# Parallel Algorithms

CS 263

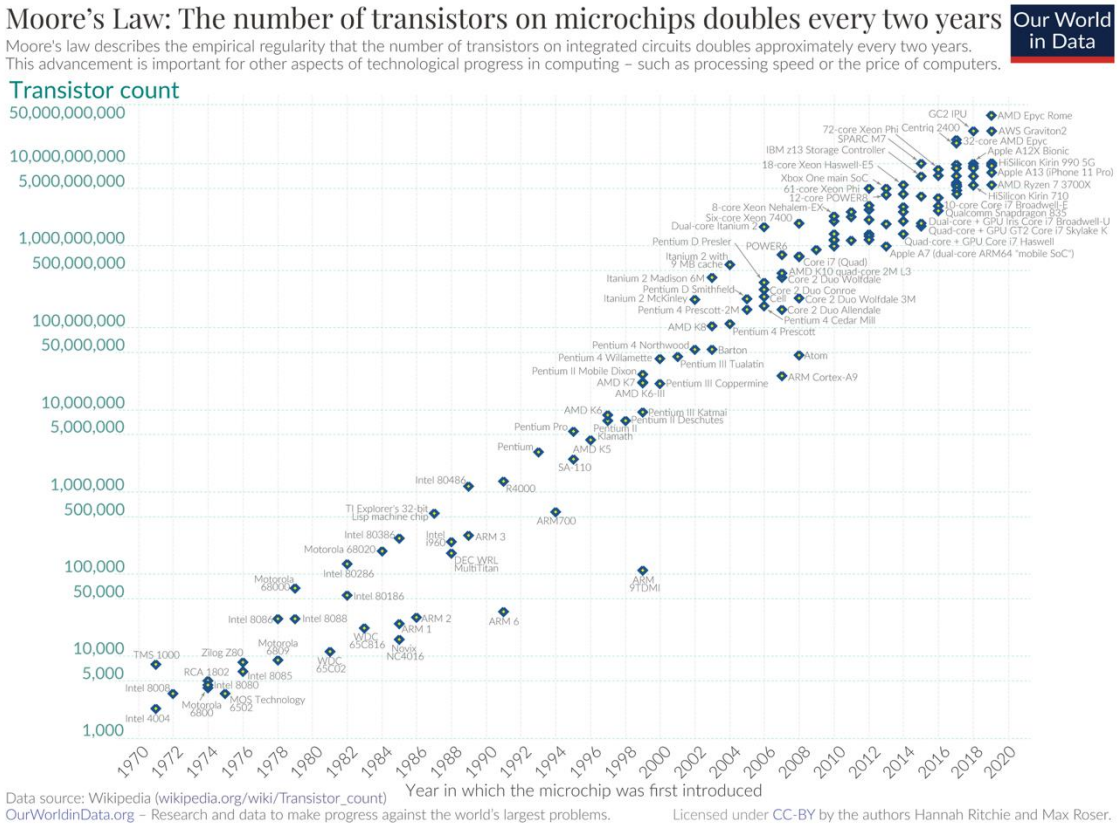
Michael T. Goodrich

University of California, Irvine

Some slides adapted from content by Joseph Jaja, Yun Gu, and Yihan Sun

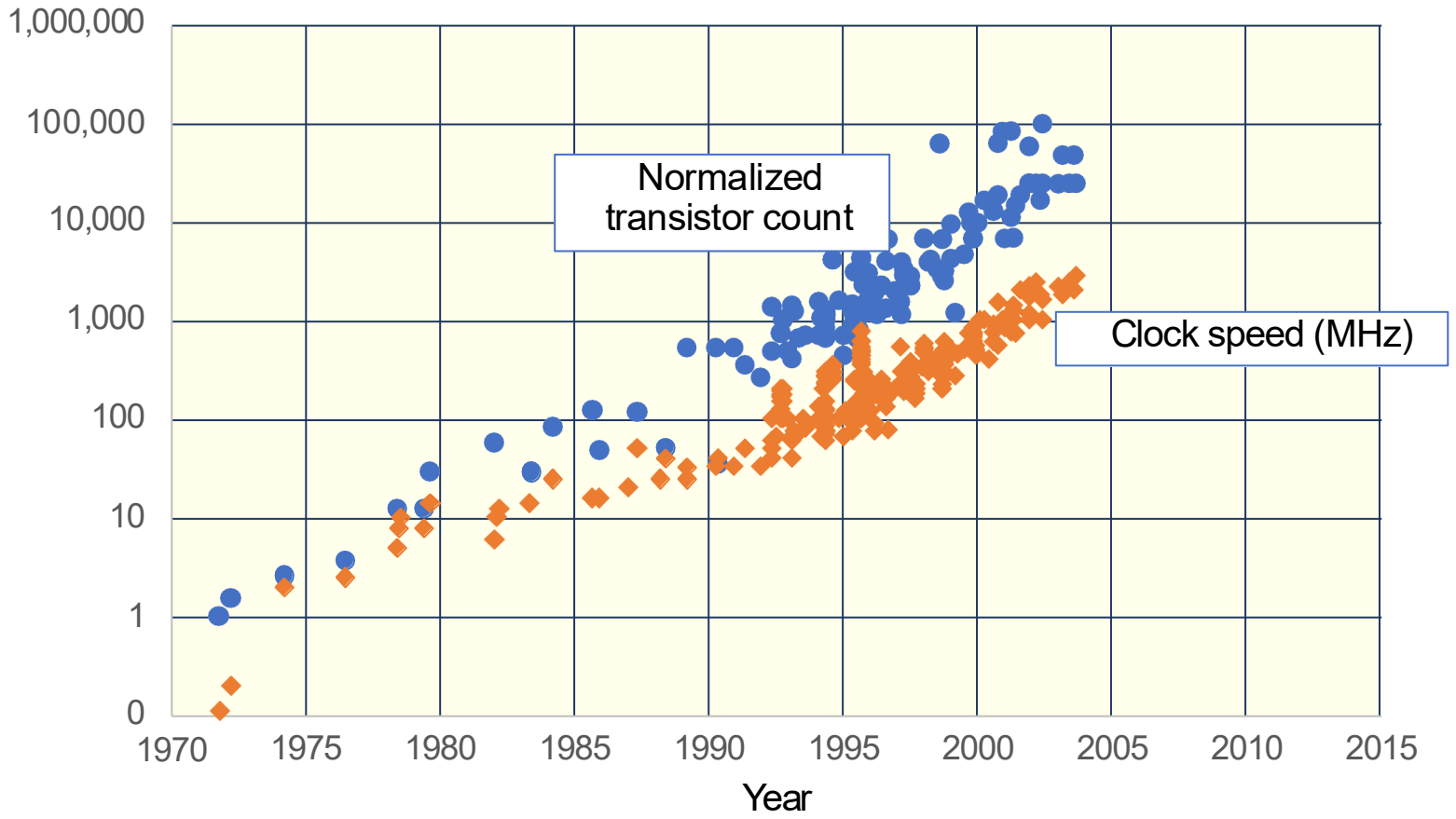
# Moore's Law

- Moore's "law" is the observation that the number of transistors in an integrated circuit doubles about every two years.




Gordon Moore
(Intel co-founder)

# Technology Scaling – Clock Speed Kept Pace Until 2004




Stanford's CPU DB [DKM12]

# Advances in Hardware

## Apple computers with similar prices from 1977 to 2004

**Apple II**

| | |
|---|---|
| Launched: | 1977 |
| Clock rate: | 1 MHz |
| Data path: | 8 bits |
| Memory: | 48 KB |
| Cost: | $1,395 |

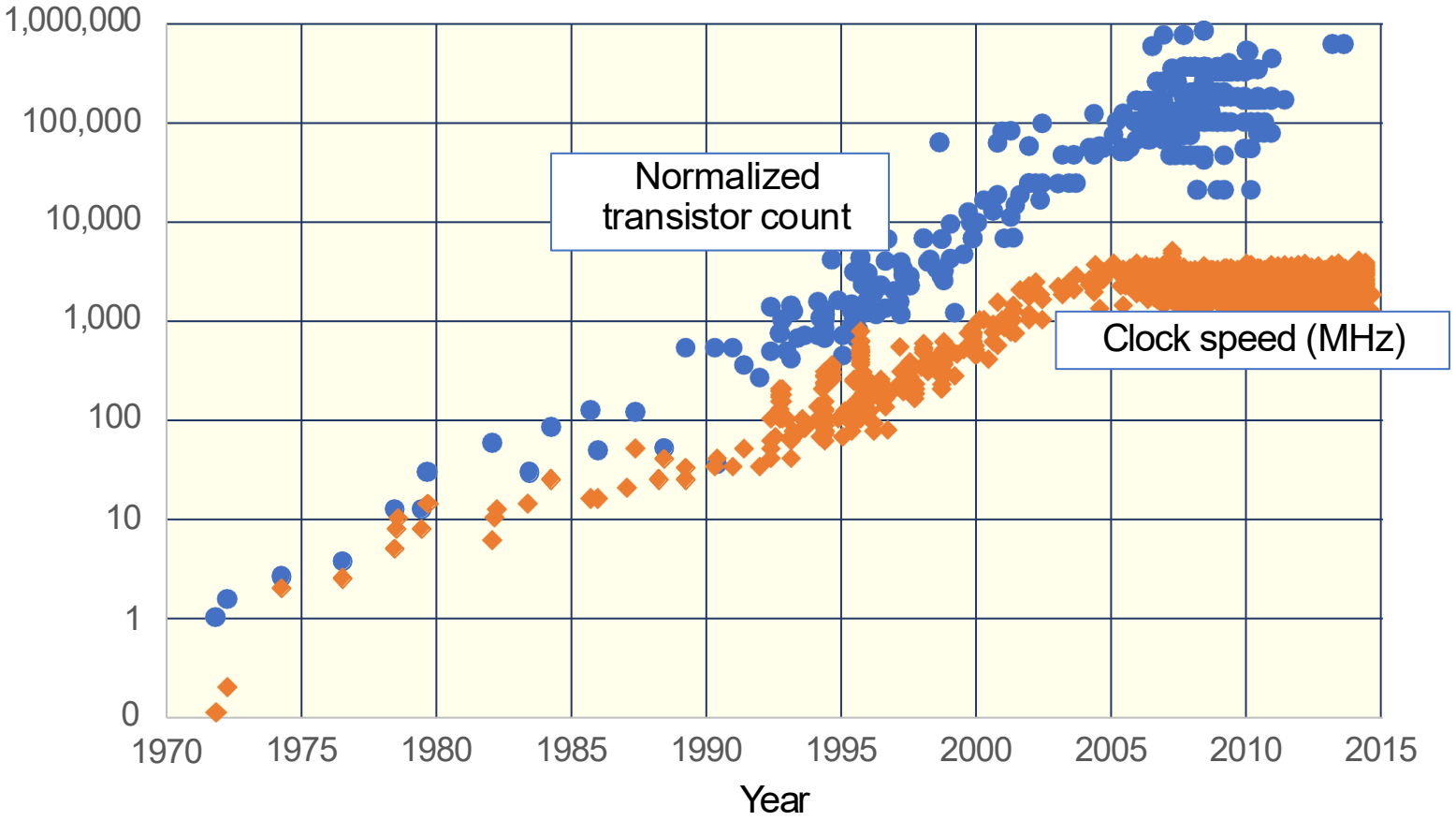
**Power Macintosh G4**

| | |
|---|---|
| Launched: | 2000 |
| Clock rate: | 400 MHz |
| Data path: | 32 bits |
| Memory: | 64 MB |
| Cost: | $1,599 |

**Power Macintosh G5**

| | |
|---|---|
| Launched: | 2004 |
| Clock rate: | 1.8 GHz |
| Data path: | 64 bits |
| Memory: | 256 MB |
| Cost: | $1,499 |

# Technology Scaling After 2004 – Clock Speed Hit a Wall

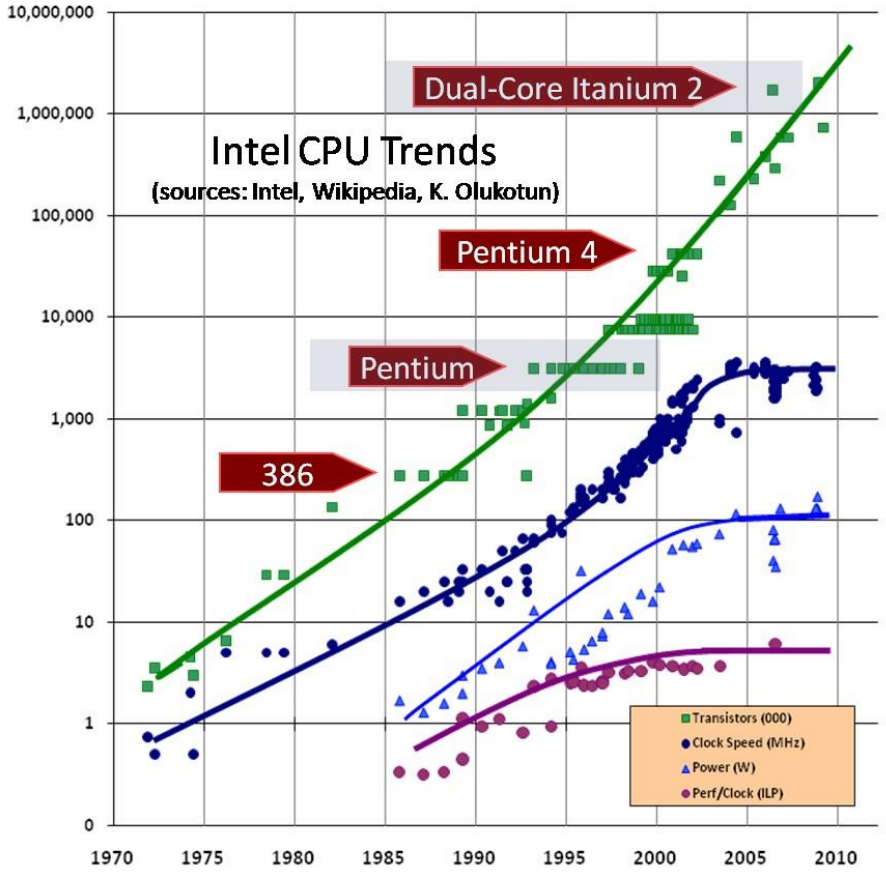



Stanford's CPU DB [DKM12]

# Moore's Law Misquote

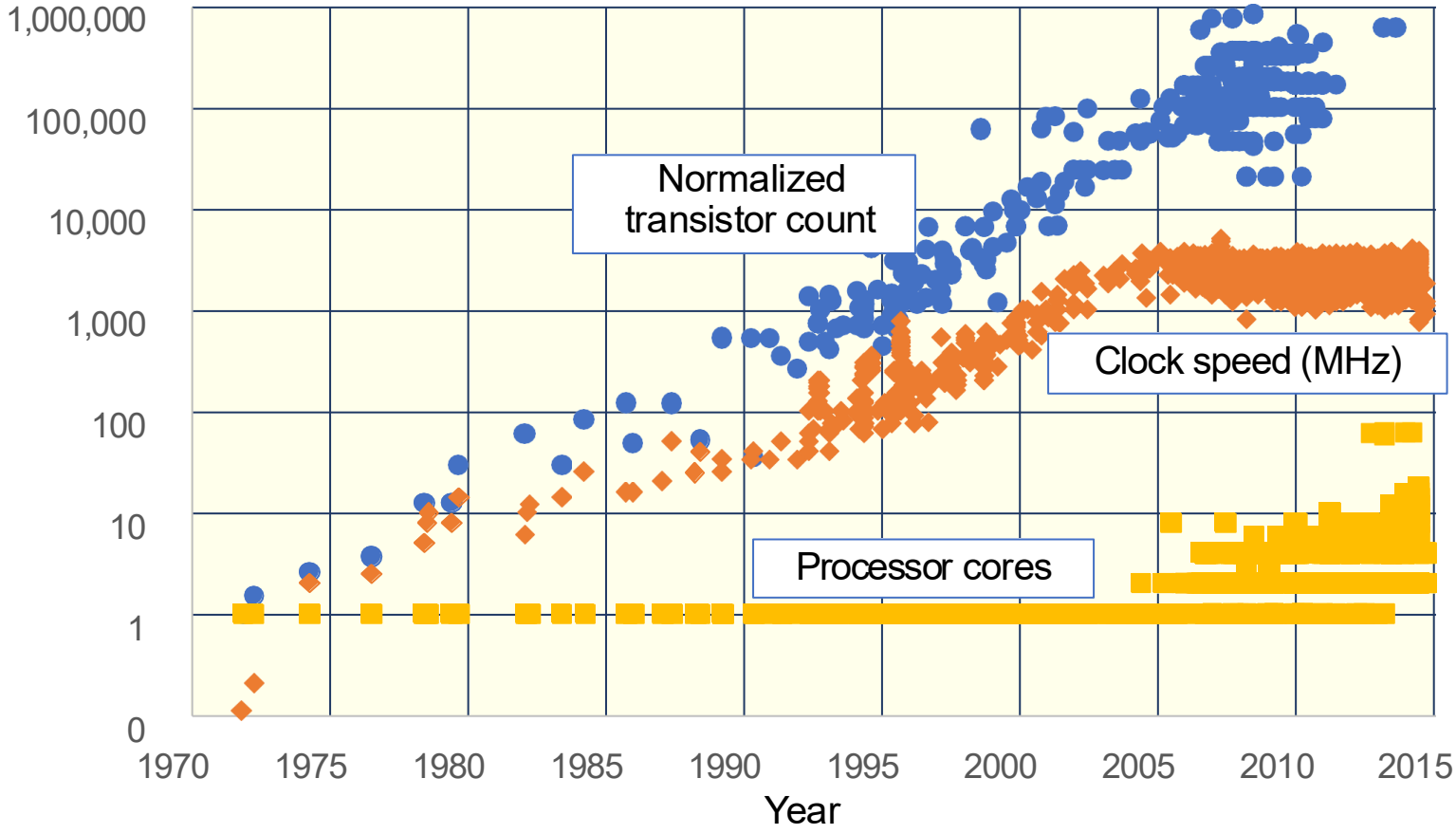
- Not clock speed

Grace Hopper gave out "nanoseconds" to prove this point.



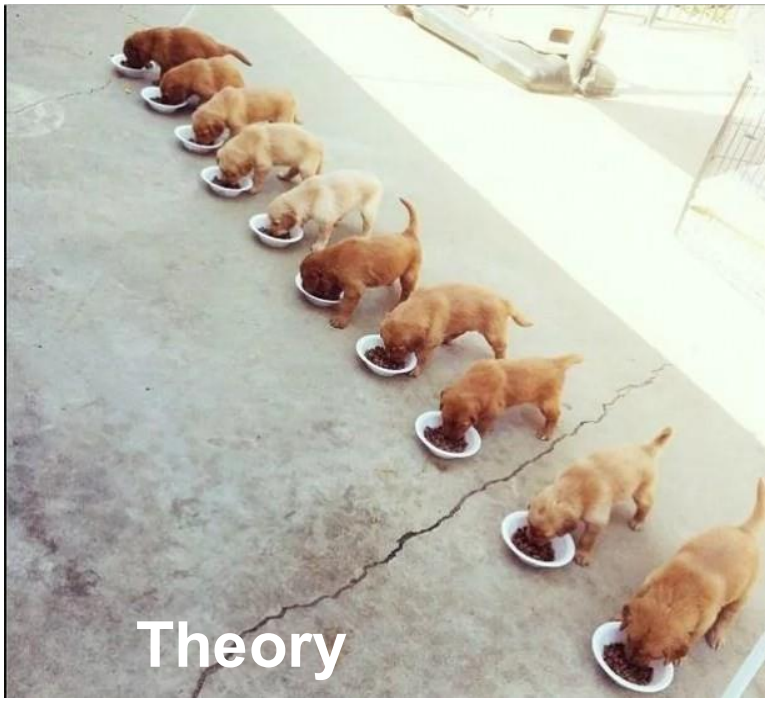

Her prediction: Computers are going to have to utilize parallel processing.

# Technology Scaling: Multicore CPUs




Stanford's CPU DB [DKM12]

# Parallel Computing: Theory and Practice

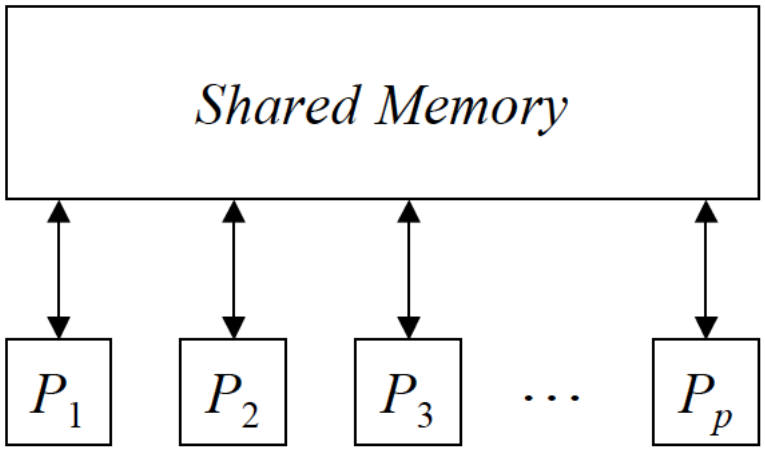




(Pictures from 9gag.com)

# The PRAM Model

- Parallel Random Access Machine (PRAM)
- Extension of RAM: each processor is a RAM
- Processors operate **synchronously**
- Memory space is shared among all processors

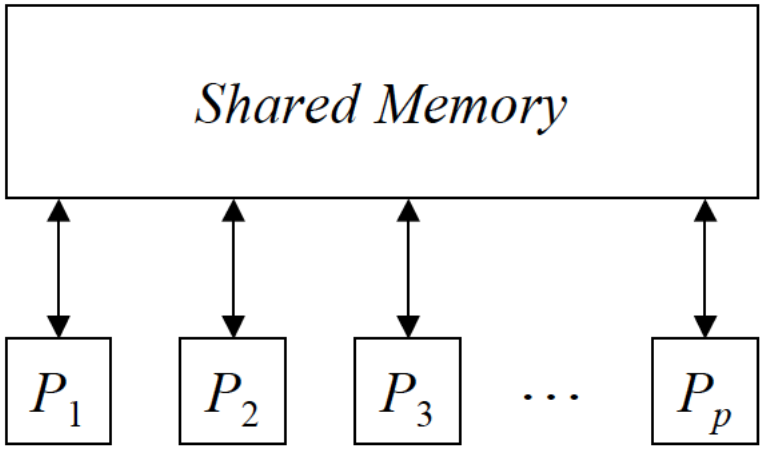


# PRAM Variations

- A PRAM step ("clock cycle") consists of three phases
  1. **Read**: each processor may read a value from shared memory
  2. **Compute**: each processor performs operations on local data
  3. **Write**: each processor may write a value to shared memory
- Model is refined for concurrent read/write capability
  - Exclusive Read Exclusive Write (EREW)
  - Concurrent Read Exclusive Write (CREW)
  - Concurrent Read Concurrent Write (CRCW)
- CRCW PRAM is further refined as follows:
  - **Common CRCW**: all processors must write the same value
  - **Arbitrary CRCW**: one of the processors succeeds in writing
  - **Priority CRCW**: processor with highest priority succeeds in
  - writing

# Work and Span for a PRAM Algorithm

- The **work**, W, done in a T-step PRAM algorithm:
  - Let $P_i$ be the number of active processors in step i
  - Then work W = sum of $P_i$ values, for i=1,2,…,T.
- The **span** for a PRAM algorithm is its number of parallel steps.
- Span corresponds to parallel **time**.
- Work is the **total number of computations** done.



# Brent's Meta-theorem for PRAMs

- A T-step PRAM algorithm with work W can be implemented with P processors in time O(T+W/P).
- This ignores the problem of how to assign tasks to the P processors.
  - But sometimes this processor allocation is easy

# Example Computations

- OR of n bits
  - O(1) time in CRCW PRAM (all versions)
  - O(log n) time in CREW/EREW PRAM
- Broadcast a value to all processors
  - O(1) time in CREW/CRCW PRAM
  - O(log n) time in EREW PRAM

# CRCW PRAM Maximum Finding

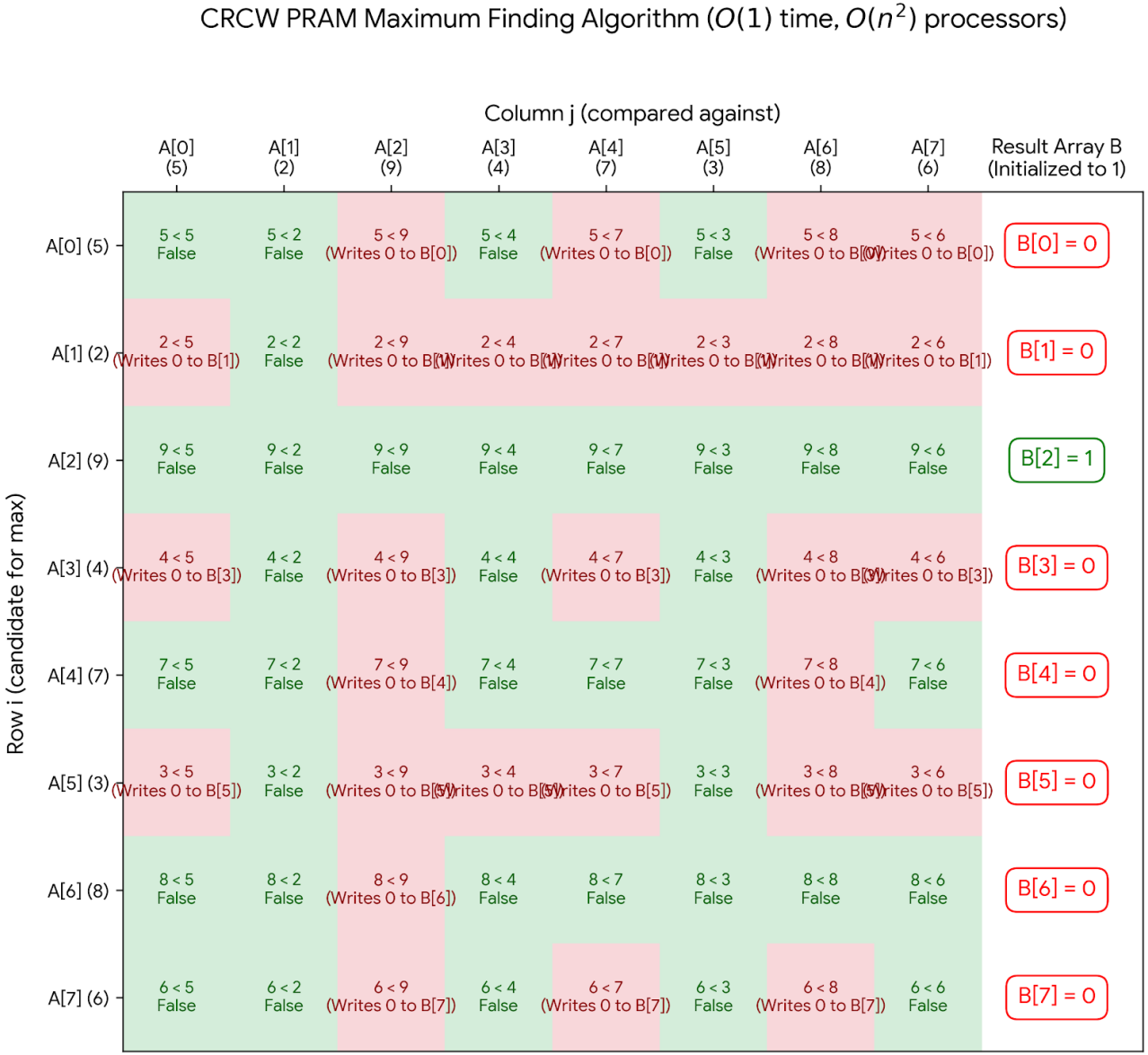




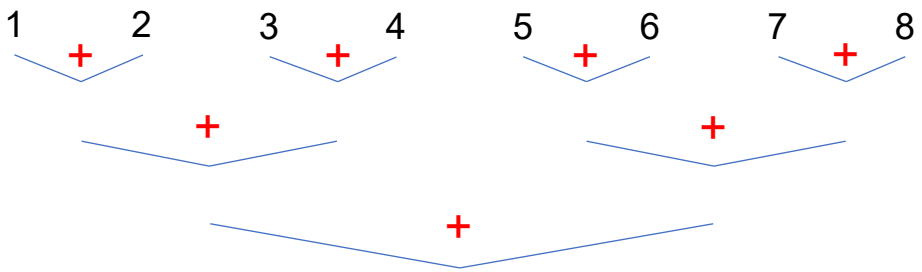

Image generated using Nano Banana Pro

# CRCW PRAM Maximum Finding

- Use divide-and-conquer but divide into $O(n^{1/2})$ groups of size $O(n^{1/2})$ each, recursively find the maximum in each group.
- Use the constant-time maximum-finding algorithm on the results.
- Time: $T(n) = T(n^{1/2}) + 1$, which implies $T(n)$ is $O(\text{loglog } n)$.
- Work: $W(n) = n^{1/2}\, W(n^{1/2}) + n$, which means $W(n)$ is $O(n \text{ loglog } n)$.

# The Reduce Operation - Summing n numbers

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




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



16

# Two algorithms to implement a reduce

**Top-down**

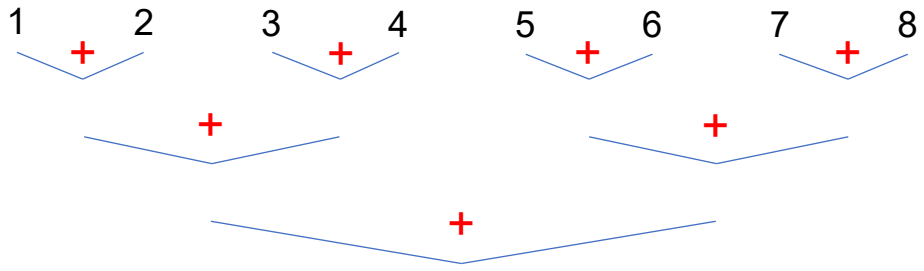
```
reduce(A, n) {
    if (n == 1) return A[0];
    parallel_do:
        L = reduce(A, n/2);
        R = reduce(A + n/2, n-n/2);
    return L+R;
}
```

**Bottom-up**

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

**Divide-and-conquer:**
Dealing with the left and right halves recursively, in parallel

**Decrease and conquer:**
Shrink the original size into a half, in parallel




Both run in O(log n) time using n processors on an EREW PRAM.

# Alternative Model: Binary fork-join

- **All computation start from a thread**
- **A thread can:**
  - Do normal operations as in RAM model (arithmetic operations, memory access)
  - **Fork**: create one new thread to run in parallel (1->2); can be **nested and/or repeated**
  - **Join**: synchronize with previous forked threads

**Top-down**

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

**Top-down using fork-join**

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

# More on Binary fork-join Model

- **All computation start from a thread**
- **A thread can:**
  - Do normal operations as in RAM model (arithmetic operations, memory access)
  - **Fork**: create one new thread to run in parallel (1->2), can be done in a **nested** manner
  - **Join**: synchronize with previous forked thread

**Bottom-up**

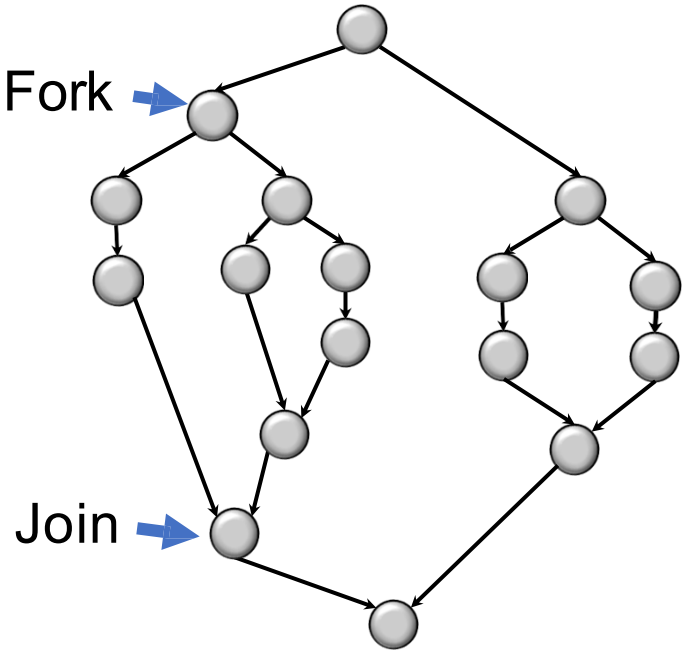
```
reduce(A, n) {
  if (n == 1) return A[0];
  if (n is odd) n=n+1;
  parallel_for i=1 to n/2
    B[i]=A[2i]+A[2i+1];
  return reduce(B, n/2); }
```

Here since it's forking out n/2 threads, it's generally more expensive than just forking out one thread – requires O(log n) steps of binary fork operations

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

# Viewing a Binary fork-join Algorithm as a DAG

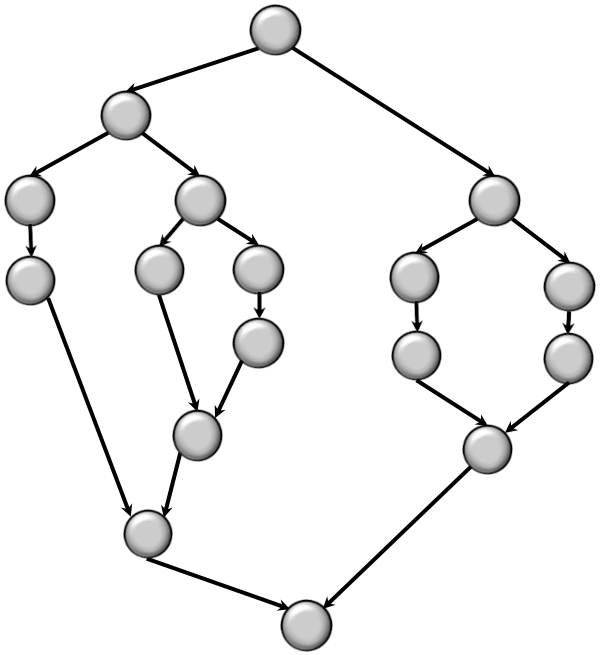
- **For all computations, we can draw a DAG (directed acyclic graph)**
  - A->B means that B can be performed only when A has been finished




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

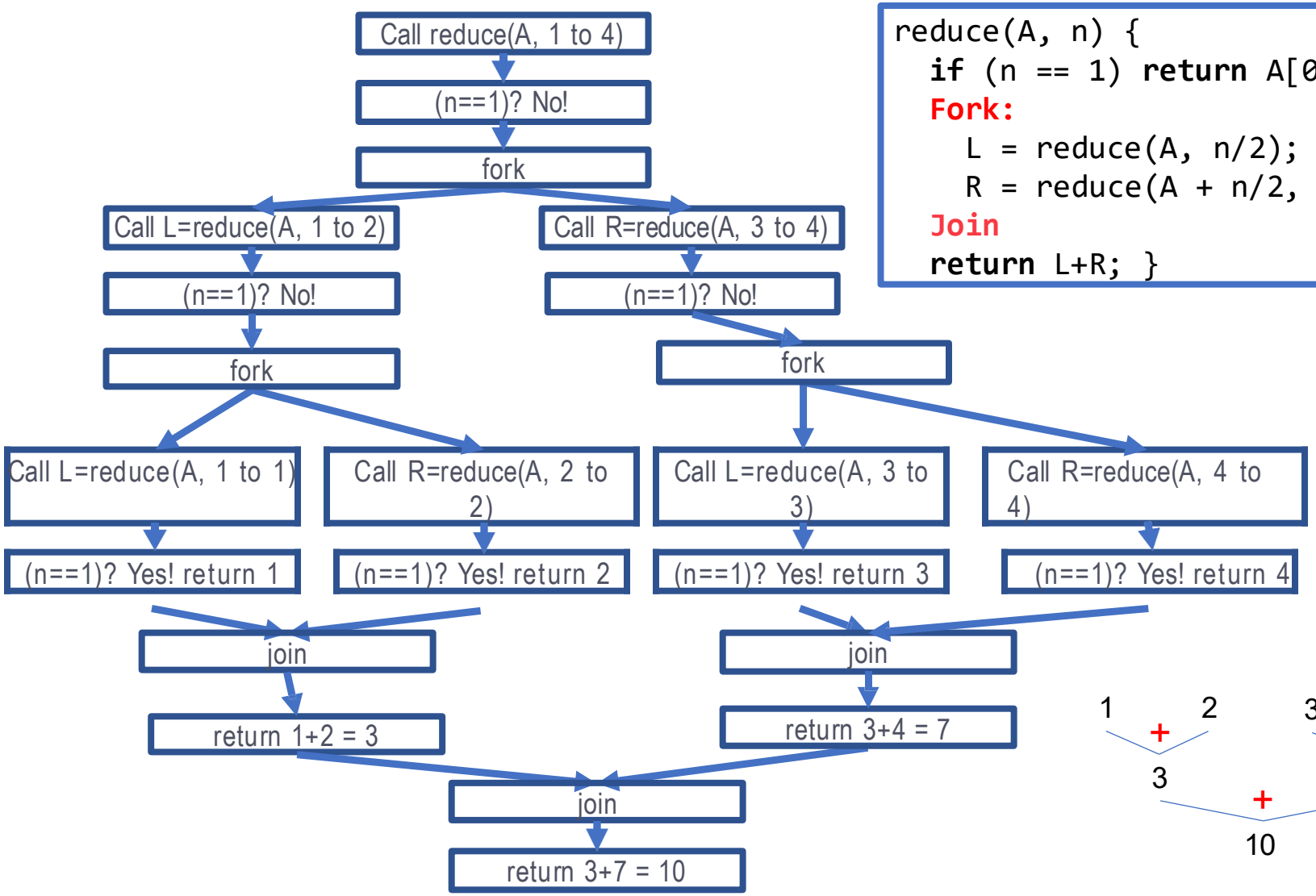
# Binary Fork-Join Work and Span

- **For all computations, imagine a DAG**
  - A->B means that B can be performed only when A has been finished
- **Unit cost per operation, including fork and join**



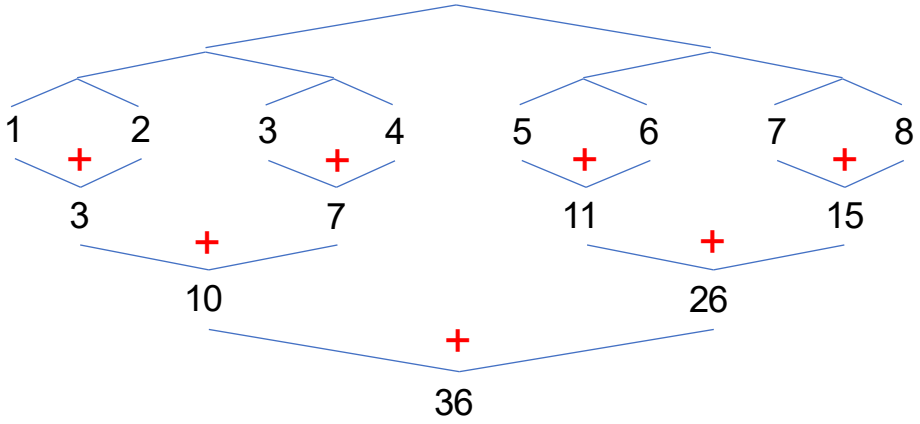
- **Work: the total number of operations**
- **Span (depth): the longest length of sequential dependency chain (the "*critical path*")**

# Computational DAG




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

# Work



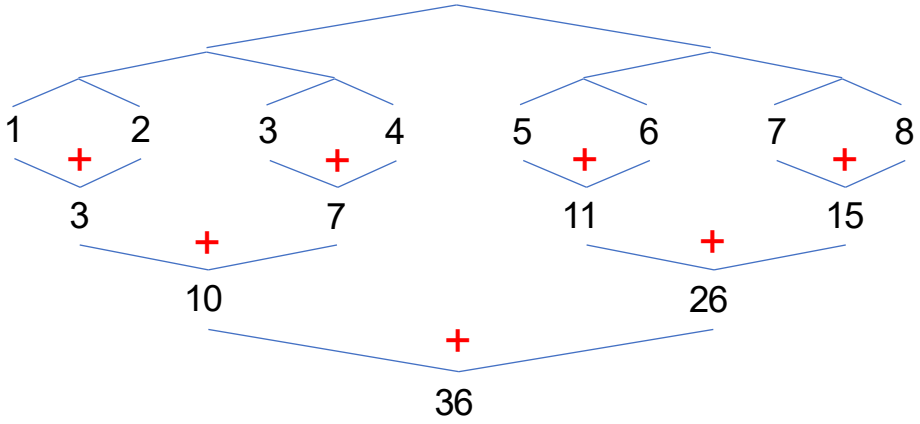

```
reduce(A, n) {
    if (n == 1) return A[0];
    parallel_do:
        L = reduce(A, n/2);
        R = reduce(A + n/2, n-n/2);
    return L+R;
}
```

What is the work?

A. O(log n)
B. O(n)
C. O(n log n)

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

# Span



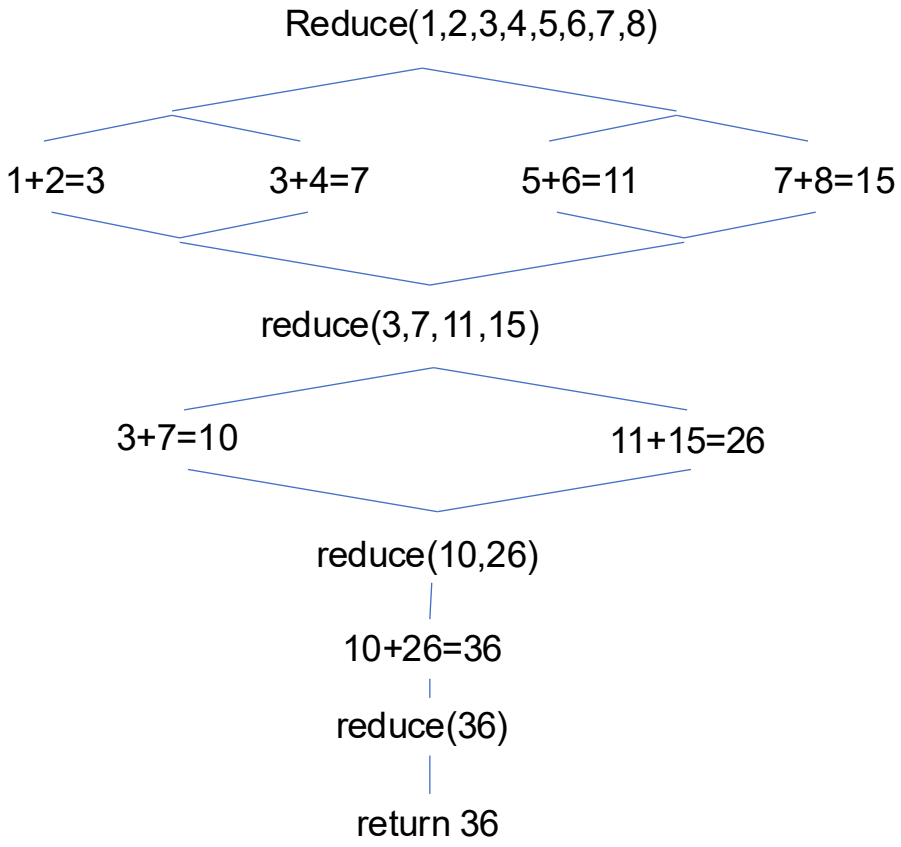

```
reduce(A, n) {
    if (n == 1) return A[0];
    parallel_do:
        L = reduce(A, n/2);
        R = reduce(A + n/2, n-n/2);
    return L+R;
}
```

What is the span?

A. O(log n)
B. $O(\log^2 n)$
C. O(n)
D. O(n log n)

- **Span (depth): The longest dependency chain**
  - Total time required if there are infinite number of processors
  - **Low span**: Logarithmic is ideal, or at least it should be asymptotically lower than work (sequential time complexity)
  - Goal: make the parallel algorithm faster and faster when more and more processors are available - scalability

# Work for the other reduce algorithm




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

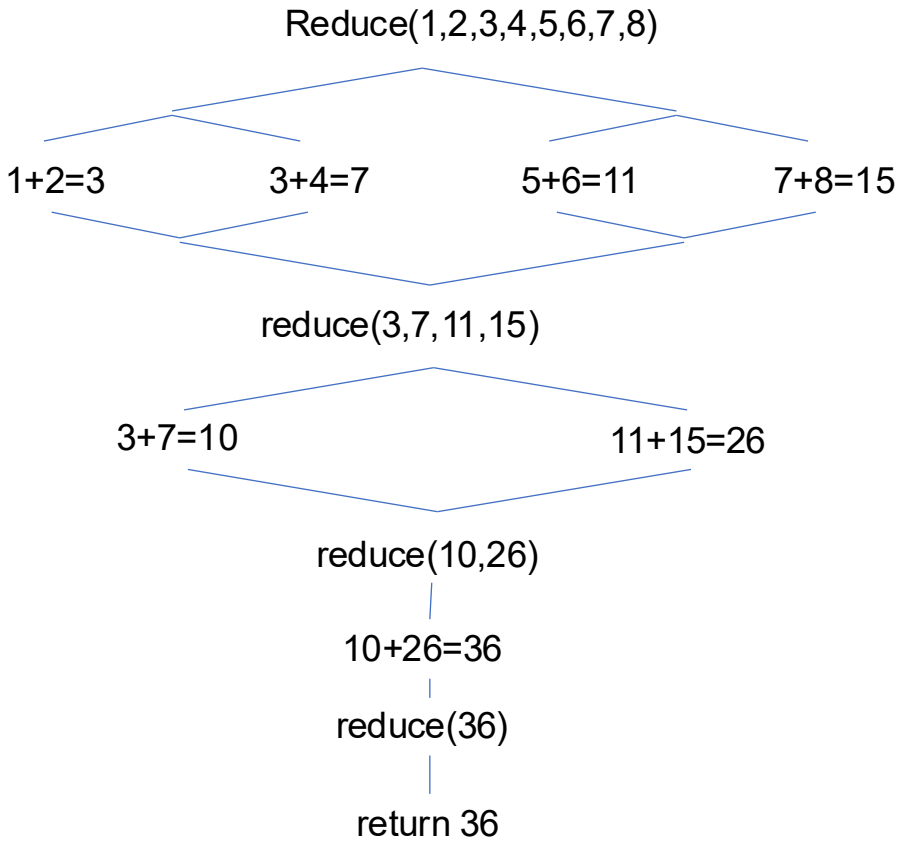
What is the work?

A. O(log n)

B. O(n)

C. O(n log n)

# Span for the other reduce algorithm




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

What is the span?

A. O(log n)
B. O($\log^2$ n)
C. O(n)
D. O(n log n)

# Simulating a PRAM in the Fork-Join Model

- Perform O(log P) forks and joins for each step to the P active processors.
- Work is the same as for the PRAM algorithm.

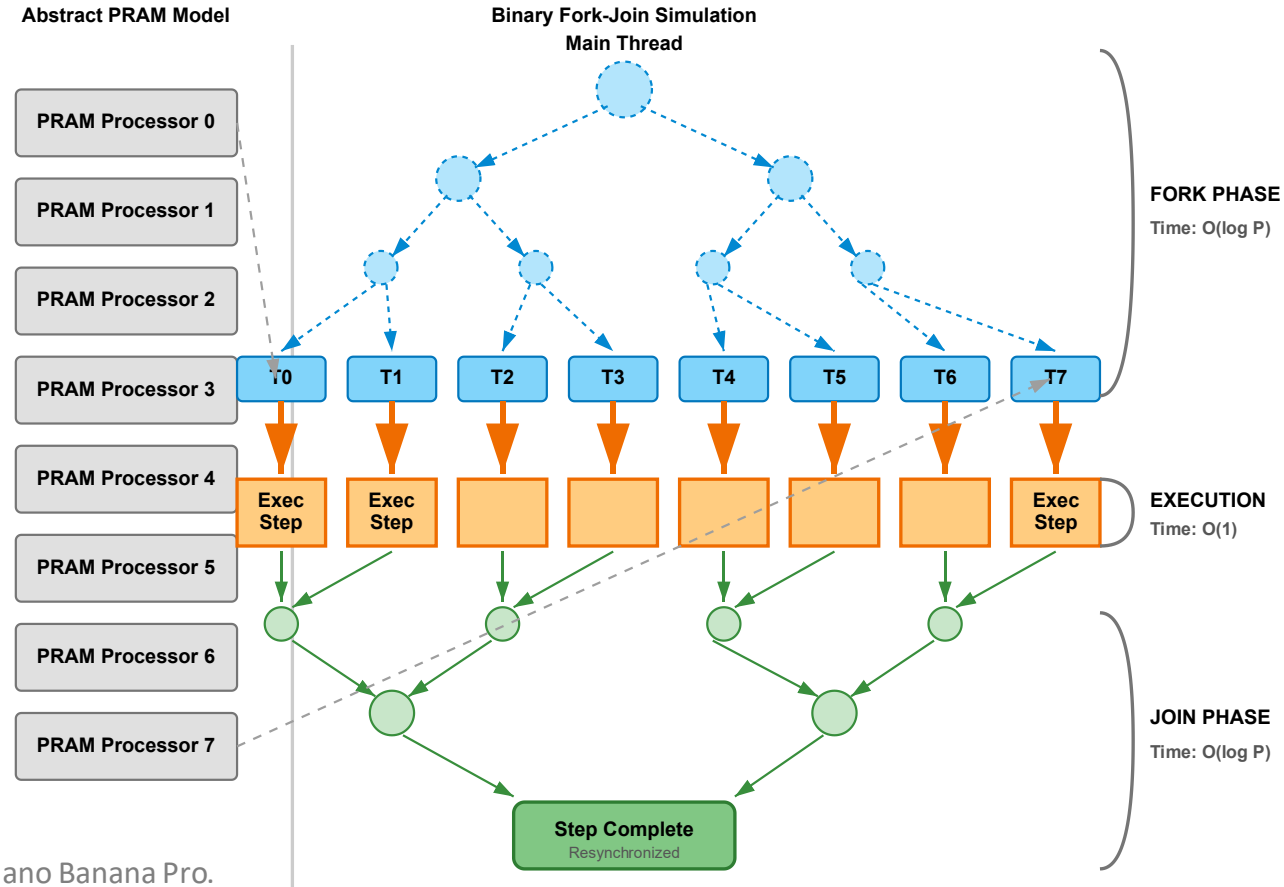
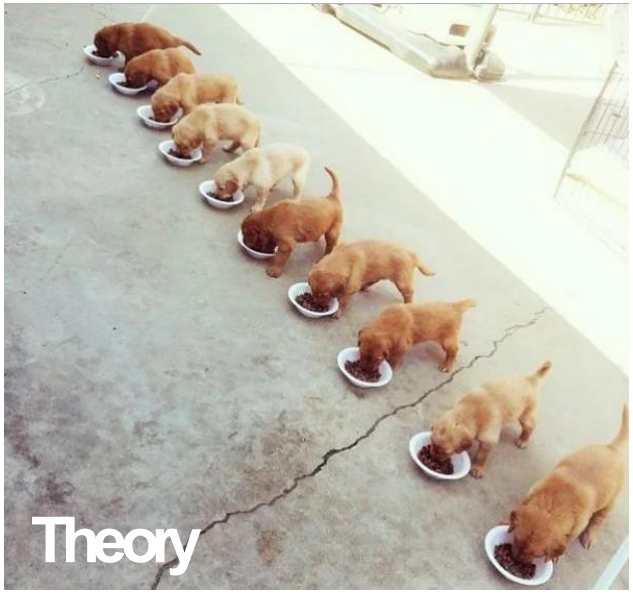



Image generated using Nano Banana Pro.

# Why is parallelism hard in practice?

**Concurrency!**





(Pictures from 9gag.com)

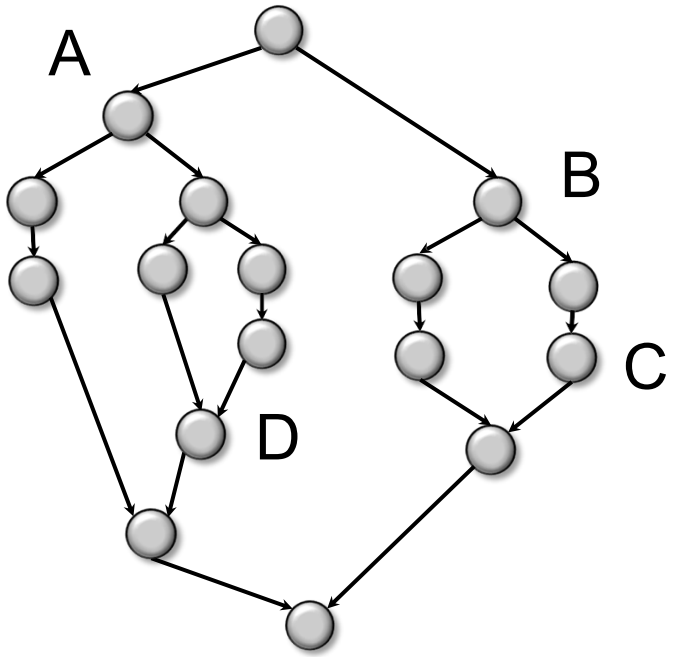
# Concurrency!

- **Things can be executed at the same time or "almost" same time**
- **Scheduling is unknown**
- **Processors are asynchronous**
  - delays from cache misses, processor pipelines, branch prediction, hyper-threading, changing clock speeds, interrupts, …
- **Relative ordering for operations is unknown**
- **Race conditions (e.g., between multiple reads and writes)**

Image from https://www.nytimes.com/2022/02/18/sports/olympics/speedskating-mass-start-joey-mantia.html

# Which tasks are “parallel”?

- **Two threads are “logically parallel” if they CAN run in parallel**
- **Are they parallel?**
  - A and B?
  - A and C?
  - A and D?
  - B and C?
  - C and D?
- **As long as X is not an ancestor of Y on the DAG, X and Y can be parallel**



# Race Conditions

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

```
int bal = 5;
void deposit(x) {
  bal = bal + x;
}
```

balance = 5

P1: deposit(3)

P2: deposit(4)

# Race Example 1

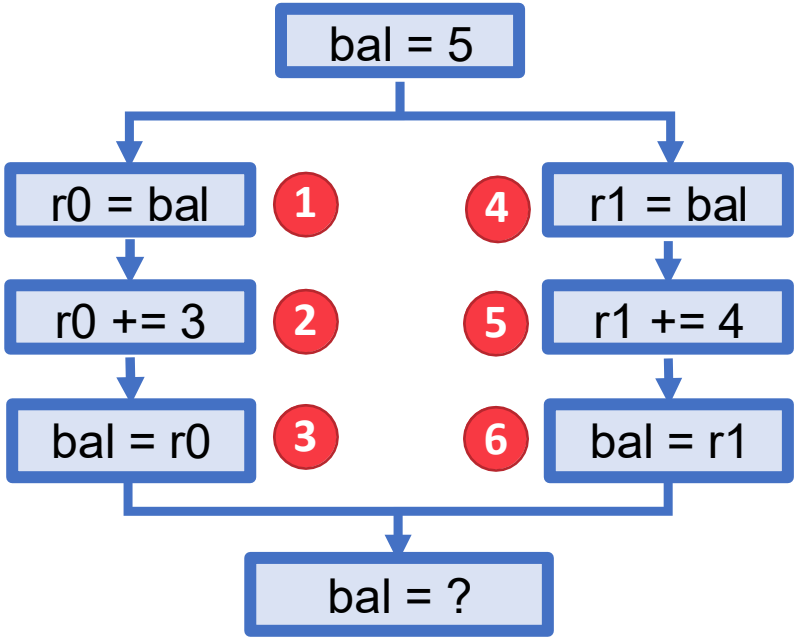
- **Race occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.**

```
int bal = 5;
void deposit(x) {
  bal = bal + x;
}
```

balance = 5

P1: deposit(3)

P2: deposit(4)



# Race Example 2

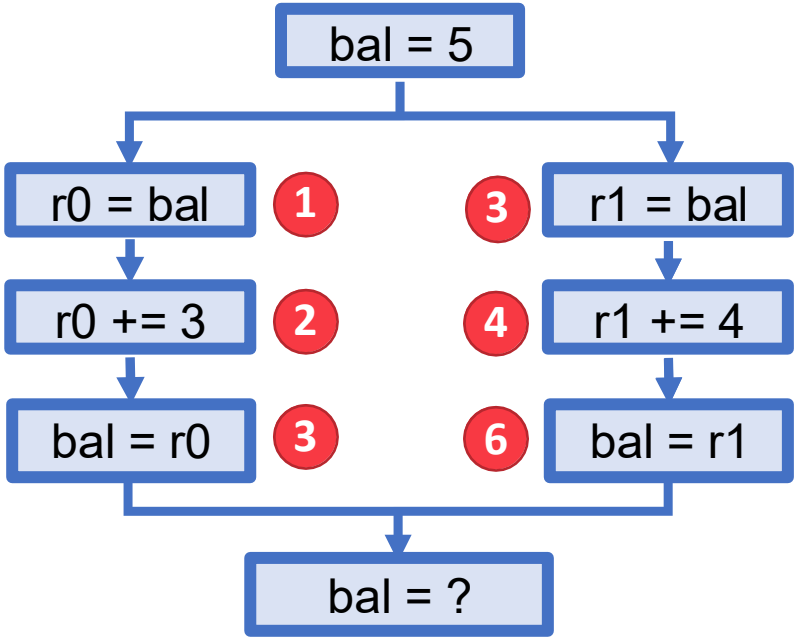
- **Race occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.**

```
int bal = 5;
void deposit(x) {
  bal = bal + x;
}
```

balance = 5

P1: deposit(3)

P2: deposit(4)



# Types of Race Conditions

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

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

# “Atomic” operations

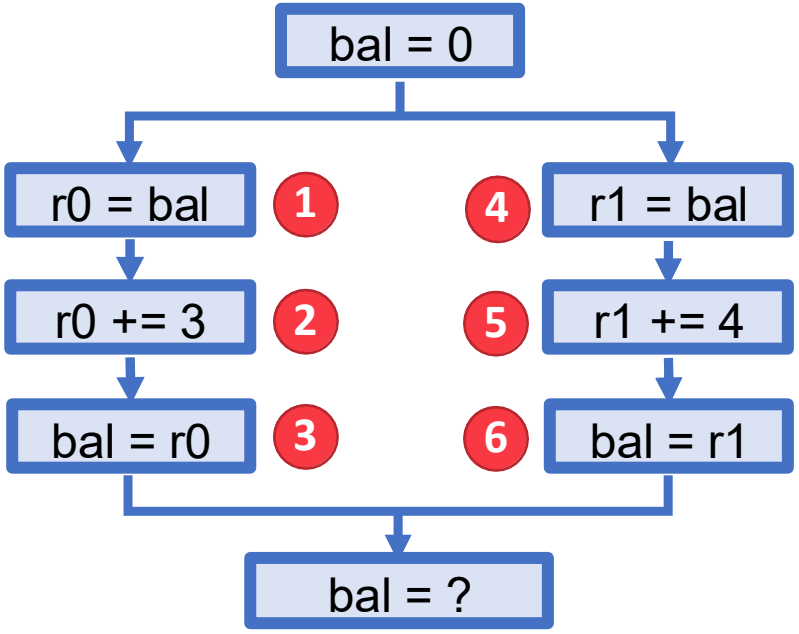
- **“Atomic operations” are operations that cannot be interfered by other operations**
- **Race occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.**

```
int bal;
void deposit(x) {
  bal = bal + x;
}
```

balance = 5

P1: deposit(3)

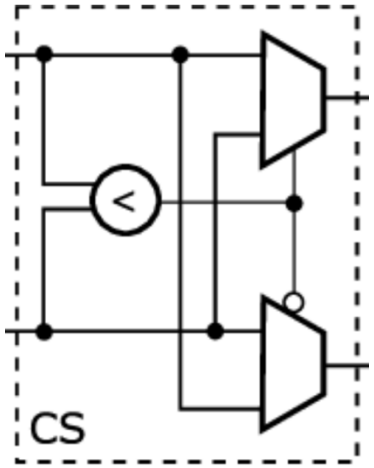
P2: deposit(4)




The process of “deposit” should happen **atomically**!
No other operations (from other threads) can interrupt in the middle!

# An Atomic Operation: compare_and_swap

- **Compare-and-swap (CAS)**
  - **bool** CAS($p$, $v_{old}$, $v_{new}$): compare the value stored in the pointer $p$ with value $v_{old}$, if they are equal, change $p$'s value to $v_{new}$ and return **true**. Otherwise do nothing and return **false**.
- **Supported by Intel and AMD machines**
- **Other operations can be simulated using compare-and-swap**
- **Can assume unit-cost**
- **Allows us to simulate some CRCW operations**



# Guarantee correctness

- **When we have concurrency, it's easy to get things wrong…**
- **Correctness is the first consideration (more important than performance!)!**
- **You must show your parallel algorithm is correct, either using PRAM synchronization, fork-join synchronization, or correctly using atomic operations.**

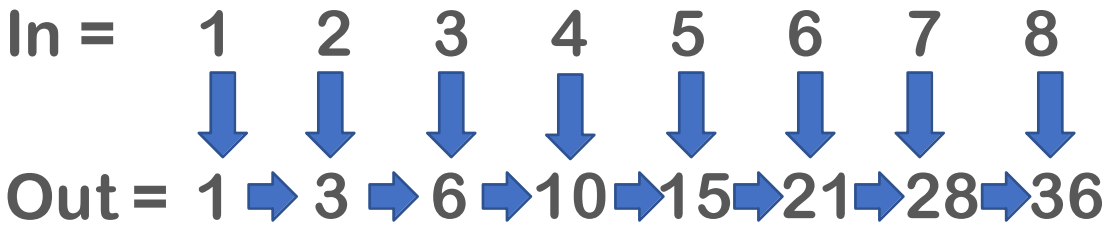
**Job interviewer:** It says in your CV that you are quick at mathematics. What is 17x19?

**Me:** 36

**Interviewer:** That's not even close

**Me:** But it was quick

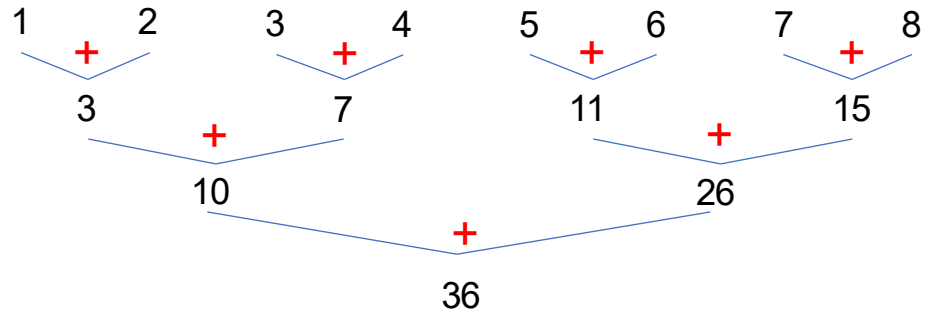
# Prefix sum



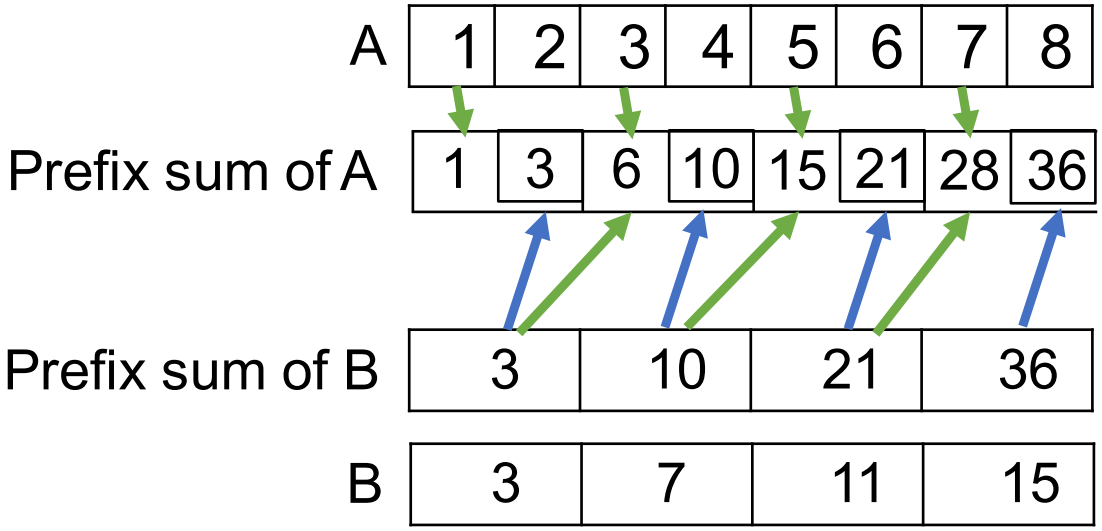

**The most widely-used building block in parallel algorithm design**

```
scan (A[1..n]) {
  B[0] = A[0];
  for i = 1 to n
    B[i] = B[i-1] + A[i];
  return B[1..n];
}
```

# Prefix Sum Parallel Algorithm




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

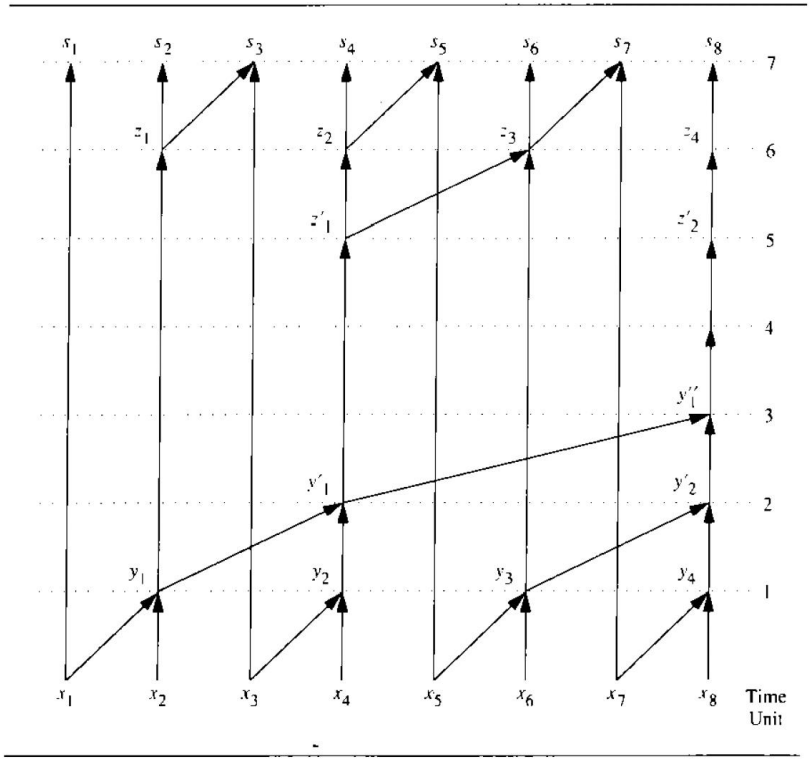



- Shrink the problem size into $\frac{n}{2}$, possibly in parallel
- Solve the same problem on $\frac{n}{2}$
- Convert the result of the subproblem to the final answer, possibly in parallel.

# Implementations of the prefix sum algorithm

- **EREW PRAM prefix sum algorithm:**
  - $O(n)$ work, $O(\log n)$ span
  - E.g., can be done in O(log n) time using O(n/log n) processors.

- **Binary fork-join algorithm:**
  - $O(n)$ work, $O(\log n)$ span
  - Slightly more complicated, e.g., can be done by two divide-and-conquer methods, one going bottom-up and the other going top-down.



# Application: Array Compaction

- Suppose you have an array of data and you want to filter out certain elements (e.g., keep only the positive numbers) and pack the remaining elements into a dense, gapless array.
- **The Problem:** In a parallel environment, if multiple processors try to write their surviving elements to a new array simultaneously, they will overwrite each other unless they know exactly ***where*** to write.
- **The Prefix Sum Solution:**
  1. Create a temporary array of 1s and 0s: assign 1 to elements you want to keep, and 0 to elements you want to discard.
  2. Run a parallel prefix sum (using addition) on this 1/0 array.
  3. The result of the prefix sum gives each surviving element its **exact, unique target index** in the new, compacted array.

# Application: Range-Sum Queries

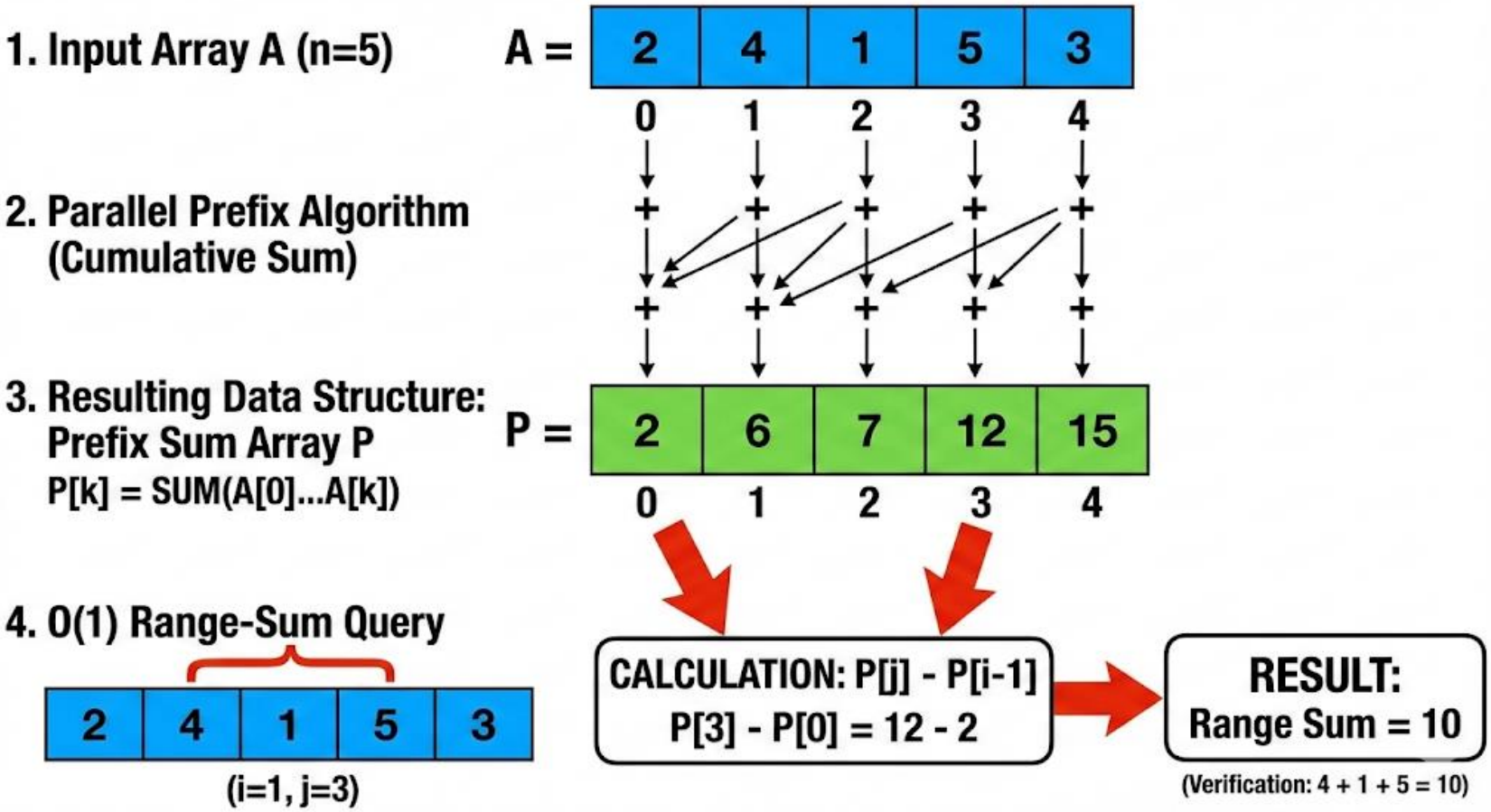



Image generated using Nano Banana Pro

# Extension: Any Associative Operation

- **Running Maximums, Minimums, or Logic**
  - The prefix algorithm doesn't just work with addition. It works with any associative operator.
- **Running Max:** If you have an array of daily stock prices and you want to find the highest price seen *up to that point* for every single day in history, you run a prefix sum but replace the addition operator + with the max() function.
- Note that this can't be used for range-minimum or range-maximum queries, since min and max don't have inverses.