c_5	$\sum_{j=2}^{n}(t_j+1)$
c_6	$\sum_{j=2}^{n} t_j$
c_7	$\sum_{j=2}^{n} t_j$
c_8	$n-1$


Algorithm A
Cost bound $f(A)$
Computational Model
Cost measure f

Binary Fork-Join Model



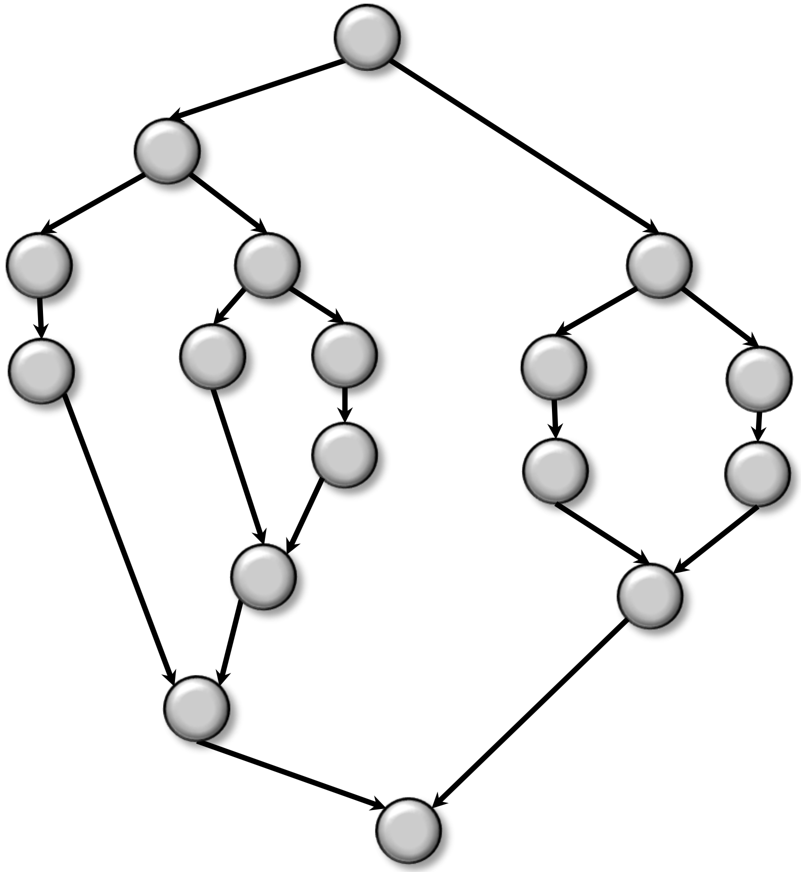
- **In addition to RAM instructions, you can also use**
  - The "in parallel" instruction: fork two tasks (functions) and they can be run in parallel (but not necessarily run in parallel)
  - The "parallel for" instruction: all iterations in this for loop can be run in parallel

```
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;
}
```

```
copy(A, B, n) {
    parallel for (i=0; i<n; i++)
        B[i] = A[i];
}
```

Cost model: work-span

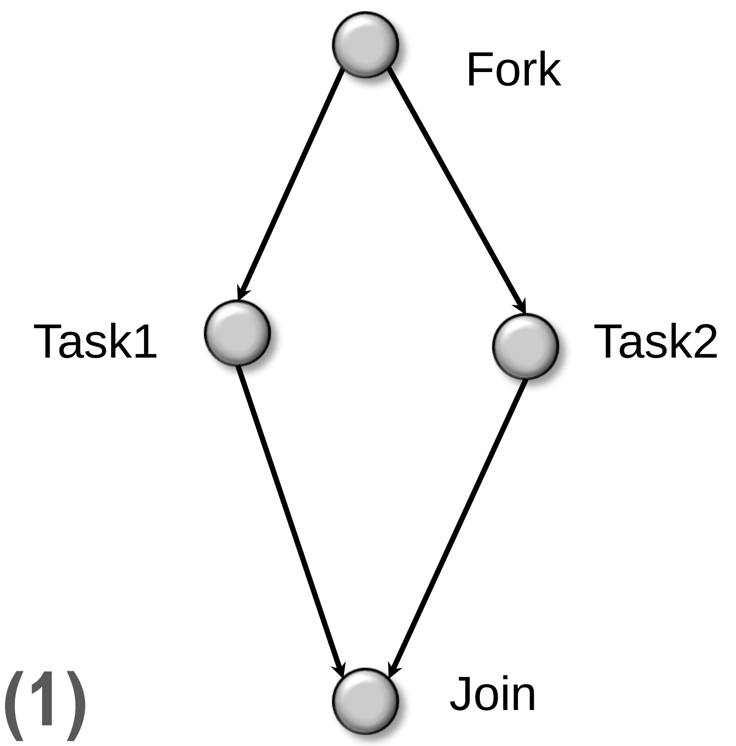
- **For all computations, draw a DAG**
  - A->B means that B can be performed only when A has been finished
- **Work: the total number of operations**
- **Span (depth): the longest length of chain**



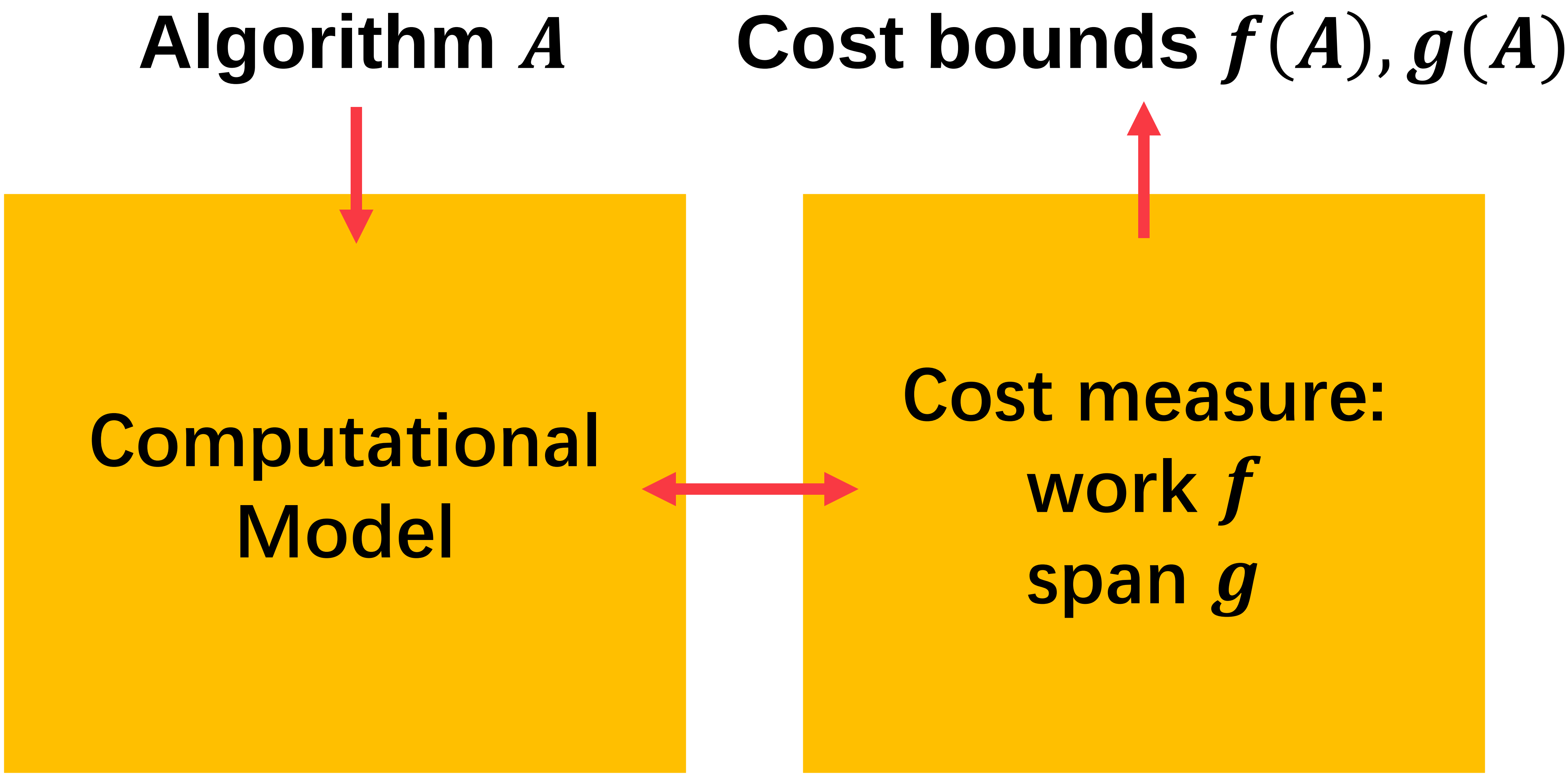
- **It shows the dependency of operations in the algorithm**

Compute work and span

- When we see a in-parallel (fork-join, spawn-sync):
  - in-parallel
  - Task1
  - Task2
- Work = work of Task1 + work of Task2+O(1)
- Span = max(span of Task1, span of Task2)+O(1)



- When you see a serial code:
- Task1
- Task2
- Work = work of Task1 + work of Task2
- Span = work of Task1 + work of Task2

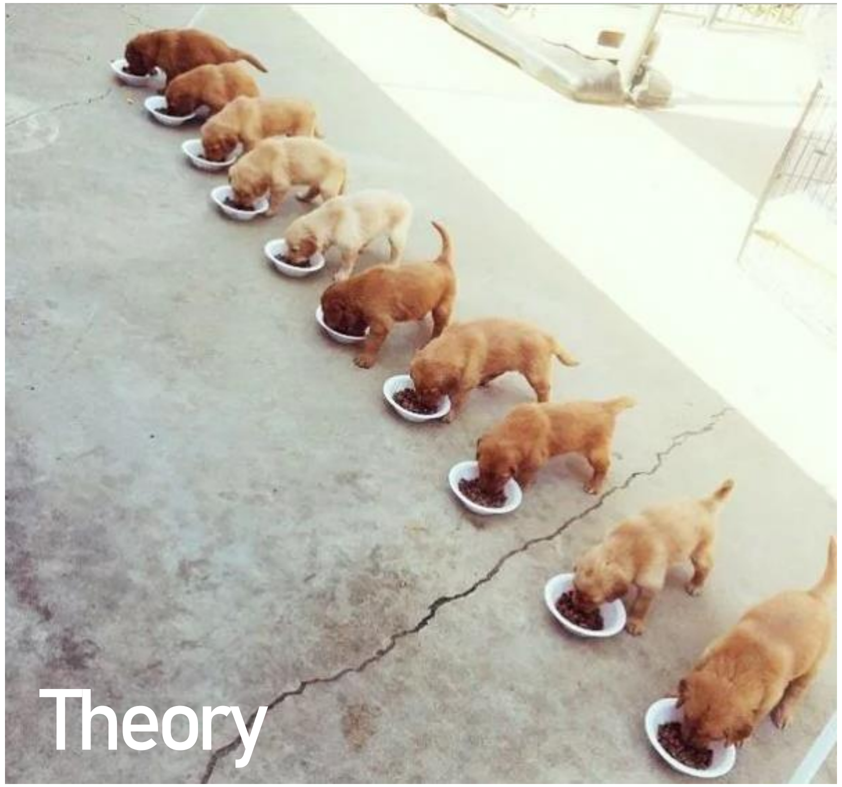

Algorithm A
Cost bounds $f(A), g(A)$
Computational Model
Cost measure:
work f
span g

Race

Some materials are from 6.172 Performance Engineering of Software Systems, credits to Charles Leiserson

Why is parallelism "hard"?

Non-determinism!!



Why is parallelism “hard”?

Non-determinism!!

- **Scheduling is unknown**
- **Relative ordering for operations is unknown**

- **Hard to debug**
  - Bugs can be **non-deterministic!**
  - Bugs can be different if you rerun the code
  - Referred to as race hazard / condition

Race hazard can cause severe consequences

- Therac-25 radiation therapy machine — killed 3 people and seriously injured many more (between 1985 and 1987). https://en.wikipedia.org/wiki/Therac-25

- North American Blackout of 2003 — left 50 million people without power for up to a week. https://en.wikipedia.org/wiki/Northeast_blackout_of_2003

- Race bugs are notoriously difficult to discover by conventional testing!

Determinacy Races

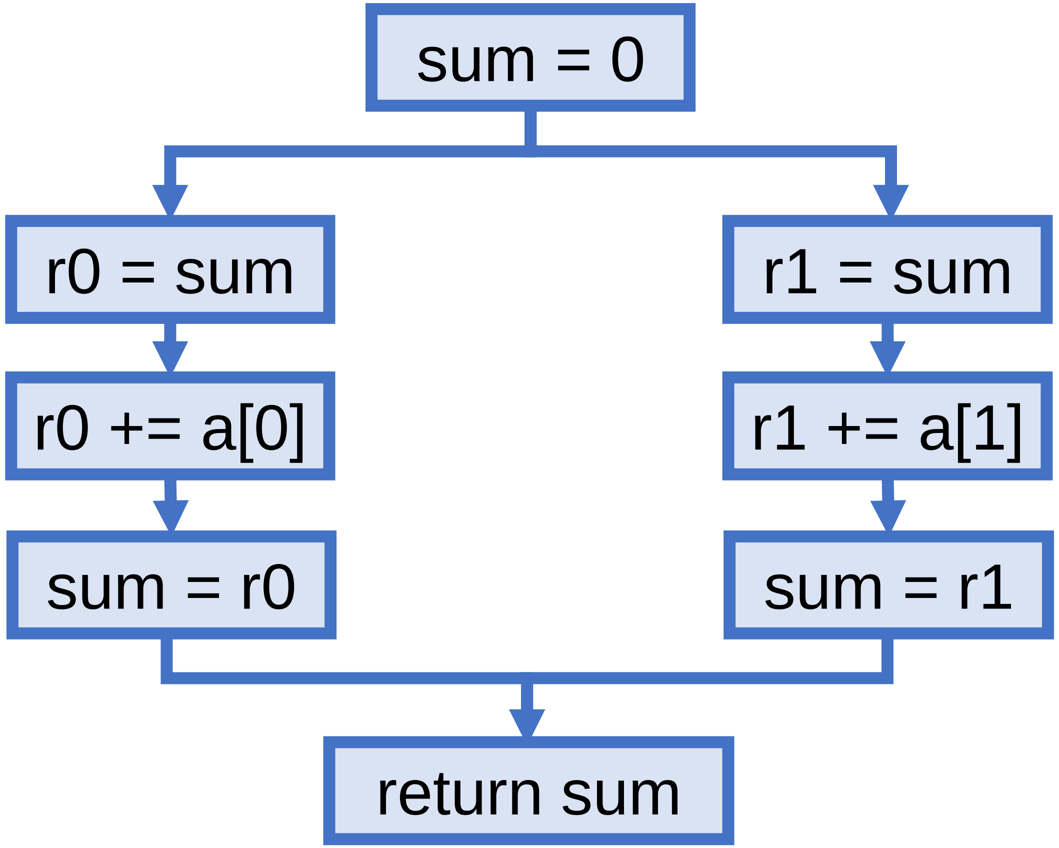
- **Definition: a determinacy race occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.**

```
direct_reduce(A, n) {
  parallel_for (i=0;i<n;i++)
    sum = sum + a[i];
  return sum;
}
```

Determinacy Races

- Definition: a **determinacy race** occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

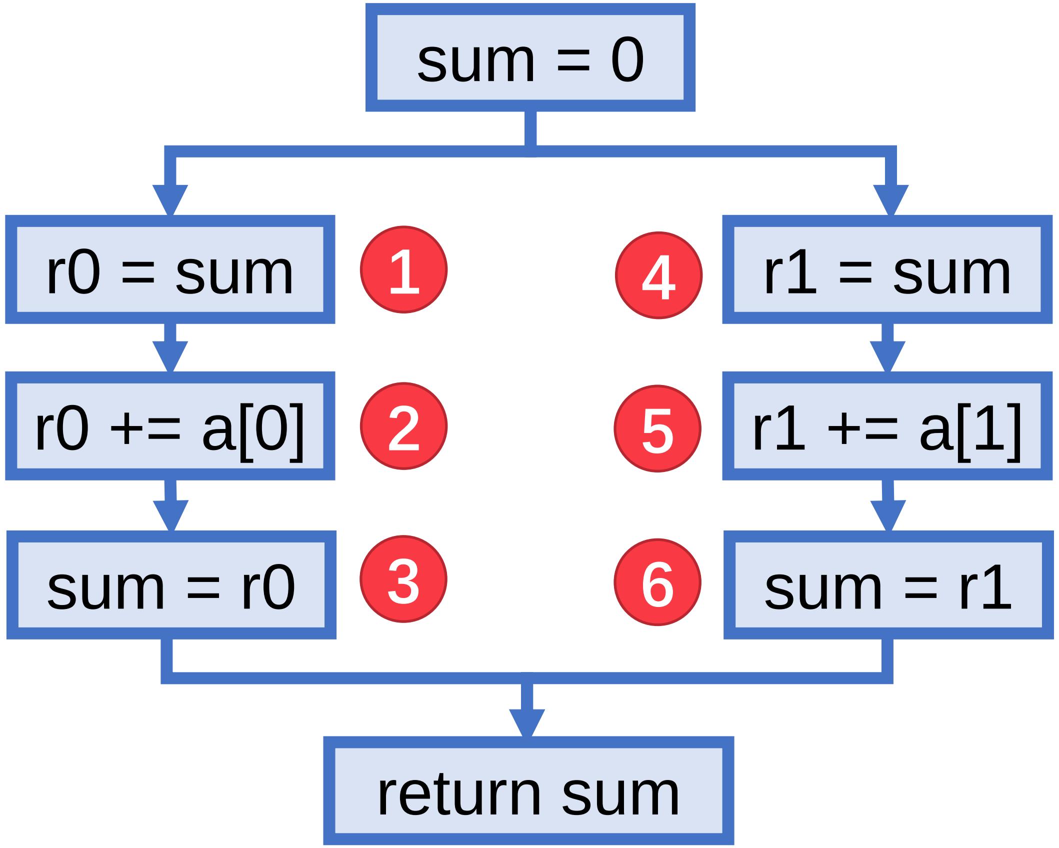
```
direct_reduce(A, n) {
  parallel_for (i=0;i<2;i++)
    sum = sum + a[i];
  return sum;
}
```



Determinacy Races

- Definition: a **determinacy race** occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

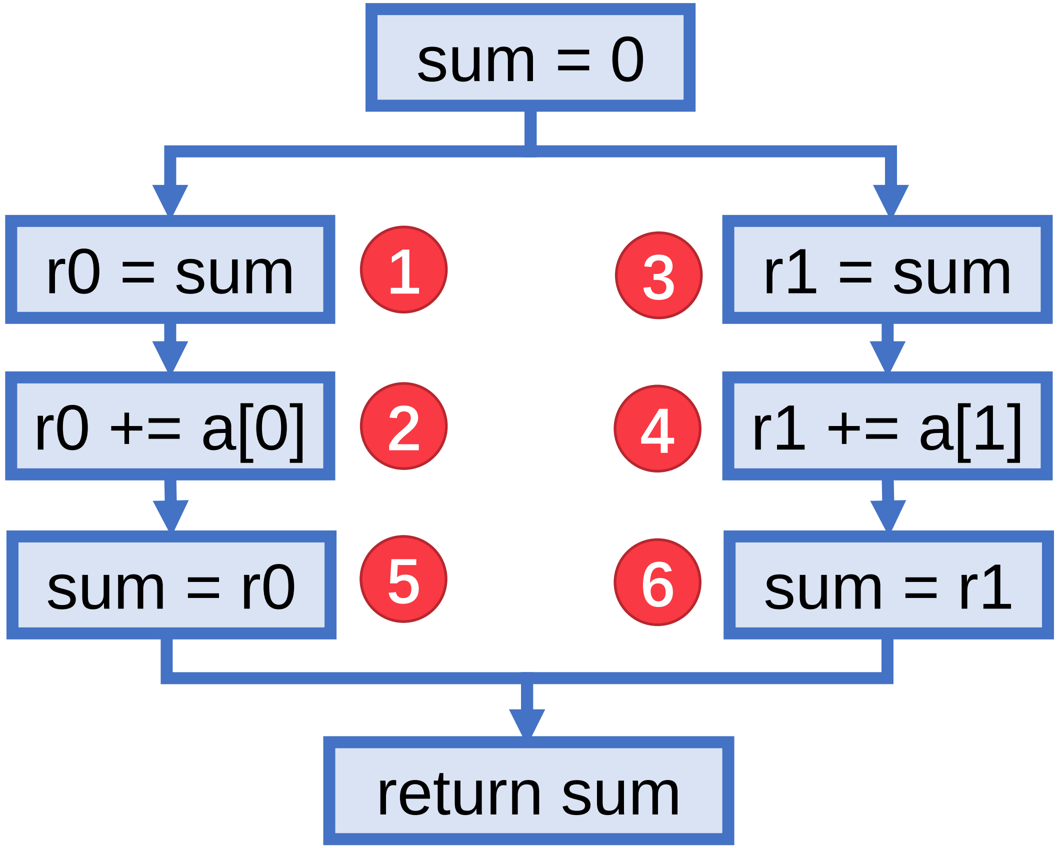
```
direct_reduce(A, n) {
  parallel_for (i=0;i<2;i++)
    sum = sum + a[i];
  return sum;
}
```



Determinacy Races

- Definition: a **determinacy race** occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

```
direct_reduce(A, n) {
  parallel_for (i=0;i<2;i++)
    sum = sum + a[i];
  return sum;
}
```



Types of Races

- Suppose that instruction A and instruction B both access a location x, and suppose that A||B (A is parallel to B).

A	B	Race Type
Read	Read	No race
Read	Write	Read race
Write	Read	Read race
Write	Write	Write race

- Two sections of code are **independent** if they have no determinacy races between them.

Avoiding races

- Iterations of a **parallel_for** loop should be independent
- Between two **in_parallel** tasks, the code of the two calls should be independent, including code executed by further in_parallel tasks

```
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;
}
```

Avoiding races

- Iterations of a **parallel_for** loop should be independent
- Between two **in_parallel** tasks, the code of the two calls should be independent, including code executed by further in_parallel tasks

```
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);
}
```

Benefit of being race-free

- **Scheduling is still unknown**
- **Relative ordering for operations is still unknown**

- **However, the computed value of each instruction is deterministic!**
  - Check the correctness of the sequential execution
  - Check if the parallel execution is the same as the sequential one

This is not the end…

- **Consider a hash table**

- **A key-value pair is inserted to a random location based on the key**

- **No guarantee that no two keys will not be inserted to the same location**

- **More relaxed definition is given in CS214 Parallel Algorithms offered in S21**
  - Other interesting concepts such as race detection and false sharing
  - More parallel algorithms, programming, and formal training

Prefix Sum (Scan)

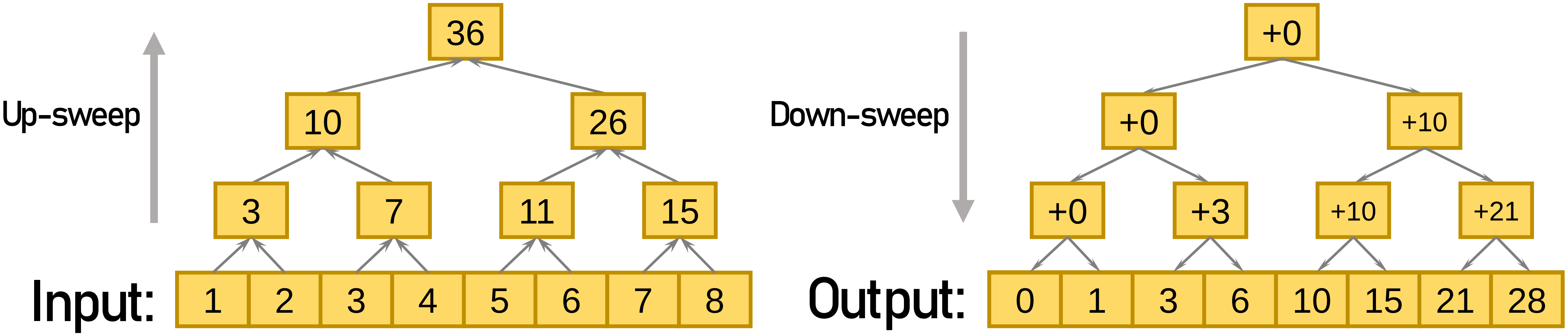
Prefix sum

In = 1 2 3 4 5 6 7 8

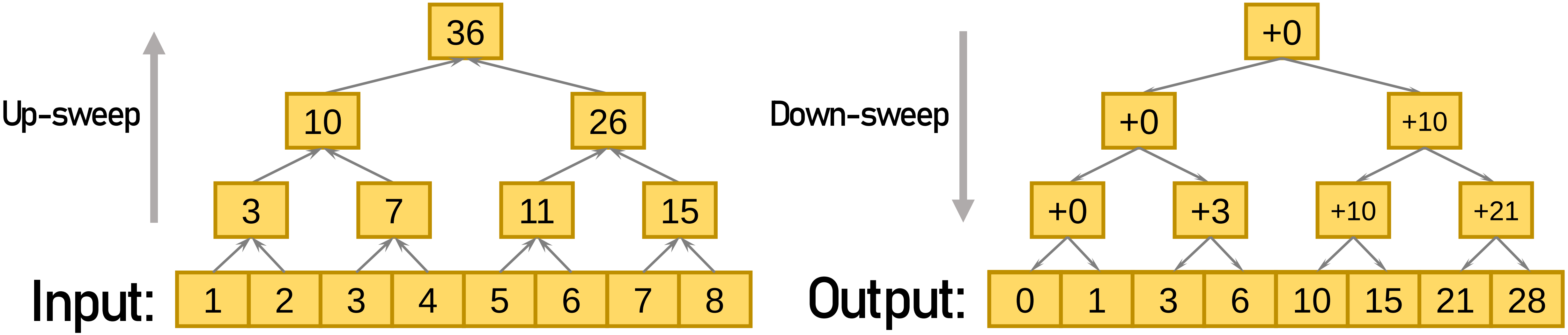
Out = 0 → 1 → 3 → 6 → 10 → 15 → 21 → 28 → 36

The most widely-used building block in parallel algorithm design

A divide-and-conquer algorithm



Pseudocode for scan




```
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 A[0..n]=L+R;
}
```

```
scan(A, n, ps) {
    if (n == 1) { A[0]=ps; return;}
    In parallel:
        scan(A, n/2, ps);
        scan(A+n/2, n-n/2, ps+LeftSum);
}
```

Filtering / packing

Parallel filtering / packing

- Given an array A of elements and a predicate function f, output an array B with elements in A that satisfy f

$$f(x) = \begin{cases} true & if\ x\ is\ odd \\ false & if\ x\ is\ even \end{cases}$$

$A =$

4	2	9	3	6	5	7	11	10	8

$B =$

9	3	5	7	11

Parallel filtering / packing

- **Sequentially, we just read the array from left to right and put those satisfying f into an input array**
- **How can we know the length of B in parallel?**
  - Count the number of red elements – parallel reduce
  - $O(n)$ work and $O(\log n)$ span

$A =$

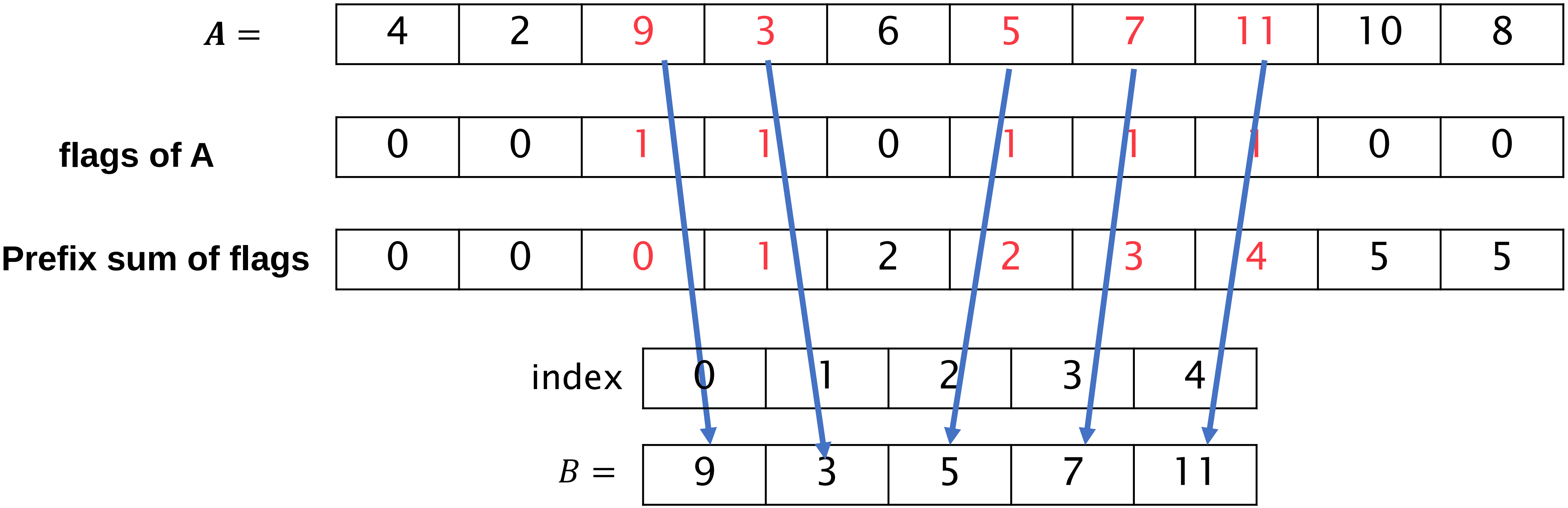
4	2	9	3	6	5	7	11	10	8

0	0	1	1	0	1	1	1	0	0

Parallel filtering / packing

- **How can we know where should 9 go?**
  - 9 is the first red element, 3 is the second, …

```
Filter(A, n, B, f) {
  new array flag[n], ps[n];
  parallel_for (i = 0 to n-1) {
    flag[i] = f(A[i]); }
  ps = prefix_sum(flag, n);
  parallel_for(i=0 to n-1) {
    if (f(A[i]))
      B[ps[i]] = A[i];
  } }
```



Parallel Filtering/packing

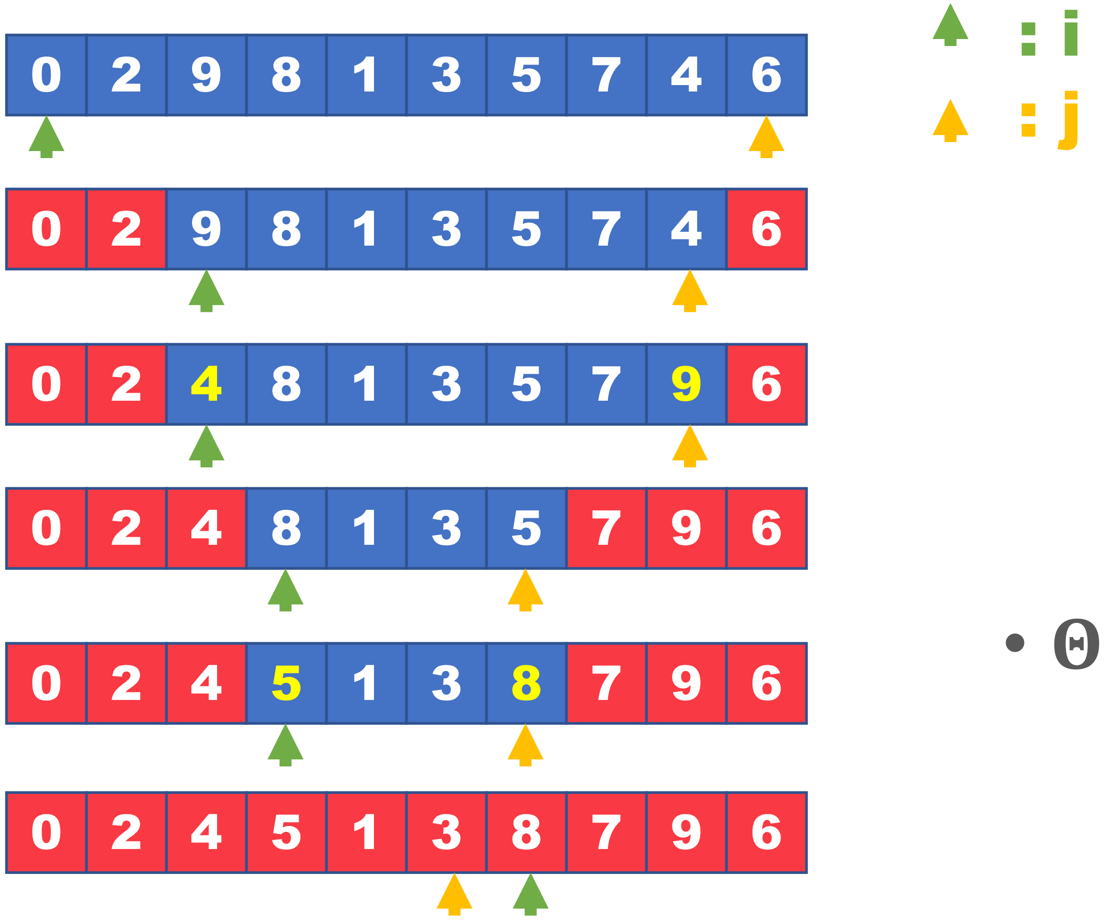
- $O(n)$ **work**
- $O(\log n)$ **span**

```
Filter(A, n, B, f) {
  new array flag[n], ps[n];
  parallel_for (i = 0 to n-1) {
    flag[i] = f(A[i]); }
  ps = prefix_sum(flag, n);
  parallel_for(i=0 to n-1) {
    if (f(A[i]))
      B[ps[i]] = A[i];
  } }
```

Parallel Quicksort

Sequential quicksort

- How to move elements around? (using 6 as a pivot)




```
Partition(A, n, x) {
  i = 0; j = n-1;
  while (i < j) {
    while (A[i] < x) i++;
    while (A[j] > x) j++;
    if (i < j) {
      swap A[i] and A[j];
      i++; j--;
    }
  }
}
```

- $\Theta(n)$ time for one round

Sequential quicksort

- Use a pivot and partition the array into two parts
- Sort each of them recursively

```
qsort(A, n) {
  t = partition(A, A[random()]);
  qsort(A, t);
  qsort(A+t, n-t);
}
```

Parallel quicksort

- Use a pivot and partition the array into two parts
- Sort each of them recursively, **in parallel**

```
qsort(A, n) {
  t = partition(A, A[random()]);
  In parallel:
    qsort(A, t);
    qsort(A+t, n-t);
}
```

Parallel quick sort

- **The partitioning algorithm costs $O(n)$ time. So even if the problem is always perfectly partitioned**
  - $W(n) = 2W\left(\frac{n}{2}\right) + O(n)$
  - $S(n) = S\left(\frac{n}{2}\right) + O(n)$
  - $S(n) = O(n)$?

```
qsort(A, n) {
  t = partition(A, A[random()]);
  In parallel:
    qsort(A, t);
    qsort(A+t, n-t);
}
```

- **Have to partition in parallel!**

Application of filter: partition in quicksort

- For an array A, move elements in A smaller than k to the left and those larger than k to the right

A

6	2	9	4	1	3	5	8	7	0

Partition by 6

Possible output:

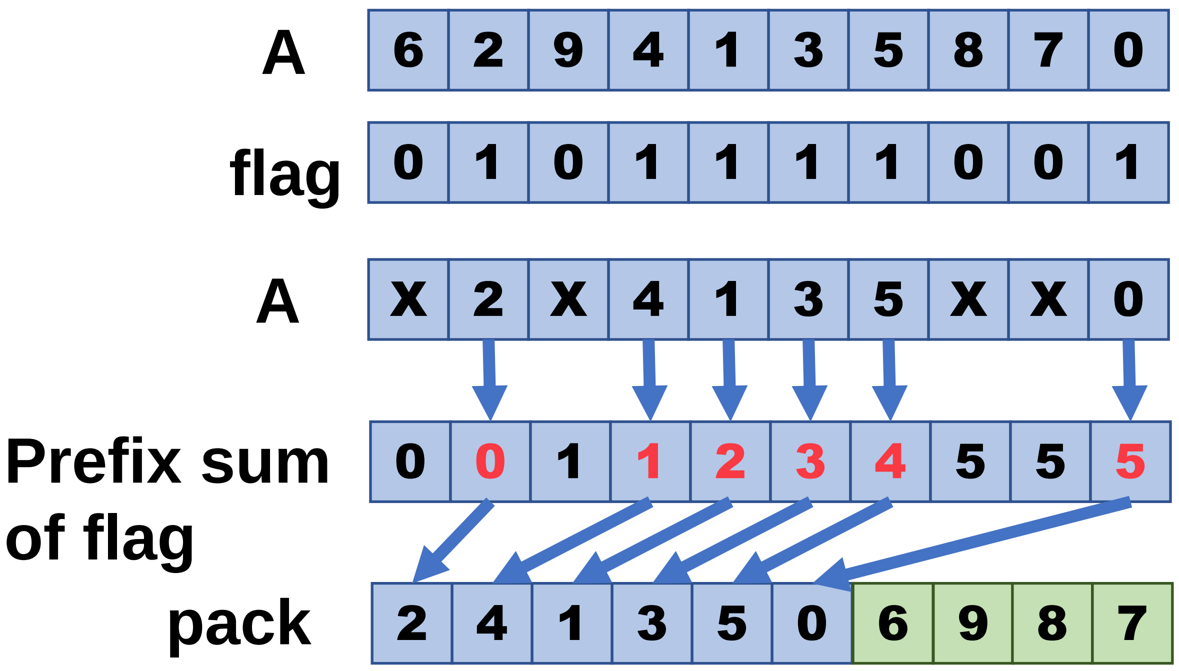
2	4	1	3	5	0	6	9	8	7

- Two filters!!

Using filter for partition

(Looking at the left part as an example)

using 6 as a pivot




```
Partition(A, n, k, B) {
  new array flag[n], ps[n];
  parallel_for (i = 1 to n) {
    flag[i] = (A[i]<k); }
  ps = scan(flag, n);
  parallel_for(i=1 to n) {
    if (A[i]<k)
      B[ps[i]] = A[i];
  }
  //symmetric for the right half
}
```

Predicator: if A[i]<pivot

Parallel quicksort

```
qsort(A, n) {
  t = parallel_partition(A, A[random()]);
  In parallel:
    qsort(A, t);
    qsort(A+t, n-t);
}
```

- **Work**
  - Exactly the same as sequential version
  - $O(n \log n)$ in expectation
- **Span**
  - $O(\log n) \times$ (#rounds of recursions) = $O(\log^2 n)$ in expectation

Summary

Topics covered today

- **Computational models and cost measures**
  - RAM model and time complexity
  - The binary fork-join model and work-span analysis
  - Will be more in the future
- **Race: two logically parallel instructions access the same memory location and at least one of the instructions performs a write**
  - Should be avoided and can be avoided
- **Filtering/packing: based on scan**
- **Quicksort: based on filtering**

What we have talked about so far: parallel algorithms

- **Avoid low-level details**
  - The binary fork-join model
  - Scheduler
  - Cost measures: work and span
- **Coarsening: for divide-and-conquer algorithms**
- **Some parallel algorithms**
  - Reduce → Scan → Pack → Partition → Quicksort
- **Parallel thinking**
  - Consider problems as primitives, and build one on top of others
  - Functional programming

The next two lectures

- An Overview of Computer Architecture
  - Instruction level parallelism (ILP)
  - Multiple processing cores
  - Vector (superscalar, SIMD) processing
  - Multi-threading (hyper-threading)
  - Caching