CPSC 4240/5240: Parallel Programming Techniques

Lecture 3: Modeling Parallelism

Lecturer: Quanquan C. Liu

Spring 2026

Special thanks to **MIT 6.106** and the FASTCODE Community for part of this lecture's content!

Parallel Loops

How does parallel for loops help us in matrix multiplication?

```
for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Which of these for loops can be parallelized without issues?

What are the dependencies for each computation?

Parallel Loops

How does parallel for loops help us in matrix multiplication?

```
for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Which of these for loops can be parallelized without issues?

Care about: **writes**; two threads should not write to the same location

Can **read in parallel**

Parallel Loops

```
parallel_for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    parallel_for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Both of these loops can theoretically be parallelized

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \cdot \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

Parallel Loops

```
parallel_for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    parallel_for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Both of these loops can theoretically be parallelized

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \cdot \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

Parallel Loops

```
parallel_for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    parallel_for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

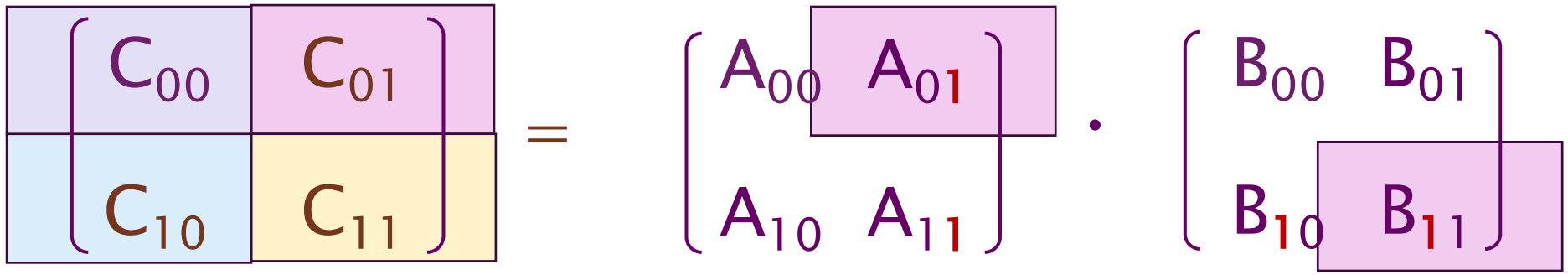
Both of these loops can theoretically be parallelized

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \cdot \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

Parallel Loops

```
parallel_for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    parallel_for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

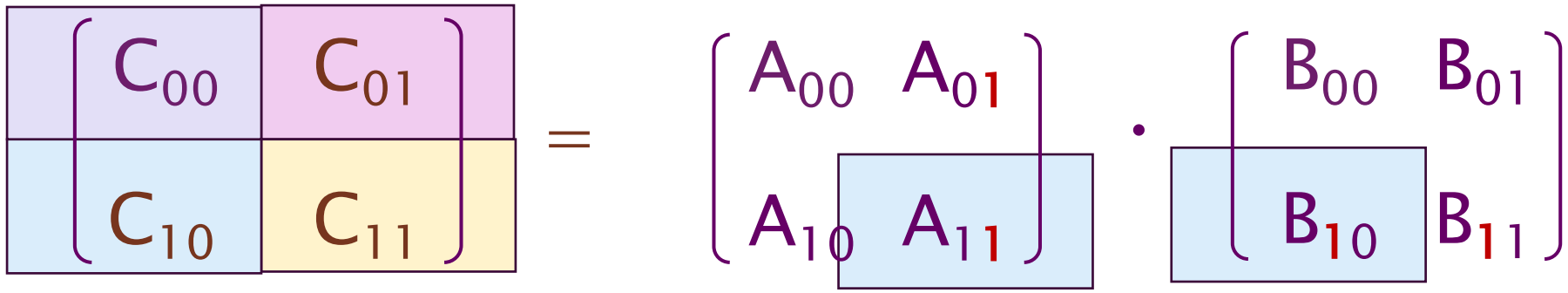
Both of these loops can theoretically be parallelized



Parallel Loops

```
parallel_for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    parallel_for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Both of these loops can theoretically be parallelized



Parallel Loops

```
parallel_for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    parallel_for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Both of these loops can theoretically be parallelized

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

Parallel Loops

```
parallel_for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    parallel_for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Both of these loops can theoretically be parallelized

Parallel Loops

```
parallel_for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    parallel_for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Both of these loops can theoretically be parallelized

Some packages allow you to parallelize **both** simultaneously easily; others do not

Parallel Loops

```
parallel_for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    parallel_for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

OpenMP does not allow you to parallelize both simultaneously

Parallel Loops

Which is better?

```
#pragma omp parallel for
for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

```
for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    #pragma omp parallel for
    for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Parallel Loops

Explore in your homework!

```
#pragma omp parallel for
for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

```
for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    #pragma omp parallel for
    for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Parallel Loops

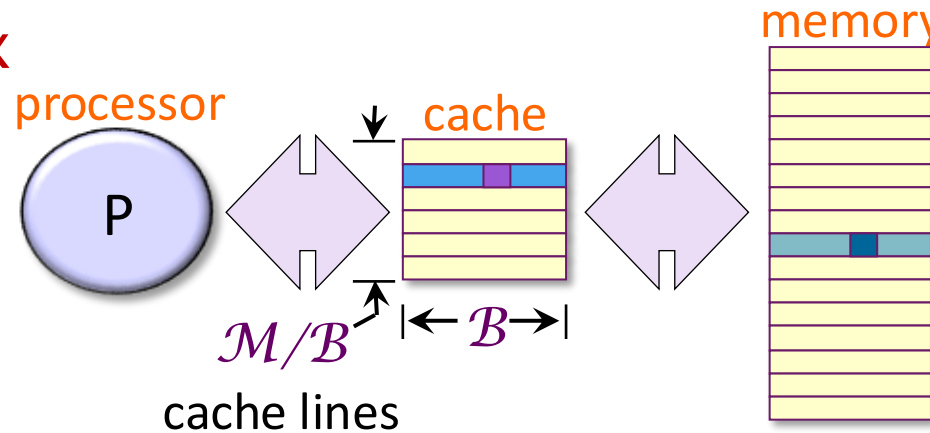
Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408

Using parallel loops gets us almost 18× speedup on 18 cores! (Disclaimer: Not all code is so easy to parallelize effectively.)

Why are we still getting less than 5% of peak?

Hardware Caches, Revisited

- IDEA: Restructure the computation to reuse data in the cache as much as possible
- Cache misses are slow, and cache hits are fast
- Try to make the most of the cache by reusing the data that's already there
- Cache Capacity
  - One Row of a matrix = 4096*8bytes = 32KB
  - L1D cache = 32KB → 1 row of **one** matrix
  - Can't fit all rows from all three matrices!



Slide courtesy of MIT 6.106

D&C Matrix Multiplication

For matrix multiplication, a recursive, parallel, divide-and-conquer algorithm uses caches almost optimally.

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

IDEA: Divide the matrices into $(n/2)\times(n/2)$ submatrices.

Slide courtesy of MIT 6.106

IDEA: For matrix multiplication, a recursive, parallel, divide-and-conquer algorithm uses caches almost optimally.

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{pmatrix} + \begin{pmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{pmatrix}$$

Slide courtesy of MIT 6.106

IDEA: For matrix multiplication, a recursive, parallel, divide-and-conquer algorithm uses caches almost optimally.

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{pmatrix} + \begin{pmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{pmatrix}$$

Slide courtesy of MIT 6.106

IDEA: For matrix multiplication, a recursive, parallel, divide-and-conquer algorithm uses caches almost optimally.

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{pmatrix} + \begin{pmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{pmatrix}$$

Slide courtesy of MIT 6.106

IDEA: For matrix multiplication, a recursive, parallel, divide-and-conquer algorithm uses caches almost optimally.

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{pmatrix} + \begin{pmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{pmatrix}$$

Slide courtesy of MIT 6.106

IDEA: For matrix multiplication, a recursive, parallel, divide-and-conquer algorithm uses caches almost optimally.

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{pmatrix} + \begin{pmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{pmatrix}$$

Slide courtesy of MIT 6.106

IDEA: For matrix multiplication, a recursive, parallel, divide-and-conquer algorithm uses caches almost optimally.

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{pmatrix} + \begin{pmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{pmatrix}$$

Slide courtesy of MIT 6.106

IDEA: For matrix multiplication, a recursive, parallel, divide-and-conquer algorithm uses caches almost optimally.

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{pmatrix} + \begin{pmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{pmatrix}$$

Slide courtesy of MIT 6.106

IDEA: For matrix multiplication, a recursive, parallel, divide-and-conquer algorithm uses caches almost optimally.

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{pmatrix} + \begin{pmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{pmatrix}$$

Slide courtesy of MIT 6.106

IDEA: For matrix multiplication, a recursive, parallel, divide-and-conquer algorithm uses caches almost optimally.

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

Each recursive call can be done in parallel!

$$= \begin{pmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{pmatrix} + \begin{pmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{pmatrix}$$

1. Compute $C_{00} \mathrel{+}= A_{00}B_{00}$; $C_{01} \mathrel{+}= A_{00}B_{01}$; $C_{10} \mathrel{+}= A_{10}B_{00}$; and $C_{11} \mathrel{+}= A_{10}B_{01}$ recursively in parallel.
2. Compute $C_{00} \mathrel{+}= A_{01}B_{10}$; $C_{01} \mathrel{+}= A_{01}B_{11}$; $C_{10} \mathrel{+}= A_{11}B_{10}$; and $C_{11} \mathrel{+}= A_{11}B_{11}$ recursively in parallel.

Slide courtesy of MIT 6.106

Parallel Divide-and-Conquer

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	Parallel divide-and-conquer	1.30	2.35	16,197	105.722	12.646

Implementation	Cache references × 10^6	L1-d cache misses × 10^6	Last-level cache misses × 10^6
Parallel loops	104,090	17,220	8,600
Parallel divide-and-conquer	58,230	9,407	64

16185.9x improvement

Slide courtesy of MIT 6.106

Modeling Parallel Algorithms

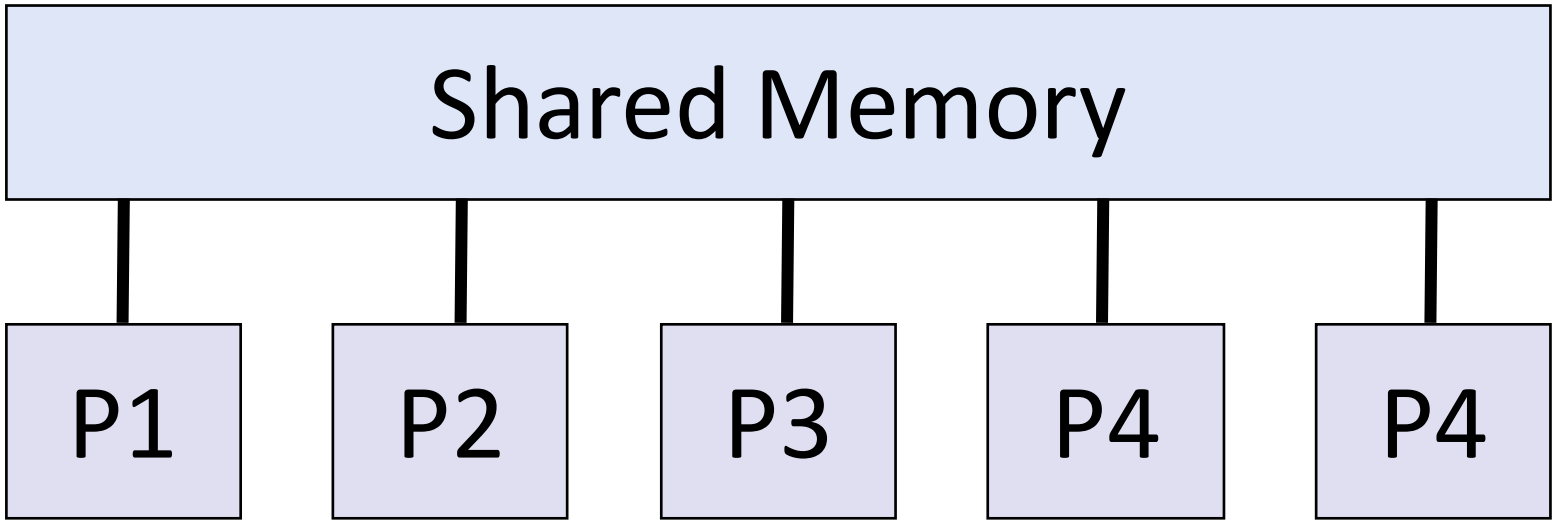
How do we prove formal guarantees about the parallel algorithms we write?

Parallel Algorithm Models

- Analyzing performance is the most important part of studying algorithms
- Cannot predict the exact running time of an algorithm
- But used to determine **how the running time grows** as a function of input size
- Hence, we need a **formal model for parallelism**
- Older models include processor-based models
- Processors connected to a common shared memory
- Performance calculated in terms of number of **instruction cycles**

Parallel Random Access Machine (PRAM)

- Assumed that all the processors can access a memory location in the shared memory simultaneously in unit time



Caveat: **unrealistic**!

Impossible for **every processor** to access a shared memory in **unit time**

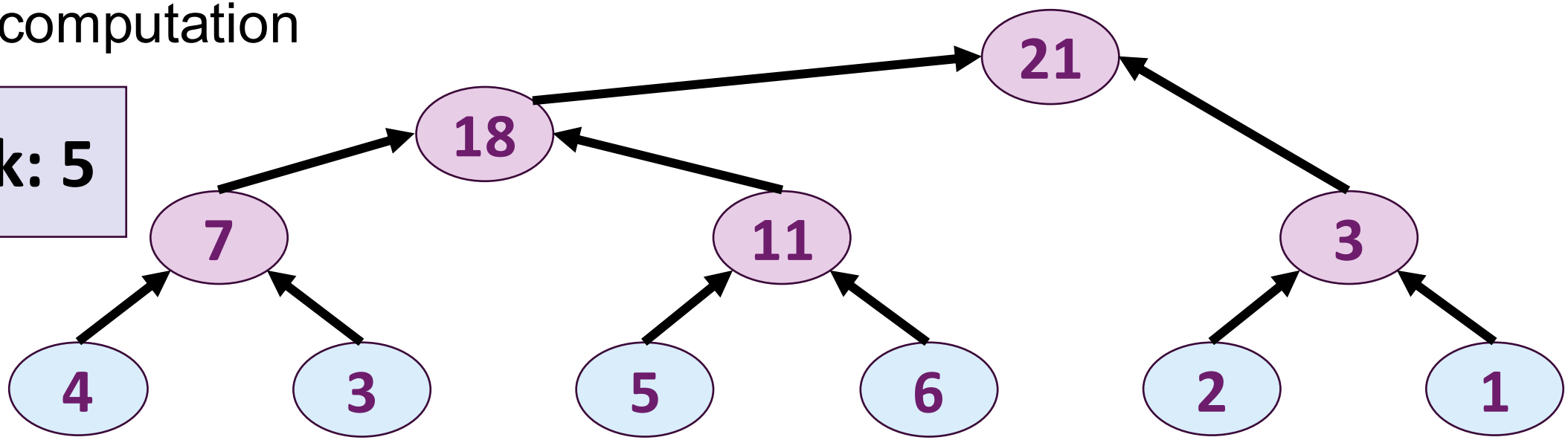
Virtual Models

- A performance model that does not attempt to **physically represent any machine**
- Higher-level model that can be **mapped onto** real machines
- PRAM is a virtual machine mapped onto more realistic physical machines
- Simulate multiple processors of PRAM on a single processor of a host machine
- Advantage of virtual models:
  - Transferred to different architectures
  - Easier to program

Work-Span Model

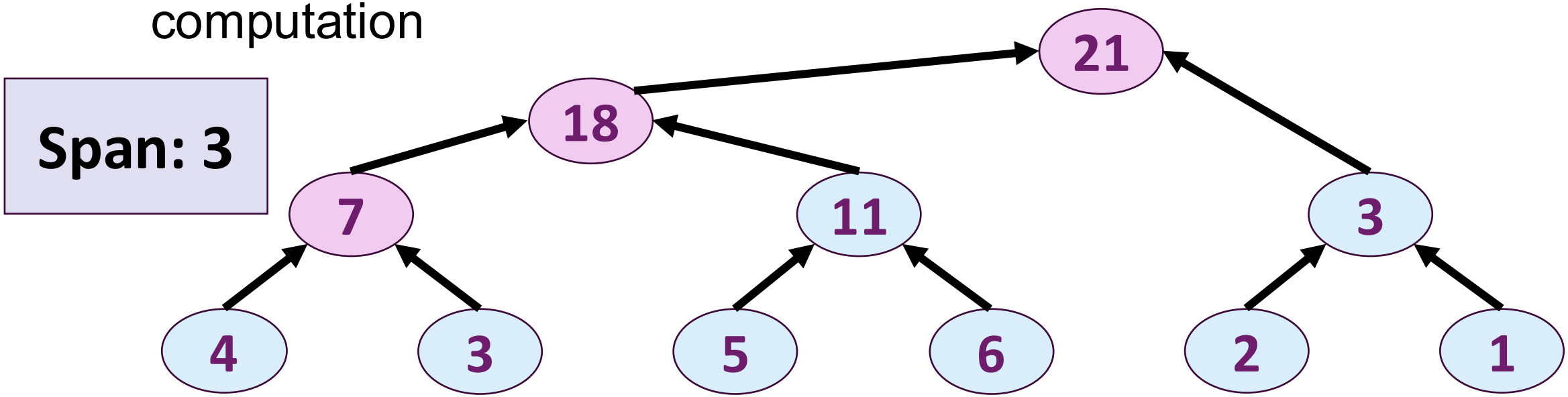
- **Work-span model** used to define measurements of performance used to find running time on a particular machine
- **Work**: total number of operations executed by a computation
- **Span**: longest chain of sequential dependencies in the computation

Work: 5



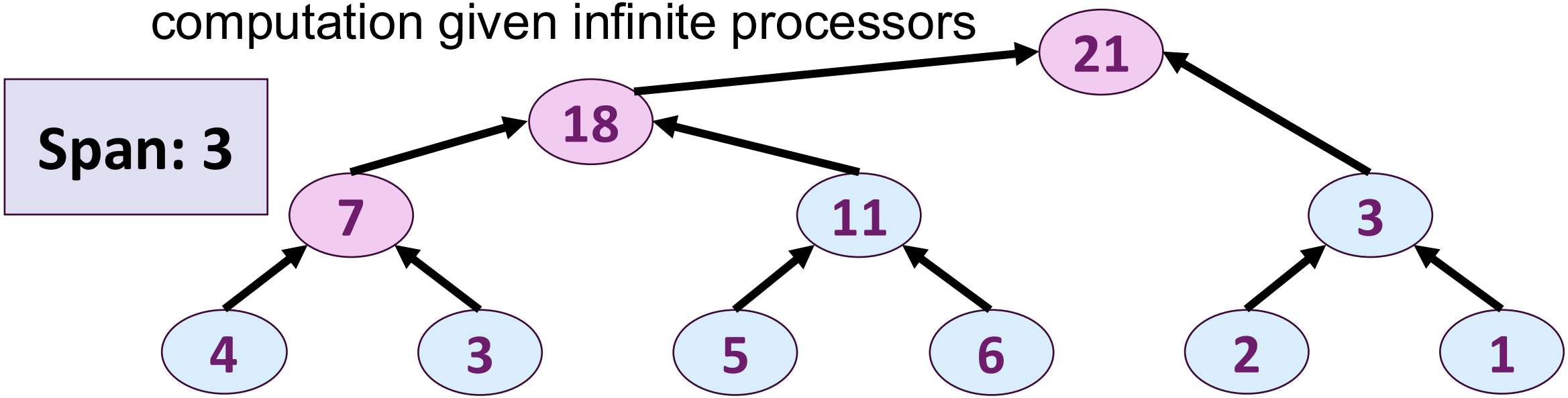
Work-Span Model

- **Work-span model** used to define measurements of performance used to find running time on a particular machine
- **Work**: total number of operations executed by a computation
- **Span**: longest chain of sequential dependencies in the computation



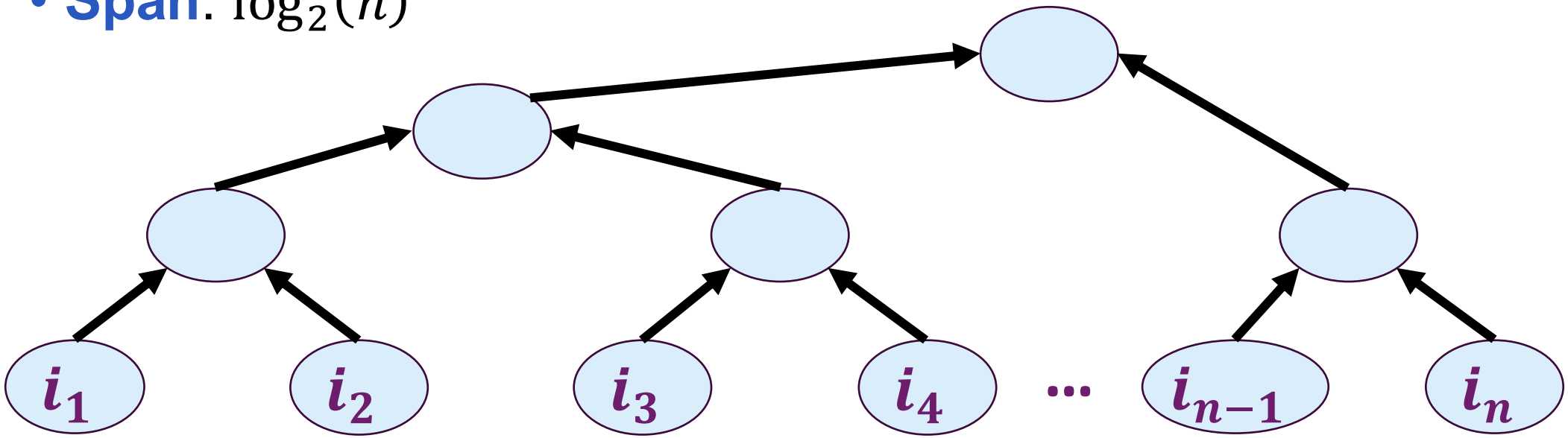
Work-Span Model

- **Work-span model** used to define measurements of performance used to find running time on a particular machine
- **Work**: total number of operations executed by a computation
- **Span**: informally—how much time it takes to perform computation given infinite processors



Work-Span Model

- Given n numbers that is summed using a balanced binary tree, what is the required work and span?
  - **Work**: $n - 1$ because $n - 1$ additions are necessary to obtain sum of n numbers
  - **Span**: $\log_2(n)$



Why Work and Span?

- Work-span model is **very simple**, performance analysis is **closely related to the code**, and code captures **parallelism**
- Example analysis with Quicksort (sequential code)

Why Work and Span?

```
Algorithm QUICKSORT(A, p, r)
 1. if p < r then
 2.      q ← PARTITION(A, p, r)
 3.      QUICKSORT(A, p, q – 1)
 4.      QUICKSORT(A, q + 1, r)

Algorithm PARTITION(A, p, r)
 1. x ← A[r]                    ▷ x is the
pivot
 2. i ← p – 1
 3. for j ← p to r – 1 do
 4.      if A[j] ≤ x then
 5.           i ← i + 1
 6.           exchange A[i] ↔ A[j]
 7. exchange A[i + 1] ↔ A[r]
 8. return i + 1
```

6 5 3 1 8 7 2 4

Average case runtime: $O(n \log n)$

Expected span of recursive calls: $O(\log n)$

Why Work and Span?

```
function parallelQuickSort(A):
        if A.size() <= 1: return A

        pivot = A[random() % A.size()]
        S1, S2, S3
        parallel_for x in A:
                if x < pivot:
                        S1.push_back(x)
                else if x == pivot:
                        S2.push_back(x)
                else: S3.push_back(x)

        future L_future = async(parallelQuickSort, S1)
        future R_future = async(parallelQuickSort, S3)

        L = L_future.get()
        R = R_future.get()

        L.insert(L.end(), S2.begin(), S2.end())
        L.insert(L.end(), R.begin(), R.end())
        return L
```

In parallel: find all elements less than pivot, equal to pivot, and smaller than pivot

Sort the remaining arrays in parallel

What is the span and work in this case?

Why Work and Span?

```
function parallelQuickSort(A):
        if A.size() <= 1: return A

        pivot = A[random() % A.size()]
        S1, S2, S3
        parallel_for x in A:
                if x < pivot:
                        S1.push_back(x)
                else if x == pivot:
                        S2.push_back(x)
                else: S3.push_back(x)

        future L_future = async(parallelQuickSort, S1)
        future R_future = async(parallelQuickSort, S3)

        L = L_future.get()
        R = R_future.get()

        L.insert(L.end(), S2.begin(), S2.end())
        L.insert(L.end(), R.begin(), R.end())
        return L
```

In parallel: find all elements less than pivot, equal to pivot, and smaller than pivot

Sort the remaining arrays in parallel

What is the span and work in this case?

Same as before!

$O(n \log n)$ expected work

$O(\log n)$ expected span

Why Work and Span?

```
function parallelQuickSort(A):
        if A.size() <= 1: return A

        pivot = A[random() % A.size()]
        S1, S2, S3
        parallel_for x in A:
                if x < pivot:
                        S1.push_back(x)
                else if x == pivot:
                        S2.push_back(x)
                else: S3.push_back(x)

        future L_future = async(parallelQuickSort, S1)
        future R_future = async(parallelQuickSort, S3)

        L = L_future.get()
        R = R_future.get()

        L.insert(L.end(), S2.begin(), S2.end())
        L.insert(L.end(), R.begin(), R.end())
        return L
```

That was so easy to analyze!

What is the span and work in this case?

Same as before!

$O(n \log n)$ expected work

$O(\log n)$ expected span

Why Work and Span?

```
function parallelQuickSort(A):
        if A.size() <= 1: return A

        pivot = A[random() % A.size()]
        S1, S2, S3
        parallel_for x in A:
                if x < pivot:
                        S1.push_back(x)
                else if x == pivot:
                        S2.push_back(x)
                else: S3.push_back(x)

        future L_future = async(parallelQuickSort, S1)
        future R_future = async(parallelQuickSort, S3)

        L = L_future.get()
        R = R_future.get()

        L.insert(L.end(), S2.begin(), S2.end())
        L.insert(L.end(), R.begin(), R.end())
        return L
```

Now suppose we are analyzing in the PRAM model with P processors…

Lots of questions: Which processors get which of the arrays? How do we partition the recursive calls across the processors? How do we synchronize the processors across the subcalls?

Why Work and Span?

```
function parallelQuickSort(A):
        if A.size() <= 1: return A

        pivot = A[random() % A.size()]
        S1, S2, S3
        parallel_for x in A:
                if x < pivot:
                        S1.push_back(x)
                else if x == pivot:
                        S2.push_back(x)
                else: S3.push_back(x)

        future L_future = async(parallelQuickSort, S1)
        future R_future = async(parallelQuickSort, S3)

        L = L_future.get()
        R = R_future.get()

        L.insert(L.end(), S2.begin(), S2.end())
        L.insert(L.end(), R.begin(), R.end())
        return L
```

This complication is unnecessary when we think about parallel algorithms

Lots of questions: Which processors get which of the arrays? How do we partition the recursive calls across the processors? How do we synchronize the processors across the subcalls?

Relationship of Work and Span to Runtime

- **Work** can be considered to be runtime when given **one processor**
- **Span** can be considered to be the runtime when given **infinite processors**
- In almost all real-life settings, we have a **fixed number of P processors**
- Run in time (W work, S span, T runtime):

$$\frac{W}{P} \leq T < \frac{W}{P} + S$$

Simple Parallel Primitives

Standard techniques we can use in our parallel algorithms

Parallel Primitives

- Parallel primitives are **building blocks** for writing your parallel algorithms
- When formulating more complicated algorithms, we can often use **subroutines** that can be found in a parallel package
- These methods are well-known and implemented in most parallel packages
- Hence, they are called **primitives**

Parallel Scan

- A **scan** or **prefix sum** operation takes a sequence A, an associative operator $\oplus$, and an identity element $\perp$ and computes the sequence

$$[\perp, \perp \oplus A[0], \perp \oplus A[0] \oplus A[1], \ldots, \perp \oplus A[0] \oplus \cdots \oplus A[n-2]]$$

- And also the overall expression: $\perp \oplus A[0] \oplus \cdots \oplus A[n-1]$
- Useful in a variety of algorithms
- **PlusScan**:$\oplus = +$ and $\perp = 0$

Parallel PlusScan Example

- Input: [1, 2, 4, 6, 8, 9]

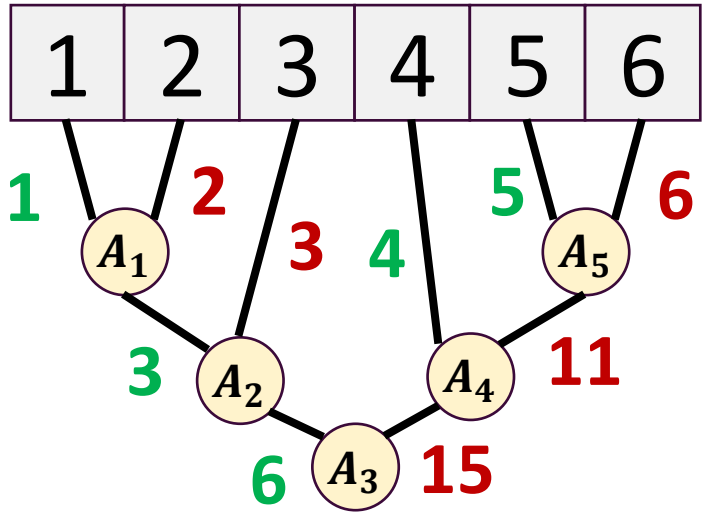
- PlusScan Output: [0, 1, 3, 7, 13, 21] and 30

Parallel Scan Algorithm

- Two operations scanUp and scanDown
- Example: Input $A = [1, 2, 3, 4, 5, 6]$

Parallel Scan Algorithm

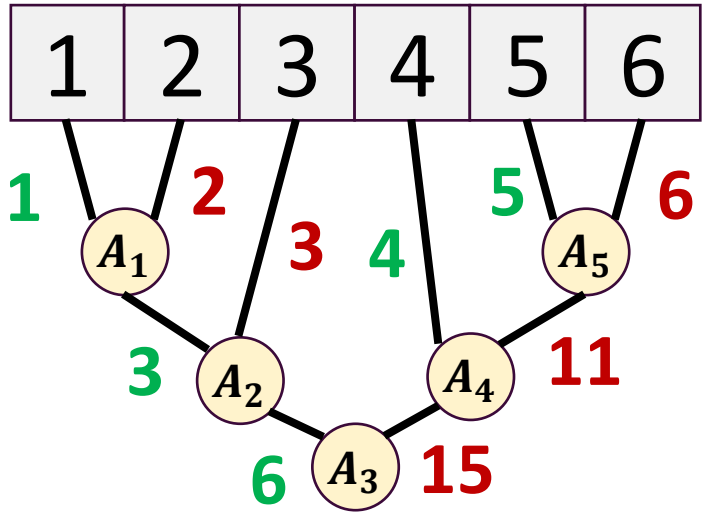
- Two operations scanUp and scanDown
- Example: Input $A = [1, 2, 3, 4, 5, 6]$



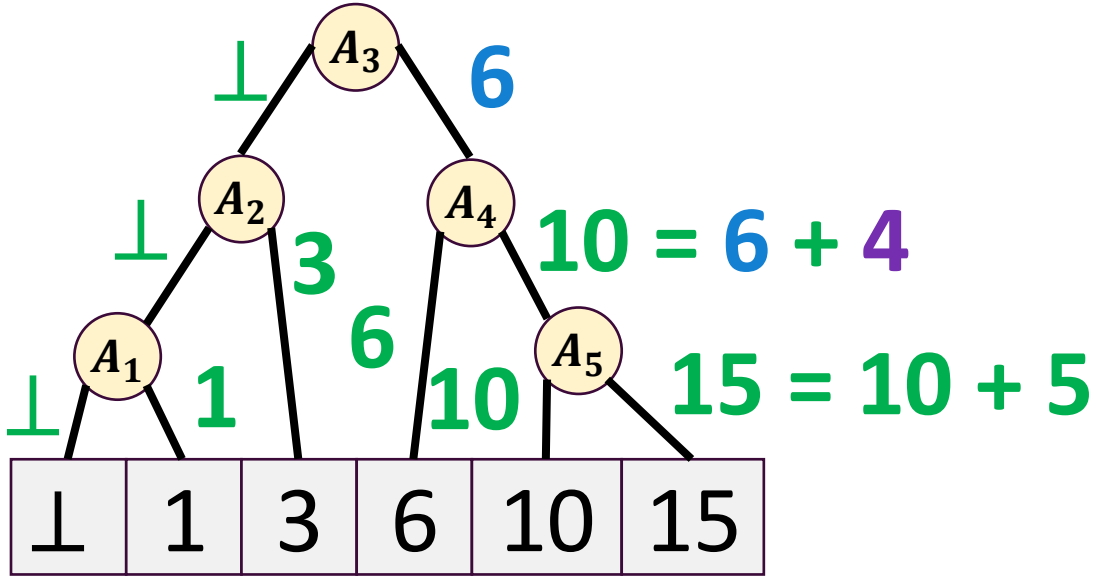
Sums = | 1 | 3 | 6 | 4 | 5 | 31

Parallel Scan Algorithm

- Two operations scanUp and scanDown
- Example: Input $A = [1, 2, 3, 4, 5, 6]$

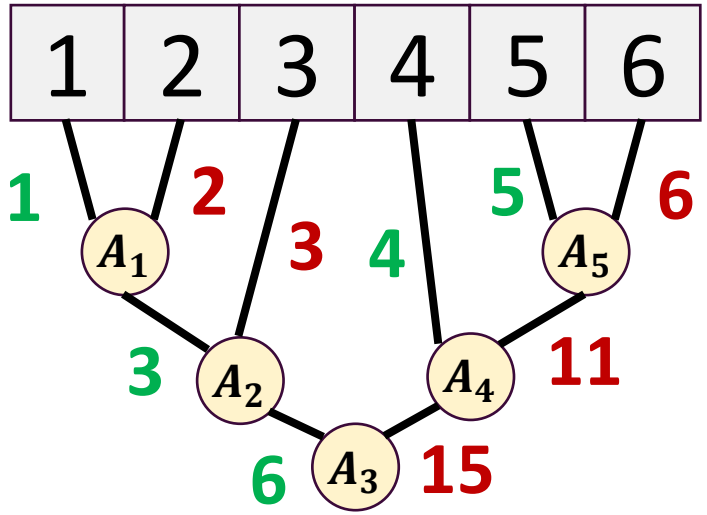


Sums = | 1 | 3 | 6 | 4 | 5 | 31

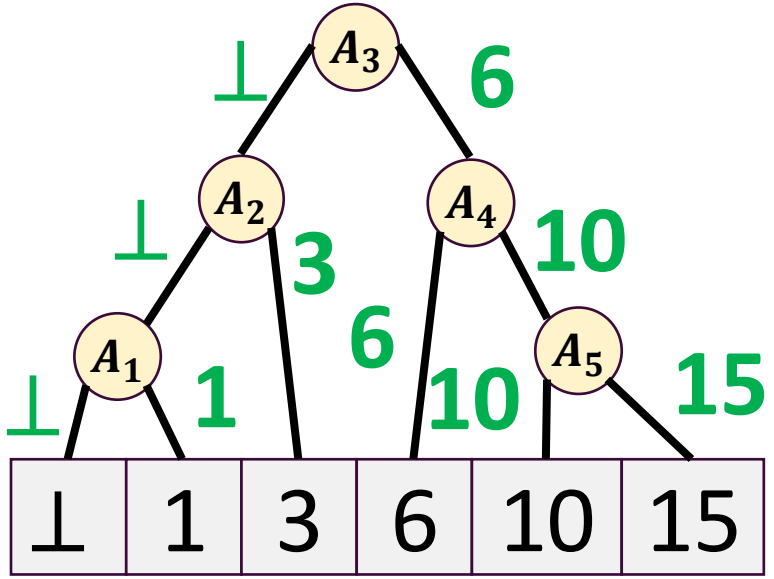


Parallel Scan Algorithm

- Two operations scanUp and scanDown
- Example: Input $A = [1, 2, 3, 4, 5, 6]$



Sums = | 1 | 3 | 6 | 4 | 5 | 31



Parallel Scan Algorithm

- What is the work and span of this algorithm? Assume array with n elements.
- Work: $W(n) = 2W\left(\frac{n}{2}\right) + O(1)$
  - Master Theorem: $O(n)$
- Span: $D(n) = D\left(\frac{n}{2}\right) + O(1)$
  - $O(\log n)$

Parallel Filter

- Takes a sequence A and a predicate p and returns an array containing $a \in A$ s.t. $p(a) = True$
- Return all elements in the same order as A
- What is a parallel algorithm we can write to solve this problem?
- Hint: how can we use PlusScan?
- Example: $A = [1, 2, 3, 4, 5, 6]$
- Predicate: divisible by 2

Parallel Filter

- Example: $A = [1, 2, 3, 4, 5, 6]$
- Predicate: divisible by 2

- Algorithm: in parallel, construct $F = [0, 1, 0, 1, 0, 1]$
  - 1 if divisible by 2
  - 0 otherwise

- PlusScan on F to obtain $\mathrm{I} = [0, 0, 1, 1, 2, 2]$ and 3

Parallel Filter

- Example: $A = [1, 2, 3, 4, 5, 6]$
- Predicate: divisible by 2

- Algorithm: in parallel, construct $F = [0, 1, 0, 1, 0, 1]$
  - 1 if divisible by 2
  - 0 otherwise

- PlusScan on F to obtain $\mathrm{I} = [0, 0, 1, 1, 2, 2]$ and 3
- Construct $C = [2, 4, 6]$ where we check in parallel whether $I[i] = I[i-1]$

Parallel Filter Algorithm

- What is the work and span of this algorithm? Assume array with n elements.
- Work: $O(n)$
- Span: $O(\log n)$

- Work and span dominated by PlusScan