CPSC 4240/5240: Parallel Programming Techniques

Lecture 4: Parallel Primitives Continued

Lecturer: Quanquan C. Liu

Spring 2026

Notes about the Homework

- You should compile using the testing script that is provided to you
- If it does not compile, your program will not compile on Gradescope
- g++ is used with the flag `-std=c++17`
- If you use a different version of C++ to compile (i.e. c++11 or c++14), it will not compile on Gradescope
  - Newer versions may be fine but switch to c++17 to be safe
  - I tested with `-std=c++20`

Notes about the Homework

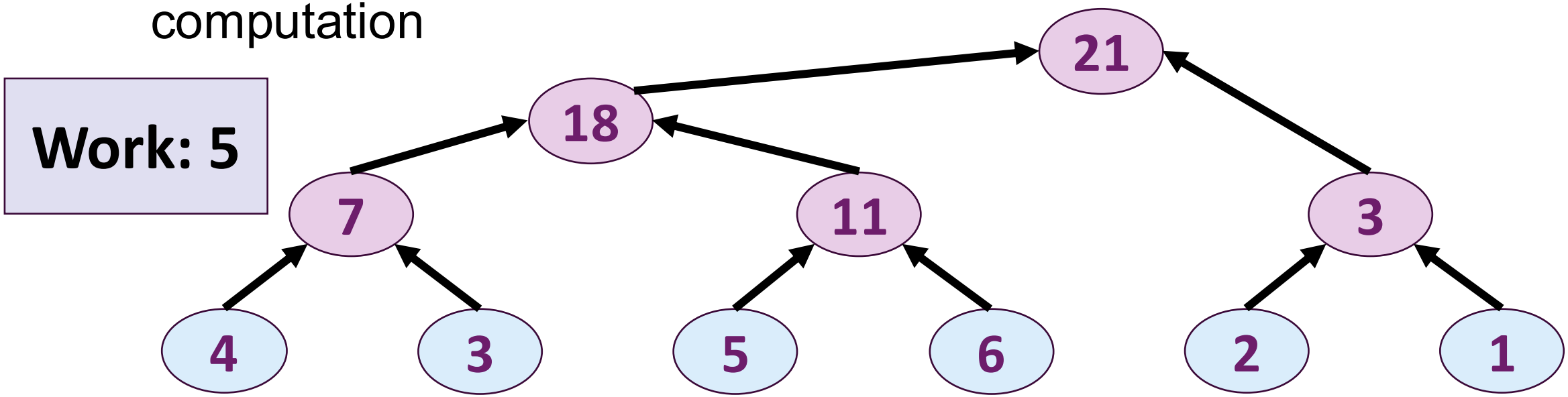
- You should see a difference in running time between the two parallel_for loop placements
- If you do not see a difference in running time then you are likely not using parallelism
- To check whether you are using parallelism
  - Run the testing script
  - In a separate commandline window, run htop
  - htop shows which CPUs are running the computation
  - If you are correctly parallelizing then see workload on more than one core
    - On the zoo, it'll be workload > 10% on more than one core

Last class on parallel primitives

We discussed Parallel Scan

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



Simple Parallel Primitives

Standard techniques we can use in our parallel algorithms

Parallel Scan

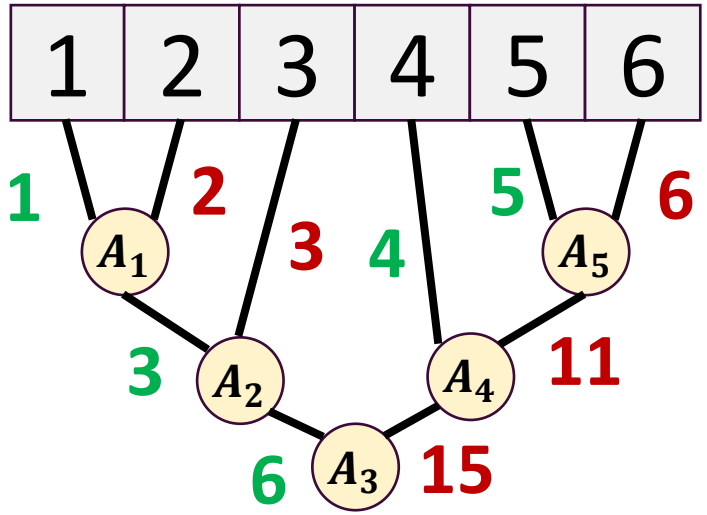
- A **scan** or **prefix sum** operation takes a sequence A, an associative operator $\oplus$, and an identity element $\perp$ and computes the sequence

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

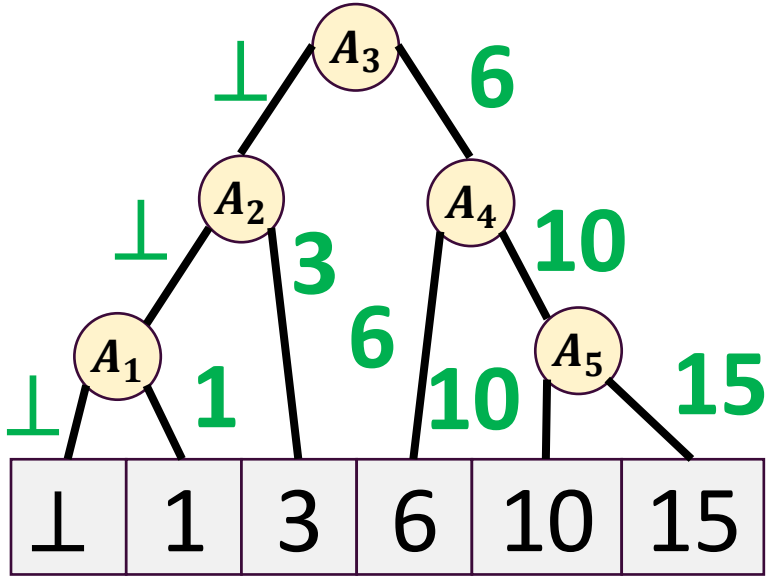
- And also the overall expression: $\perp \oplus A[0] \oplus \cdots \oplus A[n-1]$
- Useful in a variety of algorithms
- **PlusScan**:$\oplus = +$ and $\perp = 0$

Parallel 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

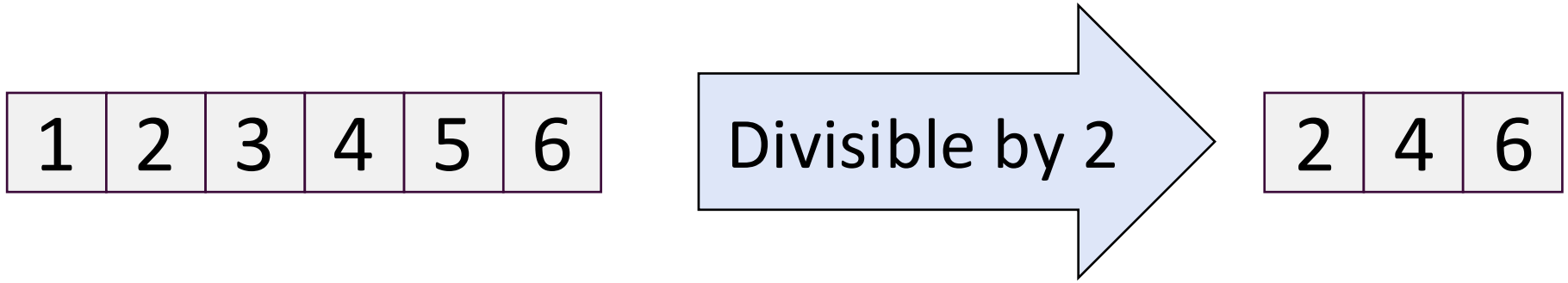
How to filter an array in parallel…

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

Parallel Filter

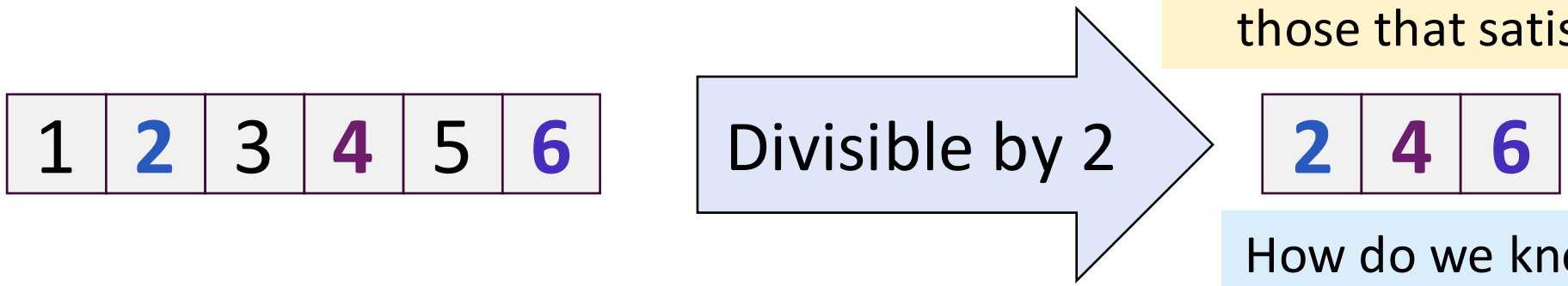
- Takes a sequence A and a predicate p and returns an array containing $a \in A$ s.t. $p(a) = True$
- Return all elements in the same order as A
- What is a parallel algorithm we can write to solve this problem?



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?
- What is the main problem?

Need to create a new array with the same **size and elements** as those that satisfy predicate



How do we know where to insert the elements?

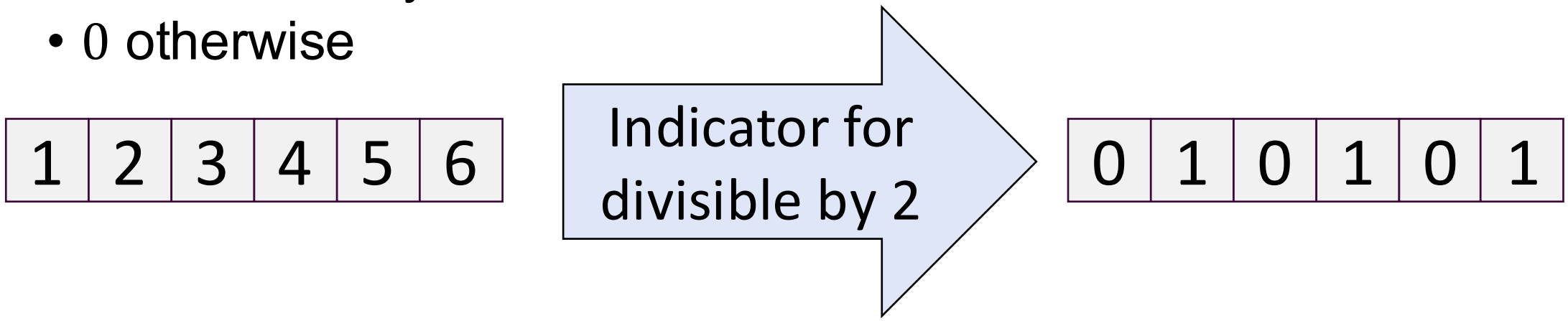
Parallel Filter

- 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?

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



Parallel Filter

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

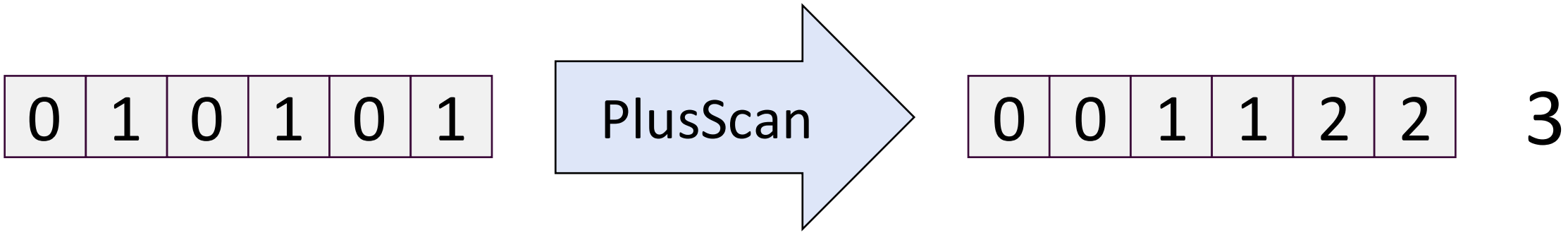
- Algorithm: in parallel, construct $F = [0, 1, 0, 1, 0, 1]$
  - 1 if divisible by 2
  - 0 otherwise

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

Parallel Filter

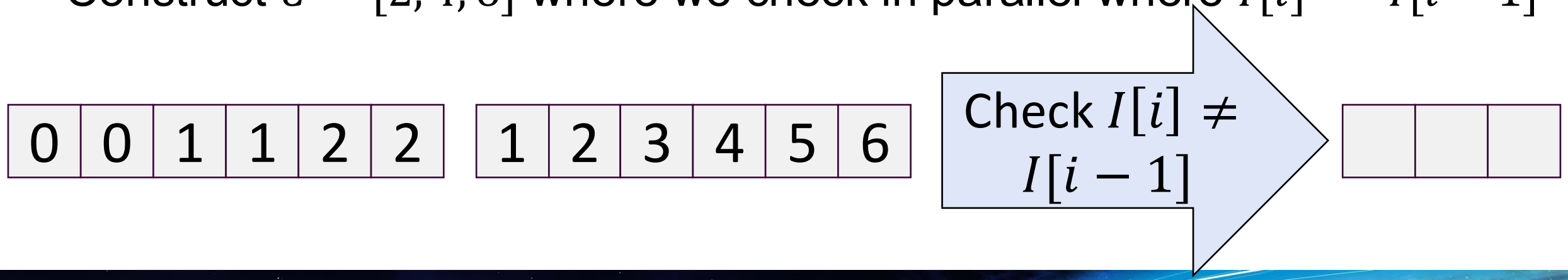
- Algorithm: in parallel, construct $F = [0, 1, 0, 1, 0, 1]$
  - 1 if divisible by 2
  - 0 otherwise

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



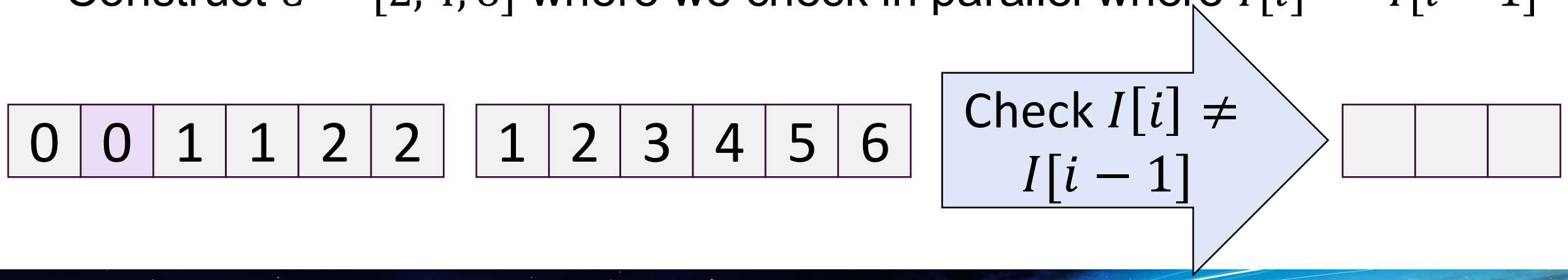
Parallel Filter

- 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 an array C of size 3
- Construct $C = [2, 4, 6]$ where we check in parallel where $I[i] = I[i-1]$



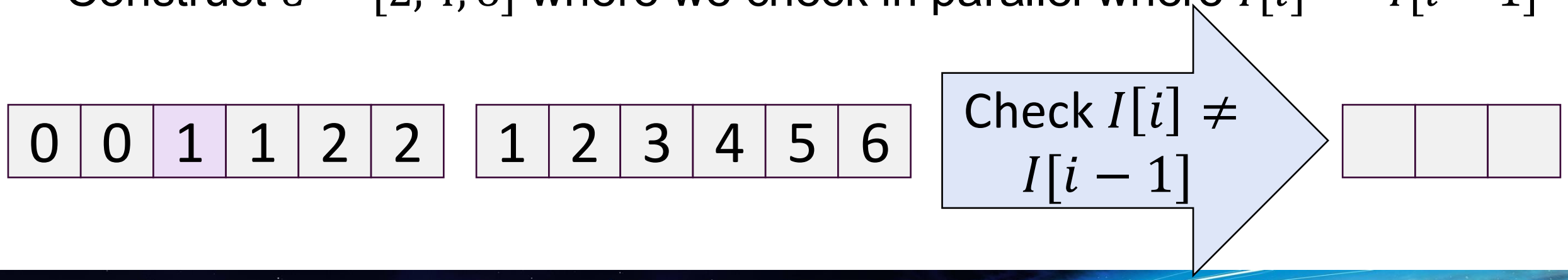
Parallel Filter

- 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 an array C of size 3
- Construct $C = [2, 4, 6]$ where we check in parallel where $I[i] = I[i-1]$



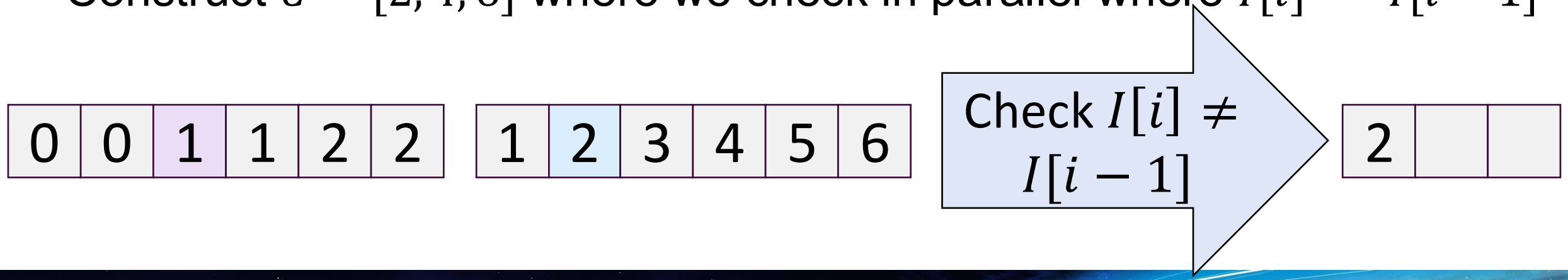
Parallel Filter

- 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 an array C of size 3
- Construct $C = [2, 4, 6]$ where we check in parallel where $I[i] = I[i-1]$



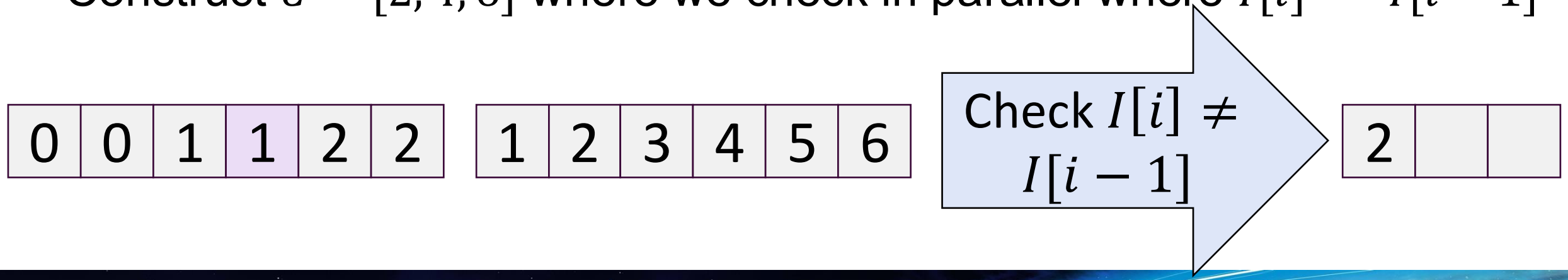
Parallel Filter

- 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 an array C of size 3
- Construct $C = [2, 4, 6]$ where we check in parallel where $I[i] = I[i-1]$



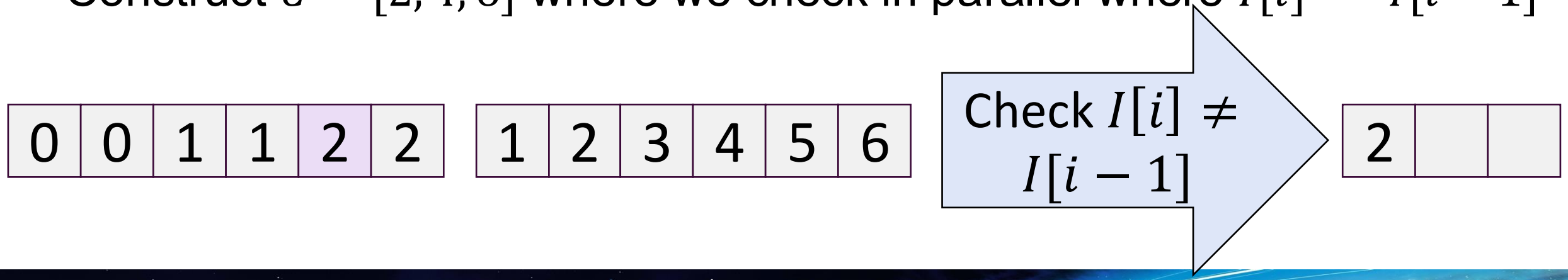
Parallel Filter

- 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 an array C of size 3
- Construct $C = [2, 4, 6]$ where we check in parallel where $I[i] = I[i-1]$



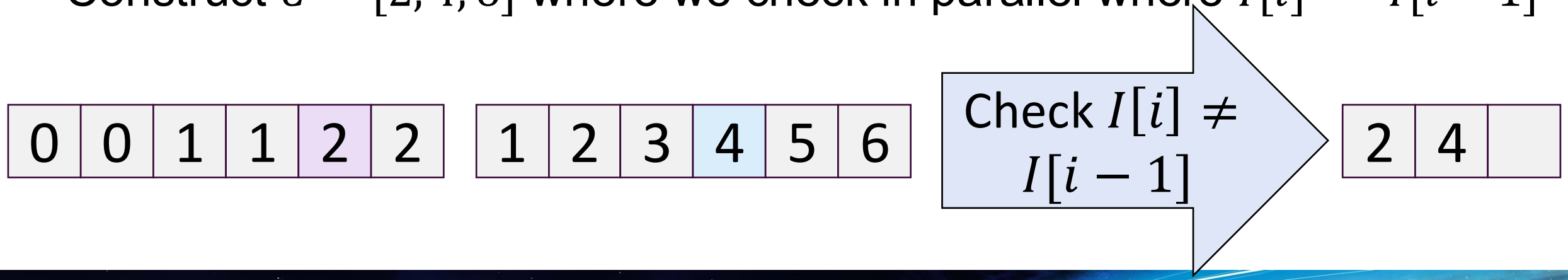
Parallel Filter

- 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 an array C of size 3
- Construct $C = [2, 4, 6]$ where we check in parallel where $I[i] = I[i-1]$



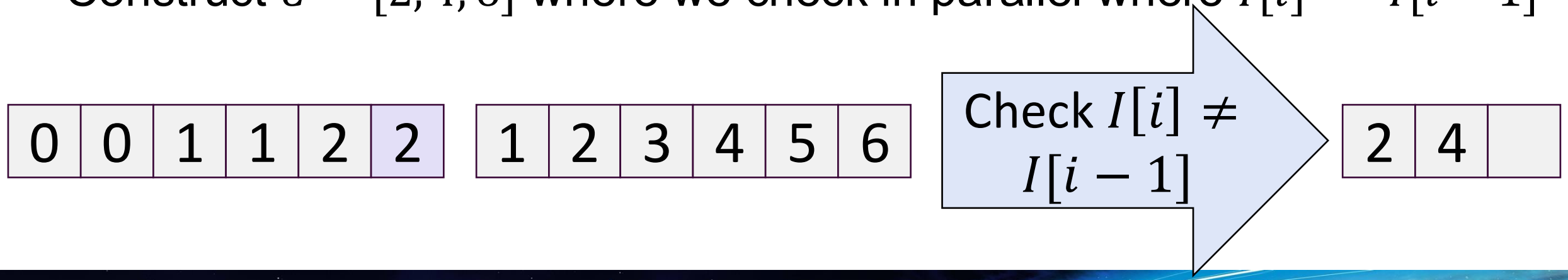
Parallel Filter

- 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 an array C of size 3
- Construct $C = [2, 4, 6]$ where we check in parallel where $I[i] = I[i-1]$



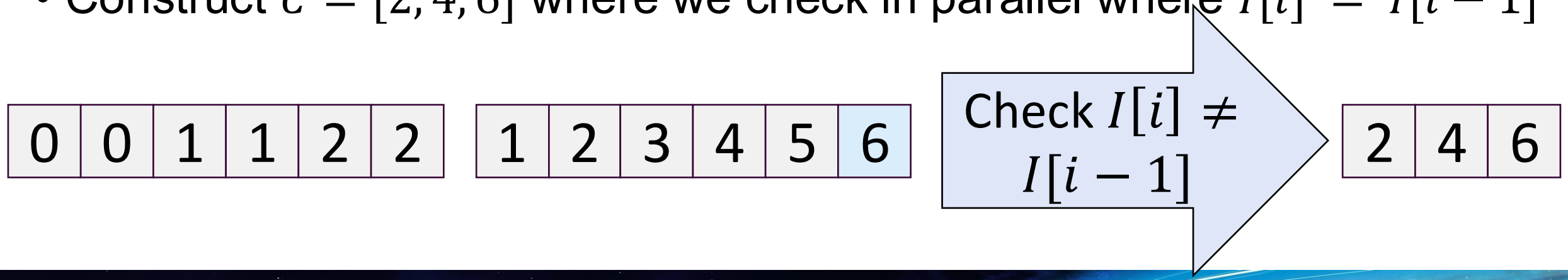
Parallel Filter

- 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 an array C of size 3
- Construct $C = [2, 4, 6]$ where we check in parallel where $I[i] = I[i-1]$



Parallel Filter

- 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 an array C of size 3
- Construct $C = [2, 4, 6]$ where we check in parallel where $I[i] = I[i-1]$



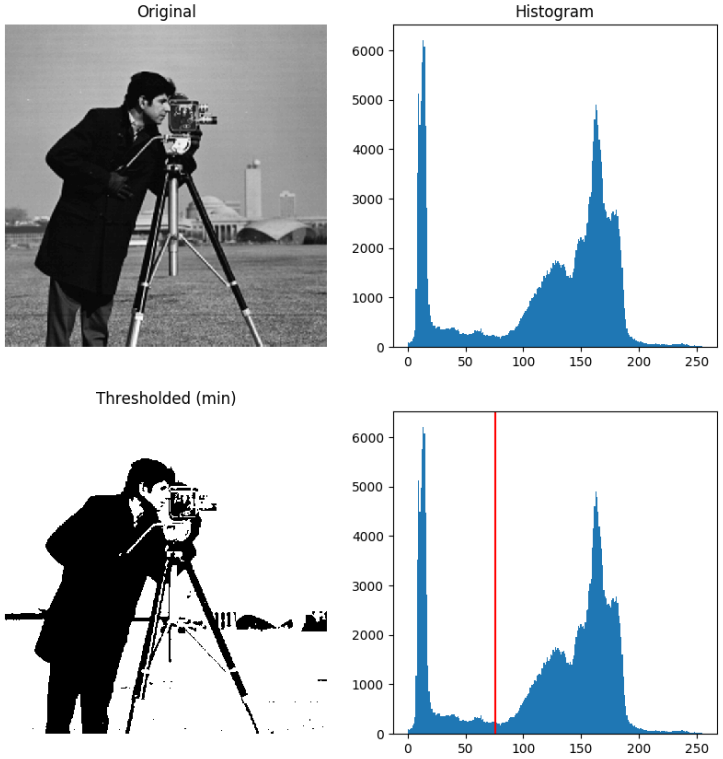
Parallel Filter Algorithm

- 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

Why do we care about Parallel Filter?

- Image processing; identify specific features—e.g. in a medical scan



https://3dprint.com/253011/what-is-metrology-part-15-inverse-filtering/

Why do we care about Parallel Filter?

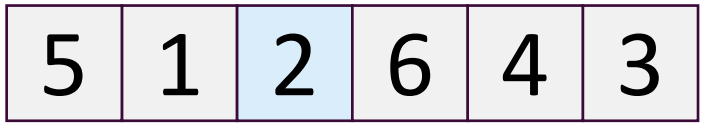
- High-frequency trading algorithms need to analyze massive streams of market data
- Parallel filters used to quickly identify relevant data points
  - Price spikes
  - Unusual trading activity
- Network security systems analyzing network traffic to detect malicious activity
  - Quickly identify suspicious patterns

Parallel Sorting

Sorting is a fundamental algorithm that can be made faster with parallelism!

Parallel Quicksort

- Parallel divide-and-conquer algorithm
- Select a pivot uniformly at random in the array



- **Parallel filter** elements smaller than pivot and larger than pivot

- Intermediate data structures present **parallel overhead**
- **Call recursively (in parallel) on the two arrays**

Parallel Quicksort

- Analysis:
  - **Expected work:** $O(n \log n)$
    - Depth of random tree is $O(\log n)$ in expectation
    - $O(n)$ work for each level
  - **Span:** Use recurrence $D(n) = D\left(\frac{n}{2}\right) + O(\log n)$
    - $O(\log n)$ term is due to span of filter
    - $O(\log^2 n)$ in expectation by Master Theorem
    - Can show **with high probability bound**

With high probability

What does it mean?

Public Service Announcement

- What does a **high probability bound** mean?
- Given a problem with input size n and the problem uses randomness
  - Want the solution complexity to hold with probability at least $1 - \frac{1}{n^c}$ for a sufficiently large constant $c \geq 1$
- Example:
  - Flipping n coins, you will see a head within $c \log_2(n)$ with probability at least $1 - \frac{1}{n^c}$
  - **With high probability**, you will see a head within $c \log_2(n)$ flips

Back to parallel sorting

Parallel Quicksort

- Practically speaking, high variance in running time
  - Choice of pivots can result in **skewed workloads**
  - Some processes get more work than others
  - **Data skew:** uneven distribution of data across different processes
- **Consider other parallel sorting algorithms with less skew**

Parallel Samplesort

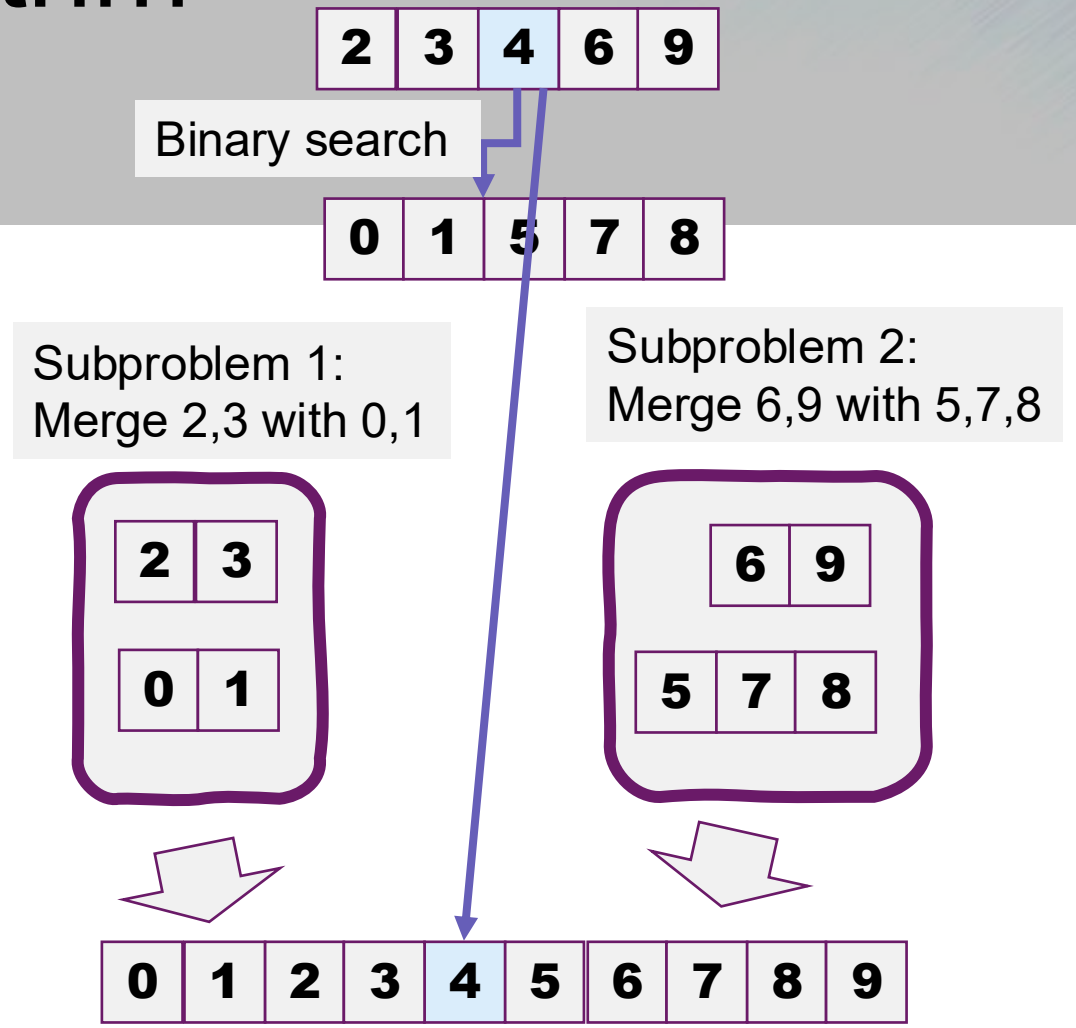
- **Samplesort** samples many pivots at once!
- Sample p pivots called **splitters**
- **Sort the splitters**
- Partition input sequence into buckets based on splitters
- **Parallel filter** the elements into buckets
- Then, put the buckets into processors to sort recursively
- Same work and depth as previously (assuming not too large p)
- But helps with skew practically!

Parallel Mergesort

- Classic divide-and-conquer algorithm in the sequential setting
- Recursively divide array at the midpoint, sort the two parts, and then merge the two sorted arrays
- Proposed algorithm:
  - Divide array in two
  - Recursively sort the two parts, **in parallel**
  - Merge two arrays back into one array
- What is the work and span of this algorithm?
  - $O(n \log n)$ work and $O(n)$ span, not good!

Parallel Merging Algorithm

- We need to parallelize the merge part!
- Here's a parallel merging algorithm:
  - Given two sorted arrays A and B, find the median of A and binary search for its position in B
  - Insert the median into the correct position
  - Recursively call left/right



Slide courtesy of CS260 – Parallel algorithms: Yihan Sun

Parallel Merging Algorithm

- What is the work and span of this algorithm?
  - $O(n)$ work
  - $O\left(\log^2(n)\right)$ span
  - Can achieve $O\left(\log(n)\right)$ span also
  - Last year: work out full proofs in homework

Parallel Mergesort

- Classic divide-and-conquer algorithm in the sequential setting
- Recursively divide array at the midpoint, sort the two parts, and then merge the two sorted arrays
- Proposed algorithm:
  - Divide array in two
  - Recursively sort the two parts, **in parallel**
  - **Parallel merge** two arrays back into one array
- What is the work and span of this algorithm?
  - $O(n \log n)$ work and $O(\log^3(n))$ span (reduced in homework)

Peak into Research: Parallel Semisort

When you only want a partial sort of the elements

Parallel Semisort

- Given an array of keys and values, compute a reordered array where values with identical keys are contiguous
- Distinct keys do not have to be sorted!
- Widely used for massively parallel systems such as MapReduce
  - Relational joins in databases
  - Parallel graph algorithms
  - “A Top-Down Parallel Semisort”: Gu-Shun-Sun-Blelloch SPAA ’15
  - $O(n)$ expected work and $O(\log n)$ depth with high probability

Lecture 4 Quiz