Chapter 9: Parallel Algorithms

Advanced Algorithms

Why parallelism? | Flynn and PRAM models | Speedup, Amdahl | Work and depth | Summation on a PRAM | Prefix sums | Pipelining | Pipelined tree | Fork/Join, MapReduce

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/advanced_algorithms/slides/Chapter_09_Parallel_Algorithms.html

Learning Outcomes

Concepts

Explain the split–assign–combine structure of parallel algorithms, their benefits and their challenges (communication, synchronization, load balancing).

Models and measures

Know Flynn's taxonomy and the PRAM model; compute speedup, efficiency, cost, work and depth; apply Amdahl's law.

Core techniques

Run PRAM tree summation, parallel prefix sums, pipelining and the pipelined tree step by step, and count their steps.

Parallelism in Java

Read examples of parallel merge sort with Fork/Join, MapReduce and data parallelism; recognize race conditions.

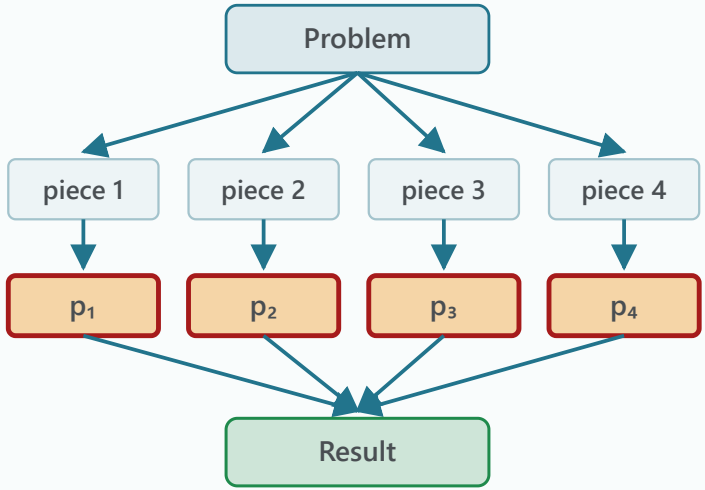
Faster Together

- Some jobs take **a very long time**: big data analysis, scientific simulation, image processing.
- **Parallel algorithms** solve a problem using **several processors** (cores) **at the same time**.
- Complex tasks finish in **much less time**.
- Single-core clock speeds have nearly stopped rising; performance gains now come from the **number of cores**.

Analogy: one teacher grades 1000 exams in 10 hours. If 10 teachers share the exams it takes ≈ 1 hour; but **handing out** the exams and **collecting** the grades also takes time.

Parallel Algorithms

- Like a group of people **working together**.
- Split a complex problem into **smaller pieces**.
- Speed up the work by **assigning the pieces to different processors** at the same time.
- When the processors finish, **combine the results**.
- The foundation of **high-performance computing** (HPC).
- Uses: big data analysis, scientific simulation, machine learning and more.



Benefits and Drawbacks

Benefits

+ Finish the same job sooner (speedup)
+ Solve large problems that do not fit on one machine (scalability)
+ Use the hardware (all cores, the GPU) efficiently

Drawbacks

– Splitting the problem and **combining** the results takes extra effort
– **Communication** and **synchronization** add overhead
– **Load balancing**: every processor should get equal work, and there must be enough processing power
– Debugging is hard (race conditions, deadlocks)
– Some problems are inherently **sequential**

With data dependences (step k waits for the result of step k − 1), that part cannot be parallelized: e.g., the recurrence $x_k = f(x_{k-1})$.

Applications

Matrix multiplication

The core operation of scientific computing; every row (even every cell) of the result can be computed independently.

Image processing

Edge detection, filtering, object recognition: the same operation on every pixel or block (data parallelism, GPU).

Computational fluid dynamics

Flow simulation in aircraft, ship and vehicle design: the domain is split into a grid, each region on one processor.

Genetic algorithms

The fitness of individuals in a population is evaluated independently and in parallel (scheduling, routing).

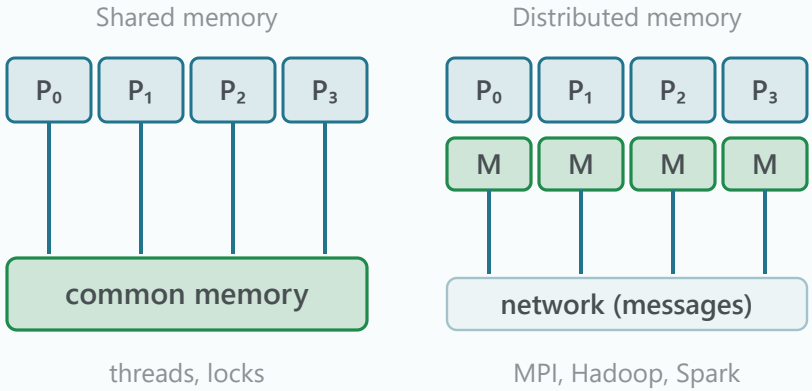
Flynn's Taxonomy

Class	Stands for	Meaning	Example
SISD	Single Instruction, Single Data	one instruction stream, one data stream	classic single-core processor
SIMD	Single Instruction, Multiple Data	the same instruction on many data items at once	vector instructions (AVX), GPU
MISD	Multiple Instruction, Single Data	different instructions on the same data	rare; fault-tolerant systems
MIMD	Multiple Instruction, Multiple Data	each processor runs its own program on its own data	multicore CPUs, clusters

Data parallelism: the same operation on different parts of the data (SIMD style).

Task parallelism: different tasks at the same time (MIMD style).

Memory Architectures



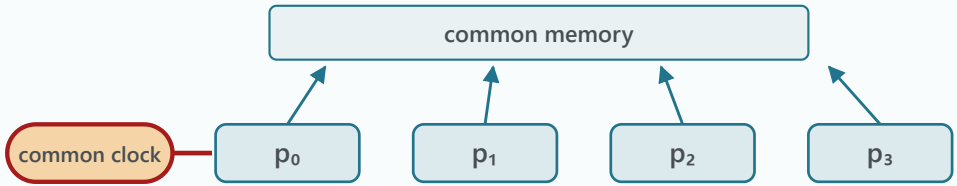
- **Shared memory**: all processors see the same address space. Communication is cheap; concurrent access requires **synchronization**.
- Java threads, `ForkJoinPool`, OpenMP.

- **Distributed memory**: each node has its own memory; data moves via **messages**.
- Scales well, but communication is expensive: MPI, Hadoop MapReduce.

The PRAM Model

- **PRAM** (Parallel Random Access Machine): the theoretical model of parallel algorithms; the parallel counterpart of the sequential RAM model.
- p processors, a shared memory and a **common clock**: the processors take steps in **lockstep**.
- In each step a processor reads from memory → computes locally → writes to memory. Each takes O(1).
- Communication and synchronization costs are **ignored**, so the model measures an algorithm's inherent parallelism.

Type	Read	Write
EREW	exclusive	exclusive
CREW	concurrent	exclusive
CRCW	concurrent	concurrent



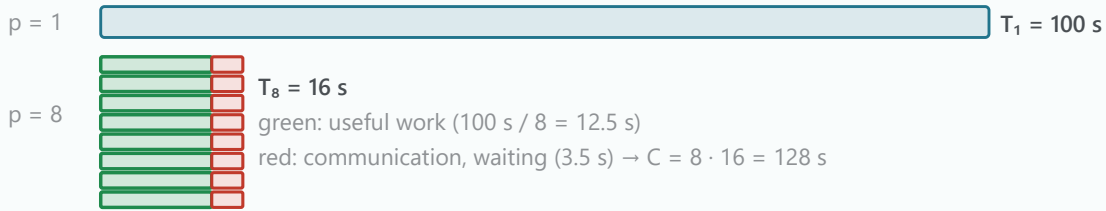
E = Exclusive (one processor per cell per step), C = Concurrent (several processors). In CRCW, conflicting writes are resolved by a rule (common value, arbitrary one, priority…).

Speedup, Efficiency, Cost

Measure	Definition
T_1	running time of the best sequential algorithm
T_p	parallel running time on p processors
Speedup	$S = T_1 / T_p$ (ideal: $S = p$)
Efficiency	$E = S / p$ $(0 < E \leq 1)$
Cost	$C = p \cdot T_p$

Example: $T_1 = 100$ s, $p = 8$, $T_8 = 16$ s
$S = 100 / 16 =$ **6.25**
$E = 6.25 / 8 \approx$ **0.78**
$C = 8 \cdot 16 = 128$ s $(> 100$ s$)$

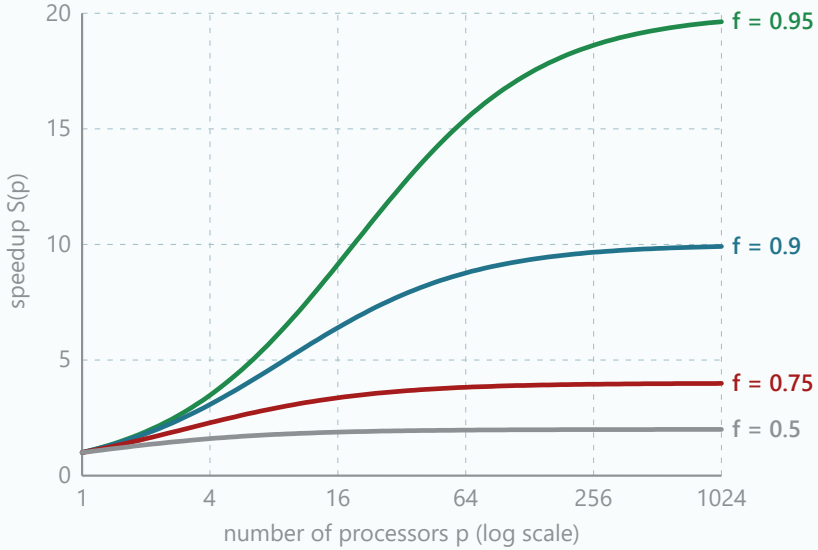
- **Cost-optimal**: $C = \Theta(T_1)$.
- Why $S < p$: communication, waiting, load imbalance, sequential parts.
- These bounds hold in the model; in practice cache effects or finding a target early in a search can give $S > p$ (**superlinear** speedup).



Amdahl's Law

- A fraction f of the program can be parallelized; the fraction 1 − f is **sequential**.
- S(p) = 1 / ((1 − f) + f / p)
- As p → ∞, S → **1 / (1 − f)**: the sequential part limits the speedup.
- f = 0.9, p = 8: S = 1 / (0.1 + 0.1125) ≈ **4.71**; the limit is 10.

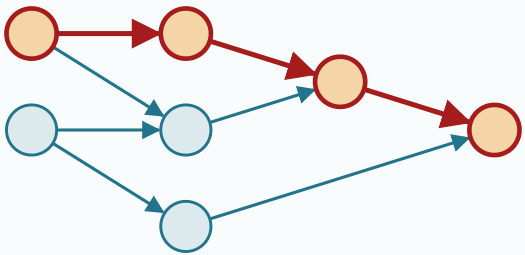
Gustafson's law: if the problem size grows with p, S = (1 − f) + f · p. f = 0.9, p = 8 → S = 7.3. Parallelism pays off more on large problems.



Work and Depth

- **Work** W: the total number of operations = T_1 (time on one processor).
- **Depth** (span) D: the longest chain of dependences = $T\infty$ (time with unlimited processors).
- **Parallelism** = W / D: the largest number of processors that can be used profitably.
- Lower bounds: $T_p \geq W / p$ and $T_p \geq D$.
- **Brent's theorem**: $T_p \leq W / p + D$.
- **Work-efficient**: W is of the same order as the best sequential algorithm.

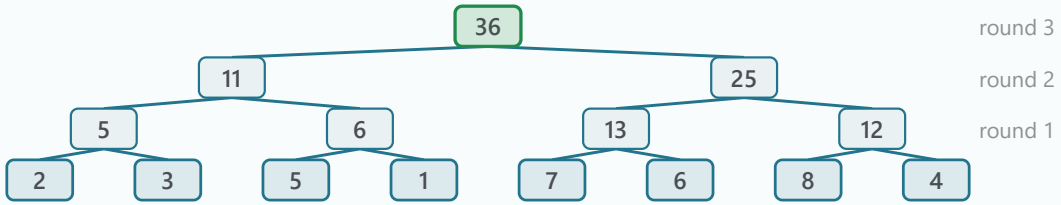
Example: summing n numbers with a tree
W = n − 1 additions, D = $\log_2$ n rounds.
n = 1024: W = 1023, D = 10, parallelism ≈ 102.
Brent with p = 32: $T_{32} \leq 1023/32 + 10 \approx 42$ steps.



W = 7 operations (nodes), D = 4 (red path)
parallelism W / D = 1.75

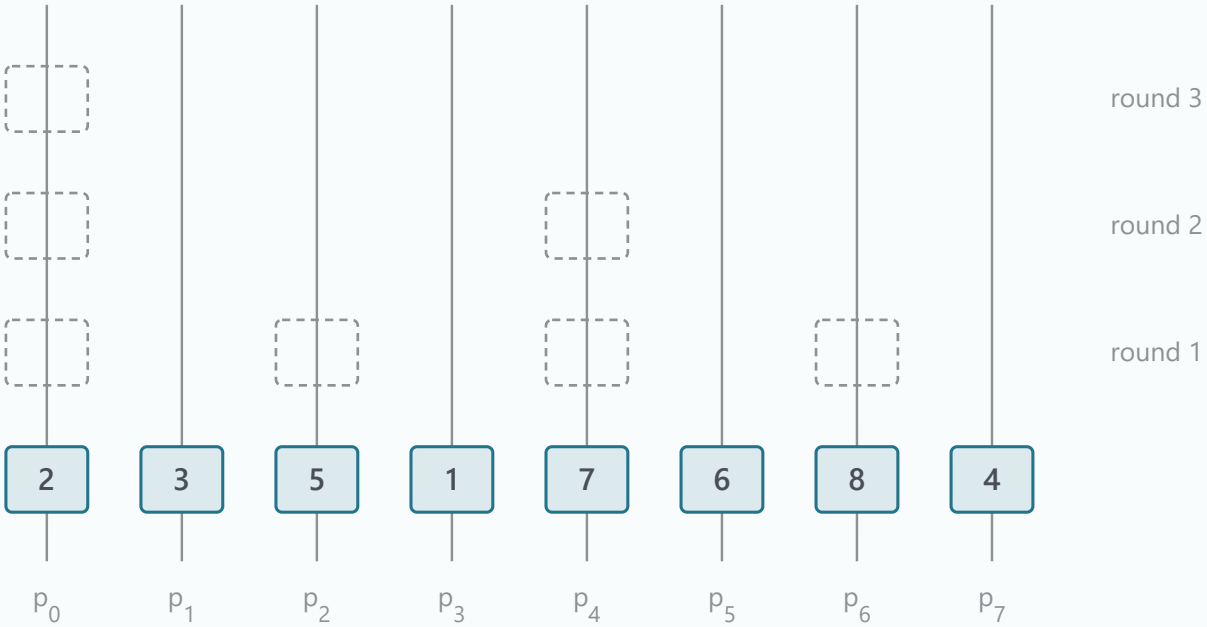
Problem: Parallel Summation

- Input: n numbers; output: their sum. The sequential algorithm performs n − 1 additions: **O(n)**.
- In the sequential loop each addition waits for the previous one: a chain of length n − 1, no parallelism.
- Since addition is **associative**, we can regroup the parentheses:
  ((2 + 3) + (5 + 1)) + ((7 + 6) + (8 + 4)) → a balanced **binary tree**.
- The sums on each level of the tree are independent → done in parallel in the same round.
- The same idea works for any associative operation: **max, min, product, AND, OR** (reduction).



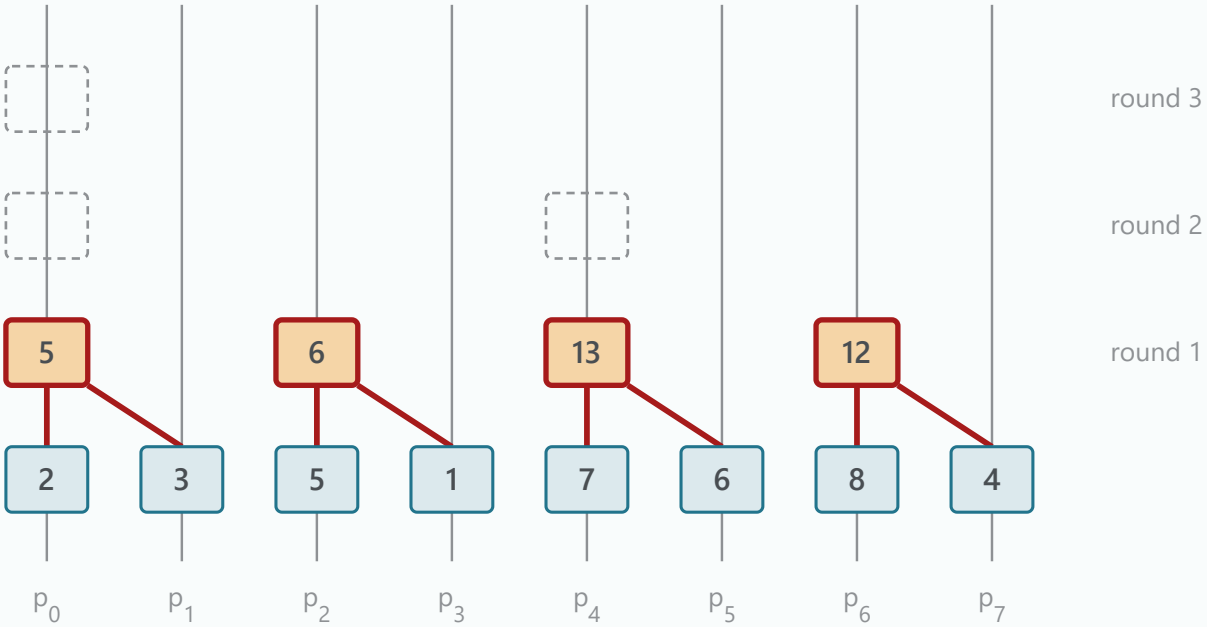
Example array: 2, 3, 5, 1, 7, 6, 8, 4 → sum 36. Eight processors p_0 … p_7 hold one number each.

PRAM Summation Step by Step



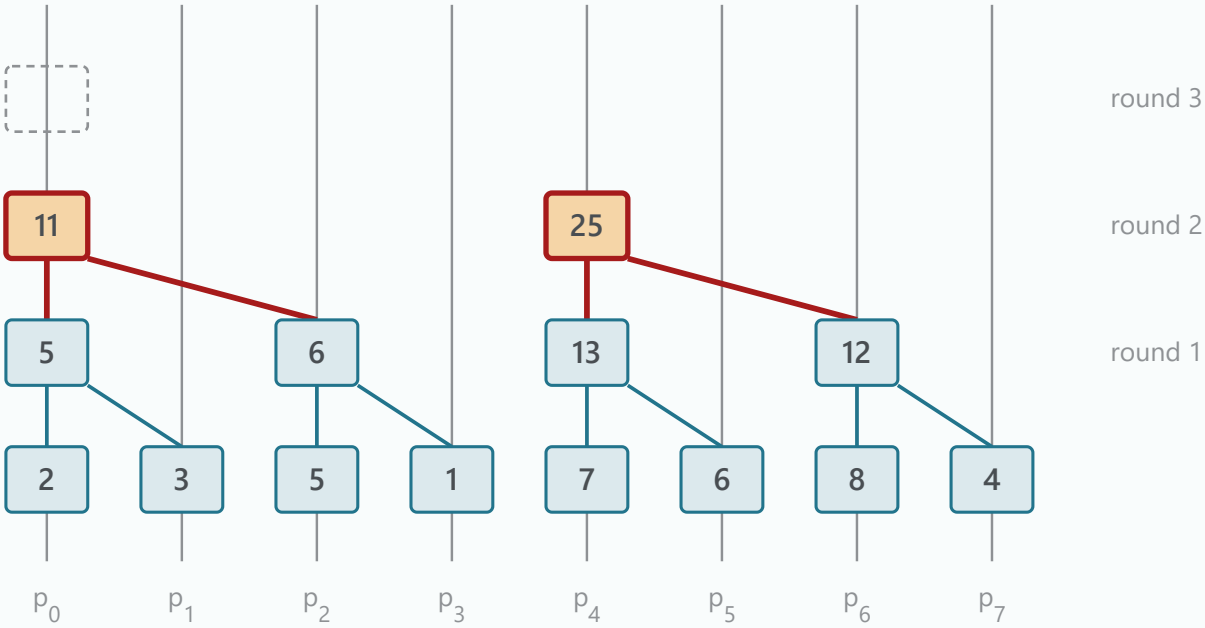
Start: the 8 numbers are distributed one per processor (p_0 … p_7). Dashed boxes are the cells written in later rounds.

PRAM Summation Step by Step



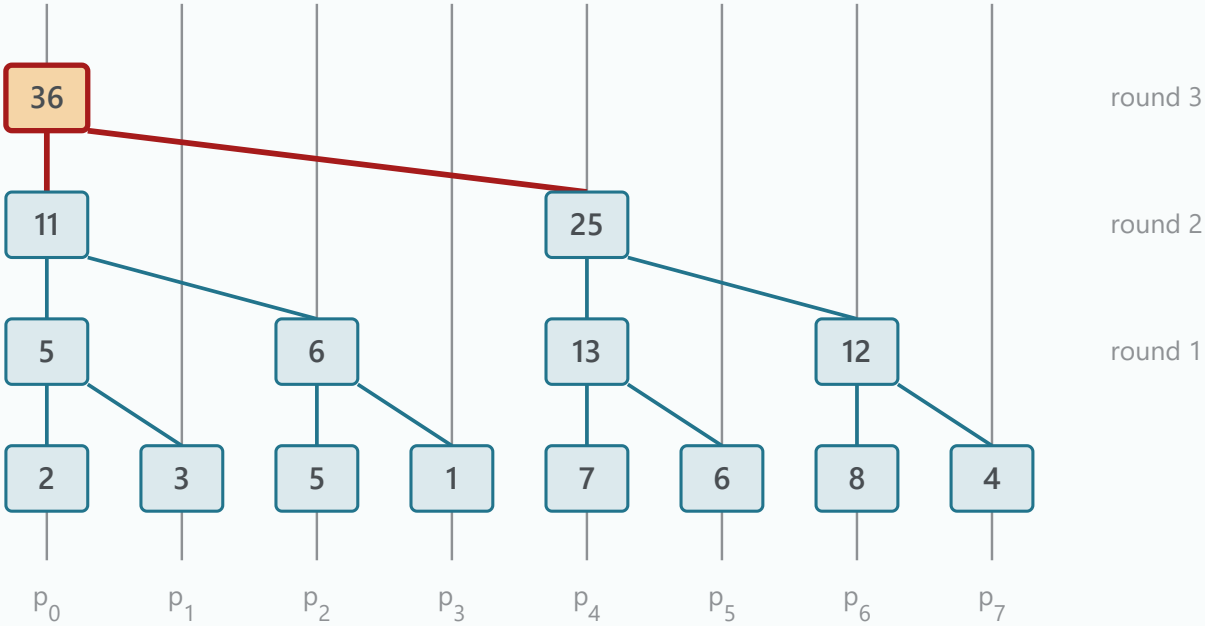
Round 1: 4 processors add adjacent pairs *simultaneously*: 2+3=5, 5+1=6, 7+6=13, 8+4=12.

PRAM Summation Step by Step



Round 2: 2 processors combine partial sums at distance 2: 5+6=11, 13+12=25.

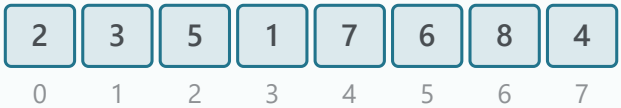
PRAM Summation Step by Step



Round 3: p_0 computes 11+25 = **36**. The 8 numbers were summed in $\log_2 8$ = **3 rounds** (sequential: 7 steps).

Tree Summation: Code Trace

```
▶ int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
▶ int n = a.length;
3 for (int step = 1; step < n; step *= 2) {
4   // all i in the same round run in parallel
5   for (int i = 0; i + step < n; i += 2 *
  step)
6     a[i] = a[i] + a[i + step];
7 }  // end of round: synchronization
8 // the sum is in a[0]
```



The array is created; `n = 8`. Each cell is held by one processor.

Step 1 / 12

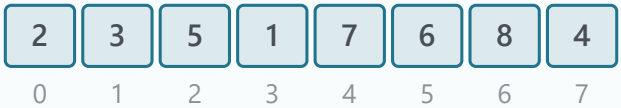
VARIABLES

`n = 8` `step = –` `i = –`

`a = [2, 3, 5, 1, 7, 6, 8, 4]`

Tree Summation: Code Trace

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
▶ for (int step = 1; step < n; step *= 2) {
4   // all i in the same round run in parallel
5   for (int i = 0; i + step < n; i += 2 *
  step)
6     a[i] = a[i] + a[i + step];
7 }  // end of round: synchronization
8 // the sum is in a[0]
```



New round: `step = 1`. This round performs 4 additions **in parallel**.

Step 2 / 12

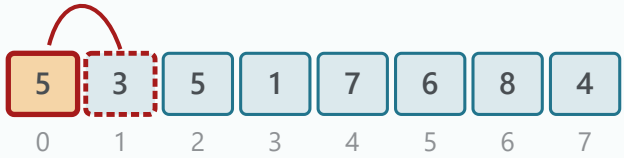
VARIABLES

`n = 8` `step = 1` `i = –`

`a = [2, 3, 5, 1, 7, 6, 8, 4]`

Tree Summation: Code Trace

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
3 for (int step = 1; step < n; step *= 2) {
4   // all i in the same round run in parallel
▶   for (int i = 0; i + step < n; i += 2 *
  step)
▶     a[i] = a[i] + a[i + step];
7 }  // end of round: synchronization
8 // the sum is in a[0]
```



`a[0] = a[0] + a[1]` = 2 + 3 = **5**.

Step 3 / 12

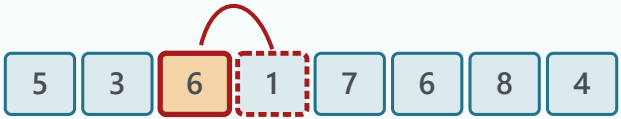
VARIABLES

`n = 8` `step = 1` `i = 0`

`a = [5, 3, 5, 1, 7, 6, 8, 4]`

Tree Summation: Code Trace

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
3 for (int step = 1; step < n; step *= 2) {
4   // all i in the same round run in parallel
▶   for (int i = 0; i + step < n; i += 2 *
    step)
▶     a[i] = a[i] + a[i + step];
7 }  // end of round: synchronization
8 // the sum is in a[0]
```



`a[2] = a[2] + a[3]` = 5 + 1 = **6**.

Step 4 / 12

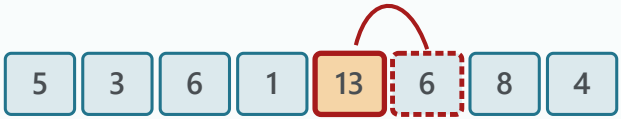
VARIABLES

`n = 8` `step = 1` `i = 2`

`a = [5, 3, 6, 1, 7, 6, 8, 4]`

Tree Summation: Code Trace

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
3 for (int step = 1; step < n; step *= 2) {
4   // all i in the same round run in parallel
▶   for (int i = 0; i + step < n; i += 2 *
    step)
▶     a[i] = a[i] + a[i + step];
7 }  // end of round: synchronization
8 // the sum is in a[0]
```



`a[4] = a[4] + a[5]` = 7 + 6 = **13**.

Step 5 / 12

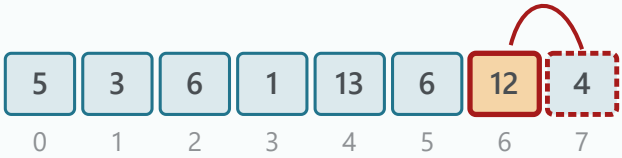
VARIABLES

`n = 8` `step = 1` `i = 4`

`a = [5, 3, 6, 1, 13, 6, 8, 4]`

Tree Summation: Code Trace

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
3 for (int step = 1; step < n; step *= 2) {
4   // all i in the same round run in parallel
▶   for (int i = 0; i + step < n; i += 2 *
    step)
▶     a[i] = a[i] + a[i + step];
7 }  // end of round: synchronization
8 // the sum is in a[0]
```



`a[6] = a[6] + a[7]` = 8 + 4 = **12**.

Step 6 / 12

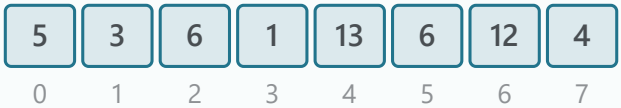
VARIABLES

`n = 8` `step = 1` `i = 6`

`a = [5, 3, 6, 1, 13, 6, 12, 4]`

Tree Summation: Code Trace

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
▶ for (int step = 1; step < n; step *= 2) {
4   // all i in the same round run in parallel
5   for (int i = 0; i + step < n; i += 2 *
  step)
6     a[i] = a[i] + a[i + step];
7 }  // end of round: synchronization
8 // the sum is in a[0]
```



New round: `step = 2`. This round performs 2 additions **in parallel**.

Step 7 / 12

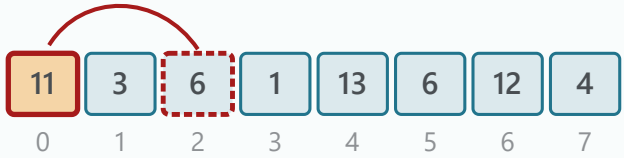
VARIABLES

`n = 8` `step = 2` `i = –`

`a = [5, 3, 6, 1, 13, 6, 12, 4]`

Tree Summation: Code Trace

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
3 for (int step = 1; step < n; step *= 2) {
4   // all i in the same round run in parallel
▶   for (int i = 0; i + step < n; i += 2 *
    step)
▶     a[i] = a[i] + a[i + step];
7 }  // end of round: synchronization
8 // the sum is in a[0]
```



`a[0] = a[0] + a[2]` = 5 + 6 = **11**.

Step 8 / 12

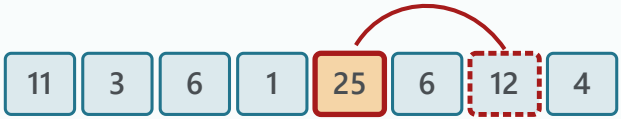
VARIABLES

`n = 8` `step = 2` `i = 0`

`a = [11, 3, 6, 1, 13, 6, 12, 4]`

Tree Summation: Code Trace

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
3 for (int step = 1; step < n; step *= 2) {
4   // all i in the same round run in parallel
▶   for (int i = 0; i + step < n; i += 2 *
    step)
▶     a[i] = a[i] + a[i + step];
7 }  // end of round: synchronization
8 // the sum is in a[0]
```



`a[4] = a[4] + a[6]` = 13 + 12 = **25**.

Step 9 / 12

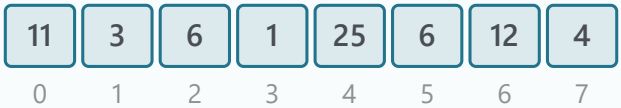
VARIABLES

`n = 8` `step = 2` `i = 4`

`a = [11, 3, 6, 1, 25, 6, 12, 4]`

Tree Summation: Code Trace

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
▶ for (int step = 1; step < n; step *= 2) {
4   // all i in the same round run in parallel
5   for (int i = 0; i + step < n; i += 2 *
  step)
6     a[i] = a[i] + a[i + step];
7 }  // end of round: synchronization
8 // the sum is in a[0]
```



New round: `step = 4`. This round performs 1 addition **in parallel**.

Step 10 / 12

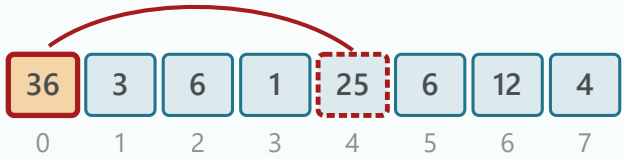
VARIABLES

`n = 8` `step = 4` `i = –`

`a = [11, 3, 6, 1, 25, 6, 12, 4]`

Tree Summation: Code Trace

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
3 for (int step = 1; step < n; step *= 2) {
4   // all i in the same round run in parallel
▶   for (int i = 0; i + step < n; i += 2 *
    step)
▶     a[i] = a[i] + a[i + step];
7 }  // end of round: synchronization
8 // the sum is in a[0]
```



`a[0] = a[0] + a[4]` = 11 + 25 = **36**.

Step 11 / 12

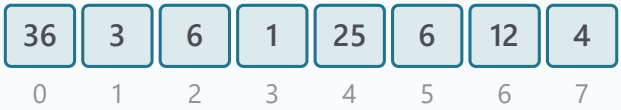
VARIABLES

`n = 8` `step = 4` `i = 0`

`a = [36, 3, 6, 1, 25, 6, 12, 4]`

Tree Summation: Code Trace

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
3 for (int step = 1; step < n; step *= 2) {
4   // all i in the same round run in parallel
5   for (int i = 0; i + step < n; i += 2 *
  step)
6     a[i] = a[i] + a[i + step];
7 }  // end of round: synchronization
▶ // the sum is in a[0]
```



`step = 8` → the loop ends. The sum is `a[0] = 36`; 3 rounds, 7 additions.

Step 12 / 12

VARIABLES

`n = 8` `step = 8` `i = –`

`a = [36, 3, 6, 1, 25, 6, 12, 4]`

PRAM Summation: Analysis

Measure	n processors (n/2 suffice)	p = n / $\log_2$ n processors
Time T_p	$\log_2 n$	O(log n)
Work W	n − 1	n − 1
Cost $p \cdot T_p$	Θ(n log n)	Θ(n)
Cost-optimal?	no	**yes**

- As the rounds go on, the number of active processors halves (4, 2, 1): most processors sit idle, so the cost is high.
- Fix: each processor first sums its own block of $\log_2$ n numbers **sequentially** (O(log n)); then the n / $\log_2$ n partial sums are combined with a tree (O(log n)).
- EREW suffices: in each round every cell is read/written by a single processor. Each round ends with a **synchronization** (barrier).

Parallel Prefix Sums

- **Prefix sum** (scan): $y_i = x_0 + x_1 + \ldots + x_i$.
- Sequential: $y_i = y_{i-1} + x_i$, O(n) but fully chained.
- **Hillis–Steele**: for d = 1, 2, 4, …, every i ≥ d does x[i] += x[i − d] in parallel.
- D = $\lceil \log_2 n \rceil$ rounds, W = O(n log n) → not work-efficient.
- The **Blelloch** algorithm (up-sweep/down-sweep) has W = O(n), D = O(log n).
- Uses: parallel sorting, compaction (filter), `Arrays.parallelPrefix` in Java.

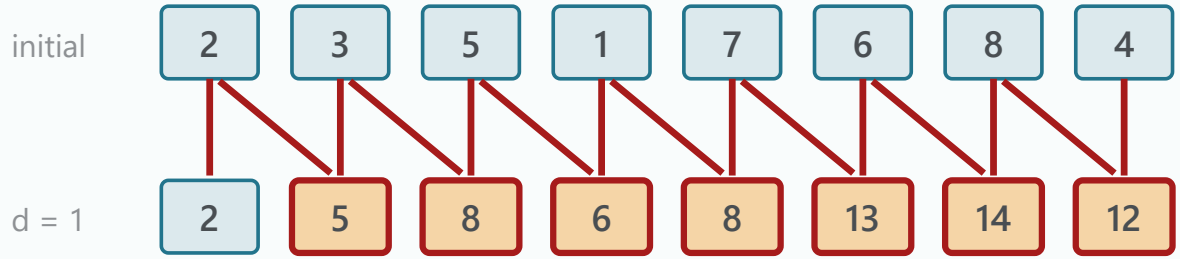
The summation tree yields only the last cell (the sum); in a prefix sum **every** cell must learn its own prefix.

Hillis–Steele Prefix Sums Step by Step



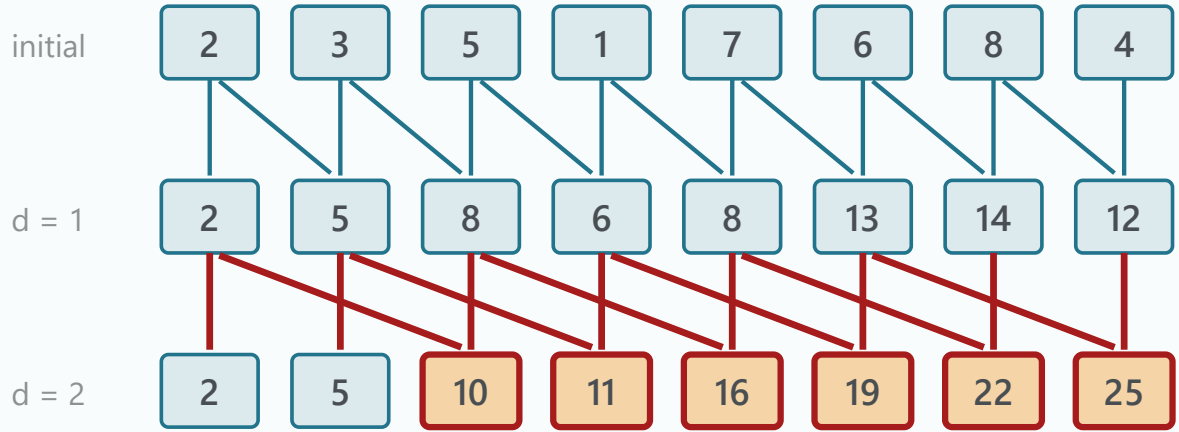
Input: 2 3 5 1 7 6 8 4. Goal: $x_0 + \ldots + x_i$ for every i (inclusive prefix sum).

Hillis–Steele Prefix Sums Step by Step



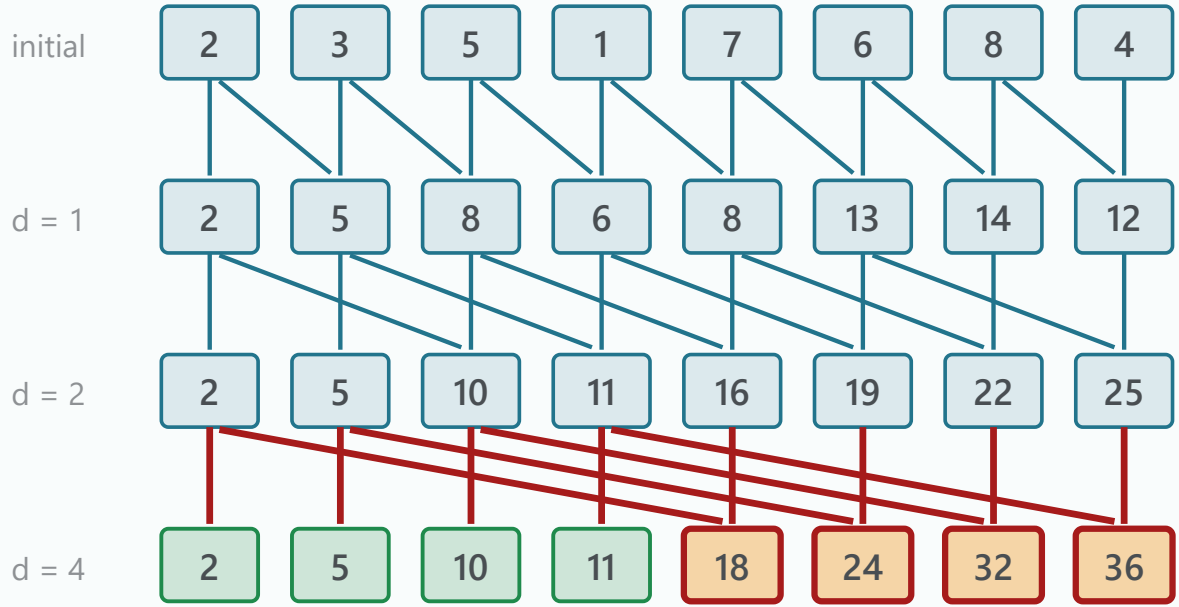
d = 1: every processor with i ≥ 1 adds its left neighbor's value (7 additions at once).

Hillis–Steele Prefix Sums Step by Step



d = 2: for i ≥ 2, add the value 2 positions to the left. Each cell now holds the sum of a 4-element window.

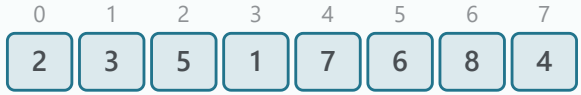
Hillis–Steele Prefix Sums Step by Step



d = 4: result 2 5 10 11 18 24 32 36. Done in $\log_2 8 = 3$ rounds, but with 7+6+4 = 17 additions in total.

Blelloch Prefix Sums: Code Trace

```
1 // up-sweep; x = 2 3 5 1 7 6 8 4
2 for (int d = 1; d < n; d *= 2)
3   for (int i = 2*d - 1; i < n; i += 2*d)
4     x[i] += x[i - d];
5 x[n - 1] = 0;
6 // down-sweep (inner loops in parallel)
7 for (int d = n / 2; d >= 1; d /= 2)
8   for (int i = 2*d - 1; i < n; i += 2*d) {
9     int t = x[i - d];
10    x[i - d] = x[i];
11    x[i] += t;
12  }
13 // x: exclusive prefix sums
```



input

Input 2 3 5 1 7 6 8 4 (n = 8). The array serves as an implicit binary tree: first up (reduction), then down.

Step 1 / 9

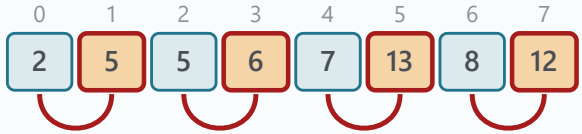
VARIABLES

d = –

x = [2, 3, 5, 1, 7, 6, 8, 4]

Blelloch Prefix Sums: Code Trace

```
1  // up-sweep; x = 2 3 5 1 7 6 8 4
▶  for (int d = 1; d < n; d *= 2)
▶    for (int i = 2*d - 1; i < n; i += 2*d)
▶      x[i] += x[i - d];
5  x[n - 1] = 0;
6  // down-sweep (inner loops in parallel)
7  for (int d = n / 2; d >= 1; d /= 2)
8    for (int i = 2*d - 1; i < n; i += 2*d) {
9      int t = x[i - d];
10     x[i - d] = x[i];
11     x[i] += t;
12   }
13 // x: exclusive prefix sums
```



up-sweep, d = 1

d = 1: x[1] += x[0], x[3] += x[2], x[5] += x[4], x[7] += x[6] (in parallel in one round, 4 additions).

Step 2 / 9

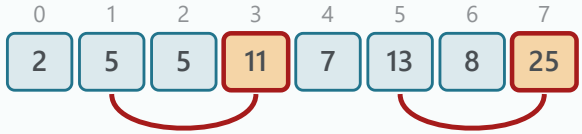
VARIABLES

d = 1

x = [2, 5, 5, 6, 7, 13, 8, 12]

Blelloch Prefix Sums: Code Trace

```
1  // up-sweep; x = 2 3 5 1 7 6 8 4
▶  for (int d = 1; d < n; d *= 2)
▶    for (int i = 2*d - 1; i < n; i += 2*d)
▶      x[i] += x[i - d];
5  x[n - 1] = 0;
6  // down-sweep (inner loops in parallel)
7  for (int d = n / 2; d >= 1; d /= 2)
8    for (int i = 2*d - 1; i < n; i += 2*d) {
9      int t = x[i - d];
10     x[i - d] = x[i];
11     x[i] += t;
12   }
13 // x: exclusive prefix sums
```



up-sweep, d = 2

d = 2: x[3] += x[1], x[7] += x[5] (in parallel in one round, 2 additions).

Step 3 / 9

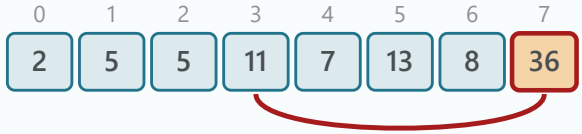
VARIABLES

d = **2**

x = **[2, 5, 5, 11, 7, 13, 8, 25]**

Blelloch Prefix Sums: Code Trace

```
1  // up-sweep; x = 2 3 5 1 7 6 8 4
▶  for (int d = 1; d < n; d *= 2)
▶    for (int i = 2*d - 1; i < n; i += 2*d)
▶      x[i] += x[i - d];
5  x[n - 1] = 0;
6  // down-sweep (inner loops in parallel)
7  for (int d = n / 2; d >= 1; d /= 2)
8    for (int i = 2*d - 1; i < n; i += 2*d) {
9      int t = x[i - d];
10     x[i - d] = x[i];
11     x[i] += t;
12   }
13 // x: exclusive prefix sums
```



up-sweep, d = 4

d = 4: x[7] += x[3] (in parallel in one round, 1 addition). The root x[7] = 36 is the total.

Step 4 / 9

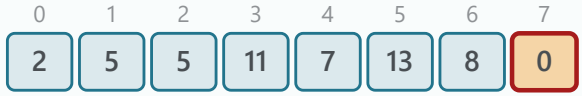
VARIABLES

d = 4

x = [2, 5, 5, 11, 7, 13, 8, 36]

Blelloch Prefix Sums: Code Trace

```
1  // up-sweep; x = 2 3 5 1 7 6 8 4
2  for (int d = 1; d < n; d *= 2)
3    for (int i = 2*d - 1; i < n; i += 2*d)
4      x[i] += x[i - d];
▶  x[n - 1] = 0;
6  // down-sweep (inner loops in parallel)
7  for (int d = n / 2; d >= 1; d /= 2)
8    for (int i = 2*d - 1; i < n; i += 2*d) {
9      int t = x[i - d];
10     x[i - d] = x[i];
11     x[i] += t;
12   }
13 // x: exclusive prefix sums
```



root set to 0

The total 36 is at the root. The root is set to 0: in the down-sweep every node carries "the sum of everything to its left".

Step 5 / 9

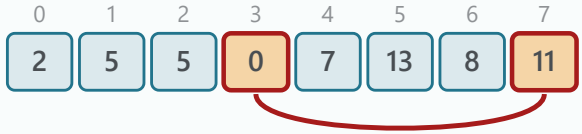
VARIABLES

`total = 36`

`x = [2, 5, 5, 11, 7, 13, 8, 0]`

Blelloch Prefix Sums: Code Trace

```
1  // up-sweep; x = 2 3 5 1 7 6 8 4
2  for (int d = 1; d < n; d *= 2)
3    for (int i = 2*d - 1; i < n; i += 2*d)
4      x[i] += x[i - d];
5  x[n - 1] = 0;
6  // down-sweep (inner loops in parallel)
▶  for (int d = n / 2; d >= 1; d /= 2)
▶    for (int i = 2*d - 1; i < n; i += 2*d) {
▶      int t = x[i - d];
▶      x[i - d] = x[i];
▶      x[i] += t;
12   }
13 // x: exclusive prefix sums
```



down-sweep, d = 4

d = 4: in each pair (i − d, i), the left child takes the parent's value and the right child takes parent + the old left value.

Step 6 / 9

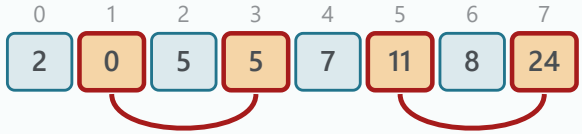
VARIABLES

d = 4

x = [2, 5, 5, 0, 7, 13, 8, 11]

Blelloch Prefix Sums: Code Trace

```
1  // up-sweep; x = 2 3 5 1 7 6 8 4
2  for (int d = 1; d < n; d *= 2)
3    for (int i = 2*d - 1; i < n; i += 2*d)
4      x[i] += x[i - d];
5  x[n - 1] = 0;
6  // down-sweep (inner loops in parallel)
▶  for (int d = n / 2; d >= 1; d /= 2)
▶    for (int i = 2*d - 1; i < n; i += 2*d) {
▶      int t = x[i - d];
▶      x[i - d] = x[i];
▶      x[i] += t;
12   }
13 // x: exclusive prefix sums
```



down-sweep, d = 2

d = 2: in each pair (i − d, i), the left child takes the parent's value and the right child takes parent + the old left value.

Step 7 / 9

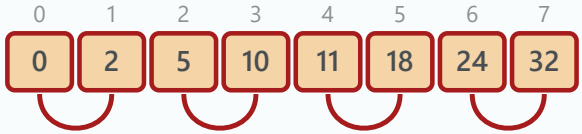
VARIABLES

d = 2

x = [2, 0, 5, 5, 7, 11, 8, 24]

Blelloch Prefix Sums: Code Trace

```
1  // up-sweep; x = 2 3 5 1 7 6 8 4
2  for (int d = 1; d < n; d *= 2)
3    for (int i = 2*d - 1; i < n; i += 2*d)
4      x[i] += x[i - d];
5  x[n - 1] = 0;
6  // down-sweep (inner loops in parallel)
▶  for (int d = n / 2; d >= 1; d /= 2)
▶    for (int i = 2*d - 1; i < n; i += 2*d) {
▶      int t = x[i - d];
▶      x[i - d] = x[i];
▶      x[i] += t;
12   }
13 // x: exclusive prefix sums
```



down-sweep, d = 1

d = 1: the last round; 4 pairs are updated at once.

Step 8 / 9

VARIABLES

`d = 1`

`x = [0, 2, 5, 10, 11, 18, 24, 32]`

Blelloch Prefix Sums: Code Trace

```
1  // up-sweep; x = 2 3 5 1 7 6 8 4
2  for (int d = 1; d < n; d *= 2)
3    for (int i = 2*d - 1; i < n; i += 2*d)
4      x[i] += x[i - d];
5  x[n - 1] = 0;
6  // down-sweep (inner loops in parallel)
7  for (int d = n / 2; d >= 1; d /= 2)
8    for (int i = 2*d - 1; i < n; i += 2*d) {
9      int t = x[i - d];
10     x[i - d] = x[i];
11     x[i] += t;
12   }
▶ // x: exclusive prefix sums
```



result (exclusive prefix)

Result 0 2 5 10 11 18 24 32: the exclusive prefix sums. $2 \cdot (n - 1) = 14$ operations, O(n) work, depth $2 \log_2 n = 6$ rounds.

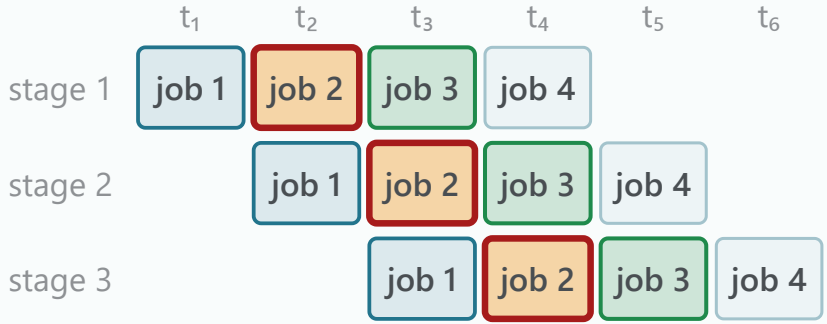
Step 9 / 9

VARIABLES

x = [0, 2, 5, 10, 11, 18, 24, 32]

The Idea of Pipelining

- A **pipeline** splits the work into consecutive **stages**; each stage is a processor.
- Like a factory assembly line: while one job moves to stage 2, stage 1 starts on the next job.
- A single job does not get faster, but the **number of jobs finished per unit time** (throughput) increases.
- k stages, m jobs: k · m steps sequentially; **k + m − 1** steps with a pipeline.
- Some stages are idle while the pipeline fills at the start and drains at the end.

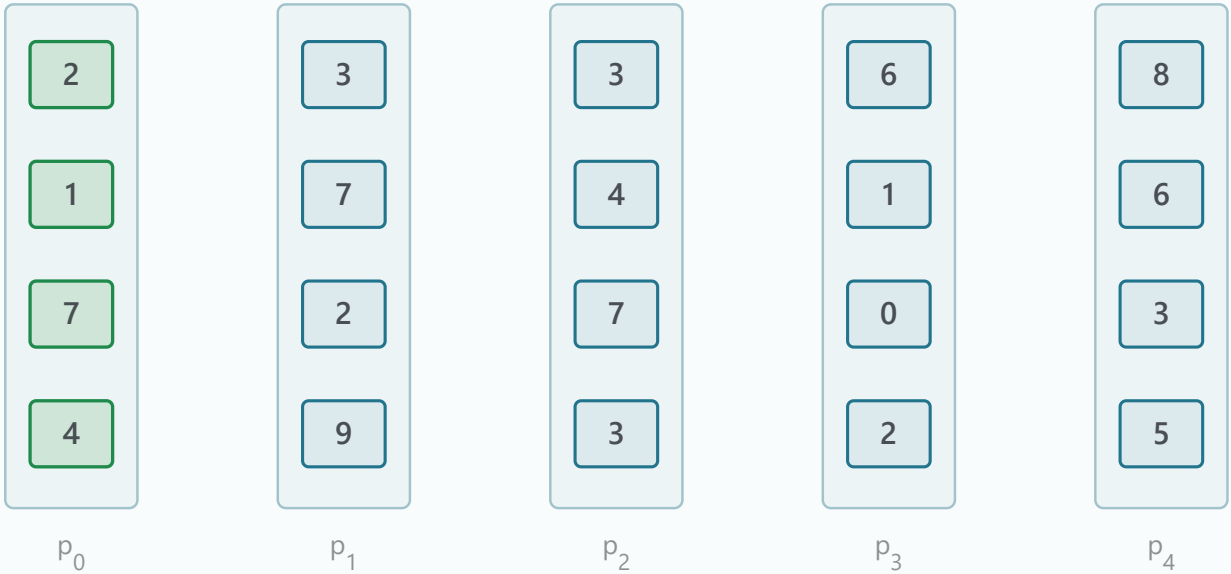


3 stages, 4 jobs: 3 + 4 − 1 = 6 steps instead of 3 · 4 = 12.

The Pipeline Problem

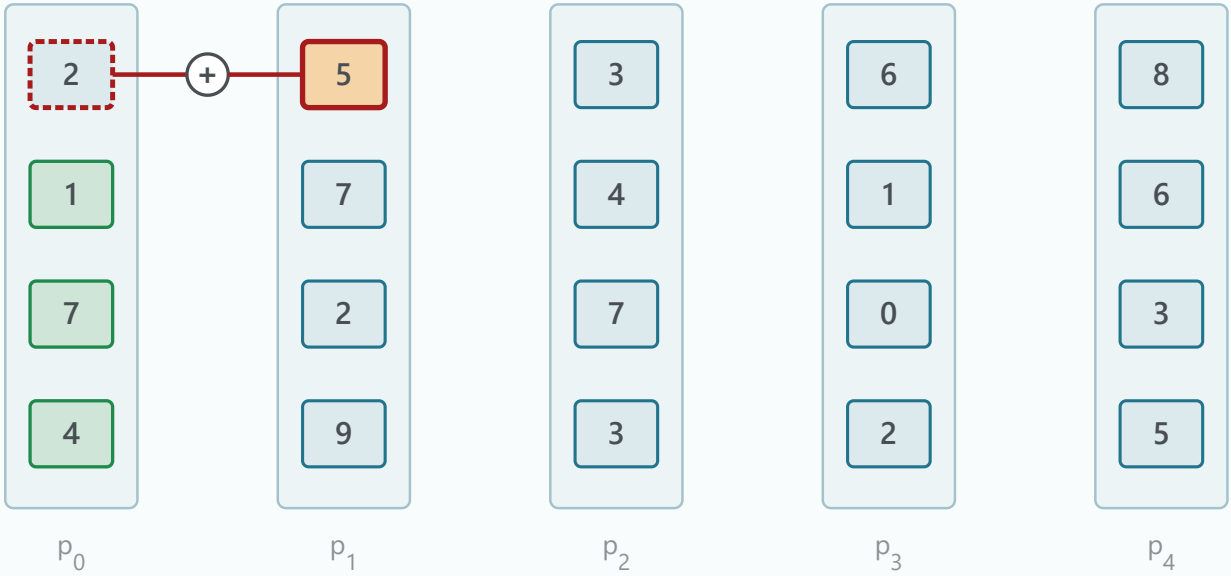
- Five processors $p_0 \dots p_4$ form a linear chain; each processor can send messages only to its **right neighbor**.
- Each p_j holds a column of 4 values (rows 0 … 3).
- Goal: for every row r, the value at p_j should become the **sum** of the values of $p_0 \dots p_j$ in that row (row-wise prefix sums). Then p_4 holds the sum of every row.
- In one step each processor sends **one** value; the receiver adds it to its own.
- Naive method: first move row 0 along the chain (4 steps), then row 1… → $4 \cdot 4 =$ **16 steps**.
- Pipeline: p_j sends row r at time **t = r + j**; the rows flow through the chain one after another.

Pipelining Step by Step: 5 Processors, 4 Rows



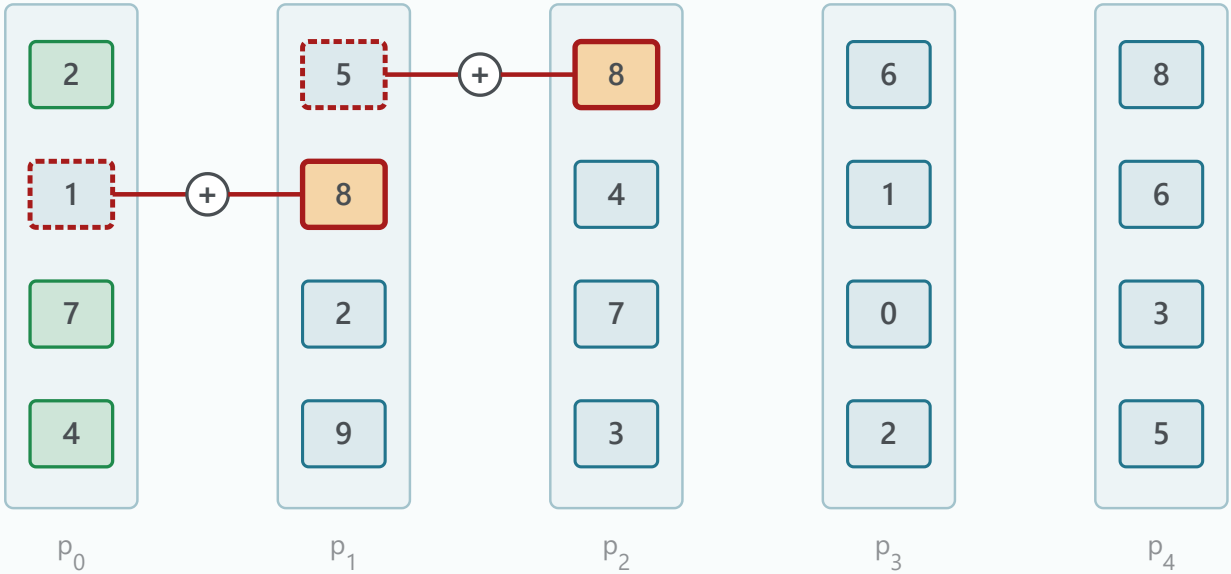
Each p_j holds a 4-row column. Goal: in every row, p_j holds the sum of the values of p_0 … p_j. p_0's values are already final.

Pipelining Step by Step: 5 Processors, 4 Rows



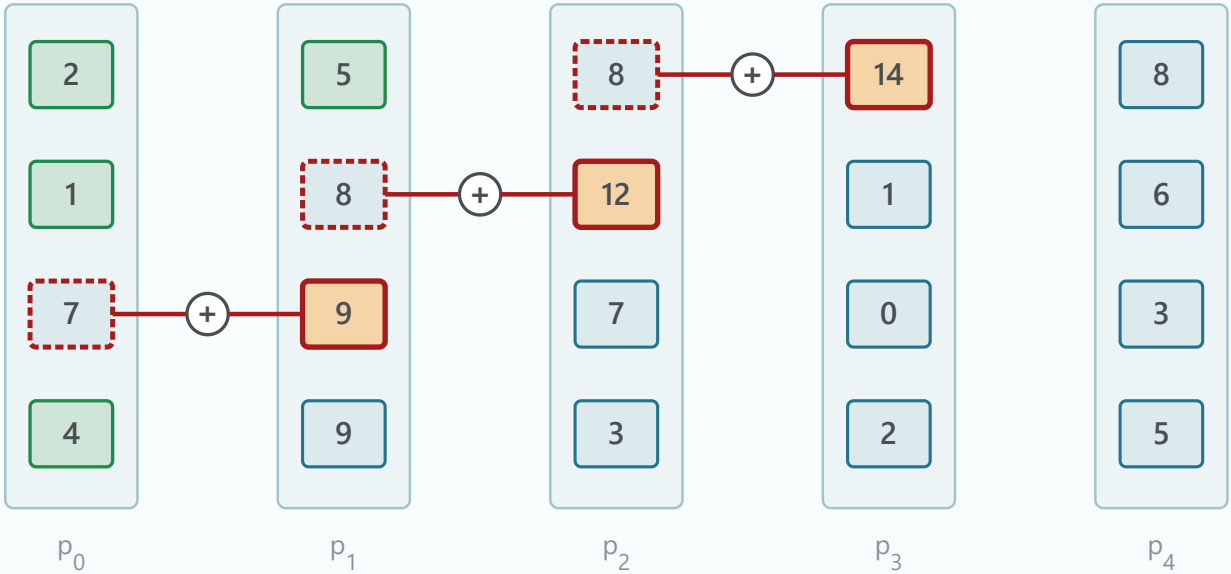
Step 1: $p_0 \rightarrow p_1$ row 0: 2+3=5.

Pipelining Step by Step: 5 Processors, 4 Rows



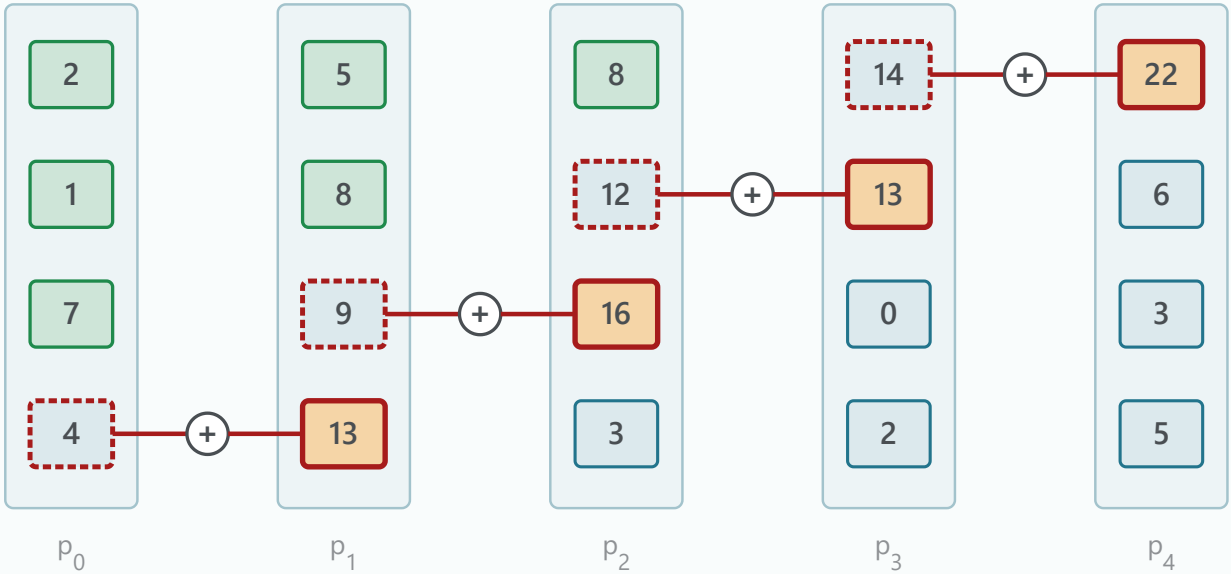
Step 2: $p_0 \to p_1$ row 1: 1+7=8, $p_1 \to p_2$ row 0: 5+3=8.

Pipelining Step by Step: 5 Processors, 4 Rows



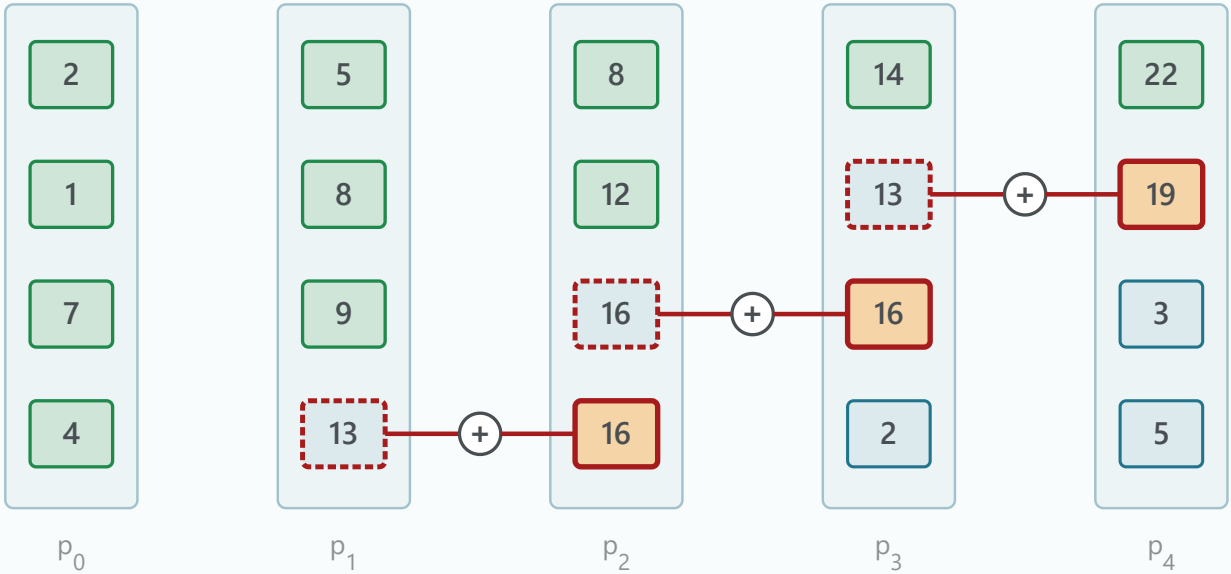
Step 3: $p_0 \to p_1$ row 2: 7+2=9, $p_1 \to p_2$ row 1: 8+4=12, $p_2 \to p_3$ row 0: 8+6=14.

Pipelining Step by Step: 5 Processors, 4 Rows



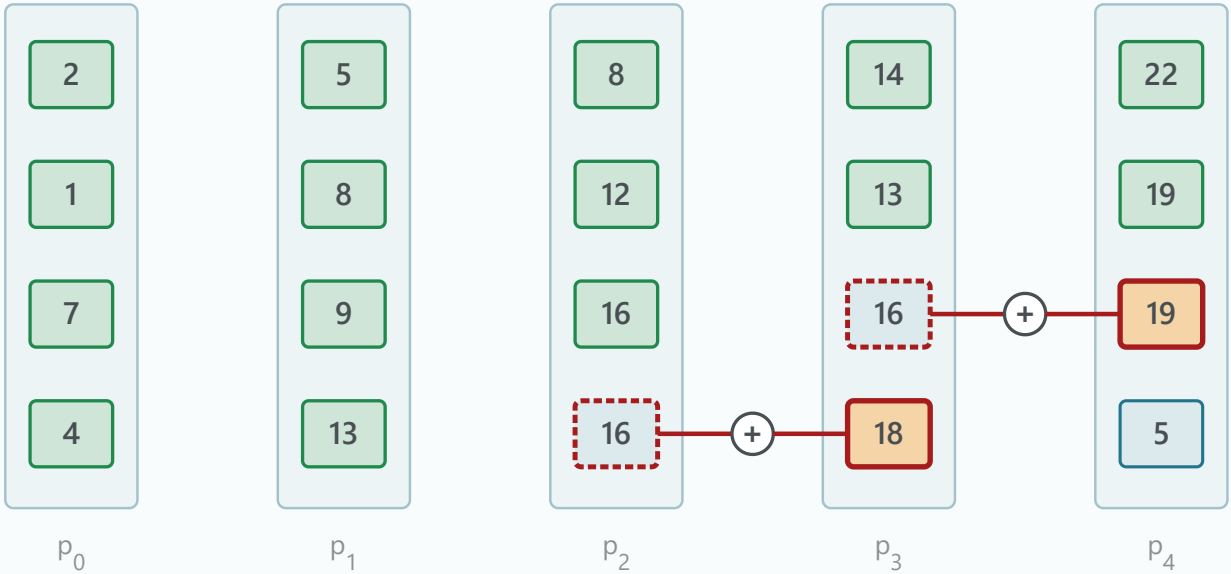
Step 4: $p_0 \to p_1$ row 3: 4+9=13, $p_1 \to p_2$ row 2: 9+7=16, $p_2 \to p_3$ row 1: 12+1=13, $p_3 \to p_4$ row 0: 14+8=22.

Pipelining Step by Step: 5 Processors, 4 Rows



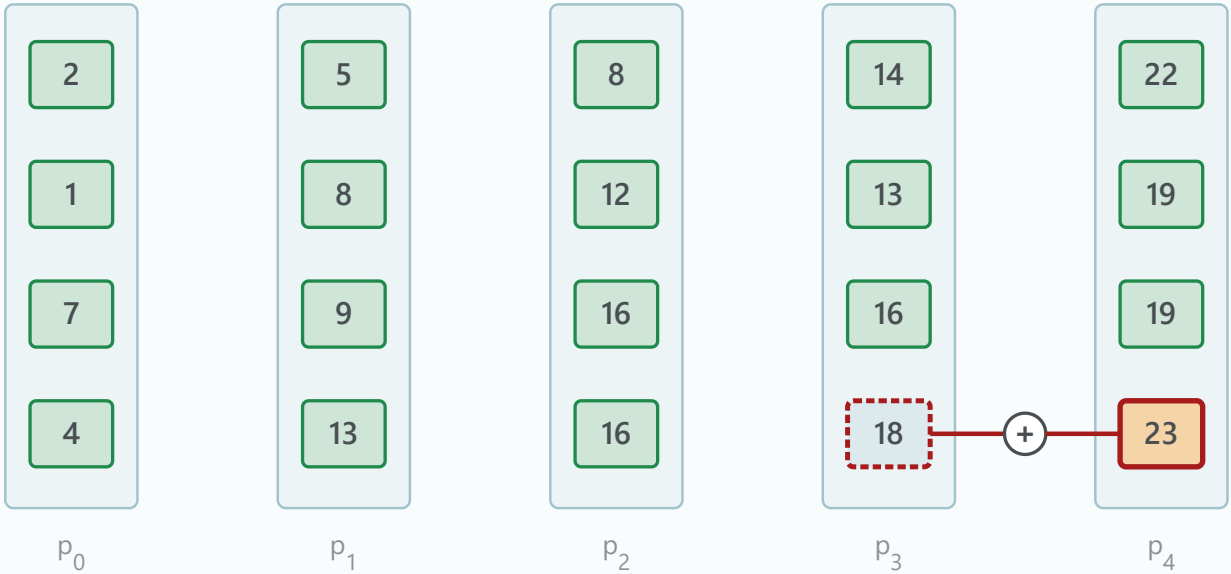
Step 5: $p_1 \to p_2$ row 3: 13+3=16, $p_2 \to p_3$ row 2: 16+0=16, $p_3 \to p_4$ row 1: 13+6=19.

Pipelining Step by Step: 5 Processors, 4 Rows



Step 6: $p_2 \to p_3$ row 3: 16+2=18, $p_3 \to p_4$ row 2: 16+3=19.

Pipelining Step by Step: 5 Processors, 4 Rows



Step 7: $p_3 \rightarrow p_4$ row 3: 18+5=23. Done in **7 steps**; p_4 holds the row sums (22 19 19 23).

Pipelining: Analysis

step	$p_0 \to p_1$	$p_1 \to p_2$	$p_2 \to p_3$	$p_3 \to p_4$
1	row 0	—	—	—
2	row 1	row 0	—	—
3	row 2	row 1	row 0	—
4	row 3	row 2	row 1	row 0
5	—	row 3	row 2	row 1
6	—	—	row 3	row 2
7	—	—	—	row 3

```
for (int t = 0; t < m + p - 2; t++)
  // all j at the same time
  for (int j = 0; j < p - 1; j++) {
    int r = t - j;
    if (r >= 0 && r < m)
      x[j + 1][r] += x[j][r];
  }
```

- Number of steps: **m + p − 2** = 4 + 5 − 2 = 7 (naive: m(p − 1) = 16).
- Efficiency: m(p − 1) additions / ((p − 1)(m + p − 2)) = m / (m + p − 2) = 4/7 ≈ 57%.
- For m ≫ p the efficiency → 1.

Pipelining: Code Trace

```
1 ▶ for (int t = 0; t < m + p - 2; t++)
2     // all j at the same time
3     for (int j = 0; j < p - 1; j++) {
4       int r = t - j;
5       if (r >= 0 && r < m)
6         x[j + 1][r] += x[j][r];
7     }
```

	p_0	p_1	p_2	p_3	p_4
0	2	3	3	6	8
1	1	7	4	1	6
2	7	2	7	0	3
3	4	9	3	2	5

p = 5 processors, m = 4 rows. Each column holds one processor's values; row r reaches p_j at time t = r + j.

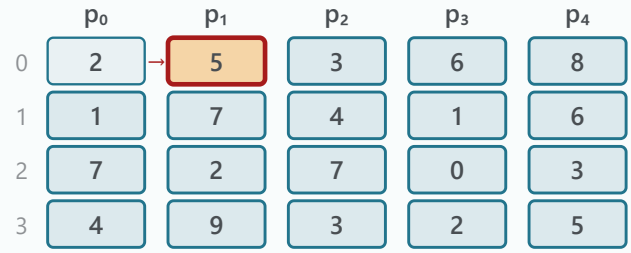
Step 1 / 9

VARIABLES

t = –

Pipelining: Code Trace

```
1 for (int t = 0; t < m + p - 2; t++)
2   // all j at the same time
    for (int j = 0; j < p - 1; j++) {
      int r = t - j;
      if (r >= 0 && r < m)
        x[j + 1][r] += x[j][r];
7   }
```



t = 0: $p_0 \rightarrow p_1$ row 0: 2 + 3 = 5.

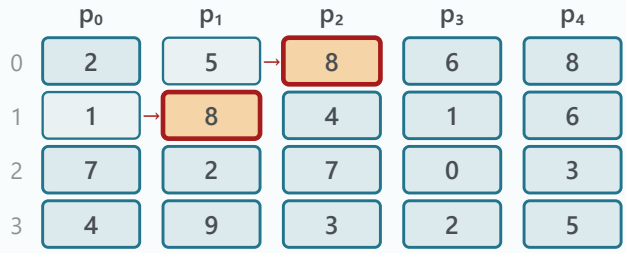
Step 2 / 9

VARIABLES

`t = 0` `p4 = [8, 6, 3, 5]`

Pipelining: Code Trace

```
1 for (int t = 0; t < m + p - 2; t++)
2   // all j at the same time
    for (int j = 0; j < p - 1; j++) {
      int r = t - j;
      if (r >= 0 && r < m)
        x[j + 1][r] += x[j][r];
7   }
```



t = 1: $p_0 \to p_1$ row 1: 1 + 7 = 8; $p_1 \to p_2$ row 0: 5 + 3 = 8.

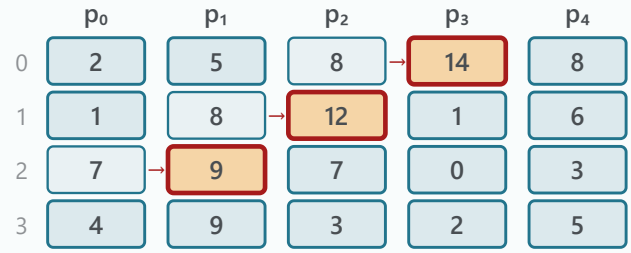
Step 3 / 9

VARIABLES

`t = 1` `p4 = [8, 6, 3, 5]`

Pipelining: Code Trace

```
1 for (int t = 0; t < m + p - 2; t++)
2   // all j at the same time
▶   for (int j = 0; j < p - 1; j++) {
▶     int r = t - j;
▶     if (r >= 0 && r < m)
▶       x[j + 1][r] += x[j][r];
7   }
```



t = 2: $p_0 \to p_1$ row 2: 7 + 2 = 9; $p_1 \to p_2$ row 1: 8 + 4 = 12; $p_2 \to p_3$ row 0: 8 + 6 = 14.

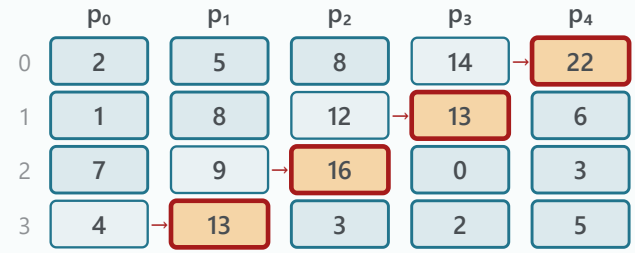
Step 4 / 9

VARIABLES

`t = 2` `p4 = [8, 6, 3, 5]`

Pipelining: Code Trace

```
1 for (int t = 0; t < m + p - 2; t++)
2   // all j at the same time
▶   for (int j = 0; j < p - 1; j++) {
▶     int r = t - j;
▶     if (r >= 0 && r < m)
▶       x[j + 1][r] += x[j][r];
7   }
```



t = 3: $p_0 \to p_1$ row 3: 4 + 9 = 13; $p_1 \to p_2$ row 2: 9 + 7 = 16; $p_2 \to p_3$ row 1: 12 + 1 = 13; $p_3 \to p_4$ row 0: 14 + 8 = 22. The pipeline is full: all 4 links are busy.

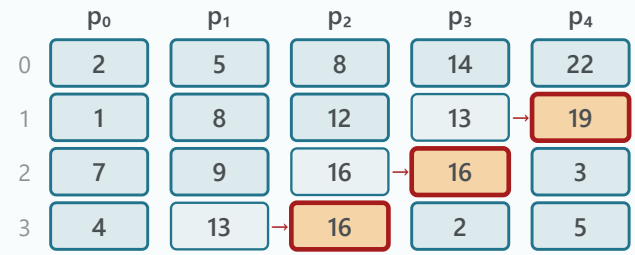
Step 5 / 9

VARIABLES

`t = 3` `p₄ = [22, 6, 3, 5]`

Pipelining: Code Trace

```
1 for (int t = 0; t < m + p - 2; t++)
2   // all j at the same time
▶   for (int j = 0; j < p - 1; j++) {
▶     int r = t - j;
▶     if (r >= 0 && r < m)
▶       x[j + 1][r] += x[j][r];
7   }
```



t = 4: $p_1 \to p_2$ row 3: 13 + 3 = 16; $p_2 \to p_3$ row 2: 16 + 0 = 16; $p_3 \to p_4$ row 1: 13 + 6 = 19.

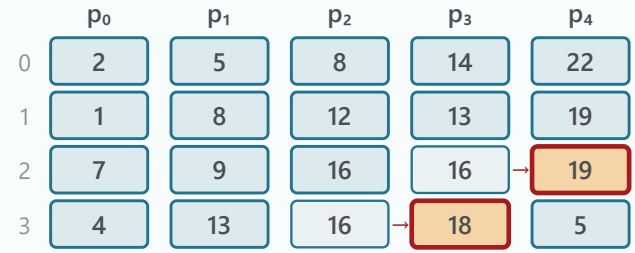
Step 6 / 9

VARIABLES

`t = 4` `p4 = [22, 19, 3, 5]`

Pipelining: Code Trace

```
1 for (int t = 0; t < m + p - 2; t++)
2   // all j at the same time
▶   for (int j = 0; j < p - 1; j++) {
▶     int r = t - j;
▶     if (r >= 0 && r < m)
▶       x[j + 1][r] += x[j][r];
7   }
```



t = 5: $p_2 \rightarrow p_3$ row 3: 16 + 2 = 18; $p_3 \rightarrow p_4$ row 2: 16 + 3 = 19.

Step 7 / 9

VARIABLES

`t = 5` `p4 = [22, 19, 19, 5]`

Pipelining: Code Trace

```
1 for (int t = 0; t < m + p - 2; t++)
2   // all j at the same time
    for (int j = 0; j < p - 1; j++) {
      int r = t - j;
      if (r >= 0 && r < m)
        x[j + 1][r] += x[j][r];
7   }
```



t = 6: $p_3 \rightarrow p_4$ row 3: 18 + 5 = 23.

Step 8 / 9

VARIABLES

`t = 6` `p4 = [22, 19, 19, 23]`

Pipelining: Code Trace

```
▶ for (int t = 0; t < m + p - 2; t++)
2   // all j at the same time
3   for (int j = 0; j < p - 1; j++) {
4     int r = t - j;
5     if (r >= 0 && r < m)
6       x[j + 1][r] += x[j][r];
7   }
```

	p_0	p_1	p_2	p_3	p_4
0	2	5	8	14	22
1	1	8	12	13	19
2	7	9	16	16	19
3	4	13	16	18	23

Done in m + p − 2 = 7 steps; p_4 holds the row sums 22 19 19 23. The naive method would take 16 steps.

Step 9 / 9

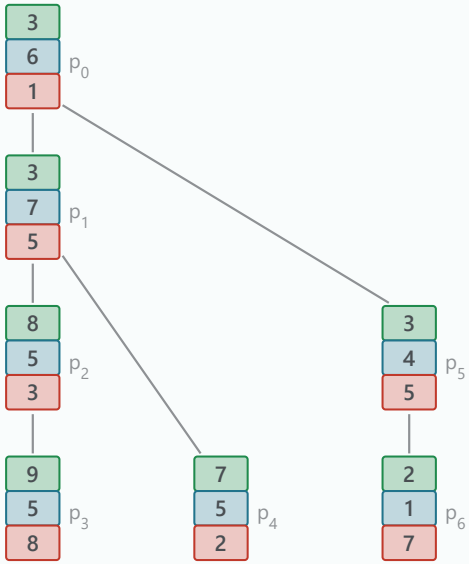
VARIABLES

`t = 7 (done)` `p₄ = [22, 19, 19, 23]`

Reduction on a Tree

- The processors are connected as a **tree**; each node sends its value to its **parent**, which adds it to its own. The result accumulates at the root.
- If the tree has depth h, reducing a single value takes h rounds (h = 3 in our example).
- Now each processor holds not one number but a **vector with k components** (k = 3 here: green, blue, red). Each component is summed separately.
- Naive method: reduce the components one after another → $k \cdot h = 9$ rounds.
- **Pipelined tree**: while one component moves up, the next one sets out from the leaves.
- Rule: a node can **receive at most one message per round** (and may also send one in the same round).

Pipelined Tree Step by Step

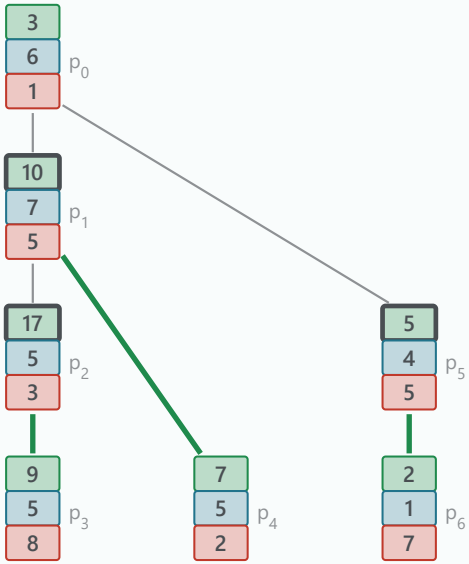


Each of the 7 processors holds a 3-component vector (green, blue, red). Goal: the sum of each component at the root p_0; values flow bottom-up.

Step 1 / 8

- Edge color = component carried this round; arrows point to the parent.
- Thick border = value updated this round.
- Each component reaches the root in 3 stages:
  (1) $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$
  (2) $p_2 \to p_1$, $p_5 \to p_0$
  (3) $p_1 \to p_0$.
- A new component starts **every 2 rounds**.

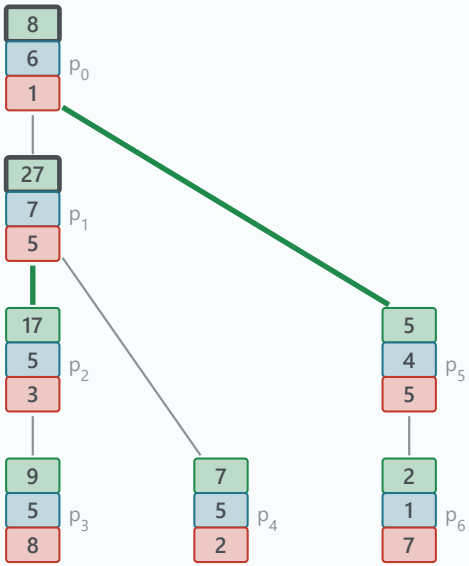
Pipelined Tree Step by Step



Round 1: green $p_3 \to p_2$: 9+8=17, $p_4 \to p_1$: 7+3=10, $p_6 \to p_5$: 2+3=5.

- Edge color = component carried this round; arrows point to the parent.
- Thick border = value updated this round.
- Each component reaches the root in 3 stages:
  (1) $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$
  (2) $p_2 \to p_1$, $p_5 \to p_0$
  (3) $p_1 \to p_0$.
- A new component starts **every 2 rounds**.

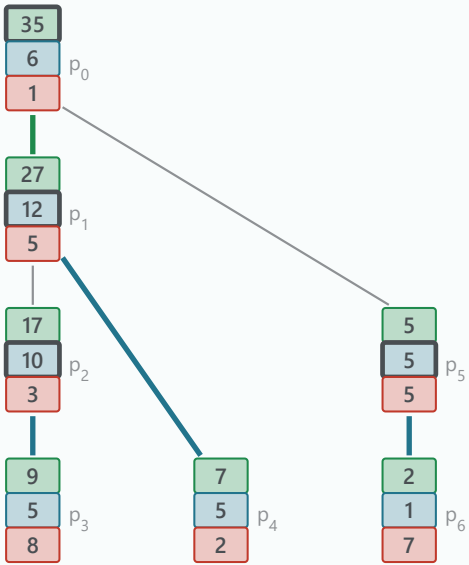
Pipelined Tree Step by Step



Round 2: green $p_2 \to p_1$: 17+10=27, $p_5 \to p_0$: 5+3=8.

- Edge color = component carried this round; arrows point to the parent.
- Thick border = value updated this round.
- Each component reaches the root in 3 stages:
  (1) $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$
  (2) $p_2 \to p_1$, $p_5 \to p_0$
  (3) $p_1 \to p_0$.
- A new component starts **every 2 rounds**.

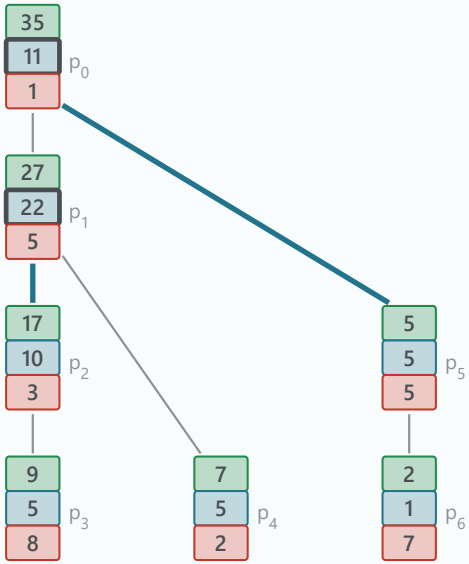
Pipelined Tree Step by Step



Round 3: green $p_1 \to p_0$: 27+8=35; blue $p_3 \to p_2$: 5+5=10, $p_4 \to p_1$: 5+7=12, $p_6 \to p_5$: 1+4=5.

- Edge color = component carried this round; arrows point to the parent.
- Thick border = value updated this round.
- Each component reaches the root in 3 stages:
  (1) $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$
  (2) $p_2 \to p_1$, $p_5 \to p_0$
  (3) $p_1 \to p_0$.
- A new component starts **every 2 rounds**.

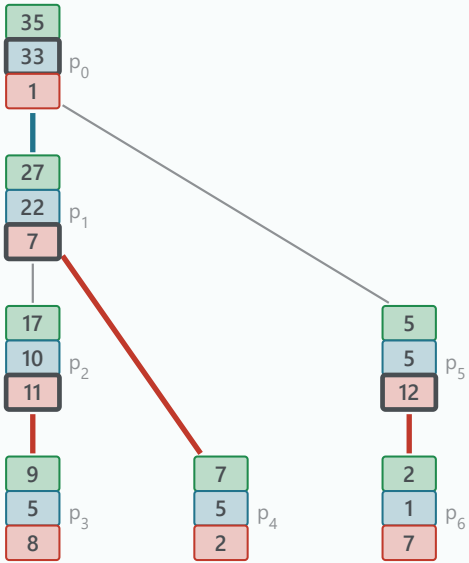
Pipelined Tree Step by Step



Round 4: blue $p_2 \to p_1$: 10+12=22, $p_5 \to p_0$: 5+6=11.

- Edge color = component carried this round; arrows point to the parent.
- Thick border = value updated this round.
- Each component reaches the root in 3 stages:
  (1) $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$
  (2) $p_2 \to p_1$, $p_5 \to p_0$
  (3) $p_1 \to p_0$.
- A new component starts **every 2 rounds**.

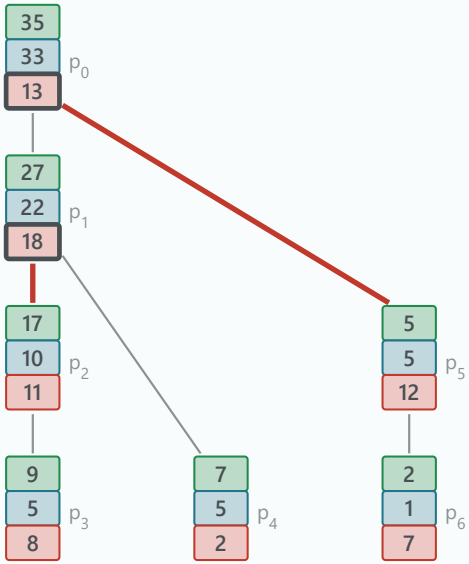
Pipelined Tree Step by Step



Round 5: blue $p_1 \to p_0$: 22+11=33; red $p_3 \to p_2$: 8+3=11, $p_4 \to p_1$: 2+5=7, $p_6 \to p_5$: 7+5=12.

- Edge color = component carried this round; arrows point to the parent.
- Thick border = value updated this round.
- Each component reaches the root in 3 stages:
  (1) $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$
  (2) $p_2 \to p_1$, $p_5 \to p_0$
  (3) $p_1 \to p_0$.
- A new component starts **every 2 rounds**.

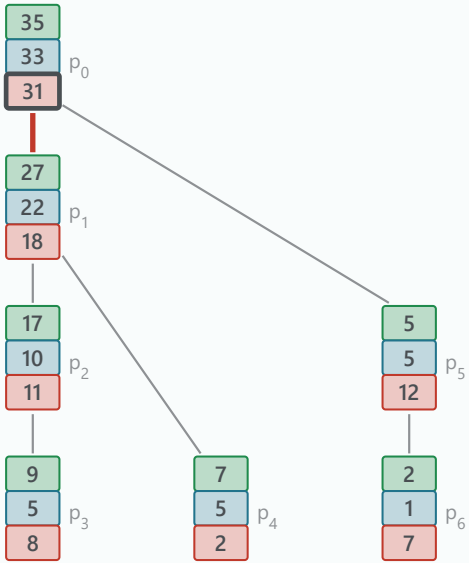
Pipelined Tree Step by Step



Round 6: red $p_2 \to p_1$: 11+7=18, $p_5 \to p_0$: 12+1=13.

- Edge color = component carried this round; arrows point to the parent.
- Thick border = value updated this round.
- Each component reaches the root in 3 stages:
  (1) $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$
  (2) $p_2 \to p_1$, $p_5 \to p_0$
  (3) $p_1 \to p_0$.
- A new component starts **every 2 rounds**.

Pipelined Tree Step by Step



Round 7: red $p_1 \to p_0$: 18+13=31. The root holds (35, 33, 31): done in 7 rounds.

- Edge color = component carried this round; arrows point to the parent.
- Thick border = value updated this round.
- Each component reaches the root in 3 stages:
  (1) $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$
  (2) $p_2 \to p_1$, $p_5 \to p_0$
  (3) $p_1 \to p_0$.
- A new component starts **every 2 rounds**.

Pipelined Tree: Analysis

round	stage 1 $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$	stage 2 $p_2 \to p_1$, $p_5 \to p_0$	stage 3 $p_1 \to p_0$
1	green	—	—
2	—	green	—
3	blue	—	green
4	—	blue	—
5	red	—	blue
6	—	red	—
7	—	—	red

- Why every 2 rounds? p_1 receives from two children (p_4 and p_2); since it can receive one message per round, each component keeps p_1 busy for **2 rounds**. The same holds for p_0.
- Total rounds: $h + (k - 1) \cdot 2 = 3 + 2 \cdot 2 = \mathbf{7}$ (naive: $k \cdot h = 9$).
- As k grows: $2k + 1 \approx 2k$ rounds versus $3k$ for the naive method. In general, rounds $\approx h + (k - 1) \cdot$ (maximum number of children).
- At the root: green 35, blue 33, red 31.

Kinds of Parallel Algorithms

Divide and conquer

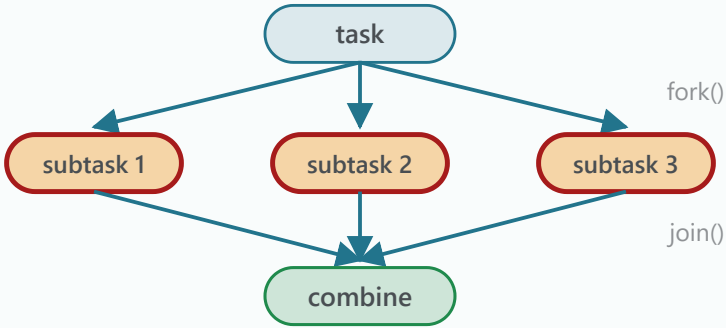
The problem is split into independent subproblems that are solved simultaneously on different cores, and the results are combined. Parallel merge sort, quicksort, matrix multiplication.

MapReduce

Big data is processed on distributed nodes: the **map** step emits intermediate key–value pairs, and the **reduce** step combines them. Hadoop, Spark; counting, indexing, log analysis.

Data parallel

The same computation is applied to different parts of the data at the same time (SIMD instructions, GPU). Scientific simulation, image processing, numerical computing.



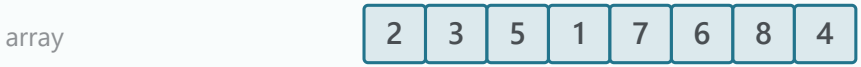
Parallel Merge Sort

```
class SortTask extends RecursiveTask<int[]> {
  private final int[] arr;
  SortTask(int[] arr) { this.arr = arr; }

  protected int[] compute() {
    if (arr.length <= 1) return arr;
    int mid = arr.length / 2;
    var left = new SortTask(
        Arrays.copyOfRange(arr, 0, mid));
    var right = new SortTask(
        Arrays.copyOfRange(arr, mid, arr.length));
    left.fork();              // start in parallel
    right.fork();
    return merge(left.join(), right.join());
  }
}
// usage:
int[] s = new ForkJoinPool().invoke(
    new SortTask(arr));
```

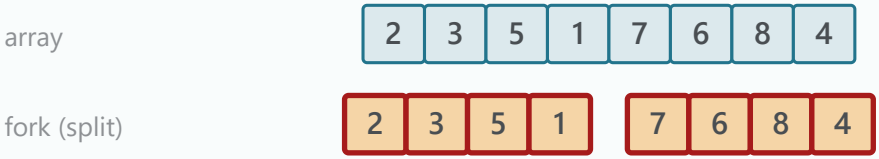
- `RecursiveTask` : a Fork/Join task that returns a result.
- `fork()` : submits the subtask to the pool (asynchronously).
- `join()` : waits for the subtask to finish and gets its result.
- `merge` : the usual sequential merge, O(n).
- `ForkJoinPool` uses work stealing to hand work to idle cores.

Parallel Merge Sort: 2 3 5 1 7 6 8 4



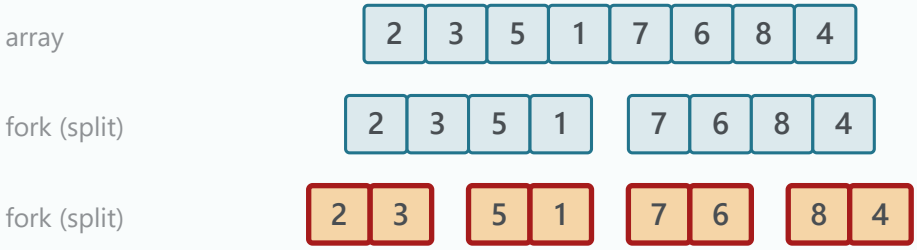
Task: sort the 8-element array. `SortTask` is the root task.

Parallel Merge Sort: 2 3 5 1 7 6 8 4



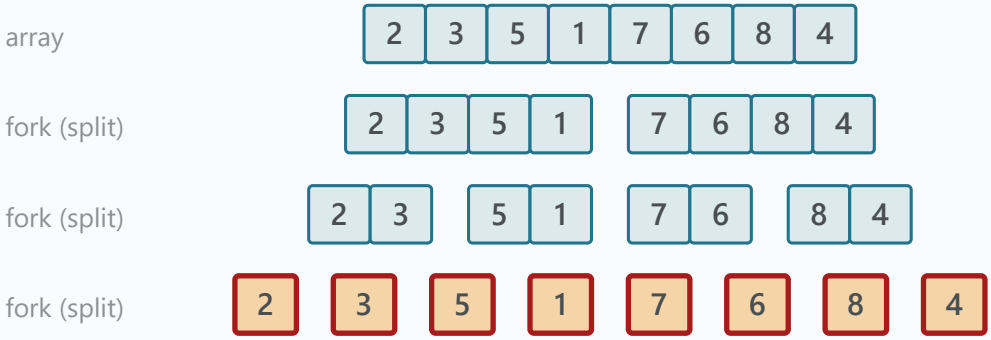
The task splits the array in two and submits both subtasks to the pool with `fork()`; the halves can be processed **at the same time**.

Parallel Merge Sort: 2 3 5 1 7 6 8 4



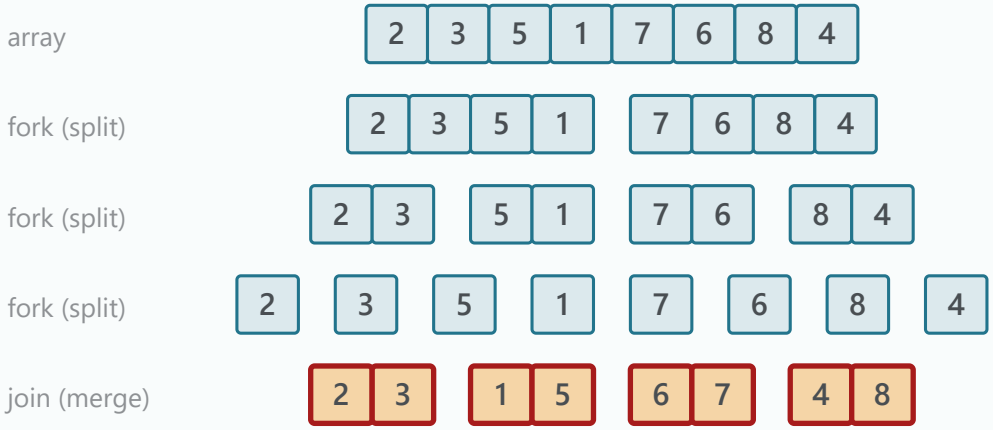
Each subtask splits again: now 4 tasks can run in parallel.

Parallel Merge Sort: 2 3 5 1 7 6 8 4



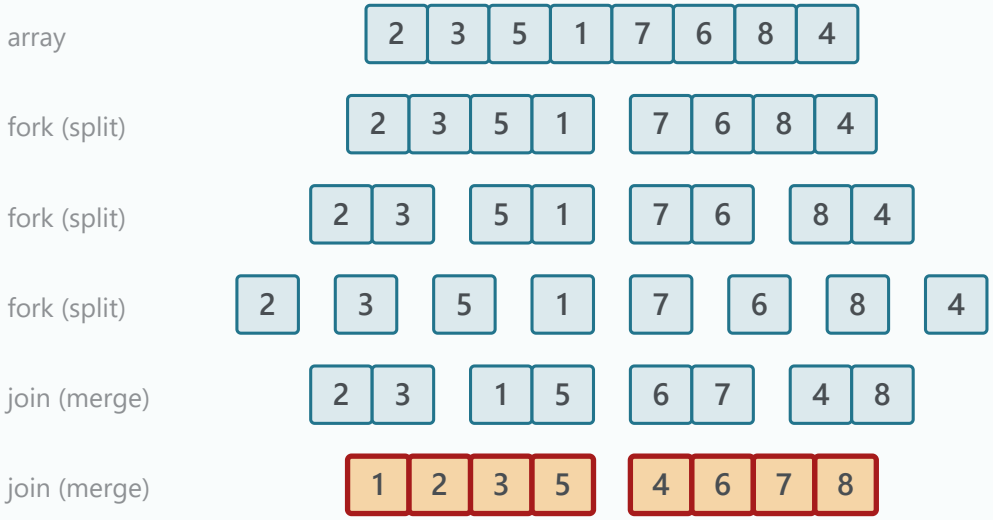
Base case: one-element arrays are already sorted (8 leaf tasks).

Parallel Merge Sort: 2 3 5 1 7 6 8 4



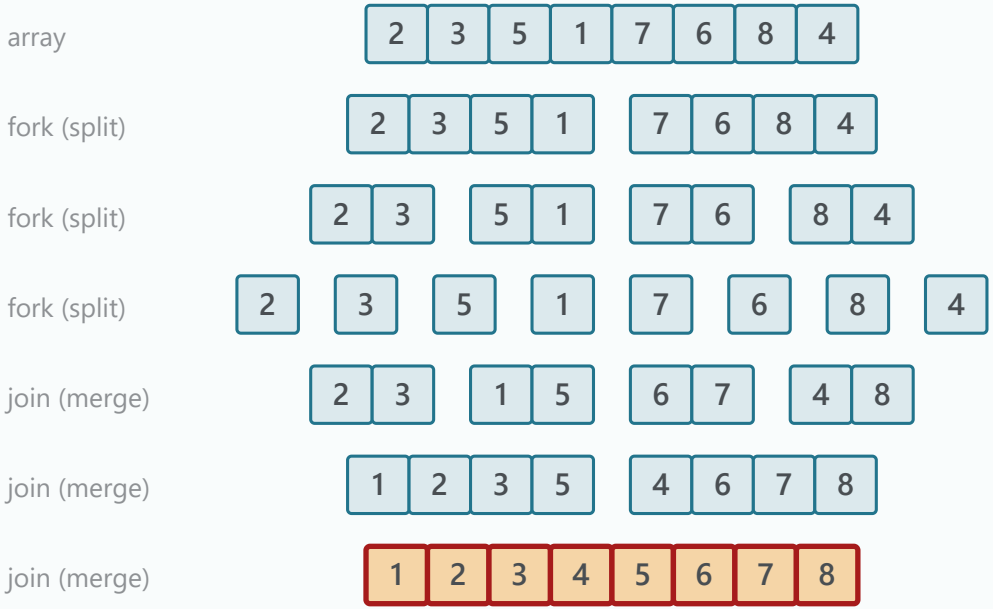
`join()` : each parent waits for its two children's results and merges them: [2 3] [1 5] [6 7] [4 8].

Parallel Merge Sort: 2 3 5 1 7 6 8 4



One level up, two merges run in parallel: [1 2 3 5] and [4 6 7 8].

Parallel Merge Sort: 2 3 5 1 7 6 8 4



The root task does the final merge **alone**: [1 2 3 4 5 6 7 8]. This sequential merge makes the depth (span) Θ(n).

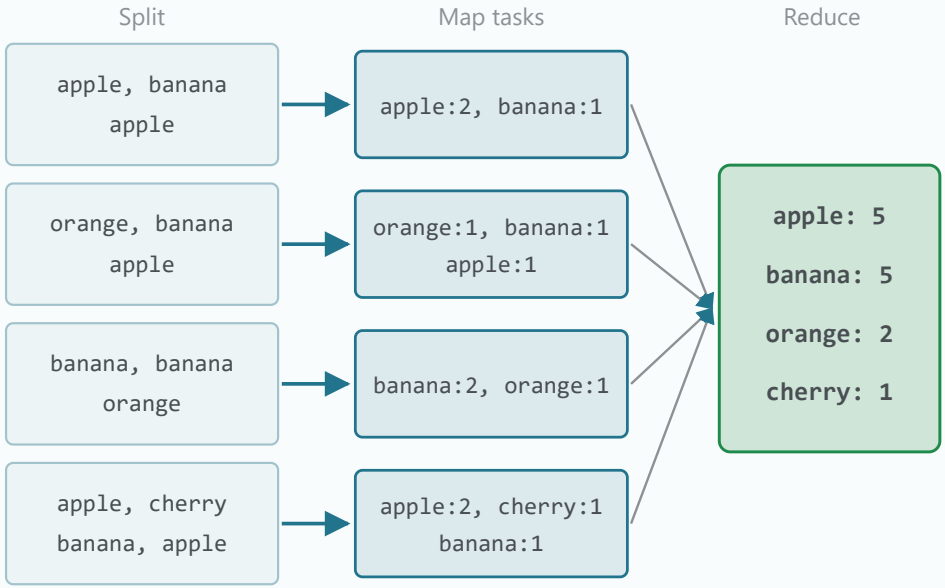
Parallel Sorting: Analysis

Version	Work W	Depth D	Parallelism W / D
Sequential merge sort	$\Theta(n \log n)$	$\Theta(n \log n)$	1
Parallel recursion + sequential merge	$\Theta(n \log n)$	$\Theta(n)$	$\Theta(\log n)$
Parallel recursion + parallel merge (CLRS)	$\Theta(n \log n)$	$\Theta(\log^3 n)$	$\Theta(n / \log^2 n)$

- Depth recurrence: $D(n) = D(n/2) + \Theta(n) = \Theta(n)$, because the two halves run in parallel but the merge is sequential.
- Parallelism is only $\Theta(\log n)$: for $n = 10^6$, more than ≈ 20 cores hardly help.
- In practice, switch to a **sequential** sort for small parts (e.g., < 1000 elements); otherwise task-creation overhead eats the gain.
- Ready-made: `Arrays.parallelSort(arr)`.

MapReduce

- **Map**: each chunk is processed independently and emits intermediate **key–value** pairs (word → count).
- **Shuffle**: equal keys are gathered at the same reducer.
- **Reduce**: the values of each key are combined (summed).
- In the example: 13 words, 4 threads → chunk size ⌈13/4⌉ = 3; the last chunk has 4 words.



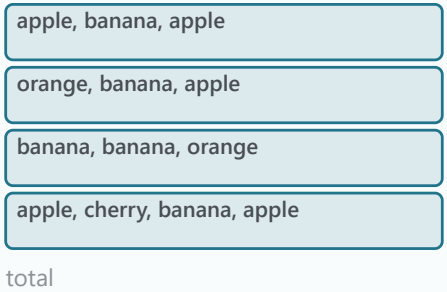
MapReduce: Code

```
// Map: count the words in one chunk
protected Map<String, Integer> compute() {
  Map<String, Integer> result = new HashMap<>();
  for (String word : chunk)
    result.merge(word, 1, Integer::sum);
  return result;
}
// Main flow
for (MapTask g : mapTasks) g.fork();
List<Map<String, Integer>> partial = new ArrayList<>();
for (MapTask g : mapTasks) partial.add(g.join());
// Reduce: combine the partial results
Map<String, Integer> total = new HashMap<>();
for (Map<String, Integer> m : partial)
  m.forEach((k, v) -> total.merge(k, v, Integer::sum));
```

- Each map task has its **own** HashMap: no shared state, no locks needed.
- Result: apple 5, banana 5, orange 2, cherry 1.
- Here reduce is a single task; in real MapReduce the keys are **partitioned** among many reducers (hash(k) mod r).

MapReduce: Code Trace

```
1  // Map: count the words in one chunk
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> result = new HashMap<>();
4    for (String word : chunk)
5      result.merge(word, 1, Integer::sum);
6    return result;
7  }
8  // Main flow
▶ for (MapTask g : mapTasks) g.fork();
10 List<Map<String, Integer>> partial = new ArrayList<>();
11 for (MapTask g : mapTasks) partial.add(g.join());
12 // Reduce: combine the partial results
13 Map<String, Integer> total = new HashMap<>();
14 for (Map<String, Integer> m : partial)
15   m.forEach((k, v) -> total.merge(k, v, Integer::sum));
```



The 13 words were split into 4 chunks; for each chunk a `MapTask` is submitted to the pool with `fork()`, and the tasks run in parallel.

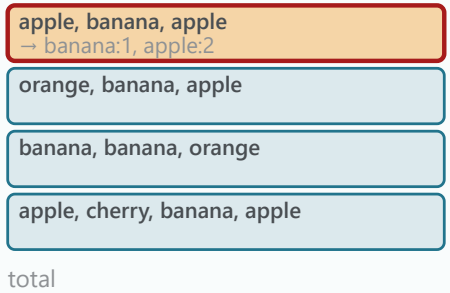
Step 1 / 10

VARIABLES

`tasks = 4`

MapReduce: Code Trace

```
1  // Map: count the words in one chunk
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> result = new HashMap<>();
4    for (String word : chunk)
5      result.merge(word, 1, Integer::sum);
6    return result;
7  }
8  // Main flow
9  for (MapTask g : mapTasks) g.fork();
10 List<Map<String, Integer>> partial = new ArrayList<>();
▶ for (MapTask g : mapTasks) partial.add(g.join());
12 // Reduce: combine the partial results
13 Map<String, Integer> total = new HashMap<>();
14 for (Map<String, Integer> m : partial)
15   m.forEach((k, v) -> total.merge(k, v, Integer::sum));
```



`join()` : task 0's own HashMap is ready → banana 1, apple 2. No shared state, no locks needed.

Step 2 / 10

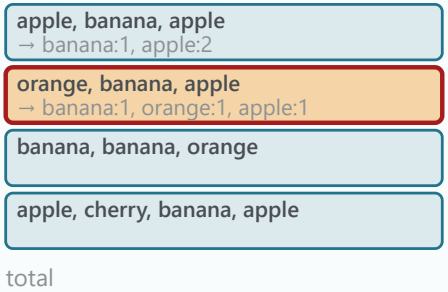
VARIABLES

`g = 0`

`partial[0] = {banana:1, apple:2}`

MapReduce: Code Trace

```
1  // Map: count the words in one chunk
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> result = new HashMap<>();
4    for (String word : chunk)
5      result.merge(word, 1, Integer::sum);
6    return result;
7  }
8  // Main flow
9  for (MapTask g : mapTasks) g.fork();
10 List<Map<String, Integer>> partial = new ArrayList<>();
▶ for (MapTask g : mapTasks) partial.add(g.join());
12 // Reduce: combine the partial results
13 Map<String, Integer> total = new HashMap<>();
14 for (Map<String, Integer> m : partial)
15   m.forEach((k, v) -> total.merge(k, v, Integer::sum));
```



`join()` : task 1's own HashMap is ready → banana 1, orange 1, apple 1. No shared state, no locks needed.

Step 3 / 10

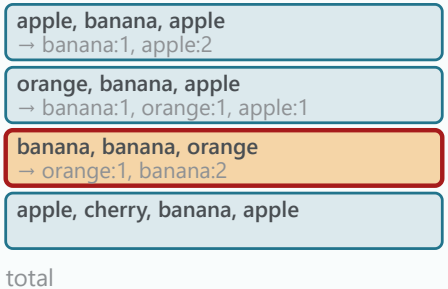
VARIABLES

`g = 1`

`partial[1] = {banana:1, orange:1, apple:1}`

MapReduce: Code Trace

```
1  // Map: count the words in one chunk
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> result = new HashMap<>();
4    for (String word : chunk)
5      result.merge(word, 1, Integer::sum);
6    return result;
7  }
8  // Main flow
9  for (MapTask g : mapTasks) g.fork();
10 List<Map<String, Integer>> partial = new ArrayList<>();
▶ for (MapTask g : mapTasks) partial.add(g.join());
12 // Reduce: combine the partial results
13 Map<String, Integer> total = new HashMap<>();
14 for (Map<String, Integer> m : partial)
15   m.forEach((k, v) -> total.merge(k, v, Integer::sum));
```



`join()` : task 2's own HashMap is ready → orange 1, banana 2. No shared state, no locks needed.

Step 4 / 10

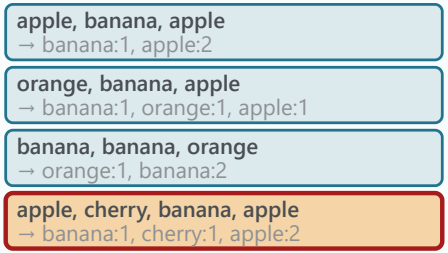
VARIABLES

`g = 2`

`partial[2] = {orange:1, banana:2}`

MapReduce: Code Trace

```
1  // Map: count the words in one chunk
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> result = new HashMap<>();
4    for (String word : chunk)
5      result.merge(word, 1, Integer::sum);
6    return result;
7  }
8  // Main flow
9  for (MapTask g : mapTasks) g.fork();
10 List<Map<String, Integer>> partial = new ArrayList<>();
▶ for (MapTask g : mapTasks) partial.add(g.join());
12 // Reduce: combine the partial results
13 Map<String, Integer> total = new HashMap<>();
14 for (Map<String, Integer> m : partial)
15   m.forEach((k, v) -> total.merge(k, v, Integer::sum));
```



`join()` : task 3's own HashMap is ready → banana 1, cherry 1, apple 2. No shared state, no locks needed.

Step 5 / 10

VARIABLES

`g = 3`

`partial[3] = {banana:1, cherry:1, apple:2}`

MapReduce: Code Trace

```
1  // Map: count the words in one chunk
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> result = new HashMap<>();
4    for (String word : chunk)
5      result.merge(word, 1, Integer::sum);
6    return result;
7  }
8  // Main flow
9  for (MapTask g : mapTasks) g.fork();
10 List<Map<String, Integer>> partial = new ArrayList<>();
11 for (MapTask g : mapTasks) partial.add(g.join());
12 // Reduce: combine the partial results
▶ Map<String, Integer> total = new HashMap<>();
14 for (Map<String, Integer> m : partial)
15   m.forEach((k, v) -> total.merge(k, v, Integer::sum));
```

apple, banana, apple
→ banana:1, apple:2

orange, banana, apple
→ banana:1, orange:1, apple:1

banana, banana, orange
→ orange:1, banana:2

apple, cherry, banana, apple
→ banana:1, cherry:1, apple:2

total

Reduce begins: an empty `total` map.

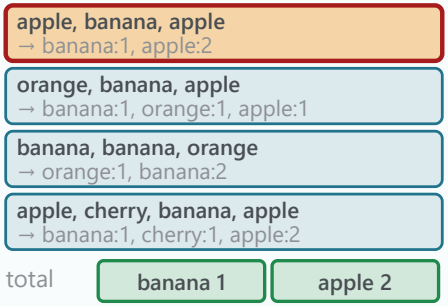
Step 6 / 10

VARIABLES

`total = {}`

MapReduce: Code Trace

```
1  // Map: count the words in one chunk
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> result = new HashMap<>();
4    for (String word : chunk)
5      result.merge(word, 1, Integer::sum);
6    return result;
7  }
8  // Main flow
9  for (MapTask g : mapTasks) g.fork();
10 List<Map<String, Integer>> partial = new ArrayList<>();
11 for (MapTask g : mapTasks) partial.add(g.join());
12 // Reduce: combine the partial results
13 Map<String, Integer> total = new HashMap<>();
▶ for (Map<String, Integer> m : partial)
▶   m.forEach((k, v) -> total.merge(k, v, Integer::sum));
```



partial[0] merged: equal keys are added with `Integer::sum`.

Step 7 / 10

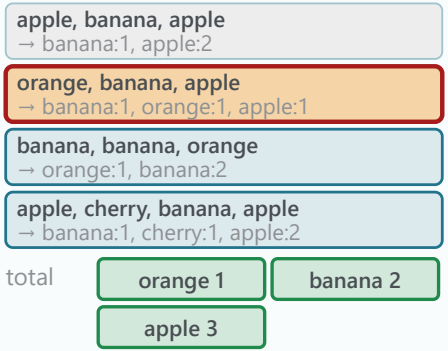
VARIABLES

`m = partial[0]`

`total = {banana:1, apple:2}`

MapReduce: Code Trace

```
1  // Map: count the words in one chunk
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> result = new HashMap<>();
4    for (String word : chunk)
5      result.merge(word, 1, Integer::sum);
6    return result;
7  }
8  // Main flow
9  for (MapTask g : mapTasks) g.fork();
10 List<Map<String, Integer>> partial = new ArrayList<>();
11 for (MapTask g : mapTasks) partial.add(g.join());
12 // Reduce: combine the partial results
13 Map<String, Integer> total = new HashMap<>();
▶ for (Map<String, Integer> m : partial)
▶   m.forEach((k, v) -> total.merge(k, v, Integer::sum));
```



partial[1] merged: equal keys are added with `Integer::sum`.

Step 8 / 10

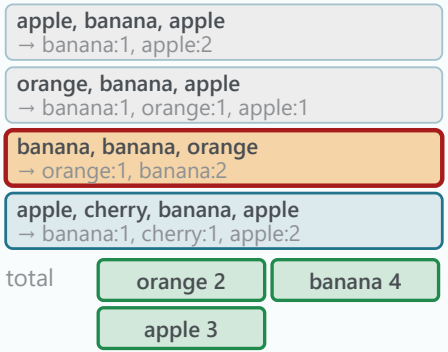
VARIABLES

`m = partial[1]`

`total = {orange:1, banana:2, apple:3}`

MapReduce: Code Trace

```
1  // Map: count the words in one chunk
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> result = new HashMap<>();
4    for (String word : chunk)
5      result.merge(word, 1, Integer::sum);
6    return result;
7  }
8  // Main flow
9  for (MapTask g : mapTasks) g.fork();
10 List<Map<String, Integer>> partial = new ArrayList<>();
11 for (MapTask g : mapTasks) partial.add(g.join());
12 // Reduce: combine the partial results
13 Map<String, Integer> total = new HashMap<>();
▶ for (Map<String, Integer> m : partial)
▶   m.forEach((k, v) -> total.merge(k, v, Integer::sum));
```



partial[2] merged: equal keys are added with `Integer::sum`.

Step 9 / 10

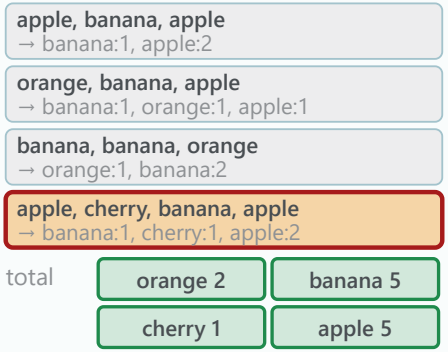
VARIABLES

`m = partial[2]`

`total = {orange:2, banana:4, apple:3}`

MapReduce: Code Trace

```
1  // Map: count the words in one chunk
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> result = new HashMap<>();
4    for (String word : chunk)
5      result.merge(word, 1, Integer::sum);
6    return result;
7  }
8  // Main flow
9  for (MapTask g : mapTasks) g.fork();
10 List<Map<String, Integer>> partial = new ArrayList<>();
11 for (MapTask g : mapTasks) partial.add(g.join());
12 // Reduce: combine the partial results
13 Map<String, Integer> total = new HashMap<>();
▶ for (Map<String, Integer> m : partial)
▶   m.forEach((k, v) -> total.merge(k, v, Integer::sum));
```



partial[3] merged: equal keys are added with `Integer::sum`. Result: orange 2, banana 5, cherry 1, apple 5.

Step 10 / 10

VARIABLES

`m = partial[3]`

`total = {orange:2, banana:5, cherry:1, apple:5}`

Data Parallelism: Matrix Multiplication

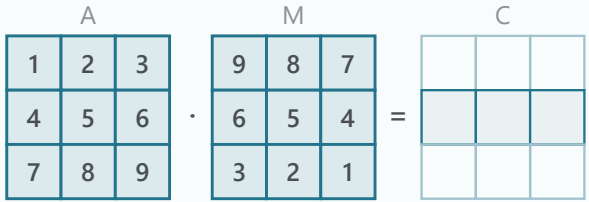
```
// each row is a separate task
class RowTask extends RecursiveAction {
  protected void compute() {
    for (int j = 0; j < n; j++) {
      C[row][j] = 0;
      for (int k = 0; k < n; k++)
        C[row][j] += A[row][k] * B[k][j];
    }
  }
}
try (var pool = new ForkJoinPool()) {
  for (int i = 0; i < n; i++)
    pool.submit(new RowTask(i));
}   // close(): waits for all tasks
```

C = A · B		
30	24	18
84	69	54
138	114	90

- A = [[1,2,3],[4,5,6],[7,8,9]], B = [[9,8,7],[6,5,4],[3,2,1]].
- Rows are independent: n tasks, each with $\Theta(n^2)$ work. W = $\Theta(n^3)$, D = $\Theta(n^2)$ (per row).
- Each task writes only to **its own row** → no race.

Row Task: Code Trace

```
1 // RowTask, row = 1
▶ protected void compute() {
3   for (int j = 0; j < n; j++) {
4     C[row][j] = 0;
5     for (int k = 0; k < n; k++)
6       C[row][j] += A[row][k] * B[k][j];
7   }
8 }
```



This task computes only row 1 of C: row 1 of A (4, 5, 6) times each column of B. The other rows are in other tasks.

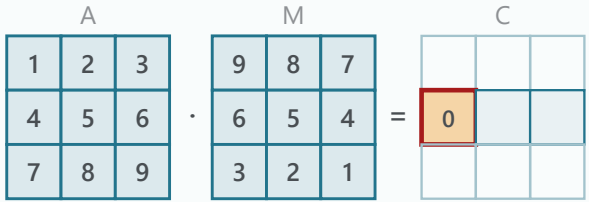
Step 1 / 8

VARIABLES

`row = 1` `C[1] = [–, –, –]`

Row Task: Code Trace

```
1 // RowTask, row = 1
2 protected void compute() {
▶   for (int j = 0; j < n; j++) {
▶     C[row][j] = 0;
5     for (int k = 0; k < n; k++)
6       C[row][j] += A[row][k] * B[k][j];
7   }
8 }
```



j = 0: C[1][0] starts at 0.

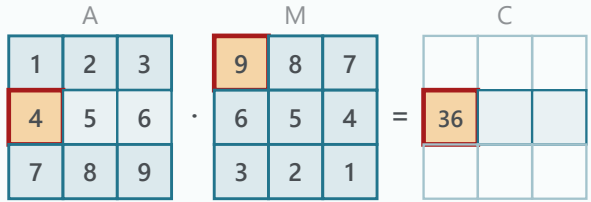
Step 2 / 8

VARIABLES

`row = 1` `j = 0` `C[1][0] = 0`

Row Task: Code Trace

```
1 // RowTask, row = 1
2 protected void compute() {
3   for (int j = 0; j < n; j++) {
4     C[row][j] = 0;
▶     for (int k = 0; k < n; k++)
▶       C[row][j] += A[row][k] * B[k][j];
7   }
8 }
```



k = 0: C[1][0] += A[1][0] · B[0][0] = 4 · 9 → **36**.

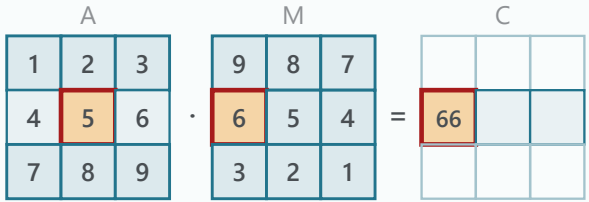
Step 3 / 8

VARIABLES

`row = 1` `j = 0` `k = 0` `C[1][0] = 36`

Row Task: Code Trace

```
1 // RowTask, row = 1
2 protected void compute() {
3   for (int j = 0; j < n; j++) {
4     C[row][j] = 0;
▶     for (int k = 0; k < n; k++)
▶       C[row][j] += A[row][k] * B[k][j];
7   }
8 }
```



k = 1: C[1][0] += A[1][1] · B[1][0] = 5 · 6 → **66**.

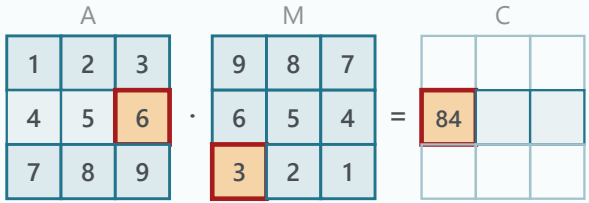
Step 4 / 8

VARIABLES

`row = 1` `j = 0` `k = 1` `C[1][0] = 66`

Row Task: Code Trace

```
1 // RowTask, row = 1
2 protected void compute() {
3   for (int j = 0; j < n; j++) {
4     C[row][j] = 0;
▶     for (int k = 0; k < n; k++)
▶       C[row][j] += A[row][k] * B[k][j];
7   }
8 }
```



k = 2: C[1][0] += A[1][2] · B[2][0] = 6 · 3 → **84**.

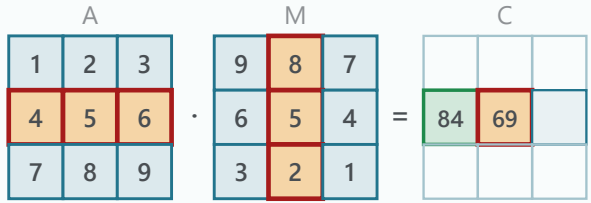
Step 5 / 8

VARIABLES

`row = 1` `j = 0` `k = 2` `C[1][0] = 84`

Row Task: Code Trace

```
1 // RowTask, row = 1
2 protected void compute() {
    for (int j = 0; j < n; j++) {
      C[row][j] = 0;
      for (int k = 0; k < n; k++)
        C[row][j] += A[row][k] * B[k][j];
7   }
8 }
```



j = 1: 4 · 8 + 5 · 5 + 6 · 2 = **69**.

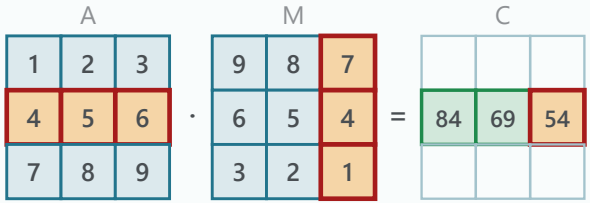
Step 6 / 8

VARIABLES

`row = 1` `j = 1` `C[1][1] = 69`

Row Task: Code Trace

```
1 // RowTask, row = 1
2 protected void compute() {
    for (int j = 0; j < n; j++) {
      C[row][j] = 0;
      for (int k = 0; k < n; k++)
        C[row][j] += A[row][k] * B[k][j];
7   }
8 }
```



j = 2: 4 · 7 + 5 · 4 + 6 · 1 = **54**.

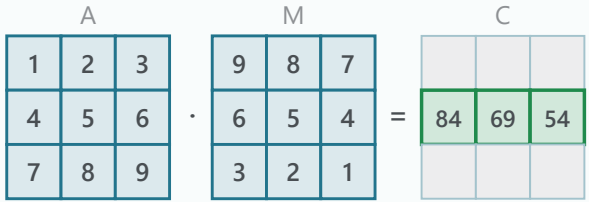
Step 7 / 8

VARIABLES

`row = 1` `j = 2` `C[1][2] = 54`

Row Task: Code Trace

```
1 // RowTask, row = 1
2 protected void compute() {
3   for (int j = 0; j < n; j++) {
4     C[row][j] = 0;
5     for (int k = 0; k < n; k++)
6       C[row][j] += A[row][k] * B[k][j];
7   }
▶ }
```



Rows 0 and 2 are computed at the same time in other tasks.

Task done: C[1] = (84, 69, 54). Each task writes only to its own row, so there is no race condition.

Step 8 / 8

VARIABLES

`row = 1` `C[1] = [84, 69, 54]`

Race Conditions

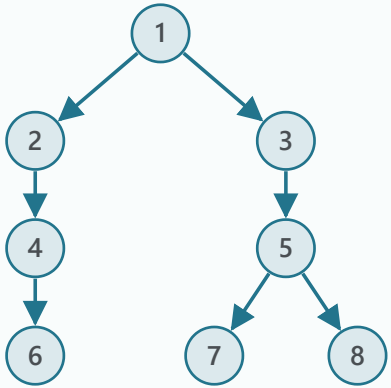
Two threads increment the shared `counter` at the same time. `counter++` is really three steps: **read** → **increment** → **write**.

time	Thread 1	Thread 2	counter
1	read (0)		0
2		read (0)	0
3	write 0 + 1		1
4		write 0 + 1	**1**

Expected 2, got 1: a **lost update**.

- **Race condition**: the result depends on the timing of the threads; the bug may not show up on every run.
- Fixes: a `synchronized` block / lock, `AtomicInteger.incrementAndGet()`, concurrent collections (`ConcurrentHashMap`).
- Best of all is to avoid shared state: each task produces its own partial result, and the results are combined afterward (reduction, MapReduce).
- Locks are not free either: they create sequential regions (Amdahl) and risk **deadlock**.

Concurrent Graph Traversal



- Idea: `fork()` a new traversal task for each unvisited neighbor; subtrees are traversed in parallel.
- The tasks **share** the `visited` set. If it is an ordinary `HashSet`, concurrent access is not safe.
- Moreover, `contains` + `add` are two separate steps: two tasks may visit the same node together.

```
Set<Node> visited = ConcurrentHashMap.newKeySet();
if (!visited.add(neighbor)) continue; // atomic check+add
```

Level-synchronous BFS: all nodes of a level are expanded in parallel, with a barrier at the end of each level. D = O(diameter) on an ideal PRAM with enough processors; the number of levels does not exceed the diameter of the part reachable from the source (O(diameter · log n) in a realistic model).

Parallelism Tools in Java

Tool	When?
`Thread`, `Runnable`	lowest level; managed by hand, rarely used directly
`ExecutorService`	running independent tasks in a fixed thread pool
`ForkJoinPool`, `RecursiveTask`	divide and conquer; recursive tasks with work stealing
`list.parallelStream()`	data-parallel map / filter / reduce (uses the common ForkJoinPool)
`Arrays.parallelSort`	sorting large arrays in parallel
`Arrays.parallelPrefix`	parallel prefix sums (scan)
`AtomicInteger`, `ConcurrentHashMap`	safe access when a shared counter/set is needed

```
Map<String, Long> counts = words.parallelStream()
    .collect(Collectors.groupingByConcurrent(w -> w, Collectors.counting()));
```

Summary

Topic	Key points
Idea	split → assign to processors → solve simultaneously → combine; the price is communication, synchronization, load balancing
Models	Flynn: SISD/SIMD/MISD/MIMD; PRAM: EREW, CREW, CRCW; shared / distributed memory
Measures	$S = T_1/T_p$, $E = S/p$, $C = p \cdot T_p$; Amdahl $S \le 1/(1 - f)$; Brent $T_p \le W/p + D$
PRAM summation	$\log_2 n$ rounds, $W = n - 1$; cost-optimal with $p = n/\log n$
Prefix sums	Hillis–Steele: $\log_2 n$ rounds, $O(n \log n)$ work; Blelloch: $O(n)$ work
Pipelining	p processors, m rows: $m + p - 2$ steps (naive $m(p - 1)$)
Pipelined tree	$h + (k - 1) \cdot 2$ rounds (example 7; naive $k \cdot h = 9$)
Java	Fork/Join (fork, join), MapReduce, one task per row; races on shared state → atomic/concurrent structures

SUMMARY

Common Mistakes

Thinking p processors = p times faster

The sequential part (Amdahl), communication and waiting make S < p. 10% sequential code caps the speedup at 10.

A shared HashSet / counter

Concurrent writes lead to race conditions. Use `ConcurrentHashMap`, `AtomicInteger`, or local results + combining.

Overly fine-grained tasks

Calling `fork()` for every element inflates task-creation overhead. Parts below a threshold are processed sequentially.

Forgetting synchronization between rounds

In PRAM summation, if round t + 1 starts before **all** writes of round t finish, stale values are read.

Looking only at depth

Hillis–Steele finishes in O(log n) rounds but does O(n log n) work; with few processors it can be slower than a sequential scan.

Miscounting pipeline steps

The fill time is forgotten: for m rows and p processors it takes **m + p − 2** steps, not m.

Sample Problems

1. n = 16 numbers are summed with a PRAM tree. How many rounds does it take, how many processors work in the first round, and how many additions are made in total?
2. How many steps does a pipeline with p = 5 processors and m = 100 rows need? Compare with the naive method.
3. In the chapter's pipelined tree example (7 processors, h = 3), how many rounds would be needed if each processor held a vector with k = 10 components?
4. 80% of a program is parallel. What is the Amdahl speedup for p = 4, and what is the upper limit?
5. T_1 = 120 s, T_6 = 30 s. Find the speedup, efficiency and cost.
6. Write the Hillis–Steele prefix sums of [1, 2, 3, 4] round by round; how many additions are made?

▶ **Show answers**

Exercises

1. Modify PRAM tree summation to find the **maximum element**. What are its time and work?
2. How many additions does Hillis–Steele make for n = 8? Compare with the sequential algorithm.
3. Make the `visited` set of the parallel traversal thread-safe.
4. Add a threshold to parallel merge sort; why is it better to `compute()` one of the two subtasks directly instead of calling `fork()`?
5. On p = 32 processors, what minimum parallel fraction gives a speedup of S = 16 (Amdahl)?
6. Express the number of steps and the efficiency of a pipeline with m rows and p processors in general.

▶ **Answers / hints**

References

- Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Chapter 26: Parallel Algorithms).
- J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.
- A. Grama, A. Gupta, G. Karypis and V. Kumar, *Introduction to Parallel Computing*, 2nd ed., Addison-Wesley, 2003.
- G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities", AFIPS, 1967.
- J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", OSDI, 2004.
- D. Lea, "A Java Fork/Join Framework", ACM Java Grande Conference, 2000.

The End

Chapter 9: Parallel Algorithms

End of the course · Previous chapter: Chapter 8: Online Algorithms