# CS170 Lecture Notes: Parallel Algorithms

Course: CS 170 — Efficient Algorithms and Intractable Problems, Spring 2026, Lecture 14
Instructors: Lijie Chen, Umesh V. Vazirani

## 1 The Work-Span Model

How do we even begin to reason about parallel computation? The first temptation is to think of parallelism as a matter of *scheduling* — which processor runs which task and when. Such scheduling decisions depend on the machine, the operating system, and countless contingencies of runtime. A clean abstraction that separates algorithmic content from implementation detail, is to model a parallel algorithm as a **directed acyclic graph** (DAG). Each vertex of the DAG represents a unit of computation; each directed edge represents a *dependency* — an ordering constraint requiring one computation to precede another. Whenever two vertices have no directed path between them in either direction, they are independent and may execute simultaneously. The entire structure of an algorithm's parallelism is encoded in the topology of this graph.

From this DAG we extract two fundamental quantities that together determine nearly everything we care about.

### 1.1 Work and Span

Let $T_p$ be the time to finish the computation on $p$ processors.

**Work** $= T_1 =$ the total number of vertices in the DAG. This is the time the algorithm would take on a single processor — no parallelism exploited. Clearly $T_p \geq Work/p = T_1/p$.

**Span** $= T_\infty =$ the length of the longest directed path in the DAG — also called the *critical path*. Even with infinitely many processors (or as many as the number of vertices), we cannot do better than $T_\infty$, because some computations are irreducibly sequential: each depends on the one before it. Span is the parallel time in the ideal limit.

**Speedup ($S_p$):** The ratio $T_1/T_p$. Max speedup is $p$ (linear speedup).

**Efficiency ($E_p$):** The ratio $T_1/pT_p$, measuring how well we utilise each processor. Perfect efficiency is 1; in practice $E_p \leq 1$.

The ratio $T_1/T_\infty$ is called the **parallelism** of the algorithm: roughly, the average amount of work available at each step along the critical path. An algorithm with high parallelism is one that exposes many independent tasks; there is "room" for many processors to be busy simultaneously.

On $p$ actual processors, the runtime $T_p$ lies between these two extremes. The relationship is made precise by one of the most elegant elementary results in the field.

### 1.2 Brent's Rule

Suppose we have an arbitrary parallel algorithm with work $T_1$ and span $T_\infty$. How long does it take on $p$ processors?

**Theorem 1.1** (Brent's Rule)**.** *For any parallel algorithm with work $T_1$ and span $T_\infty$, the runtime on $p$ processors satisfies*

$$T_p \leq \frac{T_1}{p} + T_\infty.$$

*Proof.* At each of the $T_\infty$ levels of the DAG, $p$ processors finish that level's work in at most $\lceil m_i/p \rceil \leq m_i/p + 1$ steps. Summing over all levels gives $T_p \leq T_1/p + T_\infty$. □

*Remark* (Interpretation). The bound has two terms: $T_1/p$, the time if all work were perfectly parallelised, and $T_\infty$, the serial bottleneck no processor count can reduce. As $p \to \infty$ the first term vanishes and runtime approaches $T_\infty$, confirming that span is the right notion of parallel time.

## 2 Case Studies

With the theoretical framework in hand, let us turn to concrete algorithms. The examples below are not chosen arbitrarily: each introduces a technique of broad applicability, and together they build up to a nice picture of fast parallel arithmetic.

### 2.1 Integer Multiplication: The 3-for-2 Reduction

Multiplying two $n$-bit integers is a venerable problem. The schoolbook algorithm computes $n$ partial products (one per bit of the multiplier) and adds up the $n$ numbers. It works, but it is slow. Understanding *why* it is slow, and how to fix it, will lead us to a technique of considerable generality.

The key insight is that the bottleneck is not multiplication itself — generating the partial products is fast and embarrassingly parallel. The bottleneck is *summation*. We need a way to add many numbers together that avoids long carry chains:

#### *2.1.1 The Sequential Bottleneck*

When we add two binary numbers, a carry bit may propagate all the way from the least-significant position to the most-significant one. In the worst case — adding $111\cdots1$ to $000\cdots1$ — every bit position is affected by the bit to its right. This *carry chain* is the fundamental obstacle: the $k$-th output bit cannot be determined until the carry from position $k-1$ is known, which depends on position $k-2$, and so on. This is disappointing, since it suggests that the DAG for binary addition of $n$-bit numbers looks like a path of length $n$, one vertex per bit position. i.e. $T_\infty = n$. But this is the DAG for the naive algorithm. It turns out that there is a more clever algorithm for binary addition, called carry lookahead, which has a DAG with $O(n)$ vertices and depth $O(\log n)$. i.e. work $= O(n)$ and $T_\infty = O(\log n)$.

To add $n$ binary numbers, we can pair up the numbers and add them up and repeat, resulting in a binary tree of additions. Since each addition is $O(\log n)$ parallel time and there are $\log n$ such additions going from leaf to root of the binary tree, for $T_\infty = O(\log^2 n)$ and work $= O(n^2)$. Can we do better?

#### 2.1.2 The 3-for-2 Trick (Carry-Save Addition)

Here is the central idea. Suppose we have three $n$-bit numbers $X$, $Y$, $Z$. We wish to reduce them to two numbers $S$ and $C$ such that $X + Y + Z = S + C$, in **constant time** on $n$ processors.

Examine a single bit position $k$. We are adding three bits $x_k$, $y_k$, $z_k$. Their sum is between 0 and 3, which fits in two bits $c_k s_k$:

- **Sum bit:** $s_k = x_k \oplus y_k \oplus z_k$ (the XOR, i.e., the parity of the three bits).
- **Carry bit:** $c_k = \text{majority}(x_k, y_k, z_k)$, which is 1 if and only if at least two of the three inputs are 1.

The crucial observation: The sum bit $s_k$ can be placed at position $k$ of $S$ and the carry bit $c_k$ at position $k+1$ of $C$. No carries! Consequently, **all $n$ bit positions can be processed simultaneously**. And yet, these two bits in $C$ and $S$ contribute to the sum of $C$ and $S$ exactly what $x_k$, $y_k$ and $z_k$ contribute to the sum of $X$, $Y$ and $Z$. i.e. $C + S = X + Y + Z$.

#### 2.1.3 Reducing Many Rows to Two

We now have a strategy for the full multiplication problem. We begin with $n$ partial-product rows and apply the 3-for-2 trick repeatedly:

1. Group the $n$ rows into triples. Apply the 3-for-2 trick to each triple simultaneously. This reduces $n$ rows to approximately $2n/3$ rows, in $O(1)$ time.
2. Repeat. After each round the number of rows shrinks by a factor of $2/3$. After $O(\log_{3/2} n) = O(\log n)$ rounds we have exactly two rows.
3. Add the final two rows using a fast adder (Section 2.3). This takes $O(\log n)$ time.

The total span is $O(\log n)$. The total work, counting up all the constant-time additions across all rounds, is $O(n^2)$.

### 2.2 Parallel Prefix (Scan)

We now turn to a primitive of remarkable versatility. The **parallel prefix** operation, sometimes called *scan*, takes a sequence $[x_1, x_2, \ldots, x_n]$ and an associative binary operator $\circ$ and produces the sequence of all partial results:

$$y_k \;=\; x_1 \circ x_2 \circ \cdots \circ x_k, \qquad k = 1, 2, \ldots, n.$$

The familiar example is prefix sums: $\circ = +$, so $y_k = x_1 + x_2 + \cdots + x_k$. But the operation applies equally well to multiplication, logical AND/OR, minimum, maximum, greatest common divisor, matrix multiplication, and many others. The only requirement is **associativity**: $(a \circ b) \circ c = a \circ (b \circ c)$. Commutativity is *not* required.

Associativity is the key: because the grouping of operands does not affect the result, we are free to evaluate sub-expressions in any order, and in particular in *parallel*. This is the secret ingredient that makes efficient parallel computation possible across so many settings.

#### 2.2.1 The Recursive Algorithm

A sequential scan is trivial: iterate left to right, accumulating the running total. The challenge is to do better in parallel. The following recursive strategy achieves $O(\log n)$ span with only $O(n)$ work.

**Step 1 — Reduce.**
Pair up adjacent elements and compute $z_k = x_{2k-1} \circ x_{2k}$ for $k = 1, 2, \ldots, n/2$. This takes $O(1)$ span (all pairs are independent) and produces a sequence $z$ of length $n/2$.

**Step 2 — Recurse.**
Compute the prefix scan of $z$ recursively. Let $y'$ denote the result, so that

$$y'_k \;=\; z_1 \circ z_2 \circ \cdots \circ z_k \;=\; x_1 \circ x_2 \circ \cdots \circ x_{2k}.$$

Observe that $y'_k$ is precisely the prefix of the original sequence up to position $2k$, giving us all even-indexed outputs for free.

**Step 3 — Expand.**
Recover all outputs:

- **Even indices:** $y_{2k} = y'_k$.
- **Odd indices:** $y_{2k+1} = y'_k \circ x_{2k+1}$. The prefix up to position $2k+1$ is the prefix up to $2k$ combined with the next element. All odd-indexed outputs can be computed simultaneously in $O(1)$ span.

The recurrences verify as follows. For work: $T_1(n) = T_1(n/2) + O(n)$, which solves to $T_1(n) = O(n)$ by the Master Theorem. For span: $T_\infty(n) = T_\infty(n/2) + O(1)$, which solves to $T_\infty(n) = O(\log n)$. The algorithm is *work-optimal* (it does asymptotically no more work than the sequential algorithm) and achieves logarithmic span.

### 2.3 Application: Carry-Lookahead Addition

We now have everything needed to add two $n$-bit integers in $O(\log n)$ span. This is the final ingredient in our fast multiplication algorithm, and it is also a beautiful illustration of how parallel prefix can be applied in an unexpected setting.

#### 2.3.1 The Carry Recurrence

Let $a$ and $b$ be the two numbers we are adding, and let $c_i$ be the carry into position $i$, so $c_0 = 0$. For each bit position $i$, define two signals:

$$g_i = a_i \text{ AND } b_i, \qquad p_i = a_i \text{ XOR } b_i,$$

called the *generate* and *propagate* bits respectively. $g_i = 1$ means position $i$ produces a carry regardless of its input; $p_i = 1$ means an incoming carry is passed along. The carry recurrence is:

$$c_{i+1} \;=\; g_i \;\text{ OR }\; (p_i \text{ AND } c_i).$$

This looks stubbornly serial: $c_{i+1}$ depends on $c_i$, which depends on $c_{i-1}$, all the way back to $c_0 = 0$. Computing all carries this way takes $O(n)$ time — the classic *ripple-carry* adder.

The question is whether this serial dependency is genuine or merely apparent. It turns out to be the latter.

### 2.3.2 Linearizing the Recurrence

Using the $g_i$ and $p_i$ notation already established, form the $2 \times 2$ matrix:

$$M_i \;=\; \begin{pmatrix} p_i & g_i \\ 0 & 1 \end{pmatrix}.$$

Represent the carry state as the column vector $\begin{pmatrix} c \\ 1 \end{pmatrix}$. Then the carry recurrence becomes the matrix–vector product:

$$\begin{pmatrix} c_{i+1} \\ 1 \end{pmatrix} \;=\; M_i \begin{pmatrix} c_i \\ 1 \end{pmatrix} \;=\; \begin{pmatrix} p_i & g_i \\ 0 & 1 \end{pmatrix} \begin{pmatrix} c_i \\ 1 \end{pmatrix},$$

where the entries of $M_i$ are bits, matrix "multiplication" uses AND in place of ordinary multiplication and OR in place of addition. One may verify directly: the top row gives $c_{i+1} = (p_i \text{ AND } c_i) \text{ OR } g_i$, which is exactly the carry recurrence. Unrolling, the carry out of position $n$ is:

$$\begin{pmatrix} c_n \\ 1 \end{pmatrix} \;=\; \left(M_{n-1} \cdot M_{n-2} \cdots M_0\right) \begin{pmatrix} c_0 \\ 1 \end{pmatrix}.$$

More generally, the carry into every position $k+1$ is given by the *prefix product* $M_k \cdot M_{k-1} \cdots M_0$ applied to the initial state $c_0 = 0$. We need all $n$ such prefix products.

### 2.3.3 The Punchline

Matrix multiplication is associative. (This is not a coincidence — it reflects the fact that function composition is associative, and matrix multiplication is function composition in linear algebra's clothing.) Therefore we can apply the parallel prefix algorithm with $\circ$ being matrix multiplication (under AND/OR arithmetic) over our $2 \times 2$ matrices.

This gives us all $n$ prefix matrix products — and therefore all $n$ carry bits — in $O(\log n)$ span and $O(n)$ work. Once all carries are known, the sum bits $s_i = a_i \text{ XOR } b_i \text{ XOR } c_i$ can be computed in $O(1)$ additional parallel time.

*Remark.* Stepping back: the carry-lookahead adder illustrates a general principle. When a computation appears to be a sequential chain of updates, ask whether each update is a *linear function* of the previous state. If so — and if the update functions compose associatively — then the parallel prefix framework applies immediately. The apparent serialness is illusory; the computation is in fact highly parallel. We will see this principle recur in settings ranging from dynamic programming to string matching to numerical simulation.

## 3 Summary

The three algorithms above form a coherent chain: parallel prefix on matrices gives carry-lookahead addition; carry-lookahead addition, together with carry-save reduction, gives fast multiplication.

| **Algorithm** | **Work** ($T_1$) | **Span** ($T_\infty$) | **Key Idea** |
|---|---|---|---|
| Parallel Prefix (Scan) | $O(n)$ | $O(\log n)$ | Associativity |
| Carry-Lookahead Addition | $O(n)$ | $O(\log n)$ | Linear recurrence $\to$ prefix |
| Integer Multiplication | $O(n^2)$ | $O(\log n)$ | Carry-save + scan |

These results should be understood not as isolated achievements but as instances of a single theme: *apparent serialness, when examined carefully, often dissolves into a parallel prefix computation.* The framework is so versatile that mastering it is perhaps the single most important step toward fluency in parallel algorithm design.