Performance analysis of parallel algorithms on multi-core platform

Parallel Computer Architecture and Programming

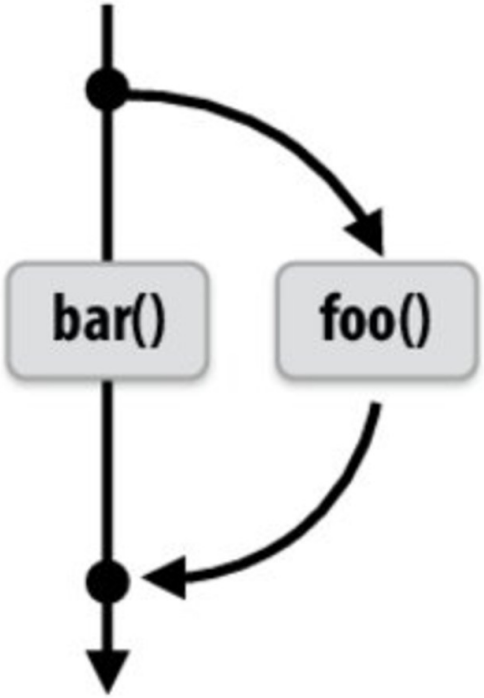
CMU/Tsinghua Summer 2017 (Lecture 7)

Today’s picture

Cilk

```
//part A
cilk_spawn foo();
bar();
cilk_sync;
//part B

//part A
In parallel
     foo()
     bar()
//part B
```



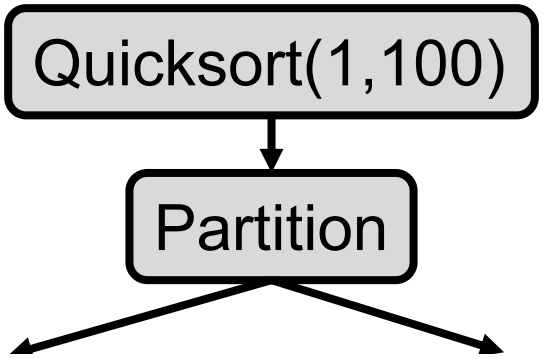
Quicksort

```
Function QuickSort (A)
    p ← random pivot
    s ← 1, t ← |A|
    WHILE s<t
        WHILE A_s<p: s++
        WHILE A_t>p: t--
        IF s<t: swap(A_{s++}, A_{t--})
    QuickSort (A_{1...s})
    QuickSort (A_{s+1...|A|})
```

Quicksort

```
Function QuickSort (A)
    if |A| is small then BaseCase
    p ← random pivot
    s ← 1, t ← |A|
    WHILE s<t
        WHILE A_s<p: s++
        WHILE A_t>p: t--
        IF s<t: swap(A_s++, A_t--)
    In Parallel
        QuickSort (A_1…s)
        QuickSort (A_s+1…|A|)
```

Quicksort



Fork-join Parallelism
(aka. Nested-Parallel Model)

Parallel primitive:

A() || B()

Can be used to implement parallel_for with $\log n$ levels

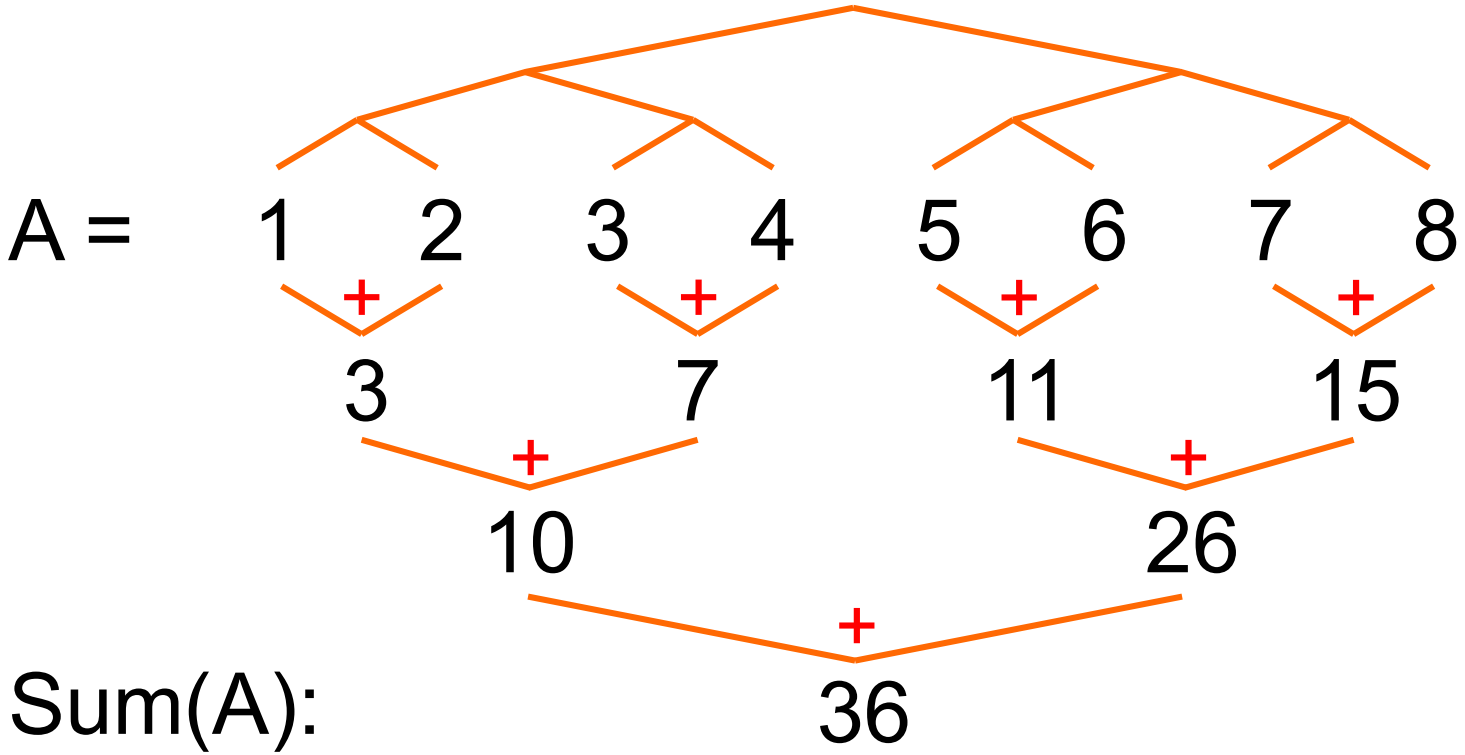
Parallelism of Randomized Work-Stealing Scheduler

6 + 15 + 15 = 36

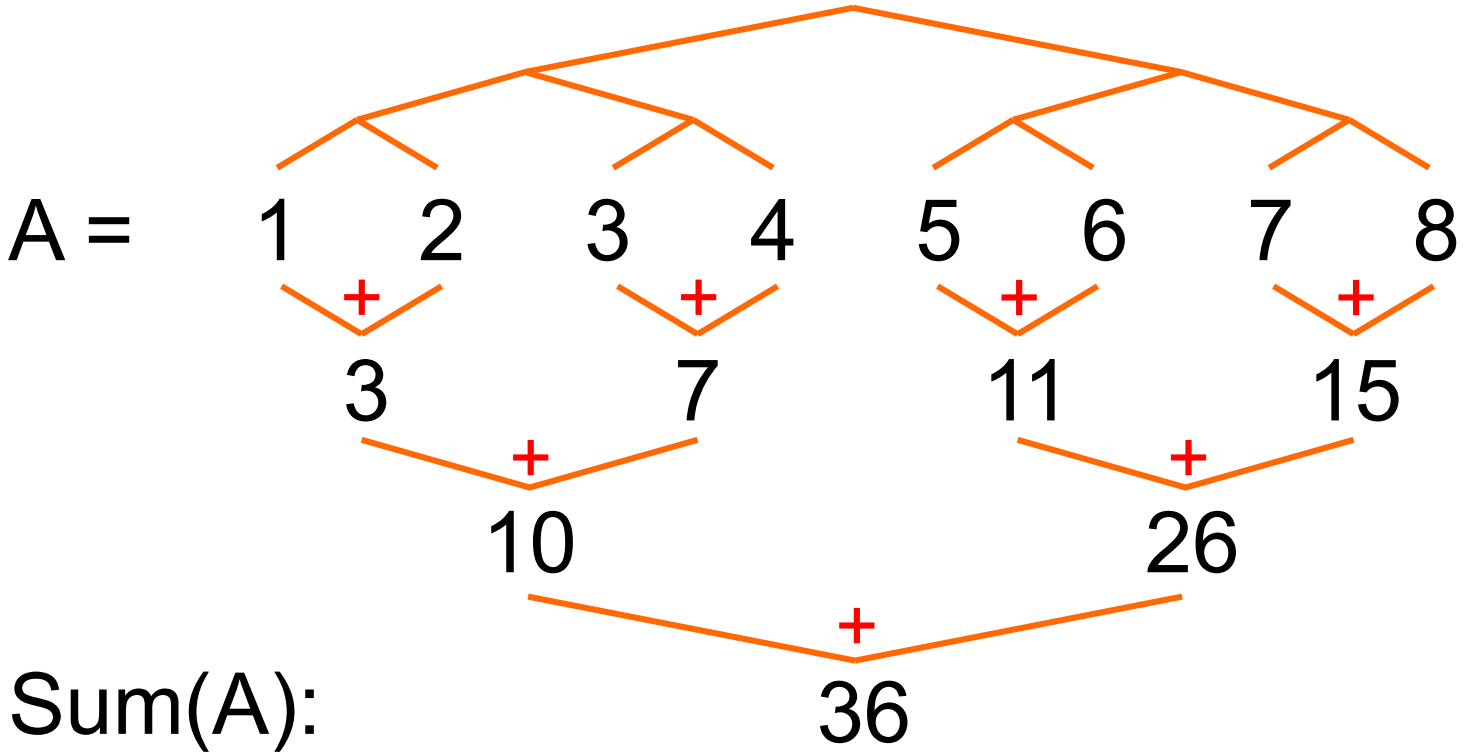
A = 1 2 3 | 4 5 6 | 7 8

Sum(A): 36

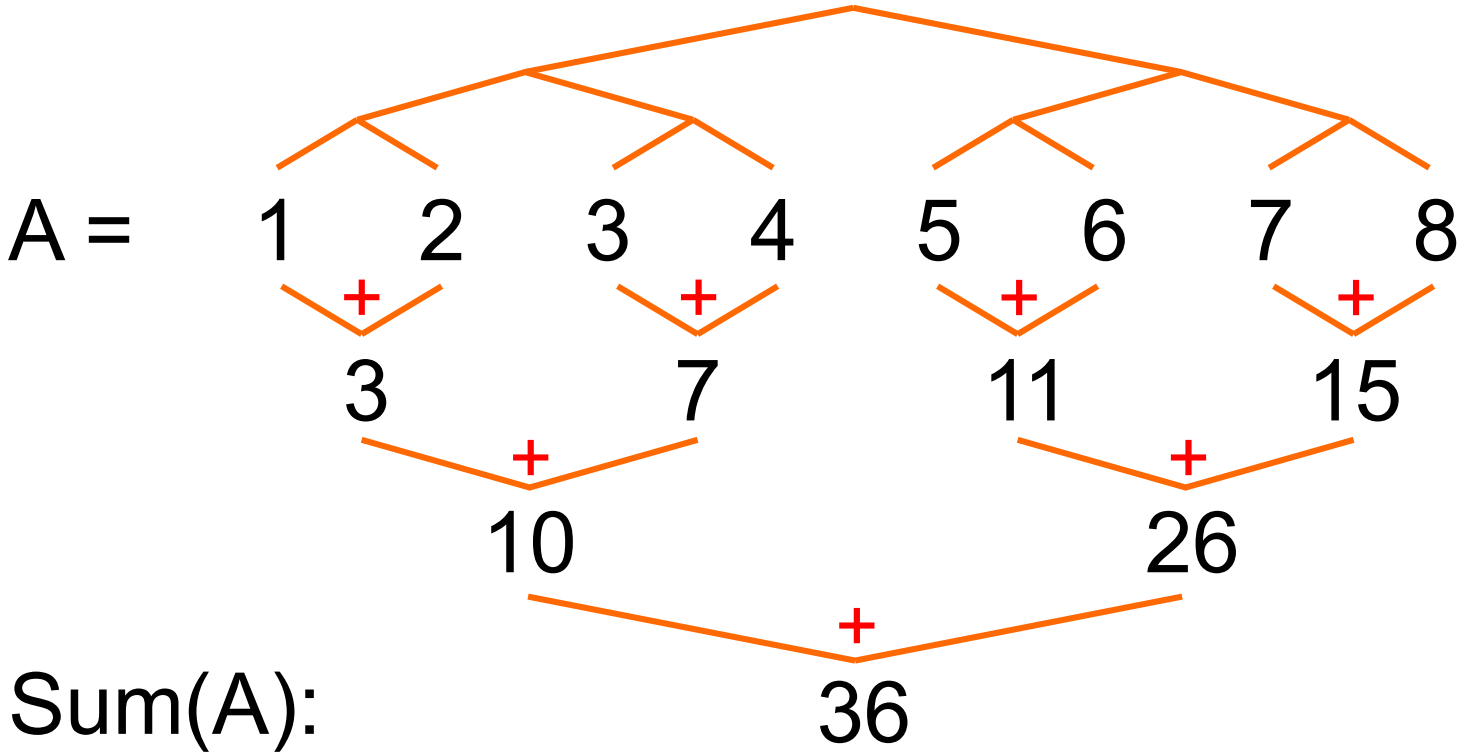
- Cut the input array into smaller segments, sum each up individually, and finally sum up the sums
- Picking the appropriate number of segments can be annoying
  - Machine parameter, runtime environment, algorithmic details




```
Function SUM(A)
        If  |A| = 1  then return  A(1)
        In Parallel
                a = SUM(first half of A)
                b = SUM(second half of A)
        return a + b
```




```
Function SUM(A)
        If  |A| = 1  then return  A(1)
        cilk_spawn a = SUM(first half of A)
                   b = SUM(second half of A)
        cilk_sync
        return a + b
```

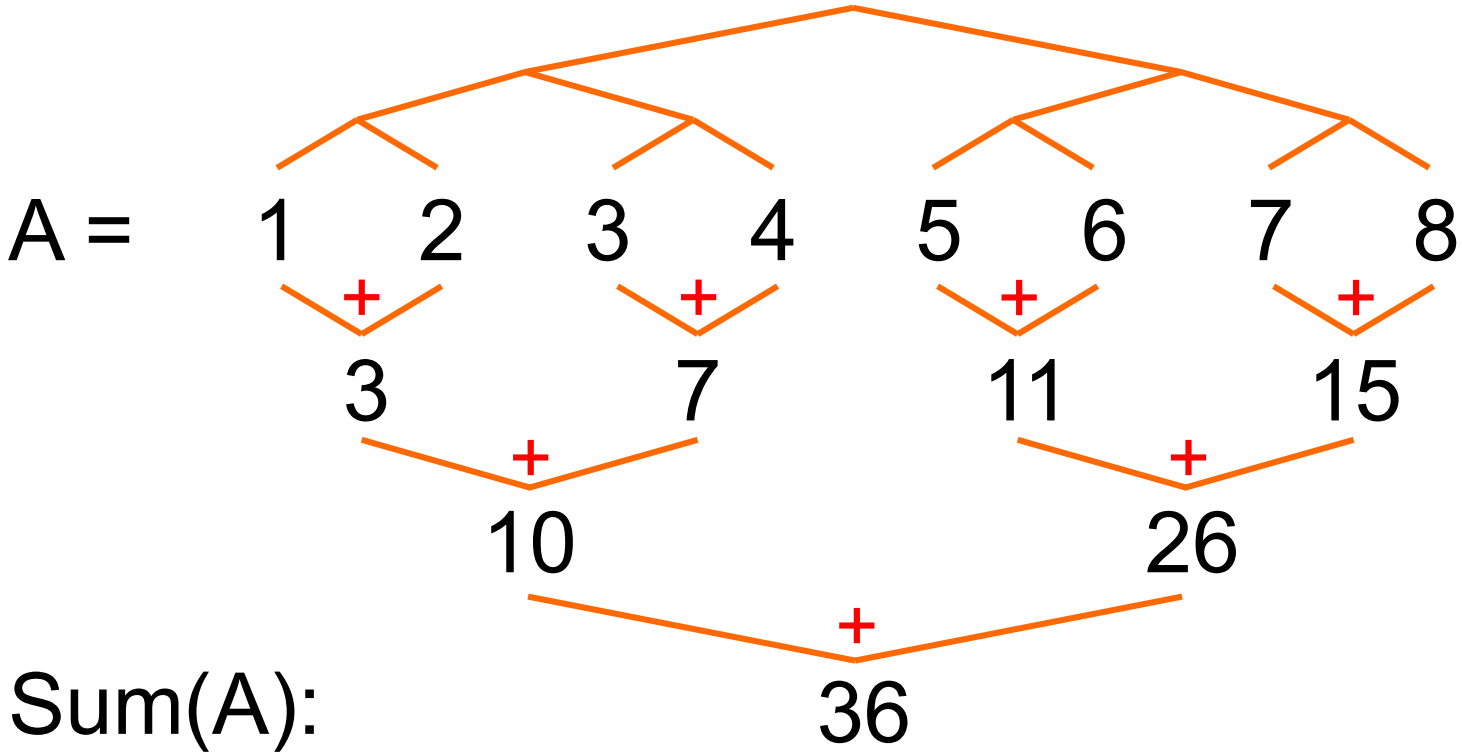


Fork-join Parallelism

Parallel primitive:

A() || B()

Can be used to implement parallel_for with $\log n$ levels



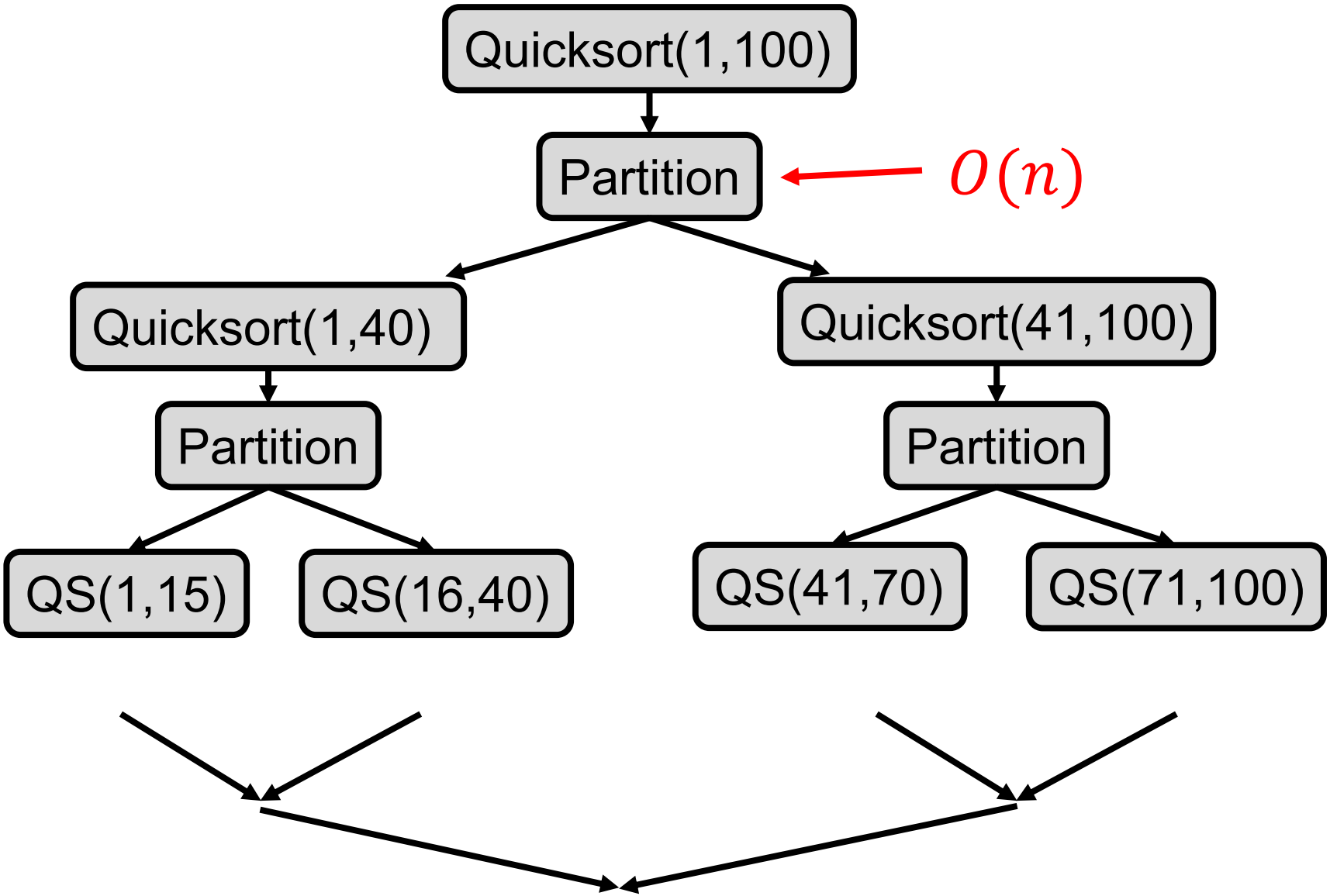
Fork-join Parallelism

Work W:	total computation	$O(n)$
Depth D:	longest chain of all dependencies	$O(\log n)$

Quicksort

```
Function QuickSort (A)
    p ← random pivot
    s ← 1, t ← |A|
    WHILE s<t
        WHILE A_s<p: s++
        WHILE A_t>p: t--
        IF s<t: swap(A_s++, A_t--)
    In Parallel
        QuickSort (A_1...s)
        QuickSort (A_s+1...|A|)
```

Quicksort


Quicksort(1,100)
Partition
$O(n)$
Quicksort(1,40)
Quicksort(41,100)
Partition
Partition
QS(1,15)
QS(16,40)
QS(41,70)
QS(71,100)

Randomized Work-stealing Scheduler

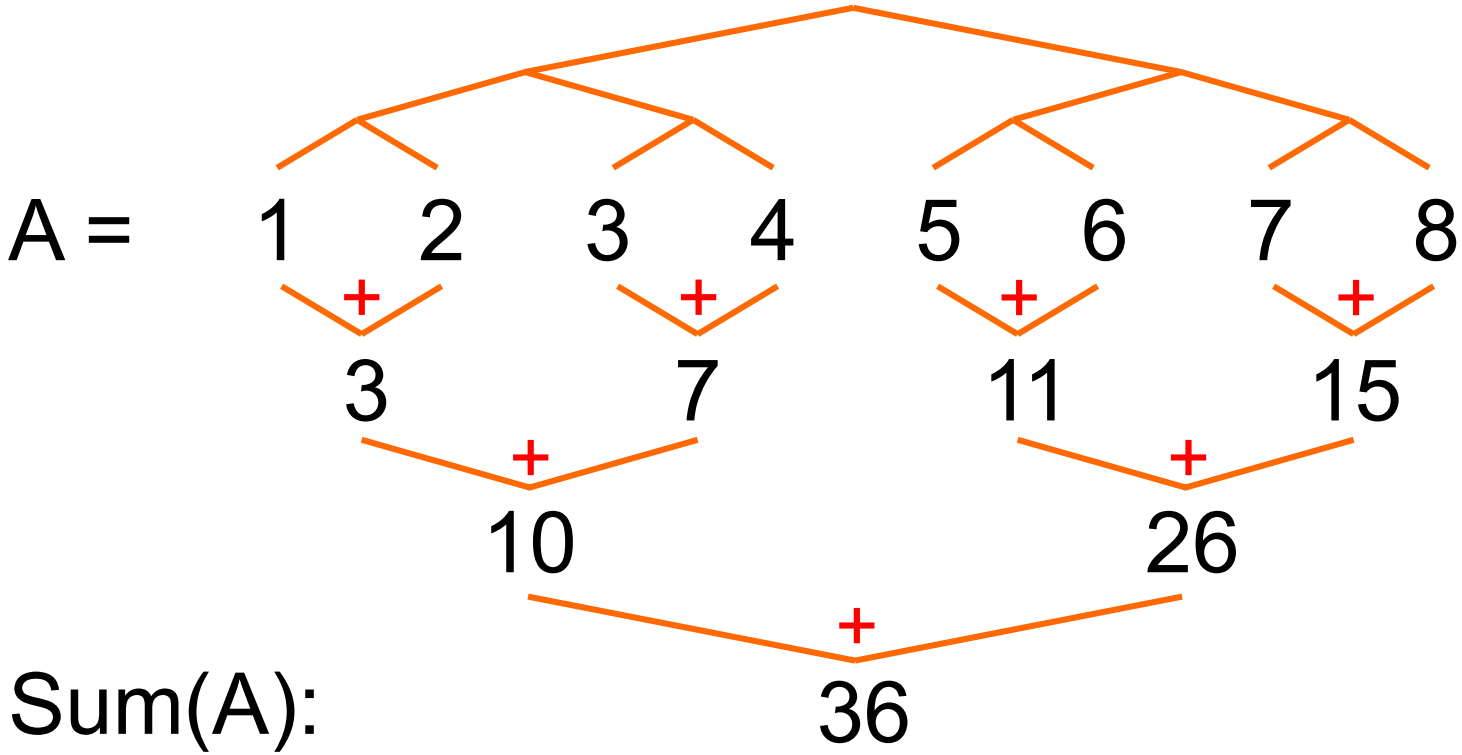
[Blumofe and Leiserson 1999]

A work-stealing scheduler can run a fork-join computation on p cores using time:

$$\frac{W}{p} + O(D)$$

Work W: total computation

Depth D: longest chain of all dependencies



A work-stealing scheduler can schedule this computation on p cores using time:

$$O\left(\frac{n}{p} + \log n\right)$$

Randomized Work-stealing Scheduler

[Blumofe and Leiserson 1999]

A work-stealing scheduler can run a fork-join computation on p cores using time:

$$\frac{W}{p} + O(D)$$

(The work-stealing scheduler is used in OpenMP, CilkPlus, Intel TBB, MS PPL, etc.)

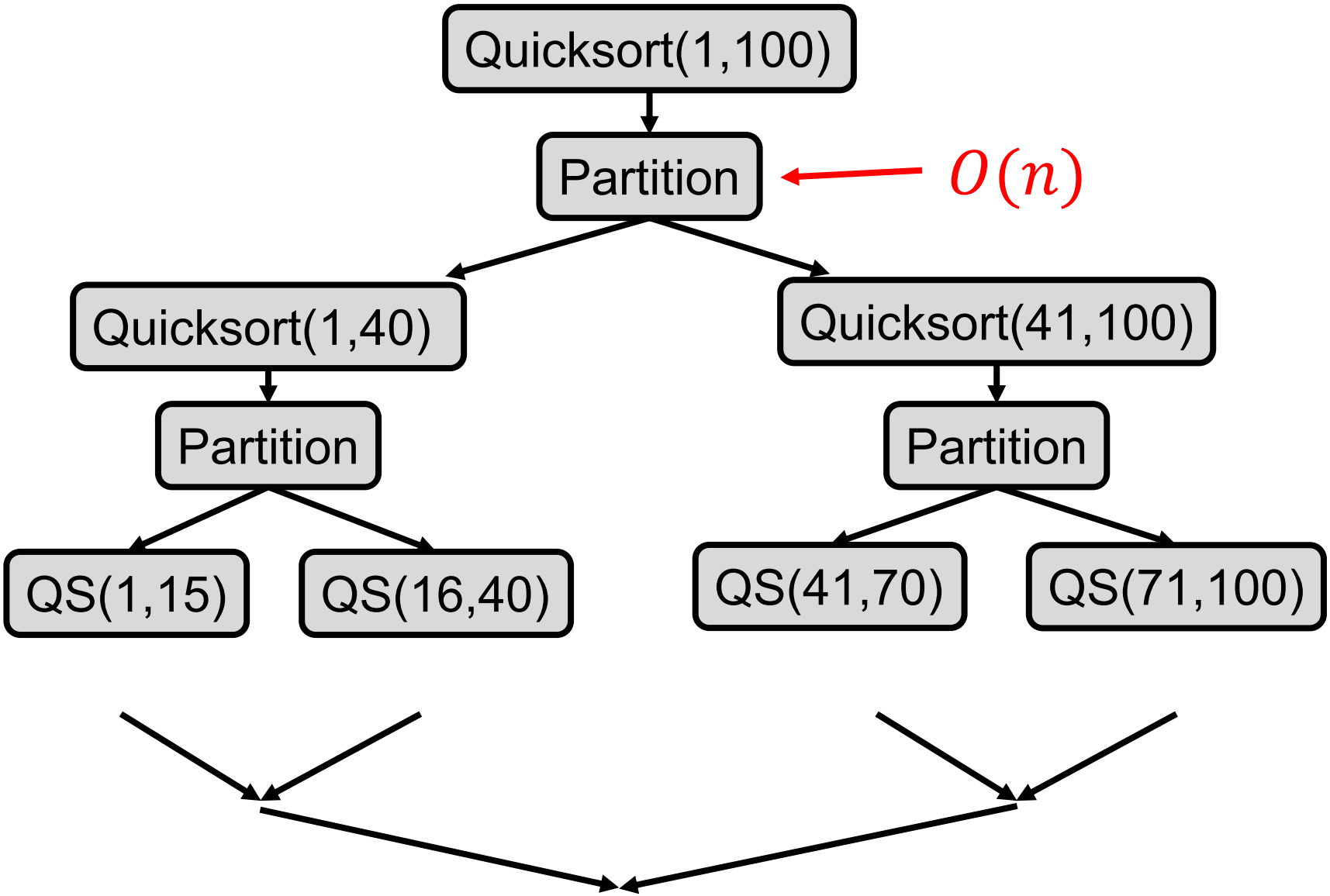
Work W: total computation

Depth D: longest chain of all dependencies

Quicksort

```
Function QuickSort (A)
    p ← random pivot
    s ← 1, t ← |A|
    WHILE s<t
        WHILE A_s<p: s++
        WHILE A_t>p: t--
        IF s<t: swap(A_s, A_t)
    In Parallel
        QuickSort (A_{1...s})
        QuickSort (A_{s+1...|A|})
```

Quicksort


Quicksort(1,100)
Partition
$O(n)$
Quicksort(1,40)
Quicksort(41,100)
Partition
Partition
QS(1,15)
QS(16,40)
QS(41,70)
QS(71,100)

Quicksort

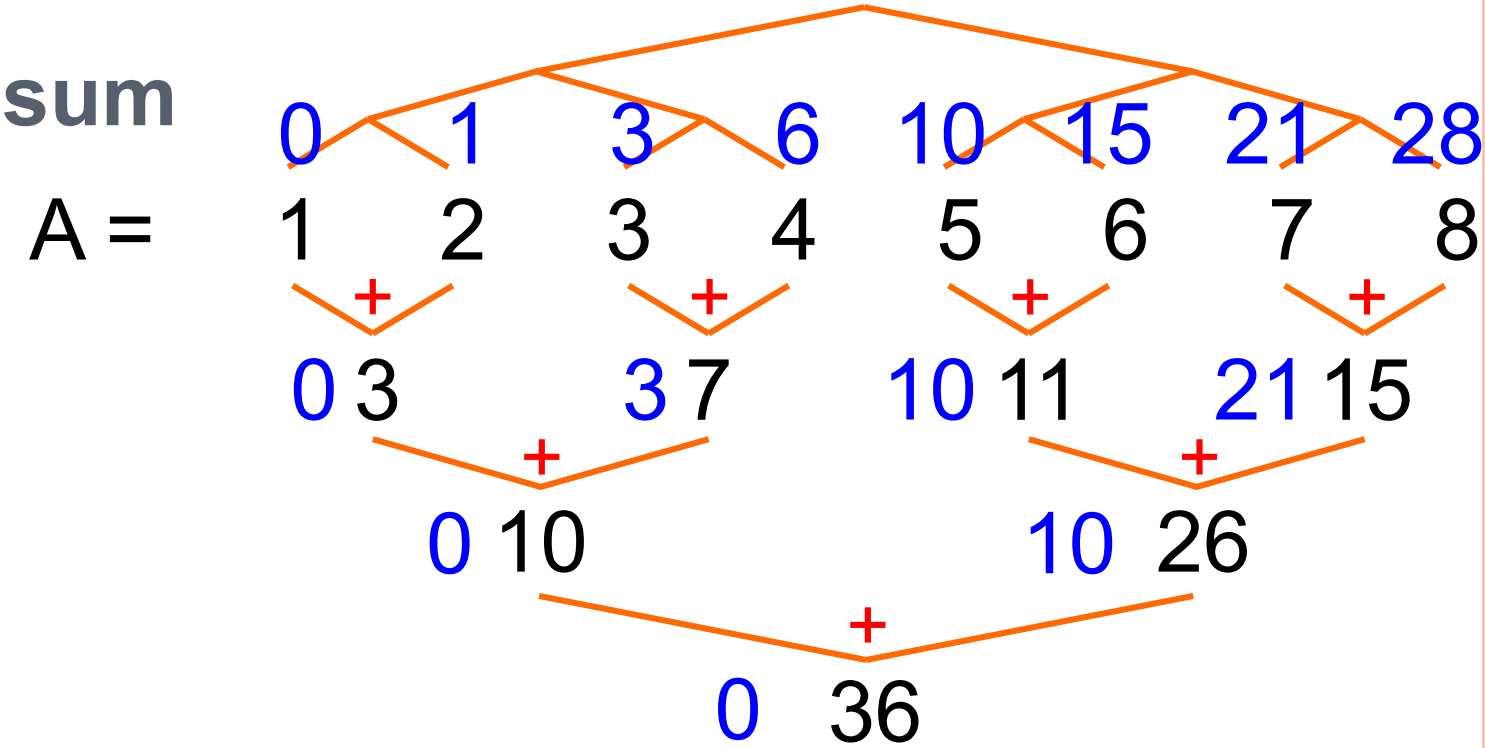
```
Function QuickSort (A)
    p ← random pivot
    s ← 1, t ← |A|
    WHILE s<t
        WHILE A_s<p: s++
        WHILE A_t>p: t--
        IF s<t: swap(A_s, A_t)
    In Parallel
        QuickSort (A_1…s)
        QuickSort (A_s+1…|A|)
```

$$O\left(\frac{n \log n}{p} + n\right) = \frac{T_1}{p} + O(n)$$

Prefix sum

A =	1	2	3	4	5	6	7	8
	0	1	3	6	10	15	21	28

Prefix sum



Function PrefixSum(s, t, ps)

If s=t then return P(s) ← ps

In Parallel

PrefixSum(s, mid, ps)

PrefixSum(mid+1, t, ps+leftSum)

Select

A =	1	2	3	4	5	6	7	8
I =	1	0	0	1	0	0	1	1
O =	1	4	7	8				
P =	0	1	1	1	2	2	2	3

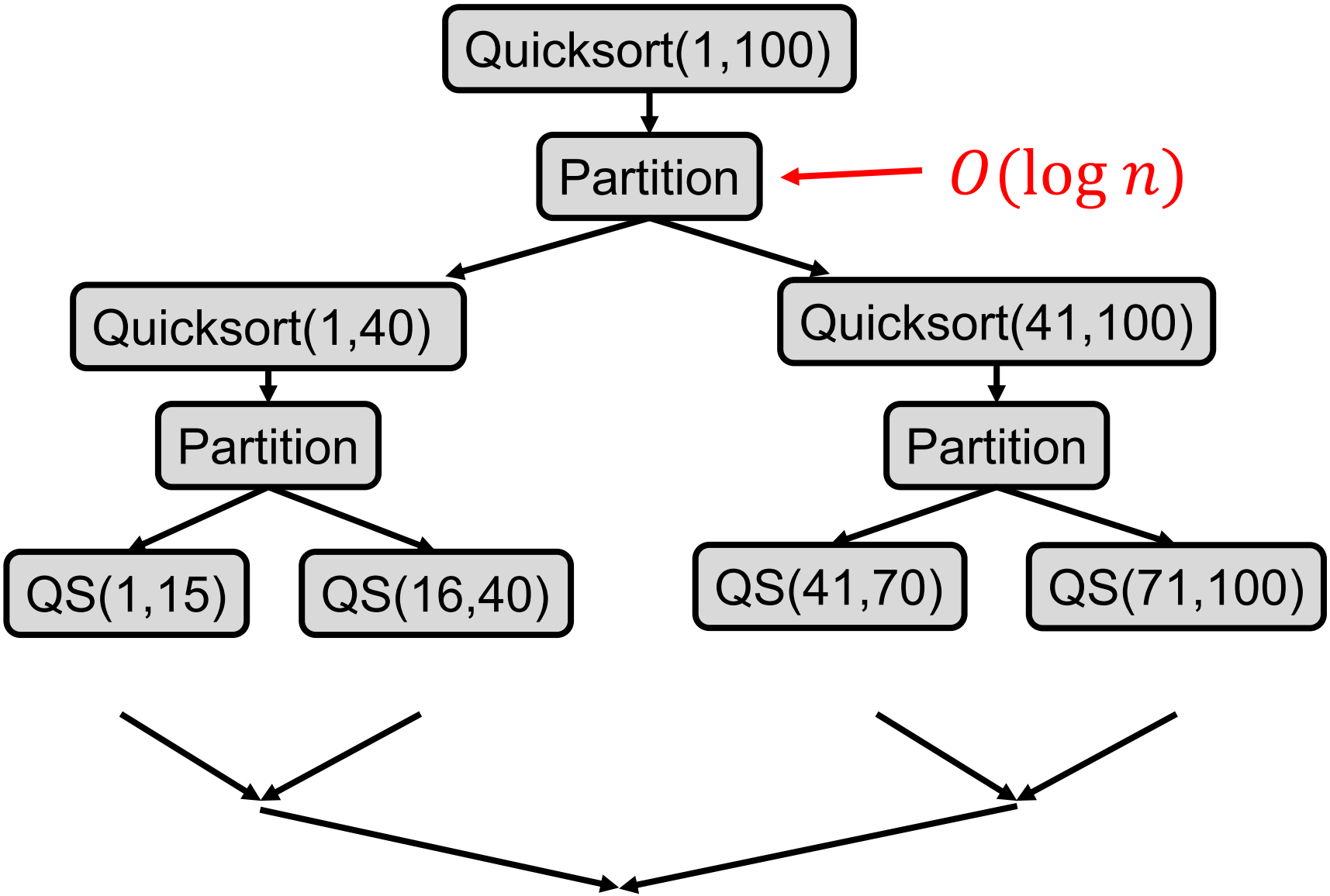
```
Function Select (A, I)
    P ← PrefixSum(I)
    Parallel_for (i ← 1 to |A|)
        if (I(i) = 1) then O(P(i)) ← A(i)
```

Quicksort

```
Function QuickSort (A)
    p ← random pivot
    L ← Select (A, <p)
    M ← Select (A, =p)
    R ← Select (A, >p)
    In parallel
        QuickSort (L)
        QuickSort (R)
    Return L + M + R
```

Work: $O(n \log n)$

Quicksort


Quicksort(1,100)
Partition
$O(\log n)$
Quicksort(1,40)
Quicksort(41,100)
Partition
Partition
QS(1,15)
QS(16,40)
QS(41,70)
QS(71,100)

Quicksort

```
Function QuickSort (A)
    p ← random pivot
    L ← Select (A, <p)
    M ← Select (A, =p)
    R ← Select (A, >p)
    In parallel
        QuickSort (L)
        QuickSort (R)
    Return L + M + R
```

Work: $O(n \log n)$

Depth: $O(\log^2 n)$
(with high probability)

Do most problems can be solved in polylogarithmic depth?

Yes:

- Reduce, prefix sum, comparison sort
- List / tree contraction / ranking
- Most graph algorithms on undirected graphs
- Matrix multiplication, FFT, most comp geo.

Yes but require (significant) extra work:

- Most graph algorithms on directed graphs, shortest-paths

No polylog depth algorithm is known:

- Radix sort, network flow

Wrap up

- As long as the depth of an algorithm is small (i.e. polylogarithmic), randomized work-stealing scheduler can have perfect load balance, and the overhead of scheduling is negligible

- Two key points:
  - Smaller (polylogarithmic) depth
  - Better sequential code (I/O to the memory)

How to write faster (parallel) code?

Algorithms with lower time complexity (less operations)?

$(n = 2 \times 10^9)$

```
parallel_for (i ← 1 to n)
     a[i] += 1
```

Running time: 0.47s

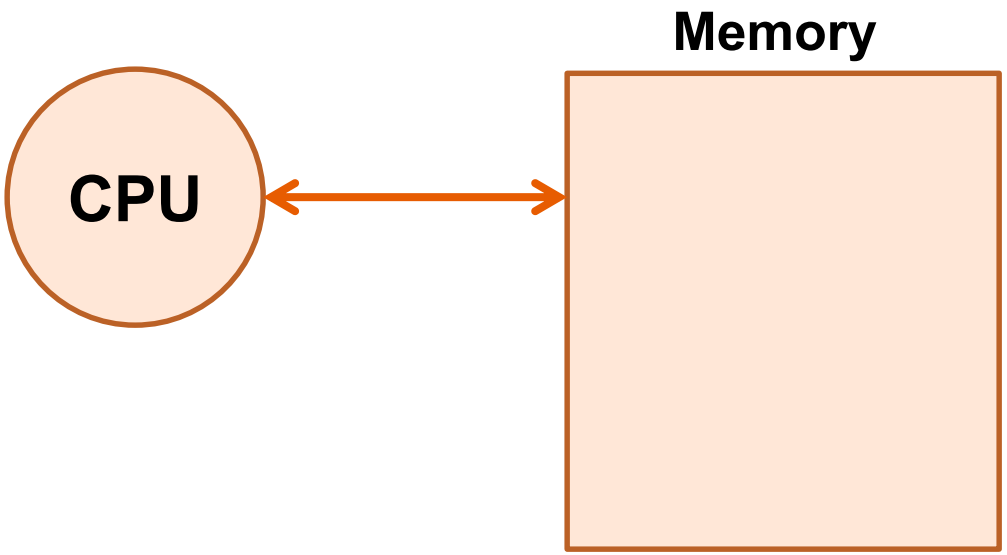
```
parallel_for (i ← 1 to n)
     t = rand() % n
     a[t] += 1
```

Running time: 2.22s

Cache complexity

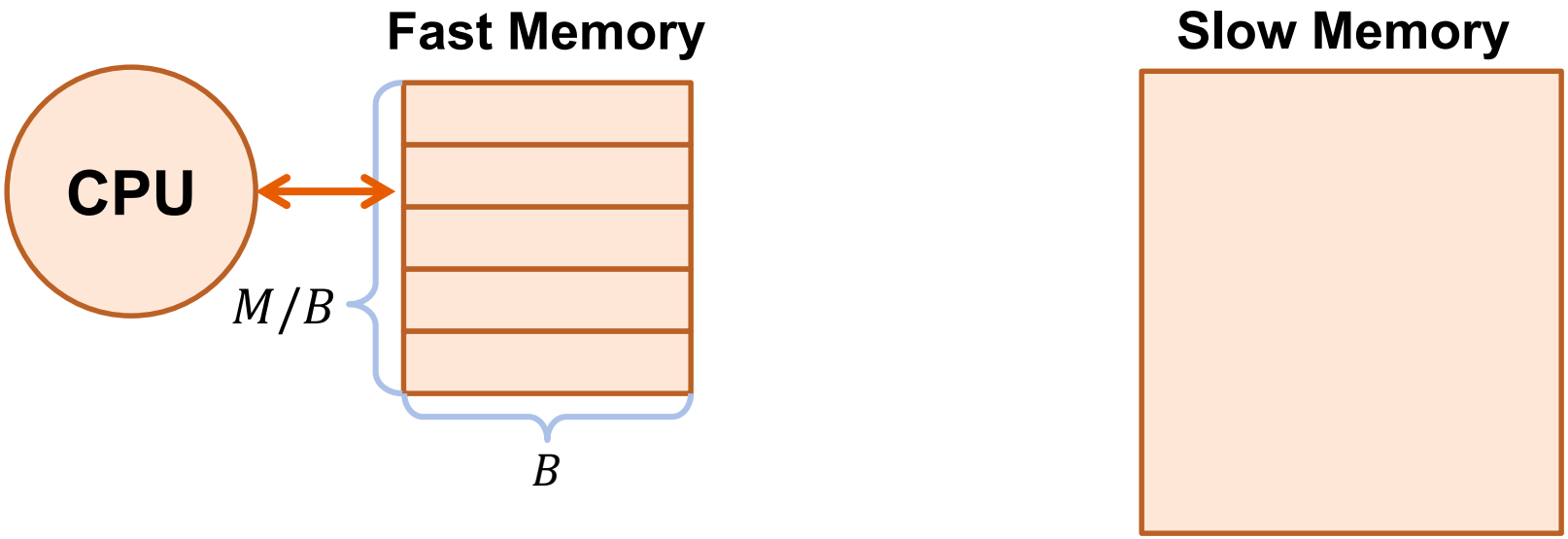
The Random-Access Machine (RAM) model

- Unit cost to read from and write to any specific memory location
  - Therefore unit cost for any operation
- **Time complexity T** of an algorithm is the minimum amount of cost of to run it on the RAM model



The External Memory (EM) model

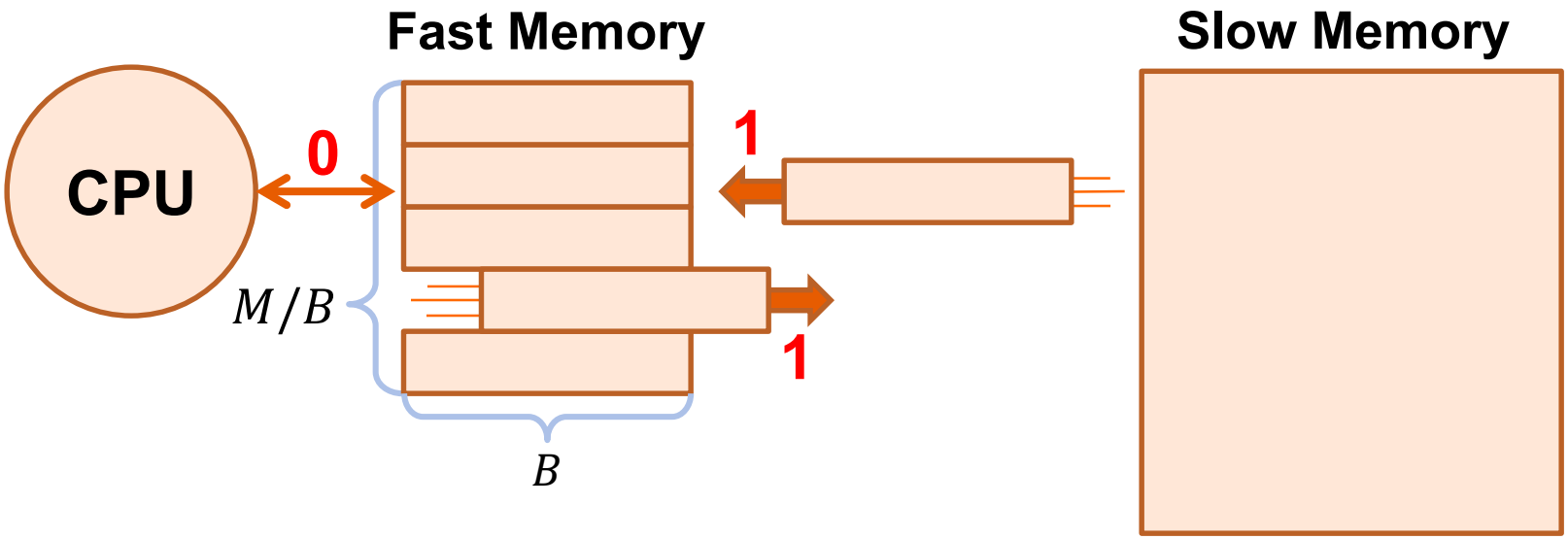
- Two-level memory hierarchy:
  - A small memory (fast memory, cache) of fixed size M
  - A large memory (slow memory) of unbounded size
- Both are partitioned into blocks of size B
- Instructions can only apply to data in primary memory, and are free



The External Memory (EM) model

- The EM model has two special *memory transfer* instructions:
  - **Read transfer**: load a block from slow memory
  - **Write transfer**: write a block to slow memory
- The complexity of an algorithm on the EM model (I/O complexity) is measured by:

$$\#(read\ transfer)\ +\ \#(write\ transfer)$$



Summaries on the EM model

- The cache can hold whatever M/B blocks YOU like, each with size B
- The algorithm can freely access the cache and apply computations
- Take unit cost to load or store an arbitrary block to the (large) memory
- The **cache complexity Q** of an algorithm is the minimum amount of replacement to finish the computation
- (Usually $M \sim 10^6$ for L3 cache, B is 64 bytes)

List access example ($n = 2 \times 10^9$)

```
parallel_for (i ← 1 to n)
      a[i] += 1
```

Scan (sequential writes)

Running time: 0.47s

$Q = 2n/B$

Scatter vs. scan: 4.72

```
parallel_for (i ← 1 to n)
      t = rand() % n
      a[t] += 1
```

Scatter (random writes)

Running time: 2.22s

$Q = 2n$

Why is this simple model good enough?

- Lemma: If an algorithm incurs $\boldsymbol{Q}$ memory transfers on an ideal cache of size $\boldsymbol{M/2}$, then it incurs at most $\boldsymbol{2Q}$ memory transfers on a fully associative cache of size $\boldsymbol{M}$ with LRU replacement

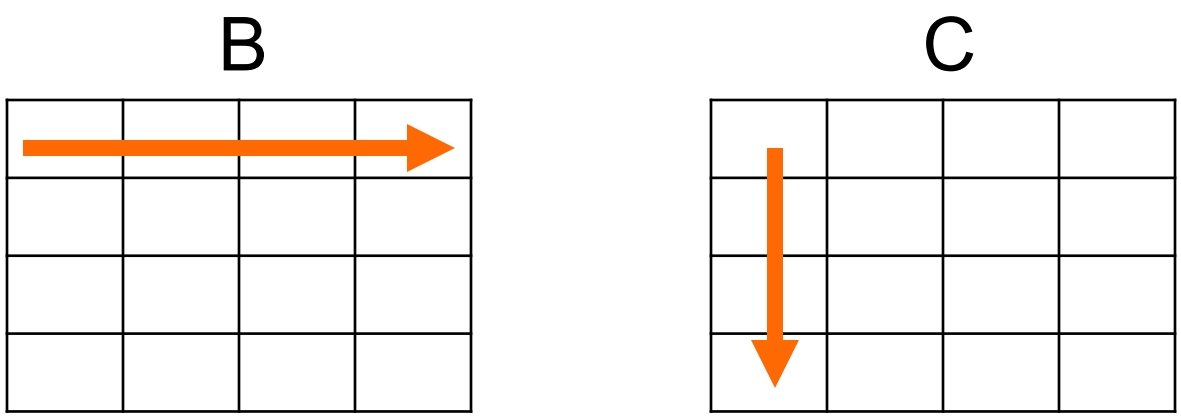
- In practice, the real cache is very clever and *almost* LRU, and the performance matches Q when the computation is I/O-bounded

Matrix multiplication ($n = 2^{13}$)

```
parallel_for (i ← 1 to n)
     for (j ← 1 to n)
          for (k ← 1 to n)
               a[i][j] += b[i][k] * c[k][j]
```

$Q = O(n^3)$, n^2 scatters

Running time: 216s

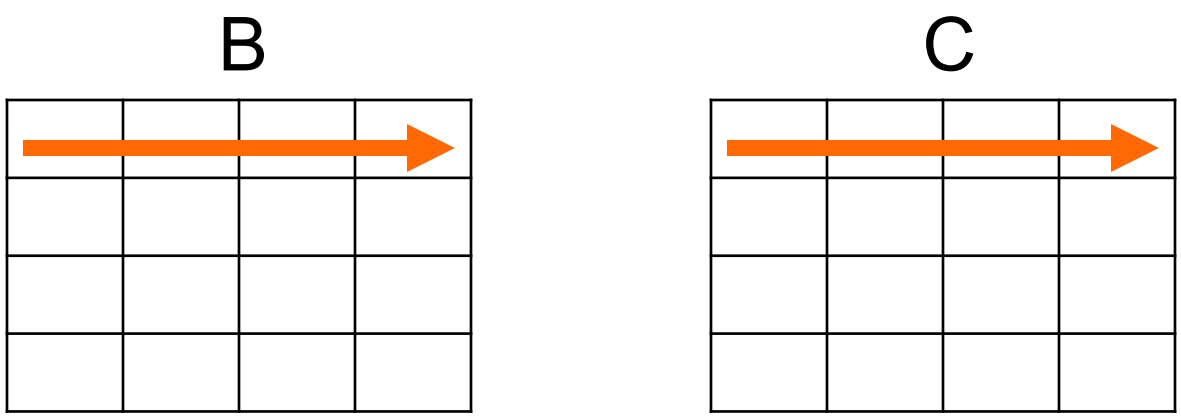


Matrix multiplication ($n = 2^{13}$)

```
Transpose(c)
parallel_for (i ← 1 to n)
     for (j ← 1 to n)
          for (k ← 1 to n)
               a[i][j] += b[i][k] * c[j][k]
```

$Q = O(n^3/B)$, n^2 scans

Running time: 69s



Matrix multiplication ($n = 2^{13}$)

```
Func A*B {
 if A is small then baseCase
 In Parallel
  C11 = A11*B11 + A12*B21
  C12 = A11*B12 + A12*B22
  C21 = A21*B11 + A22*B21
  C22 = A21*B12 + A22*B22
 return C
}
```

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

$$B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$Q = O\left(n^3/B\sqrt{M}\right)$$

Running time: 9.8s

Parallel cache complexity

[Acar, Blelloch, Blumofe 2000]

- Practically, as long as the depth is small (i.e. polylogarithmic) , the additional cache complexity caused by parallelism is bounded and usually negligible

- Using work-stealing scheduler and with private caches, the cache complexity of a parallel algorithm using p cores satisfies (*):

$$Q_p = Q_{seq} + O(PDM/B)$$

loose upper bound

(*): with high probability

Goals to design faster parallel algorithms

- The cost of a parallel algorithm:
  - Computations (CPU operations): captured by **time complexity**, should be as good as the best sequential algorithm
  - Memory access (I/Os to main memory): captured by **cache complexity**
    - Can usually be decomposed in the numbers of scatters and scans. Try to minimize them (including the constants)
  - Extra cost by parallel execution: captured by the **depth** (ideally should be polylogarithmic)

Algorithms that are I/O-efficient

- Memory access (I/Os to main memory): captured by **cache complexity**

- Algorithms that optimize it are called **I/O-efficient algorithms**. Interesting results:
  - Matrix multiplication: $\Theta\left(\frac{n^3}{B\sqrt{M}}\right)$
  - Comparison sorting: $\Theta\left(\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B}\right)$
  - Searching: $\Theta(\log_B n)$

- Algorithms that are not aware of cache parameters (M, B) are called **cache-oblivious algorithms**

Goals to design faster parallel algorithms

- The cost of a parallel algorithm:
  - Computations (CPU operations): captured by **time complexity**, should be as good as the best sequential algorithm
  - Memory access (I/Os to main memory): captured by **cache complexity**
    - Can usually be decomposed in the numbers of scatters and scans. Try to minimize them (including the constants)
  - Extra cost by parallel execution: captured by the **depth** (ideally should be polylogarithmic)

Algorithms with low depth

- Extra cost by parallel execution: captured by the depth (ideally should be polylogarithmic)

- Problems that can be solved in $O(1)$ depth are in NC^0
- Problems that can be solved in $\tilde{O}(\log n)$ depth are in NC^1
- Problems that can be solved in $\tilde{O}(\log^2 n)$ depth are in NC^2

- $NC^0 \subseteq NC^1 \subseteq NC^2 \subseteq \cdots \subseteq NC$

- Does $\boldsymbol{NC = P}$?

Goals to design faster parallel algorithms

- The cost of a parallel algorithm:
  - Computations (CPU operations): captured by **time complexity**, should be as good as the best sequential algorithm
  - Memory access (I/Os to main memory): captured by **cache complexity**
    - Can usually be decomposed in the numbers of scatters and scans. Try to minimize them (including the constants)
  - Extra cost by parallel execution: captured by the **depth** (ideally should be polylogarithmic)

Useful links:

- CMU 15-210: Parallel and Sequential Data Structures and Algorithms
  - An overview of basic algorithms and data structures that makes no distinction between sequential and parallel
- MIT 6.172: Performance Engineering of Software Systems
  - A more thorough explanation on performance analysis on parallel (multi-core) and distributed setting
- CMU: 15-853: Algorithms in the “Real World”
  - A graduate-level course that provides many useful links to parallel algorithms and I/O-efficient algorithms