What is a reduction computation?

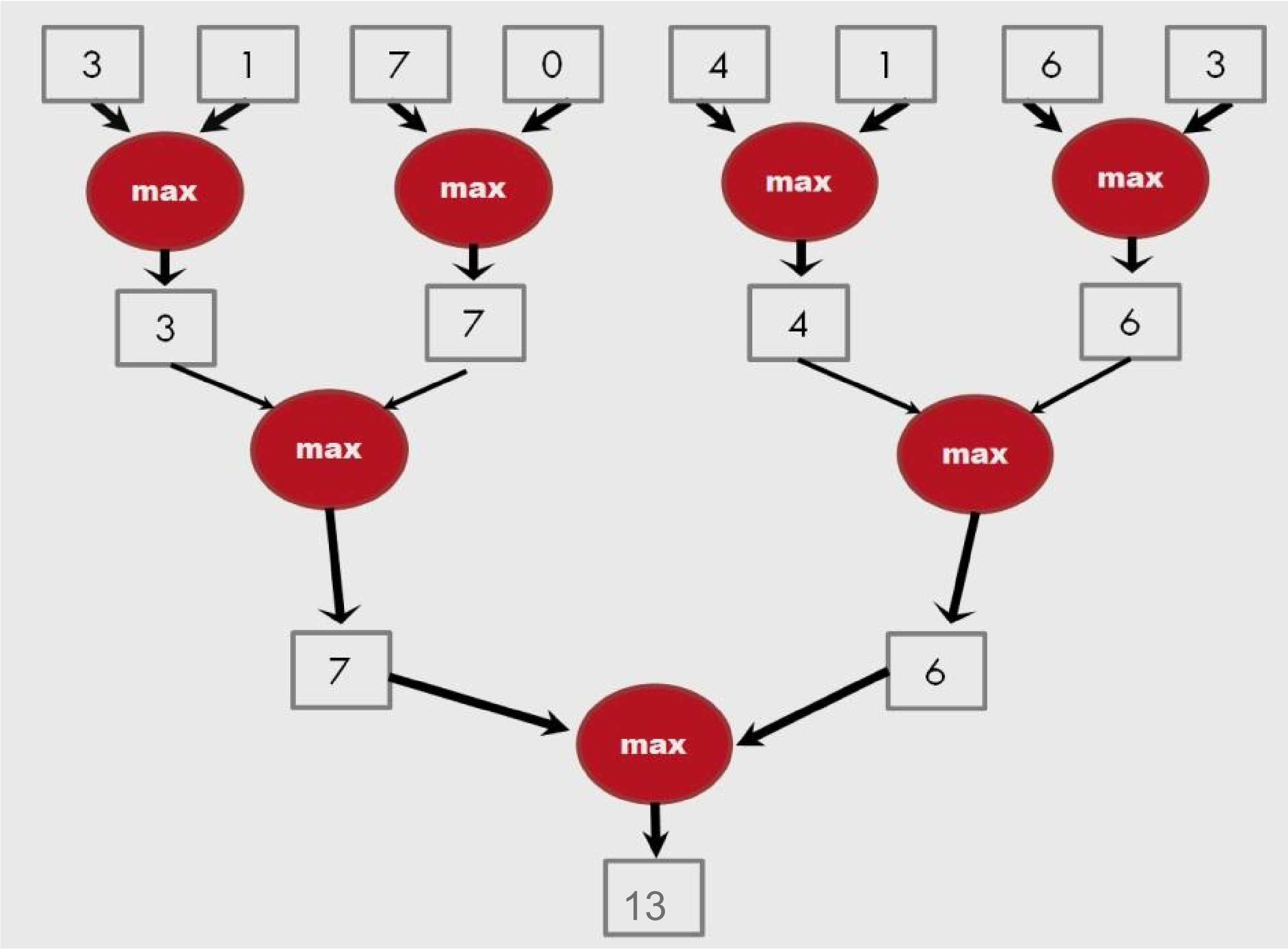
- Summarize a set of input values into one value using a "reduction operation"
  - Max
  - Min
  - Sum
  - Product
- Often used with a user defined reduction operation function as long as the operation
  - Is associative and commutative
  - Has a well-defined identity value (e.g., 0 for sum)
  - For example, the user may supply a custom "max" function for 3D coordinate data sets where the magnitude for the each coordinate data tuple is the distance from the origin.

An example of "collective operation"

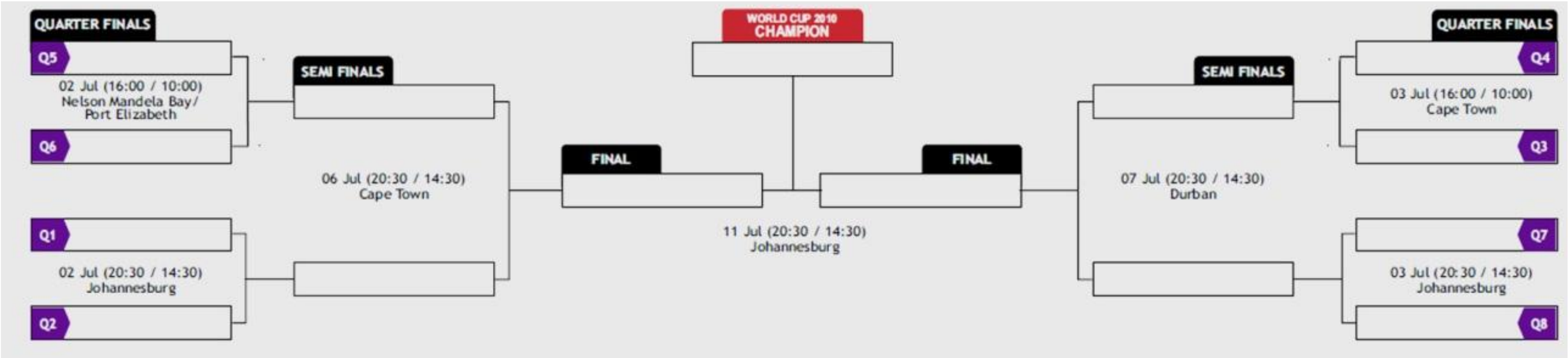
An Efficient Sequential Reduction O(N)

- Initialize the result as an identity value for the reduction operation
  - Smallest possible value for max reduction
  - Largest possible value for min reduction
  - 0 for sum reduction
  - 1 for product reduction
- Iterate through the input and perform the reduction operation between the result value and the current input value
  - N reduction operations performed for N input values
  - Each input value is only visited once – an O(N) algorithm
  - This is a computationally efficient algorithm.

A parallel reduction tree algorithm performs N-1 operations in log(N) steps



A tournament is a reduction tree with "max" operation



A Quick Analysis

- For N input values, the reduction tree performs
  - (1/2)N + (1/4)N + (1/8)N + … (1)N = (1- (1/N))N = N-1 operations
  - In Log (N) steps – 1,000,000 input values take 20 steps
    - Assuming that we have enough execution resources
  - Average Parallelism (N-1)/Log(N))
    - For N = 1,000,000, average parallelism is 50,000
    - However, peak resource requirement is 500,000
    - This is not resource efficient
- This is a work-efficient parallel algorithm
  - The amount of work done is comparable to an efficient sequential algorithm
  - Many parallel algorithms are not work efficient

Objective

- To master parallel scan (prefix sum) algorithms
  - Frequently used for parallel work assignment and resource allocation
  - A key primitive in many parallel algorithms to convert serial computation into parallel computation
  - A foundational parallel computation pattern
  - Work efficiency in parallel code/algorithms

Inclusive Scan (Prefix-Sum) Definition

Definition: *The* scan *operation takes a binary associative operator* $\oplus$ (pronounced as circle plus), *and an array of n elements*

$$[x_0, x_1, \ldots, x_{n-1}],$$

and returns the array

$$[x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1})].$$

Example: If $\oplus$ is addition, then scan operation on the array would return

[3 1 7 0 4 1 6 3], [3 4 11 11 15 16 22 25].

An Inclusive Scan Application Example

- Assume that we have a 100-inch sandwich to feed 10 people
- We know how much each person wants in inches
  - [3 5 2 7 28 4 3 0 8 1]
- How do we cut the sandwich quickly?
- How much will be left?

- Method 1: cut the sections sequentially: 3 inches first, 5 inches second, 2 inches third, etc.

- Method 2: calculate prefix sum:
  - [3, 8, 10, 17, 45, 49, 52, 52, 60, 61] (39 inches left)

Typical Applications of Scan

– Scan is a simple and useful parallel building block

  – Convert recurrences from sequential:

```
for(j=1;j<n;j++)
  out[j] = out[j-1] + f(j);
```

  – Into parallel:

```
forall(j) { temp[j] = f(j) };
  scan(out, temp);
```

– Useful for many parallel algorithms:

- Radix sort
- Quicksort
- String comparison
- Lexical analysis
- Stream compaction
- Polynomial evaluation
- Solving recurrences
- Tree operations
- Histograms, ….

Other Applications

- Assigning camping spots
- Assigning Farmer’s Market spaces
- Allocating memory to parallel threads
- Allocating memory buffer space for communication channels
- …

An Inclusive Sequential Addition Scan

Given a sequence $[x_0, x_1, x_2, \ldots]$

Calculate output $[y_0, y_1, y_2, \ldots]$

Such that
$$y_0 = x_0$$
$$y_1 = x_0 + x_1$$
$$y_2 = x_0 + x_1 + x_2$$
$$\ldots$$

Using a recursive definition

$$y_i = y_{i-1} + x_i$$

A Work Efficient C Implementation

```
y[0] = x[0];
for (i = 1; i < Max_i; i++) y[i] = y [i-1] + x[i];
```

Computationally efficient:

N additions needed for N elements - O(N)!

Only slightly more expensive than sequential reduction.

A Naïve Inclusive Parallel Scan

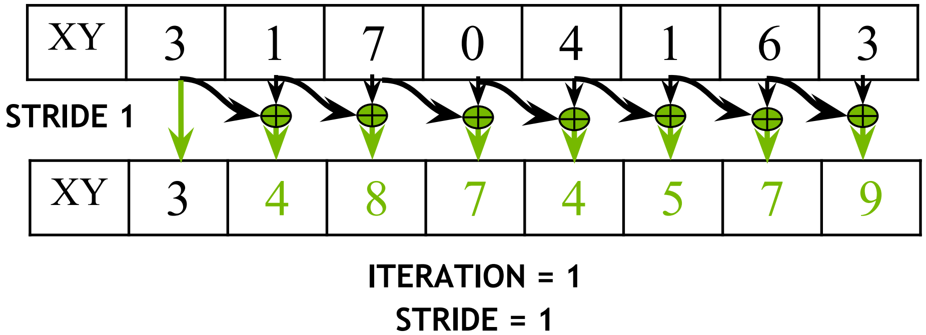
- Assign one thread to calculate each y element
- Have every thread to add up all x elements needed for the y element

$$y_0 = x_0$$
$$y_1 = x_0 + x_1$$
$$y_2 = x_0 + x_1 + x_2$$

"Parallel programming is easy as long as you do not care about performance."

A Better Parallel Scan Algorithm

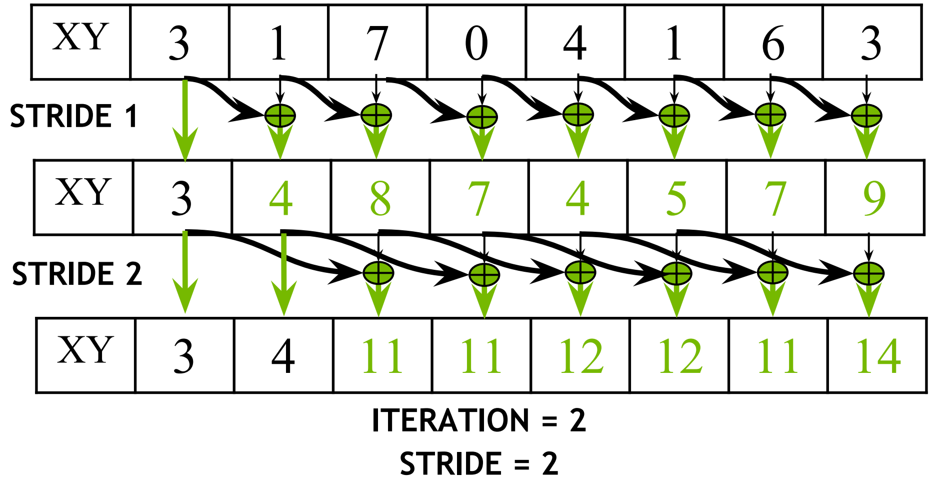
1. Read input from device global memory to shared memory
2. Iterate log(n) times; stride from 1 to n-1: double stride each iteration



- Active threads *stride* to n-1 (n-stride threads)
- Thread *j* adds elements *j* and *j-stride* from shared memory and writes result into element j in shared memory
- Requires barrier synchronization, once before read and once before write

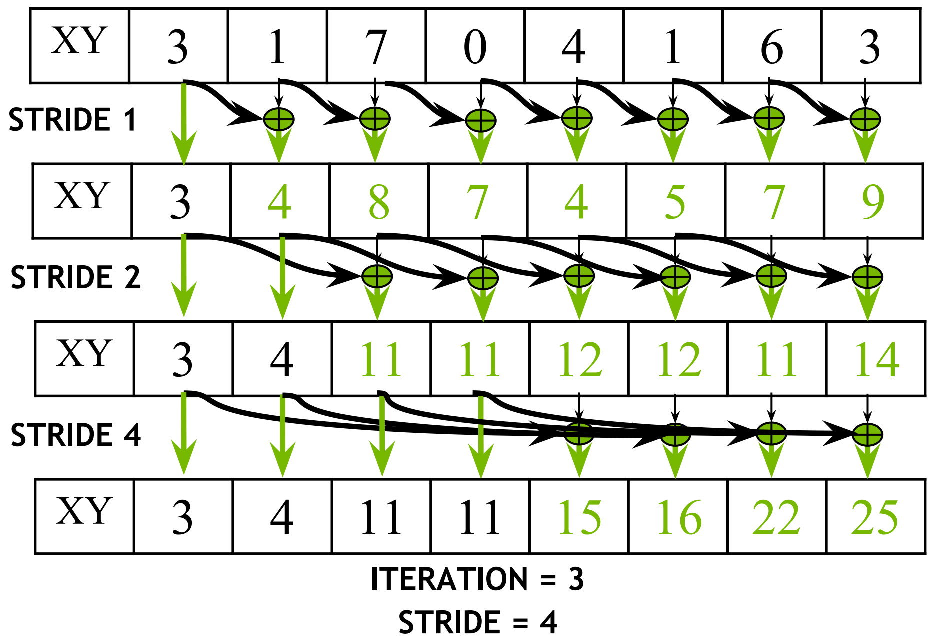
A Better Parallel Scan Algorithm

1. Read input from device to shared memory
2. Iterate log(n) times; stride from 1 to n-1: double stride each iteration.



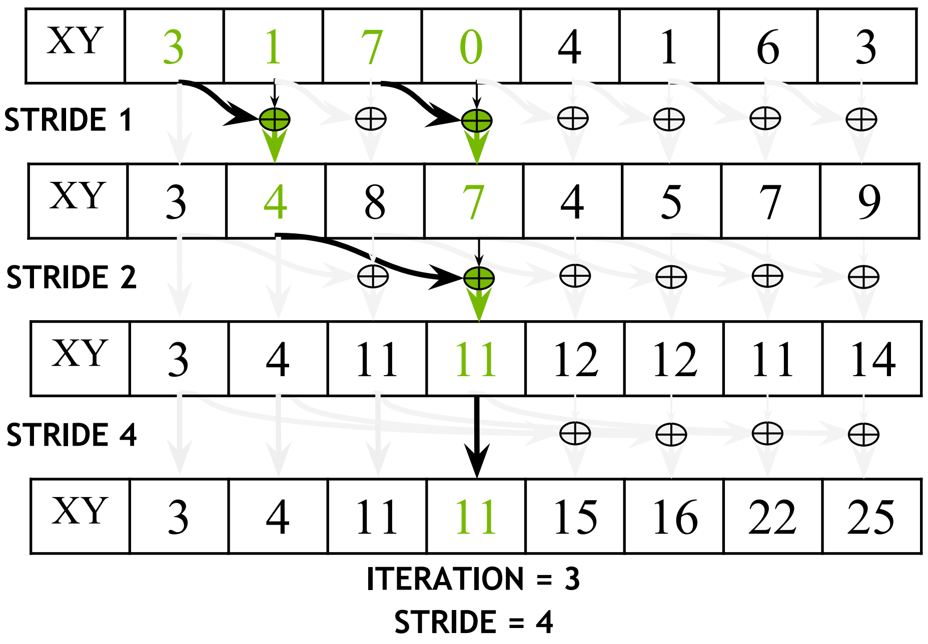
A Better Parallel Scan Algorithm

1. Read input from device to shared memory
2. Iterate log(n) times; stride from 1 to n-1: double stride each iteration
3. Write output from shared memory to device memory



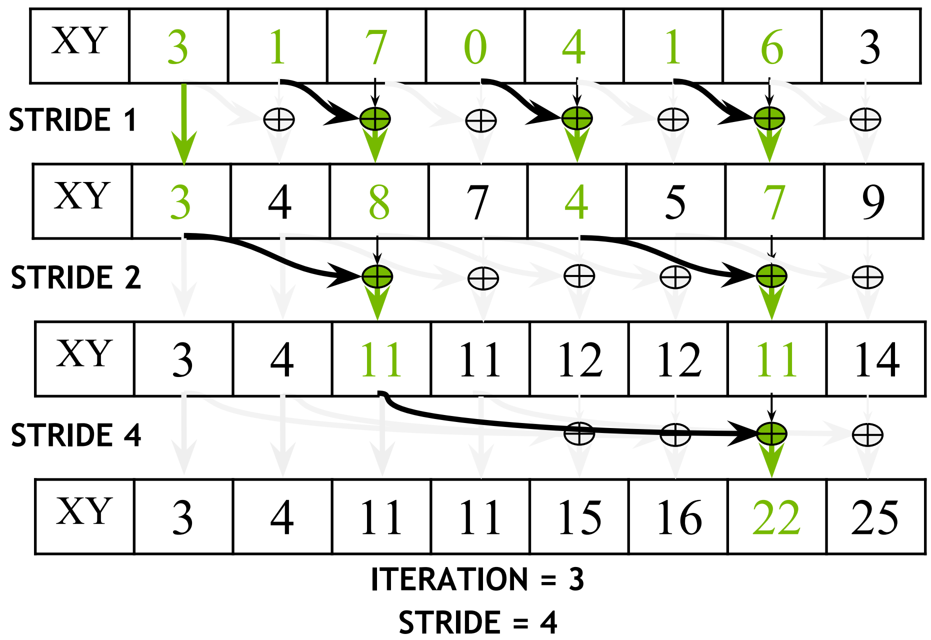
A Better Parallel Scan Algorithm

1. Read input from device to shared memory
2. Iterate log(n) times; stride from 1 to n-1: double stride each iteration
3. Write output from shared memory to device memory



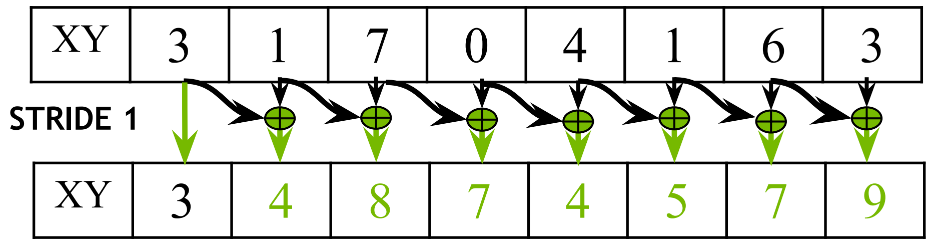
A Better Parallel Scan Algorithm

1. Read input from device to shared memory
2. Iterate log(n) times; stride from 1 to n-1: double stride each iteration
3. Write output from shared memory to device memory



Handling Dependencies

- During every iteration, each thread can overwrite the input of another thread
  - Barrier synchronization to ensure all inputs have been properly generated
  - All threads secure input operand that can be overwritten by another thread
  - Barrier synchronization is required to ensure that all threads have secured their inputs
  - All threads perform addition and write output



ITERATION = 1
STRIDE = 1

Work Efficiency Considerations

- This Scan executes log(n) parallel iterations
  - The iterations do (n-1), (n-2), (n-4),..(n- n/2) adds each
  - Total adds: n * log(n) - (n-1) → O(n*log(n)) work

- This scan algorithm is not work efficient
  - Sequential scan algorithm does *n* adds
  - A factor of log(n) can hurt: 10x for 1024 elements!

- A parallel algorithm can be slower than a sequential one when execution resources are saturated from low work efficiency

Improving Efficiency

- *Balanced Trees*
  - Form a balanced binary tree on the input data and sweep it to and from the root
  - Tree is not an actual data structure, but a concept to determine what each thread does at each step
- For scan:
  - Traverse down from leaves to the root building partial sums at internal nodes in the tree
    - The root holds the sum of all leaves
  - Traverse back up the tree building the output from the partial sums

Recap: Prefix Sums

- Given **A**: set of *n* integers
- Find **B**: *prefix sums*

$$B[i] = \sum_{k=1}^{i} A[k]$$

A:	3	1	1	7	2	5	9	2	4	3	3
B:	3	4	5	12	14	19	28	30	34	37	40

Credits: Nodari Sitchinava

Iterative prefix sum

- 2 phases: up-sweep, down-sweep
- Up-sweep pseudocode:

$\textsc{PrefixSum}(A[0, ..., n-1])$

for $i = 0$ to $n-1$ **in parallel do**
    $B[0][i] = A[i]$
end for
for $h = 1$ to $\log n$ **do**
    for $i = 0$ to $\frac{n}{2^h} - 1$ **in parallel do**
        $B[h][i] = B[h-1][2i] + B[h-1][2i+1]$
    end for
end for

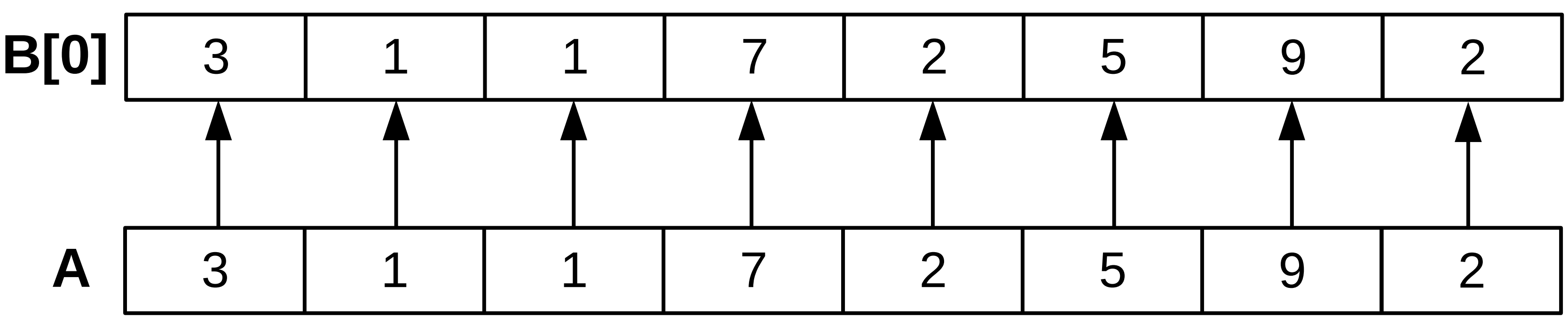
Credits: Nodari Sitchinava

Up-sweep phase

for $i = 0$ to $n - 1$ **in parallel do**

    $B[0][i] = A[i]$

end for

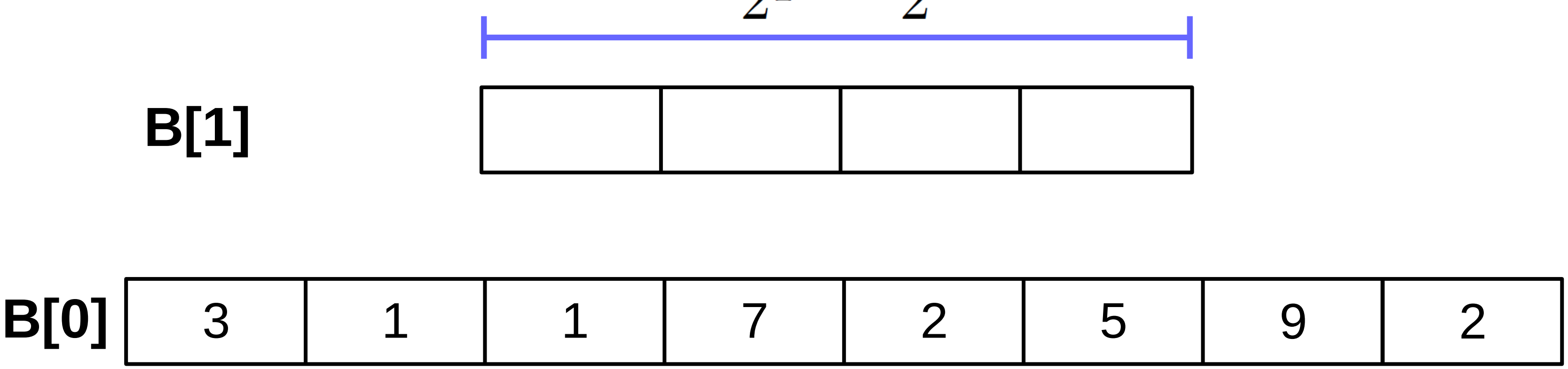


Credits: Nodari Sitchinava

Up-sweep phase

for $h = 1$ to $\log n$ **do**
    for $\boxed{i = 0 \text{ to } \frac{n}{2^h} - 1}$ **in parallel do**
        $B[h][i] = B[h-1][2i] + B[h-1][2i+1]$
    end for
end for

$$\frac{n}{2^1} = \frac{n}{2}$$

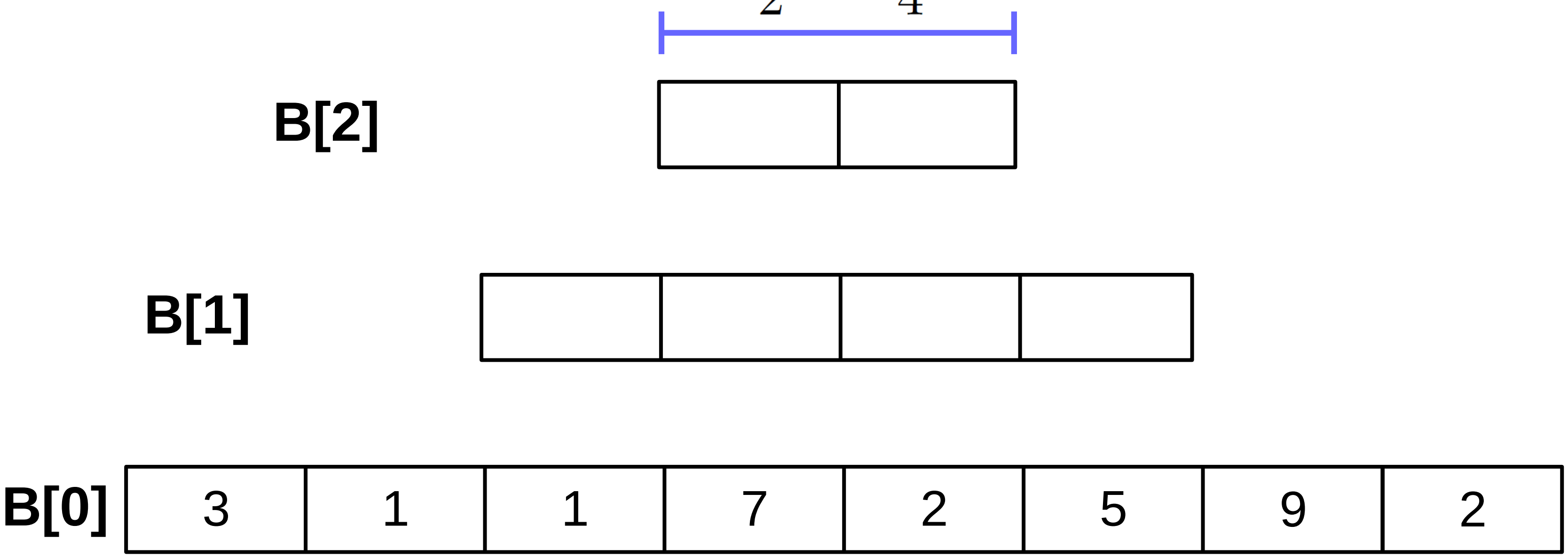


Credits: Nodari Sitchinava

Up-sweep phase

for $h = 1$ to $\log n$ **do**
  for $\boxed{i = 0 \text{ to } \frac{n}{2^h} - 1}$ **in parallel do**
    $B[h][i] = B[h-1][2i] + B[h-1][2i+1]$
  end for
end for

$$\frac{n}{2^2} = \frac{n}{4}$$

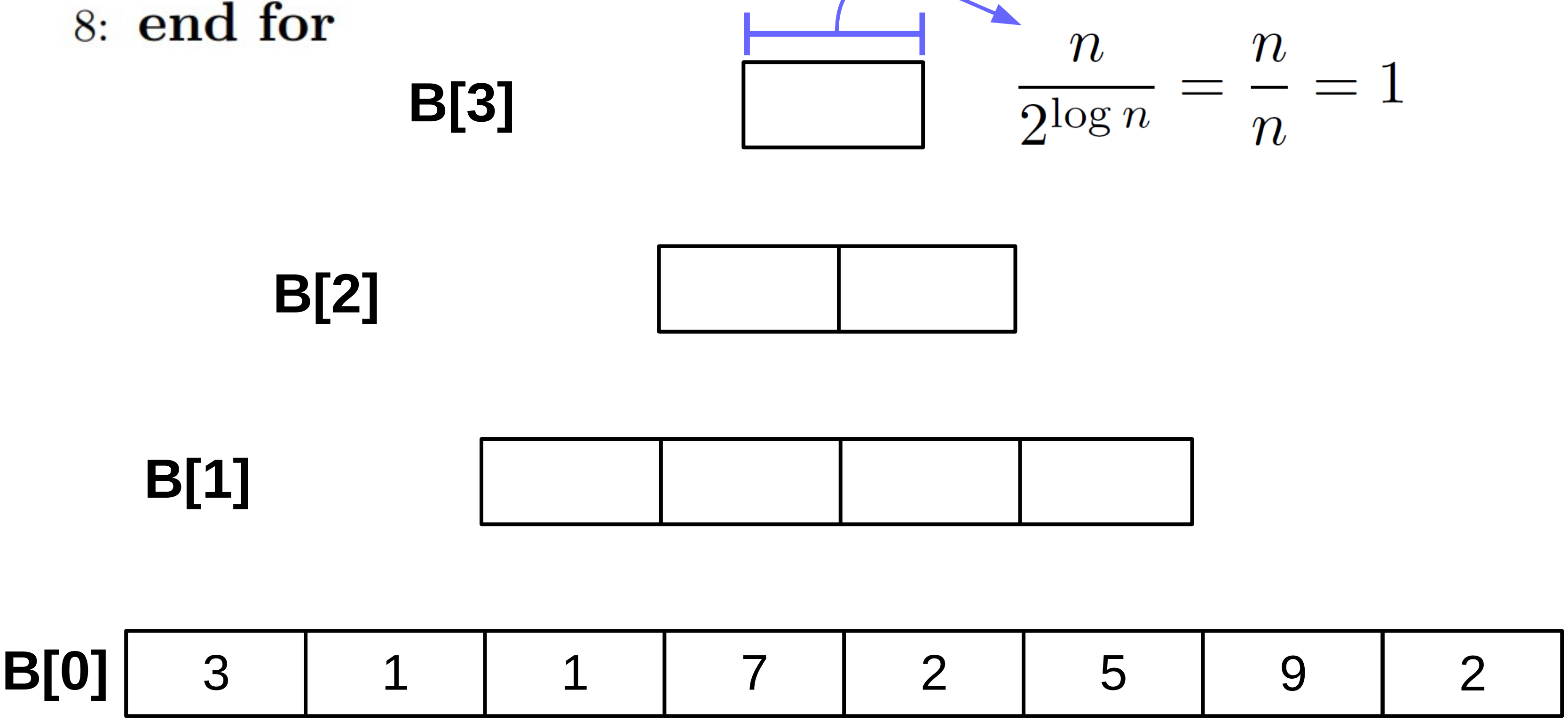


Credits: Nodari Sitchinava

Up-sweep phase

```
for h = 1 to log n do
    for [i = 0 to n/2^h − 1] in parallel do
        B[h][i] = B[h−1][2i] + B[h−1][2i+1]
    end for
end for
```

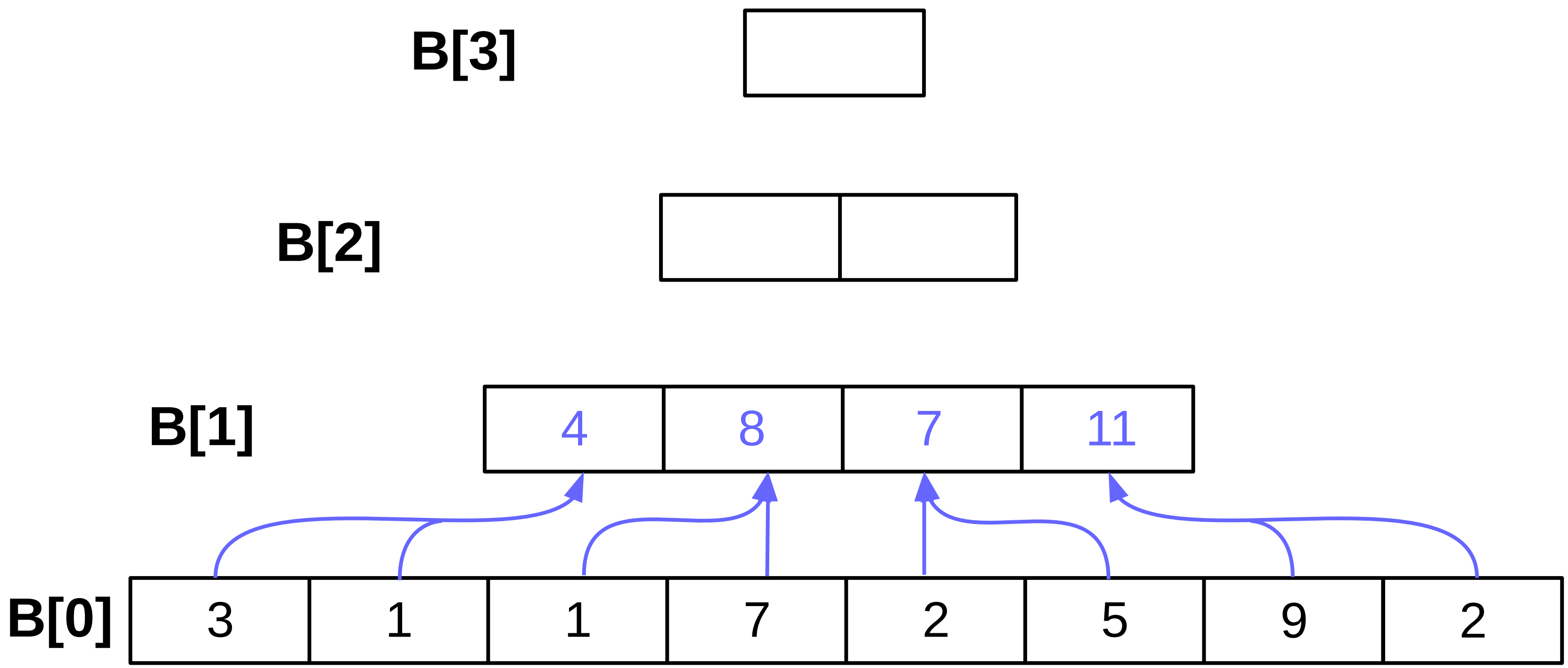
$$\frac{n}{2^{\log n}} = \frac{n}{n} = 1$$



Credits: Nodari Sitchinava

Up-sweep phase

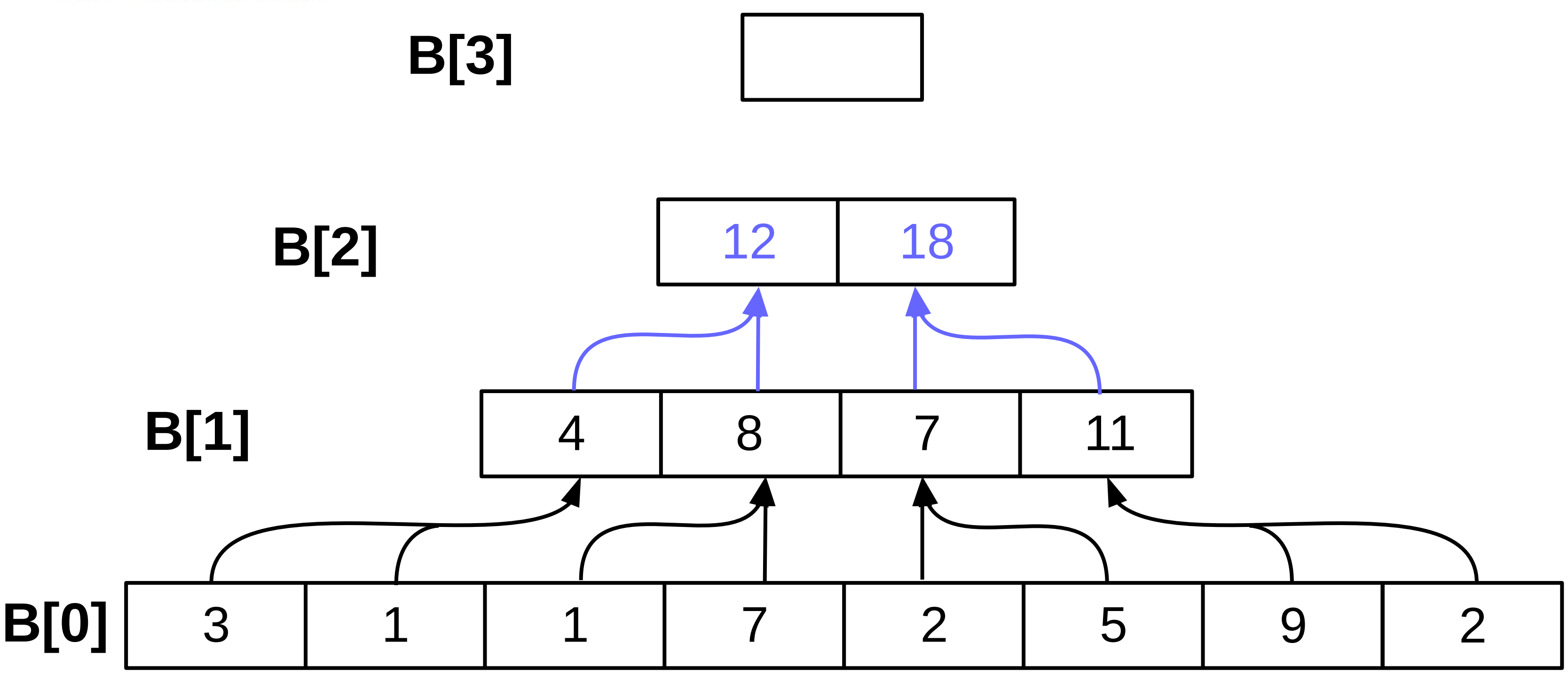
$$
\begin{array}{l}
\textbf{for } h = 1 \text{ to } \log n \textbf{ do} \\
\quad \textbf{for } i = 0 \text{ to } \frac{n}{2^h} - 1 \textbf{ in parallel do} \\
\quad\quad B[h][i] = B[h-1][2i] + B[h-1][2i+1] \\
\quad \textbf{end for} \\
\textbf{end for}
\end{array}
$$



Credits: Nodari Sitchinava

Up-sweep phase

$$
\begin{array}{l}
\textbf{for } h = 1 \text{ to } \log n \textbf{ do} \\
\quad \textbf{for } \ i = 0 \text{ to } \frac{n}{2^h} - 1 \ \ \textbf{in parallel do} \\
\quad\quad B[h][i] = B[h-1][2i] + B[h-1][2i+1] \\
\quad \textbf{end for} \\
\textbf{end for}
\end{array}
$$



Credits: Nodari Sitchinava

Up-sweep phase

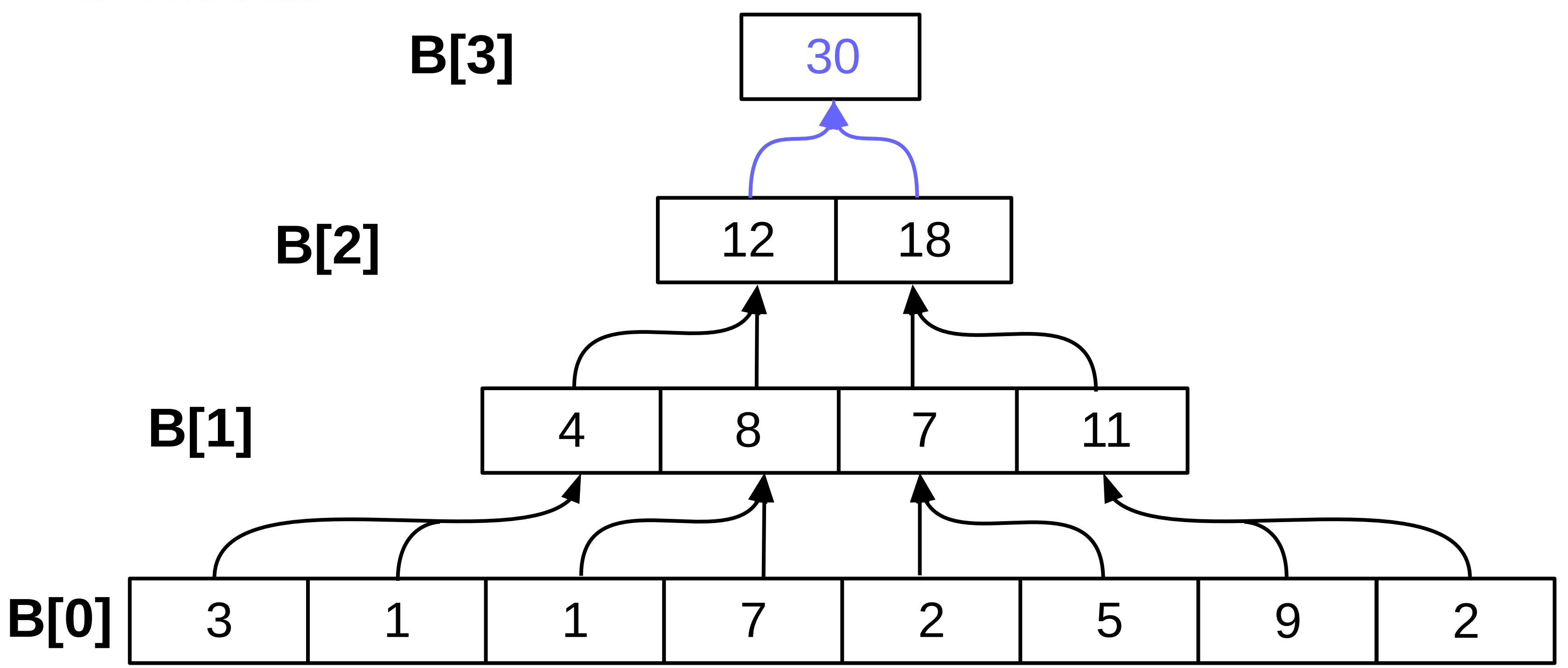
for $h = 1$ to $\log n$ **do**

  for $i = 0$ to $\frac{n}{2^h} - 1$ **in parallel do**

    $B[h][i] = B[h-1][2i] + B[h-1][2i+1]$

  end for

end for



Credits: Nodari Sitchinava

Down-sweep phase

$C[\log n][0] = 0$
for $h = \log n - 1$ down to 0 **do**
    for $i = 0$ to $\frac{n}{2^h} - 1$ **in parallel do**
        if $i \% 2 == 0$ **then**
            $C[h][i] = C[h+1][i/2]$
        else
            $C[h][i] = C[h+1][\frac{i-1}{2}] + B[h][i-1]$
        end if
    end for
end for
for $i = 0$ to $n - 1$ **in parallel do**
    $A[i] = A[i] + C[0, i]$
end for

Credits: Nodari Sitchinava

Down-sweep phase

$$9:\ C[\log n][0] = 0$$

C[3]

0

B[2]

12	18

B[1]

4	8	7	11

B[0]

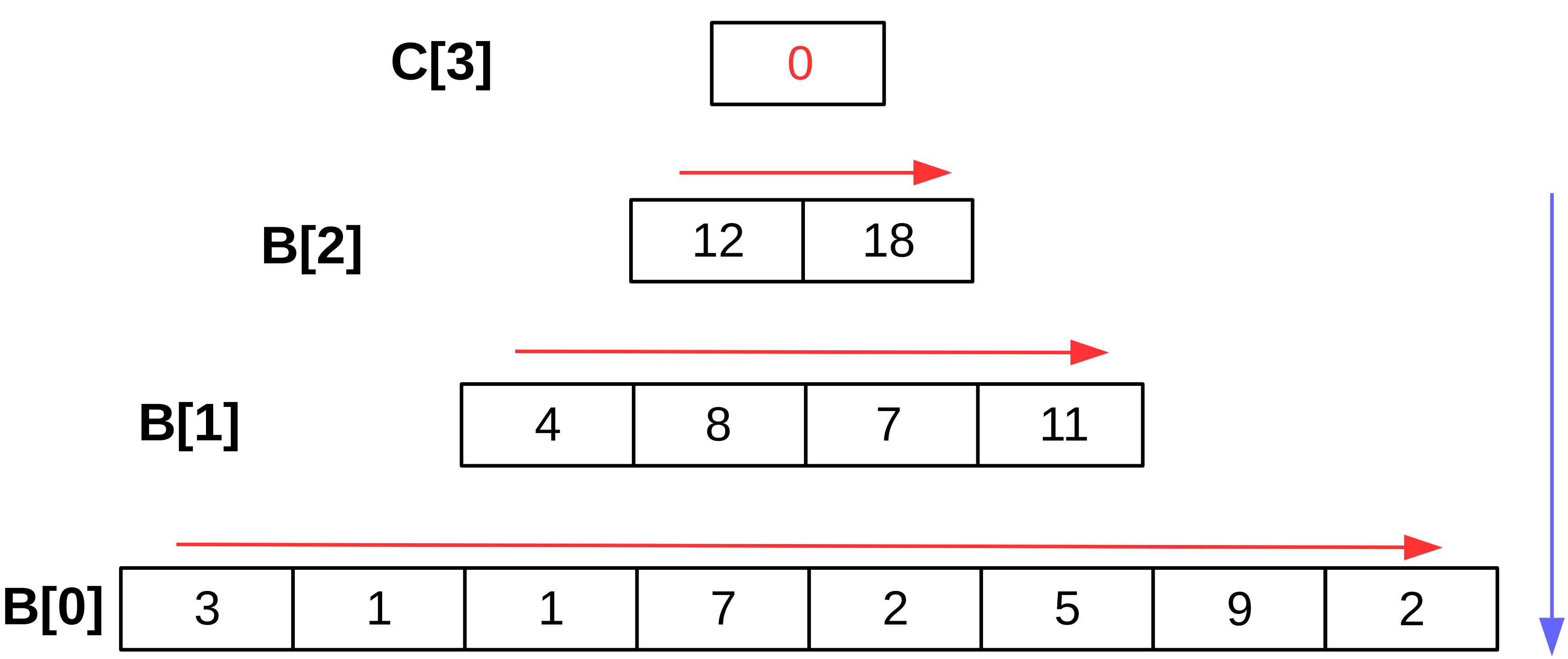
3	1	1	7	2	5	9	2

Credits: Nodari Sitchinava

Down-sweep phase

for $h = \log n - 1$ down to 0 **do**

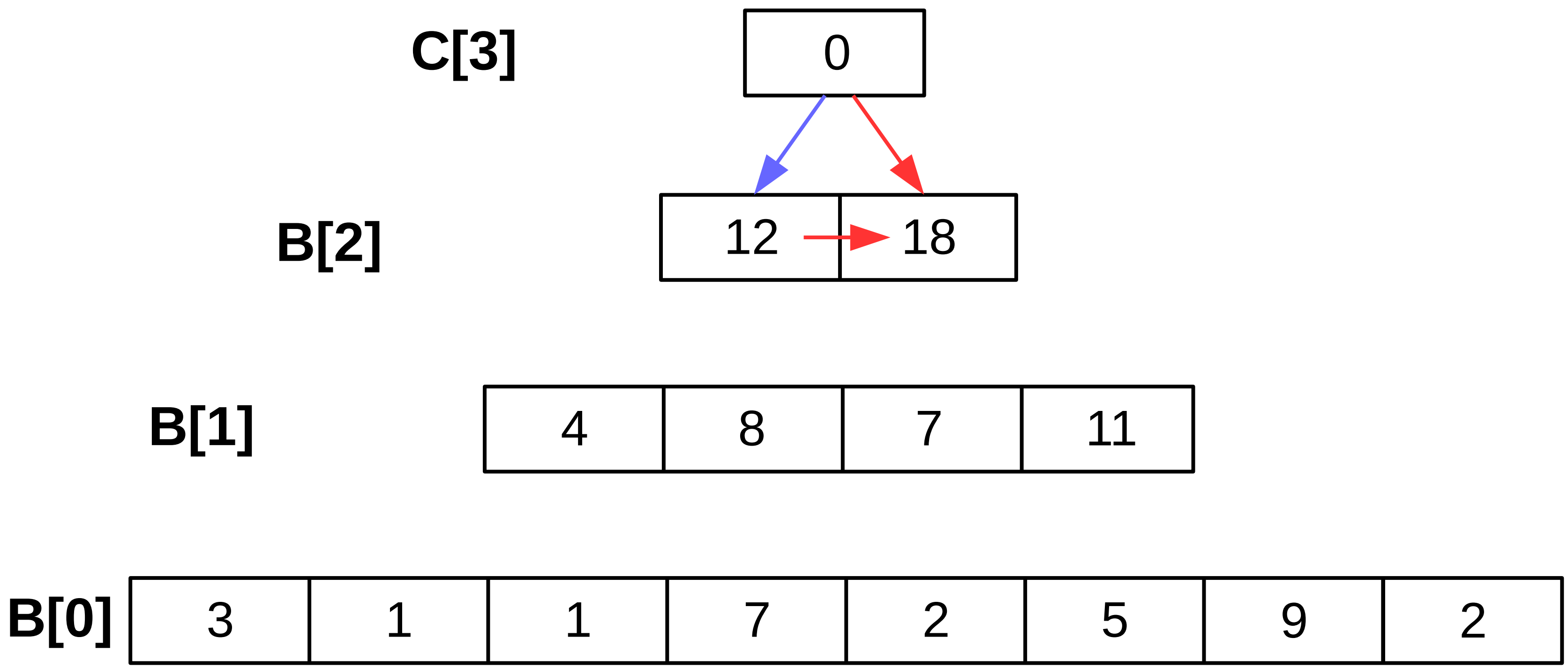
    for $i = 0$ to $\frac{n}{2^h} - 1$ **in parallel do**



Credits: Nodari Sitchinava

Down-sweep phase

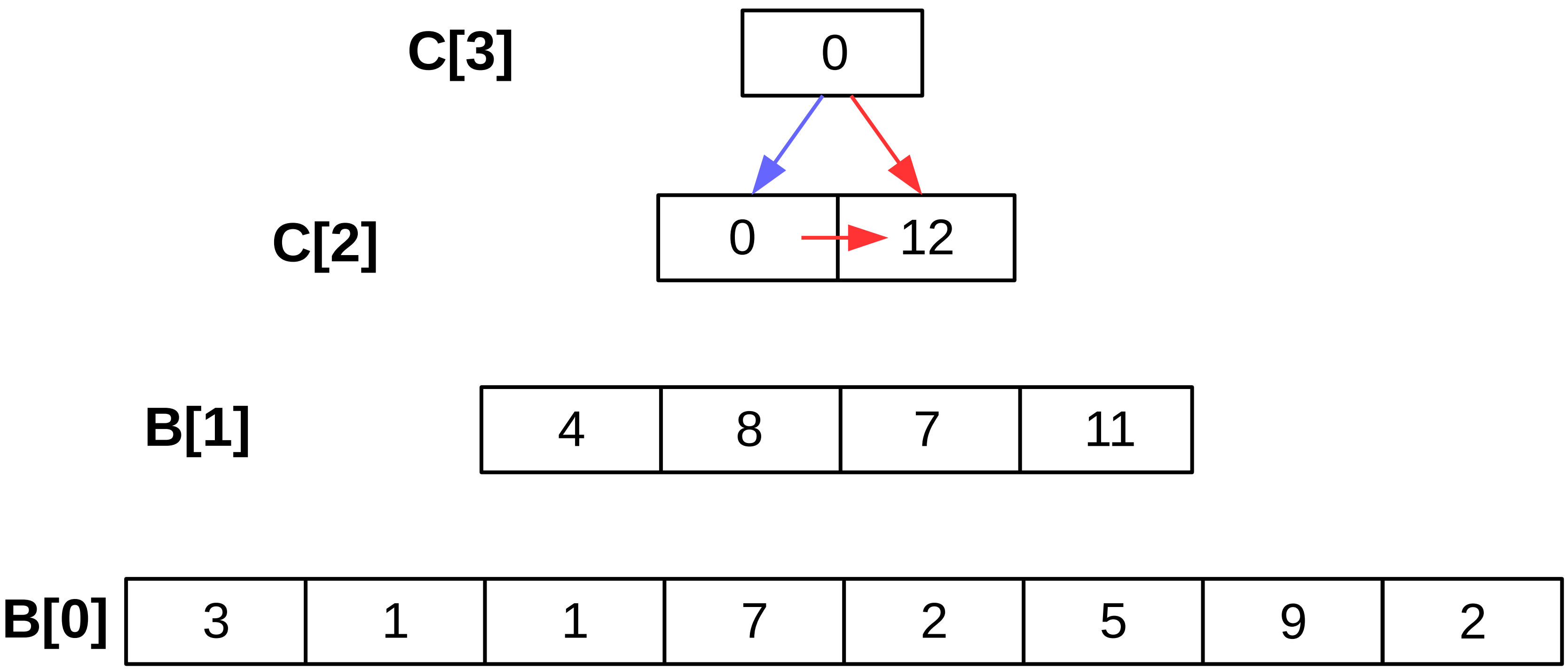
if $i \ \% \ 2 == 0$ **then**
    $C[h][i] = C[h+1][i/2]$
else
    $C[h][i] = C[h+1][\frac{i-1}{2}] + B[h][i-1]$



Credits: Nodari Sitchinava

Down-sweep phase

$$
\begin{array}{l}
\textbf{if } i \ \% \ 2 == 0 \textbf{ then} \\
\quad C[h][i] = C[h+1][i/2] \\
\textbf{else} \\
\quad C[h][i] = C[h+1][\frac{i-1}{2}] + B[h][i-1]
\end{array}
$$



Credits: Nodari Sitchinava

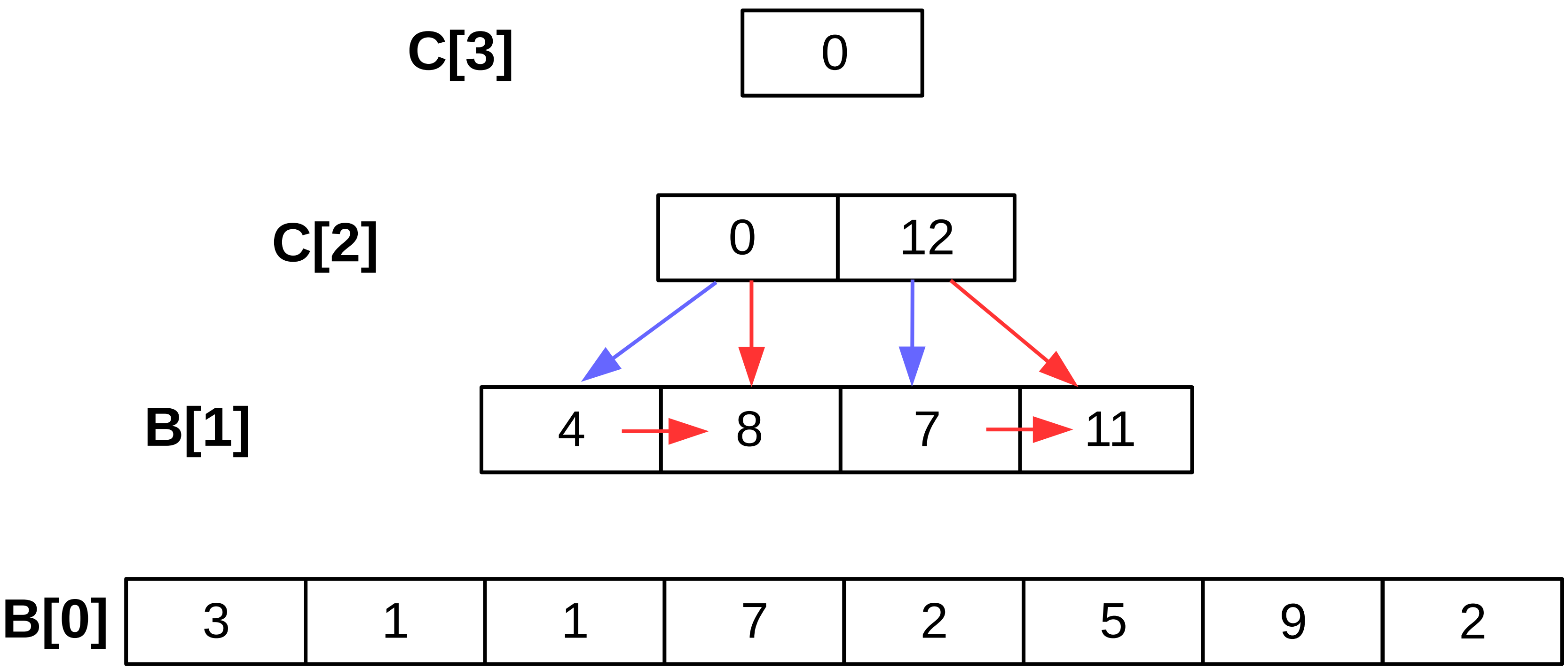
Down-sweep phase

if $i \ \% \ 2 == 0$ **then**

$\quad C[h][i] = C[h+1][i/2]$

else

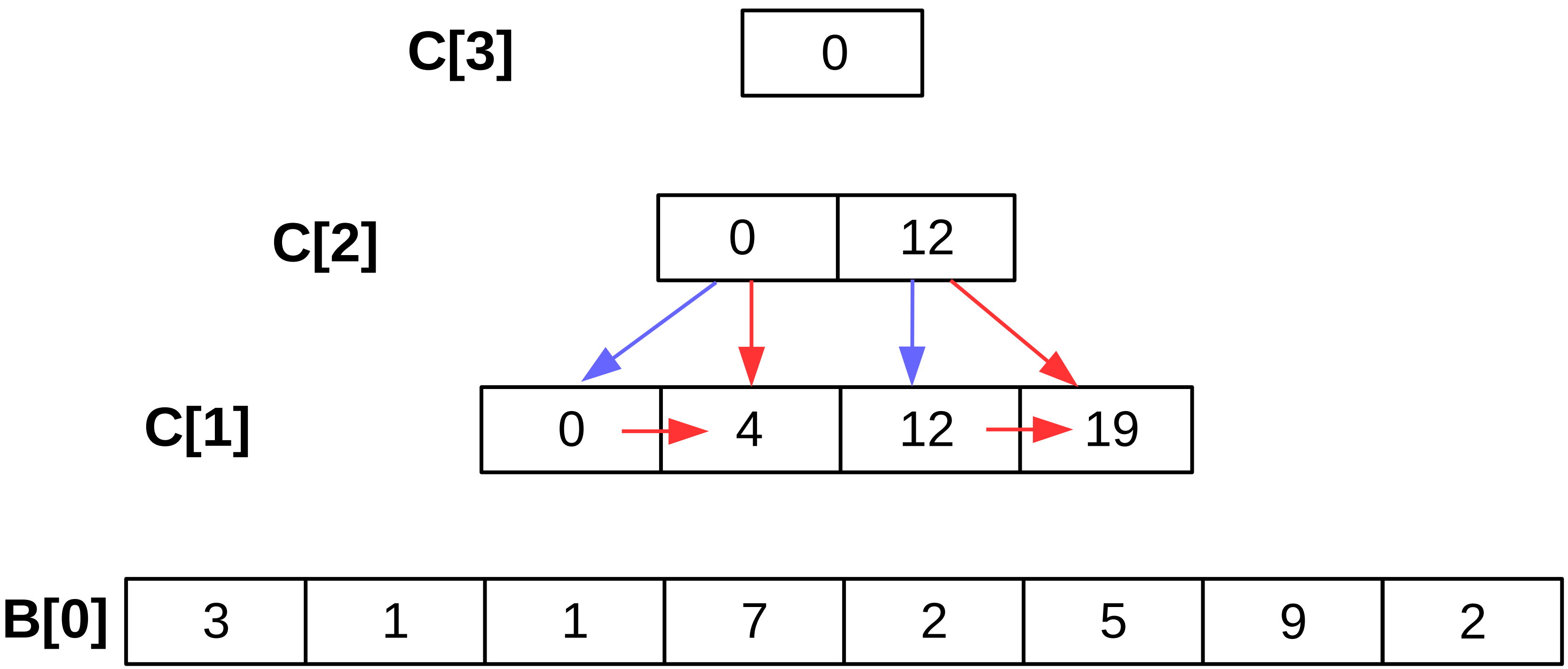
$\quad C[h][i] = C[h+1][\frac{i-1}{2}] + B[h][i-1]$



Credits: Nodari Sitchinava

Down-sweep phase

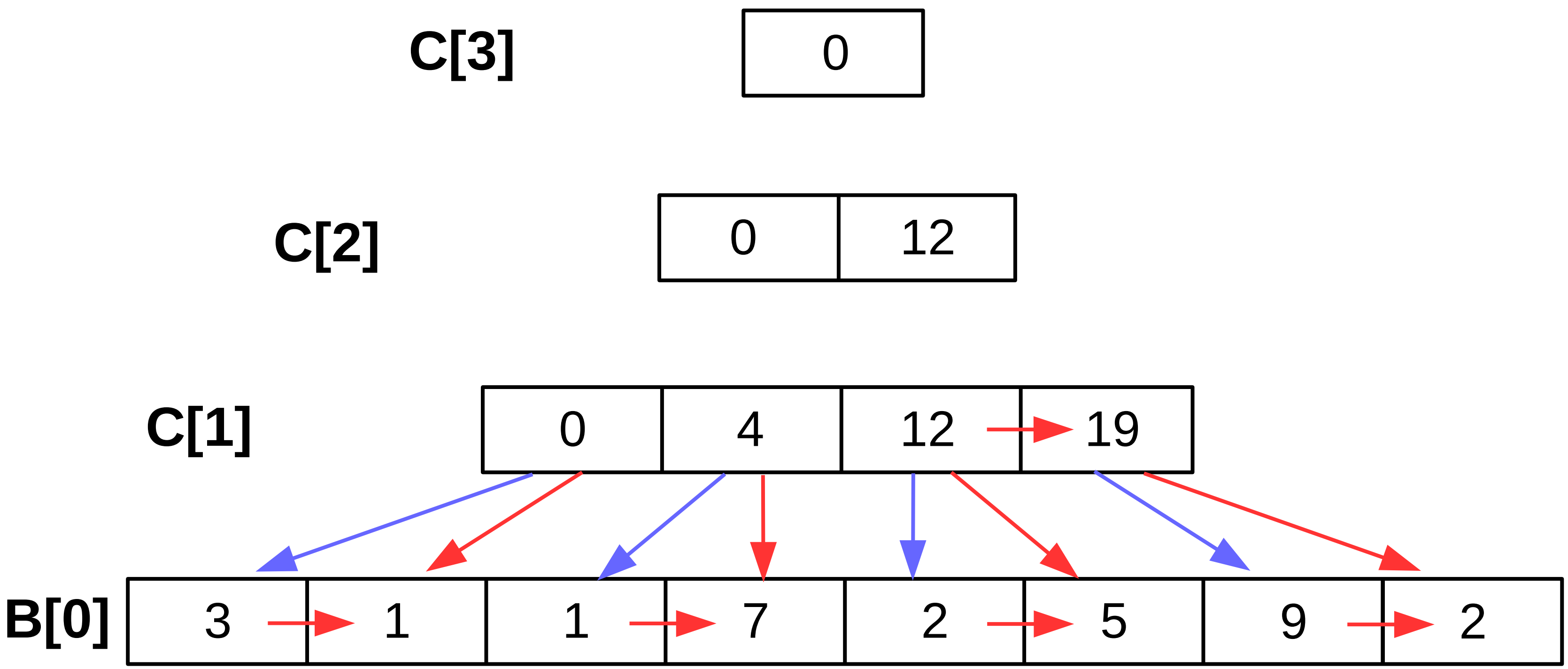
$$
\begin{array}{l}
\textbf{if } i \% 2 == 0 \textbf{ then} \\
\quad C[h][i] = C[h+1][i/2] \\
\textbf{else} \\
\quad C[h][i] = C[h+1][\frac{i-1}{2}] + B[h][i-1]
\end{array}
$$



Credits: Nodari Sitchinava

Down-sweep phase

if $i \% 2 == 0$ **then**
  $C[h][i] = C[h+1][i/2]$
else
  $C[h][i] = C[h+1][\frac{i-1}{2}] + B[h][i-1]$



Credits: Nodari Sitchinava

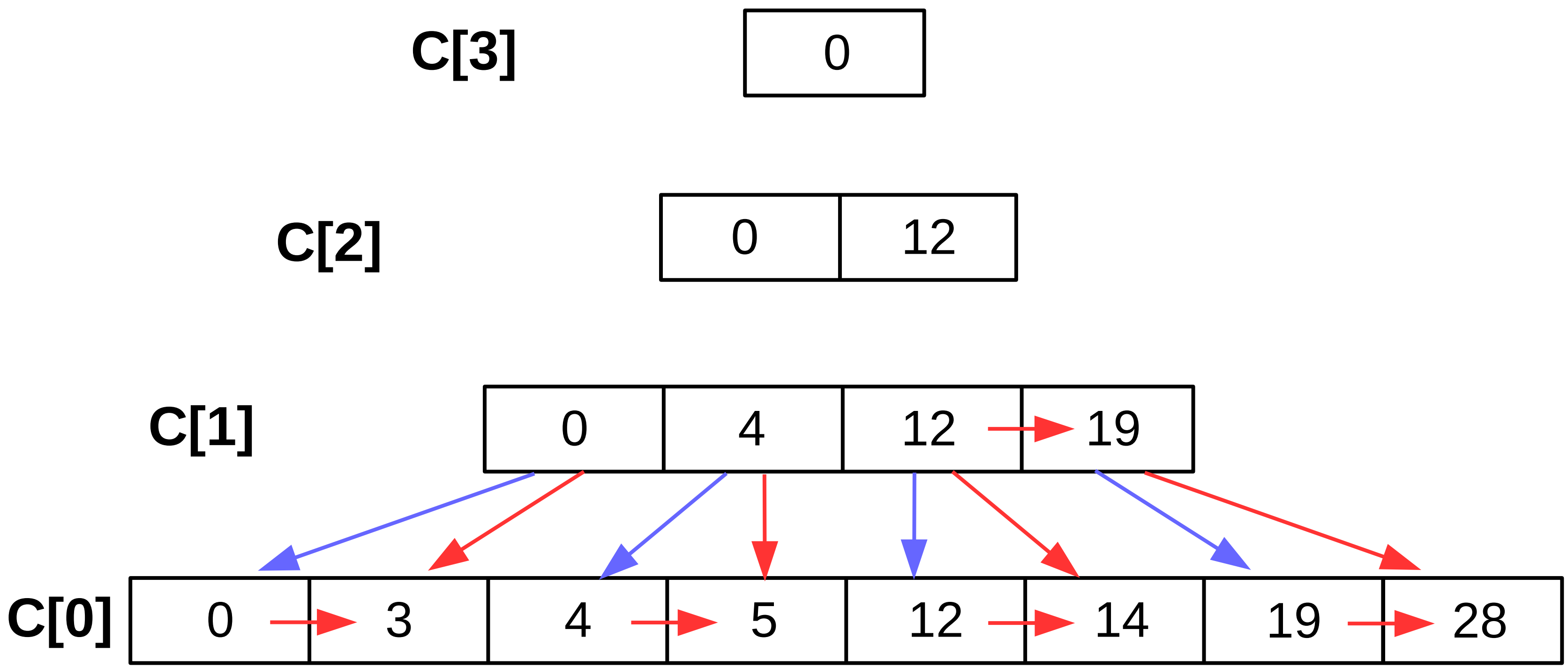
Down-sweep phase

if i % $2 == 0$ **then**

$\quad C[h][i] = C[h+1][i/2]$

else

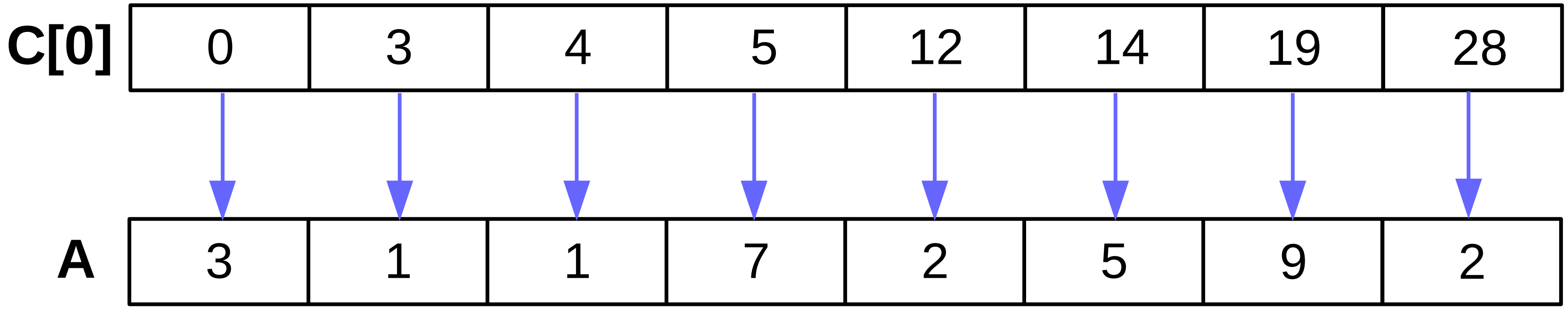
$\quad C[h][i] = C[h+1][\frac{i-1}{2}] + B[h][i-1]$



Credits: Nodari Sitchinava

Down-sweep phase

for $i = 0$ to $n - 1$ **in parallel do**
    $A[i] = A[i] + C[0, i]$
end for



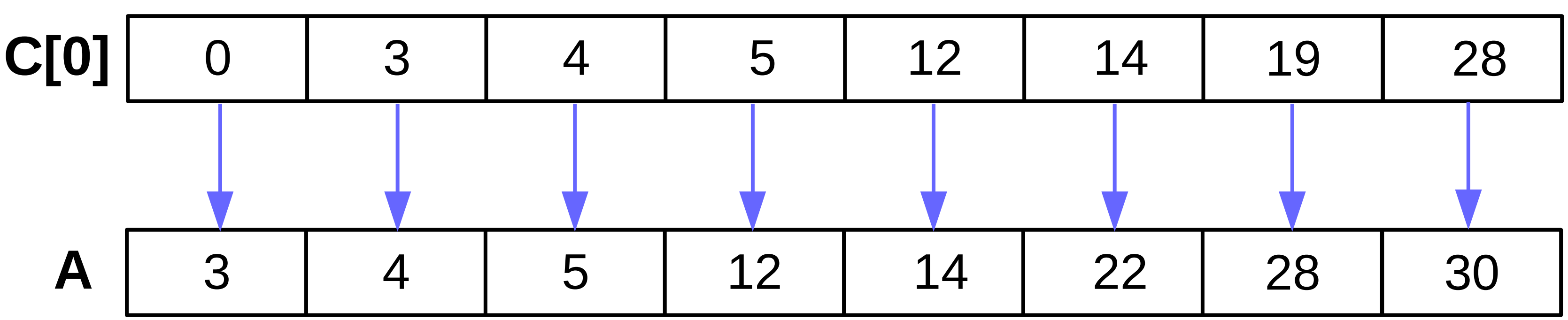
Credits: Nodari Sitchinava

Down-sweep phase

for $i = 0$ to $n - 1$ **in parallel do**

$\quad A[i] = A[i] + C[0, i]$

end for



Credits: Nodari Sitchinava

Work Analysis of the Work Efficient Kernel

- The work efficient kernel executes log(n) parallel iterations in the reduction step
  - The iterations do n/2, n/4,..1 adds
  - Total adds: (n-1) $\rightarrow$ O(n) work

- It executes log(n)-1 parallel iterations in the post-reduction reverse step
  - The iterations do 2-1, 4-1, …. n/2-1 adds
  - Total adds: (n-2) – (log(n)-1) $\rightarrow$ O(n) work

- Both phases perform up to no more than 2x(n-1) adds

- The total number of adds is no more than twice of that done in the efficient sequential algorithm
  - The benefit of parallelism can easily overcome the 2X work when there is sufficient hardware

Applications of prefix sums

- More useful than it seems:
  - Create an array of 1s and 0s
  - Prefix sums gives # of 1s up to each point
  - Used to ***separate*** an array into 2
    - Using almost **any** criteria!
- Examples:
  - separate array into upper-case and lower-case letters
  - separate array into numbers >x and <x

Credits: Nodari Sitchinava

Stream Compaction

- A common use case for parallel scans
- Stream compaction is the removal of unwanted or irrelevant elements from an input stream based on some predicate
- The elements which pass the predicate test are placed in contiguous memory

Stream Compaction

0	7	0	0	4	0	1	0	0	0	8	4	0	0	6	0

Stream Compaction

0	7	0	0	4	0	1	0	0	0	8	4	0	0	6	0

Predicate: x > 0

Stream Compaction

0	7	0	0	4	0	1	0	0	0	8	4	0	0	6	0

Predicate: x > 0

0	1	0	0	1	0	1	0	0	0	1	1	0	0	1	0

Stream Compaction

0	7	0	0	4	0	1	0	0	0	8	4	0	0	6	0

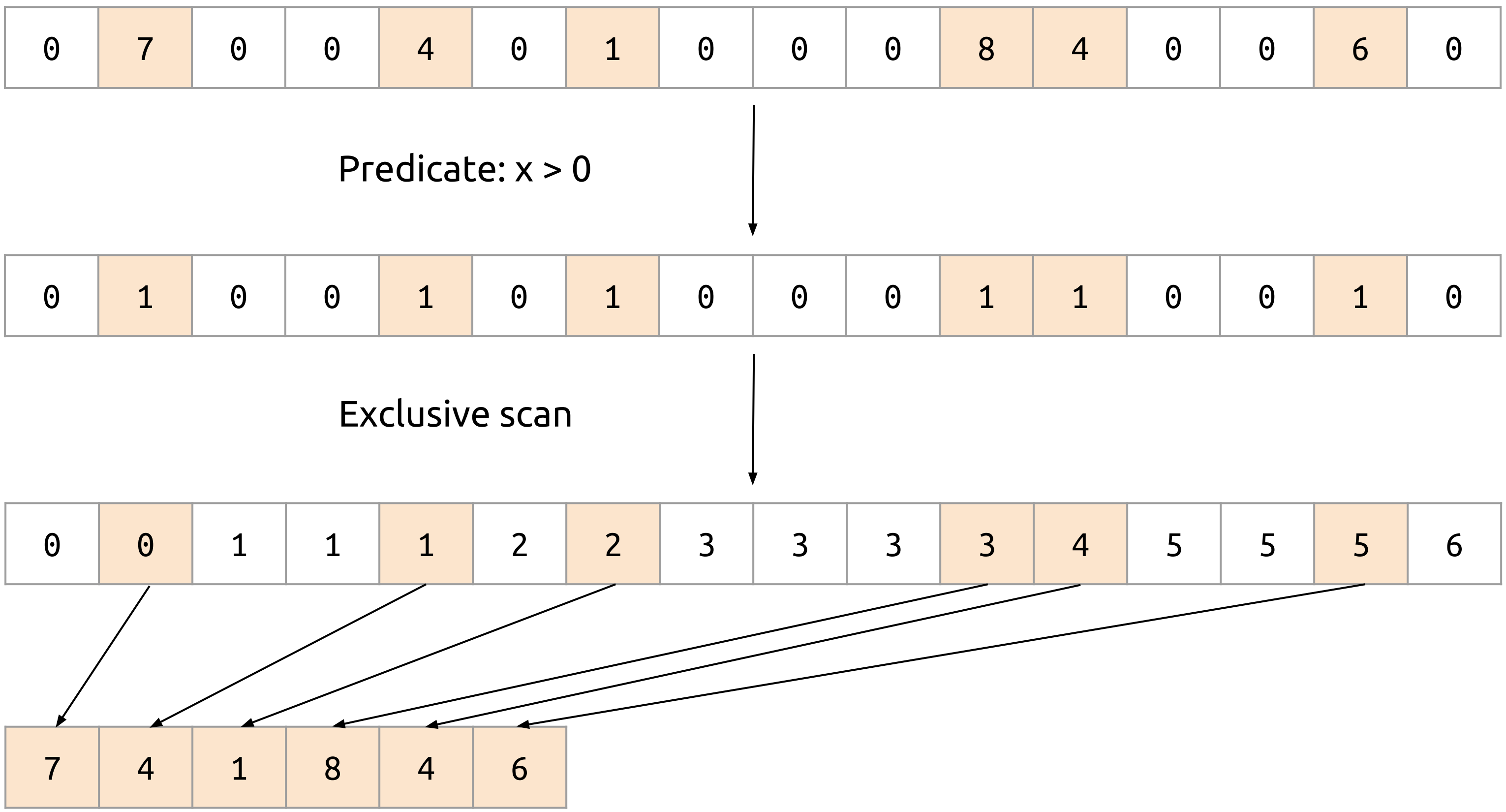
Predicate: x > 0

0	1	0	0	1	0	1	0	0	0	1	1	0	0	1	0

Exclusive scan

0	0	1	1	1	2	2	3	3	3	3	4	5	5	5	6

Stream Compaction




```
if (predicate(input[x])) {
  output[scan[x]] = input[x];
}
```

Example: string separation

- Separate array **A** into lower-case and upper-case:

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S

Credits: Nodari Sitchinava

Example: string separation

- Create bitstring B:
- 1 if upper-case, 0 otherwise

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S

Credits: Nodari Sitchinava

Example: string separation

- Create bitstring B:
- 1 if upper-case, 0 otherwise

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B	0	1	0	0	1	1	0	1	1	0	1	0	1	1	0	1	1

- Time/work to do this in parallel?

Credits: Nodari Sitchinava

Example: string separation

- Create bitstring B:
- 1 if upper-case, 0 otherwise

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B	0	1	0	0	1	1	0	1	1	0	1	0	1	1	0	1	1

- Time/work to do this in parallel?

$$W(n) = O(n)$$

$$T(n) = O(1)$$

Credits: Nodari Sitchinava

Example: string separation

- Perform **prefix sums** on B

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B	0	1	0	0	1	1	0	1	1	0	1	0	1	1	0	1	1

Credits: Nodari Sitchinava

Example: string separation

- Perform **prefix sums** on B

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B	0	1	1	1	2	3	3	4	5	5	6	6	7	8	8	9	10

- What is B[i]?

Credits: Nodari Sitchinava

Example: string separation

- Perform **prefix sums** on B

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B	0	1	1	1	2	3	3	4	5	5	6	6	7	8	8	9	10

- What is B[i]?
  - The number of capital letters with index ≤ i

Credits: Nodari Sitchinava

Example: string separation

- Copy capital letters into C

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B	0	1	1	1	2	3	3	4	5	5	6	6	7	8	8	9	10
C																	

- How can we use B to write **only capitals** into C?

Credits: Nodari Sitchinava

Example: string separation

- Copy capital letters into C

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B	0	1	1	1	2	3	3	4	5	5	6	6	7	8	8	9	10
C																	

- How can we use B to write **only capitals** into C?
  - B[i] is the **index** of each capital in C!

Credits: Nodari Sitchinava

Example: string separation

- Copy capital letters into C

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B	0	1	1	1	2	3	3	4	5	5	6	6	7	8	8	9	10
C	P	R	E	F	I	X	S	U	M	S							
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

- How can we use B to write **only capitals** into C?
  - B[i] is the **index** of each capital in C!

Credits: Nodari Sitchinava

Example: string separation

- Create **B‘**
- 1 for lower-case, 0 otherwise

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B‘	1	0	1	1	0	0	1	0	0	1	0	1	0	0	1	0	0
C	P	R	E	F	I	X	S	U	M	S							
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Credits: Nodari Sitchinava

Example: string separation

- Prefix sums on **B‘**

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B‘	1	1	2	3	3	3	4	4	4	5	5	6	6	6	7	7	7
C	P	R	E	F	I	X	S	U	M	S							
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Credits: Nodari Sitchinava

Example: string separation

- Copy lower-case into the rest of C

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B'	1	1	2	3	3	3	4	4	4	5	5	6	6	6	7	7	7
C	P	R	E	F	I	X	S	U	M	S							
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Credits: Nodari Sitchinava

Example: string separation

- Copy lower-case into the rest of C

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B‘	1	1	2	3	3	3	4	4	4	5	5	6	6	6	7	7	7
C	P	R	E	F	I	X	S	U	M	S	a	r	e	c	o	o	l
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
											1	2	3	4	5	6	7

- A[i] = C[j]
  - where j = B[n] + B‘[i] = 10 + B‘[i]

Credits: Nodari Sitchinava

Example: string separation

	$W(n)$	$T(n)$
Create **B** and **B'**	$O(n)$	$O(1)$
Prefix sums		
Copy into **C**		
Total algorithm		

Credits: Nodari Sitchinava

Example: string separation

	$W(n)$	$T(n)$
Create **B** and **B'**	$O(n)$	$O(1)$
Prefix sums	$O(n)$	$O(\log n)$
Copy into **C**		
Total algorithm		

Credits: Nodari Sitchinava

Example: string separation

	$W(n)$	$T(n)$
Create **B** and **B'**	$O(n)$	$O(1)$
Prefix sums	$O(n)$	$O(\log n)$
Copy into **C**	$O(n)$	$O(1)$
Total algorithm	$O(n)$	$O(\log n)$

Credits: Nodari Sitchinava

Quicksort Review

- Quicksort is a popular sorting algorithm
  - Works **in-place**
  - O(n^2) worst-case
  - BUT O(n log n) **expected**

- Each recursive call:
  - Find pivot
  - Partition around pivot

Credits: Nodari Sitchinava

Sequential Quicksort

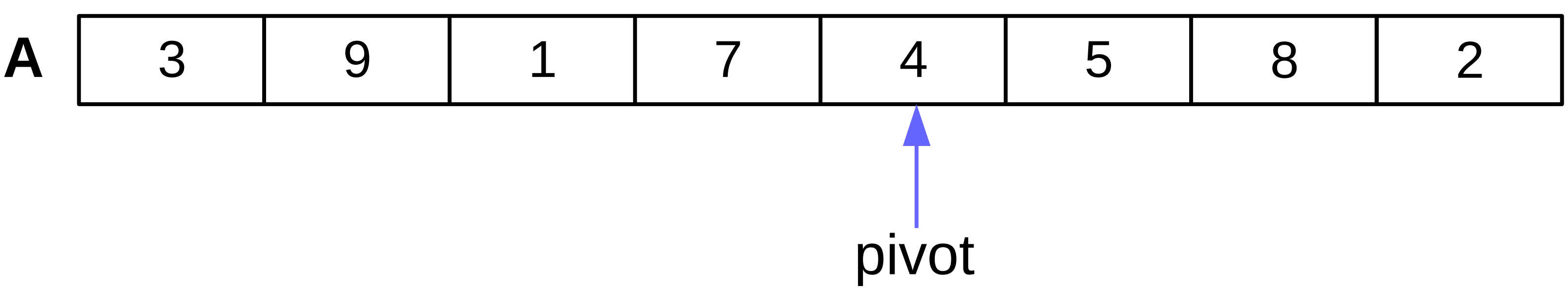
Quicksort($A[0, \cdots, n-1]$)

```
pivot = random(1 ··· n)
swap(A[0], A[pivot])
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then
        swap(A[i], A[part])
        part++
    end
end
if part > 2 then
    Quicksort(A[0,··· ,part-1])
end
if part < n-1 then
    Quicksort(A[part,··· ,n-1])
end
```

Credits: Nodari Sitchinava

Select pivot

$$\text{pivot} = \text{random}(1 \cdots n)$$
$$\text{swap}(A[0], A[\text{pivot}])$$



Credits: Nodari Sitchinava

Select pivot

$$\text{pivot} = \text{random}(1 \cdots n)$$
$$\text{swap}(A[0], A[\text{pivot}])$$

A	4	9	1	7	3	5	8	2

Credits: Nodari Sitchinava

Partition elements

```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then
        swap(A[i], A[part])
        part++
    end
end
```

A	4	9	1	7	3	5	8	2

Credits: Nodari Sitchinava

Partition elements

```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then
        swap(A[i], A[part])
        part++
    end
end
```

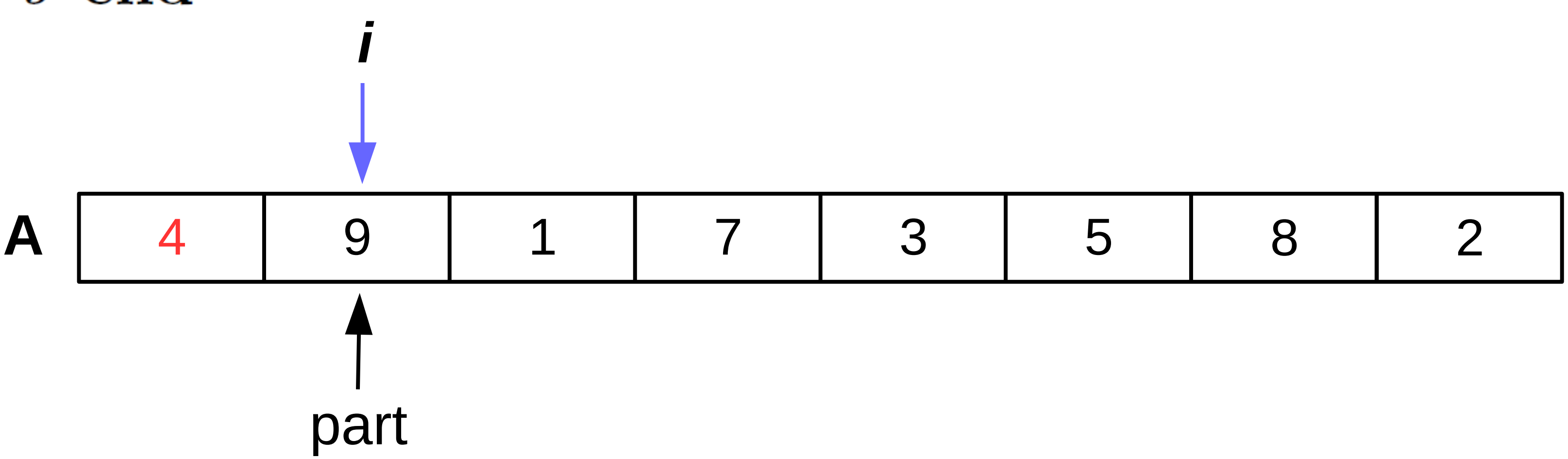
A	4	9	1	7	3	5	8	2

part

Credits: Nodari Sitchinava

Partition elements

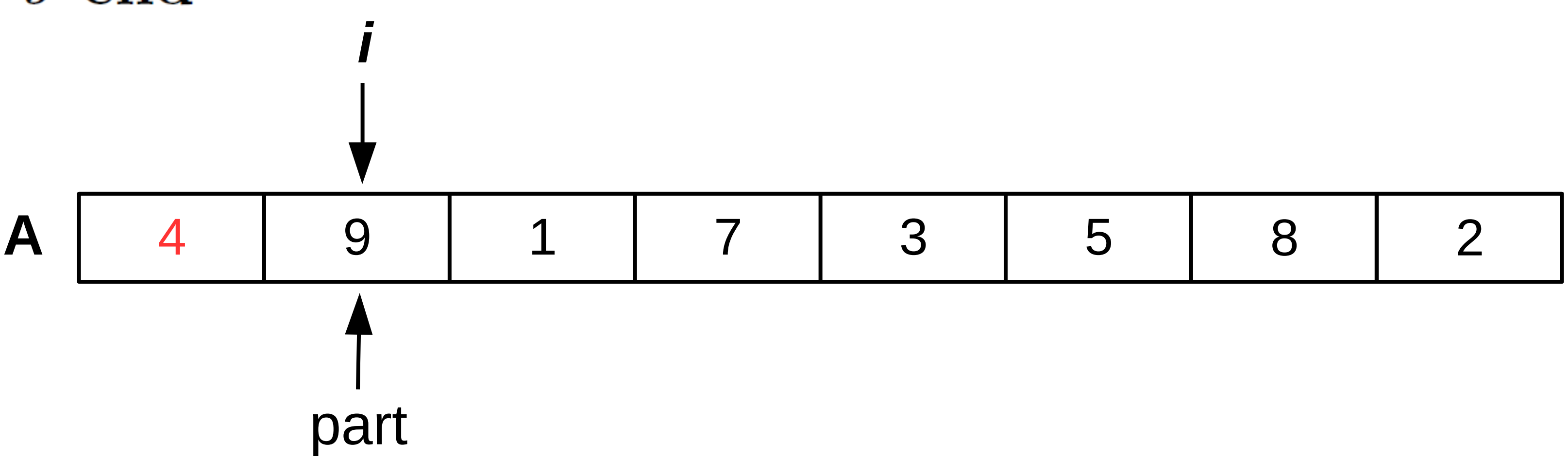
```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then
        swap(A[i], A[part])
        part++
    end
end
```



Credits: Nodari Sitchinava

Partition elements

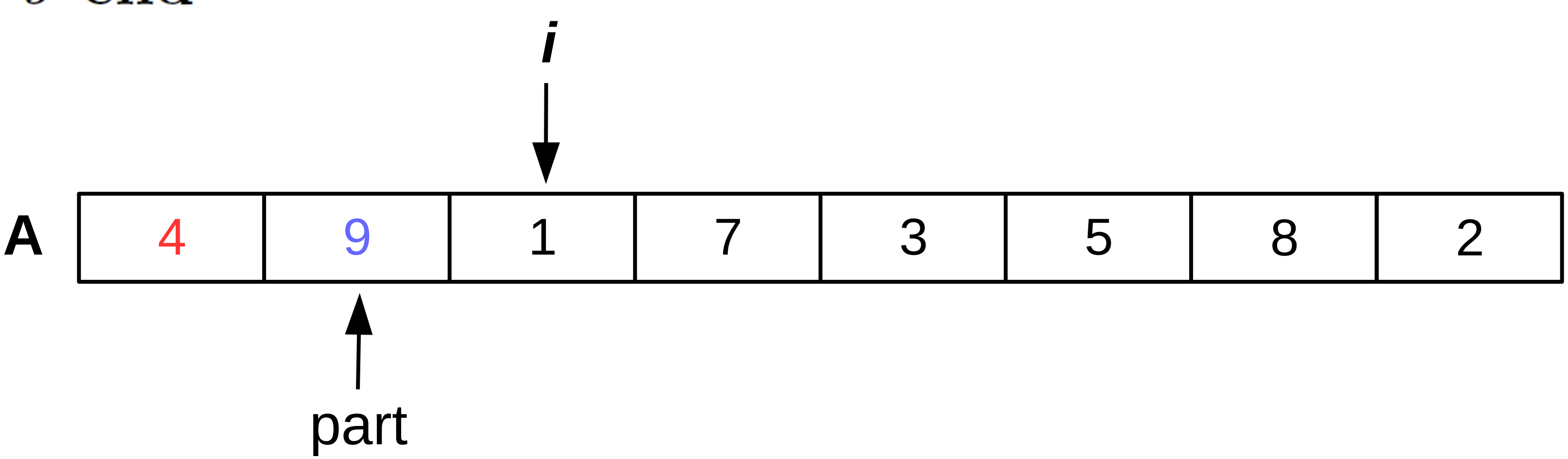
```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then      FALSE
        swap(A[i], A[part])
        part++
    end
end
```



Credits: Nodari Sitchinava

Partition elements

```
part = 1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then        TRUE
        swap(A[i], A[part])
        part++
    end
end
```



Credits: Nodari Sitchinava

Partition elements

part =1

for *i = 1 to n-1* **do**

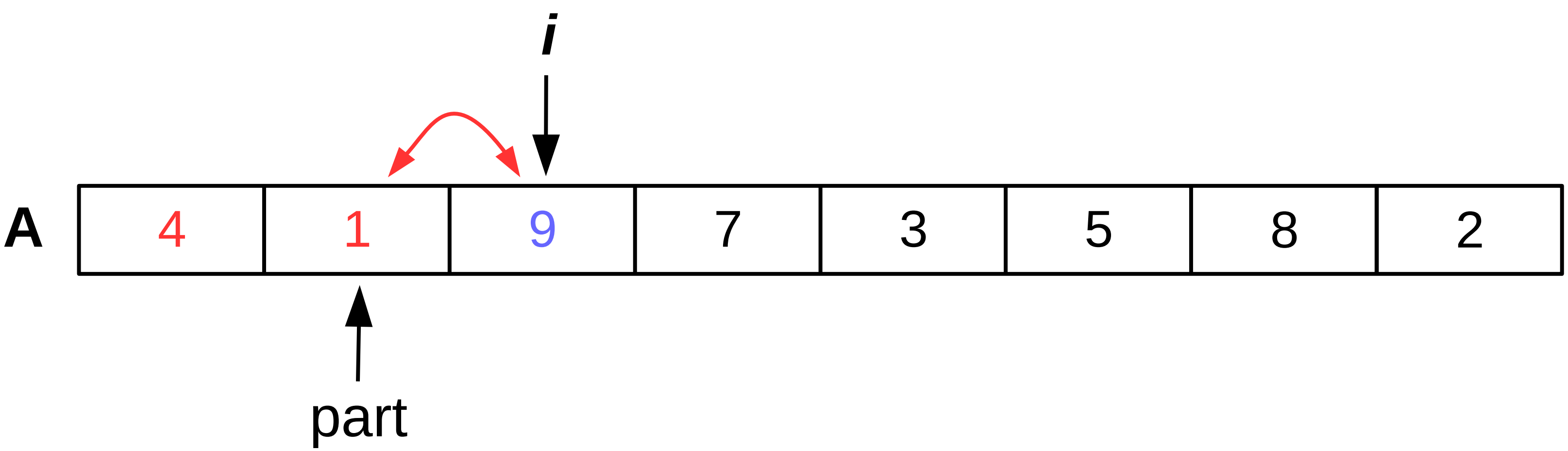
    if $A[i] \leq A[0]$ **then**   TRUE

        swap(A[i], A[part])

        part++

    end

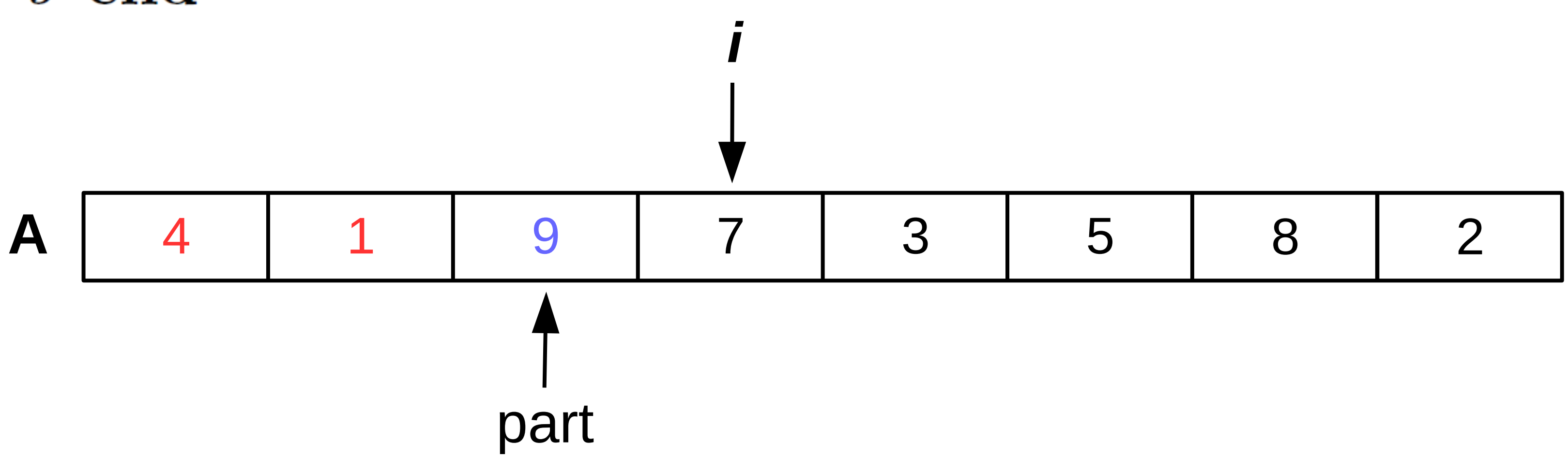
end



Credits: Nodari Sitchinava

Partition elements

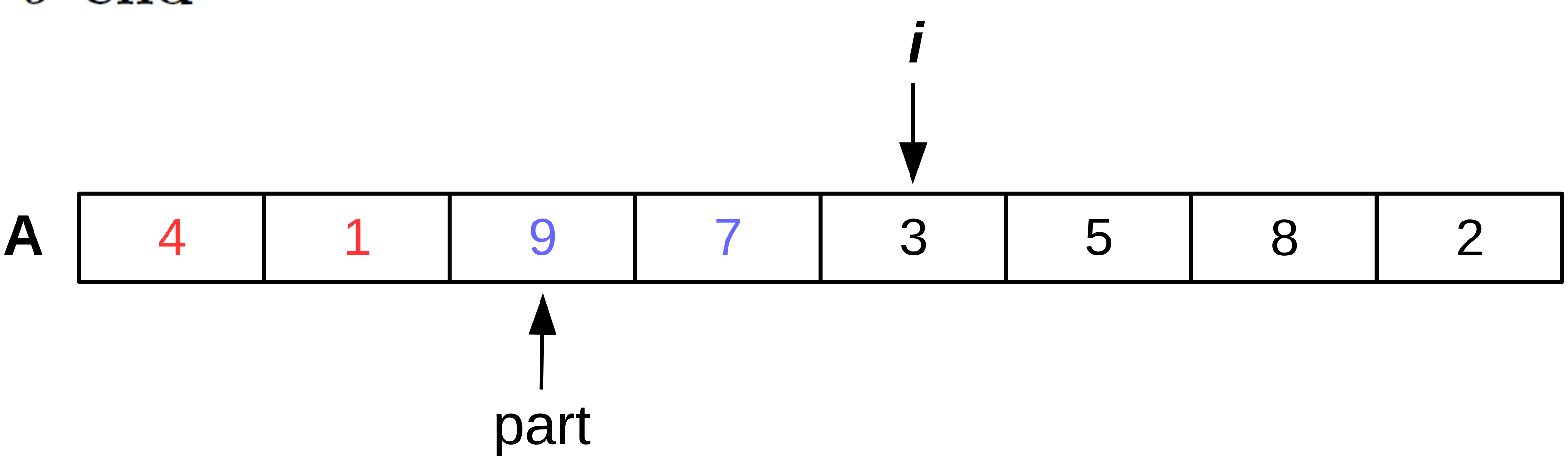
```
part = 1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then        FALSE
        swap(A[i], A[part])
        part++
    end
end
```



Credits: Nodari Sitchinava

Partition elements

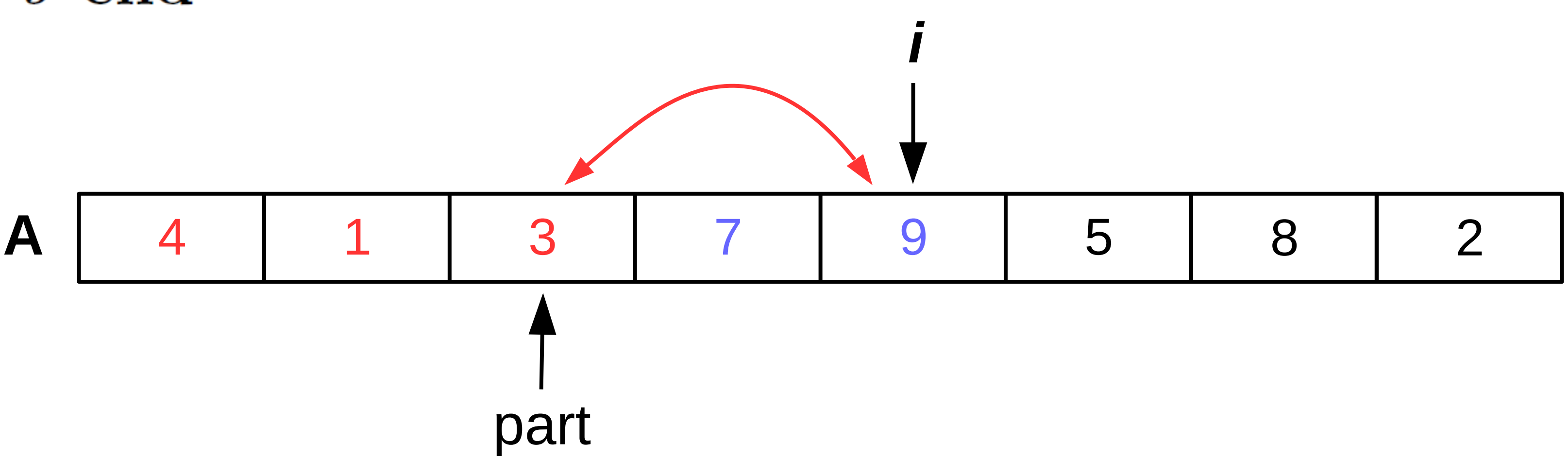
```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then      TRUE
        swap(A[i], A[part])
        part++
    end
end
```



Credits: Nodari Sitchinava

Partition elements

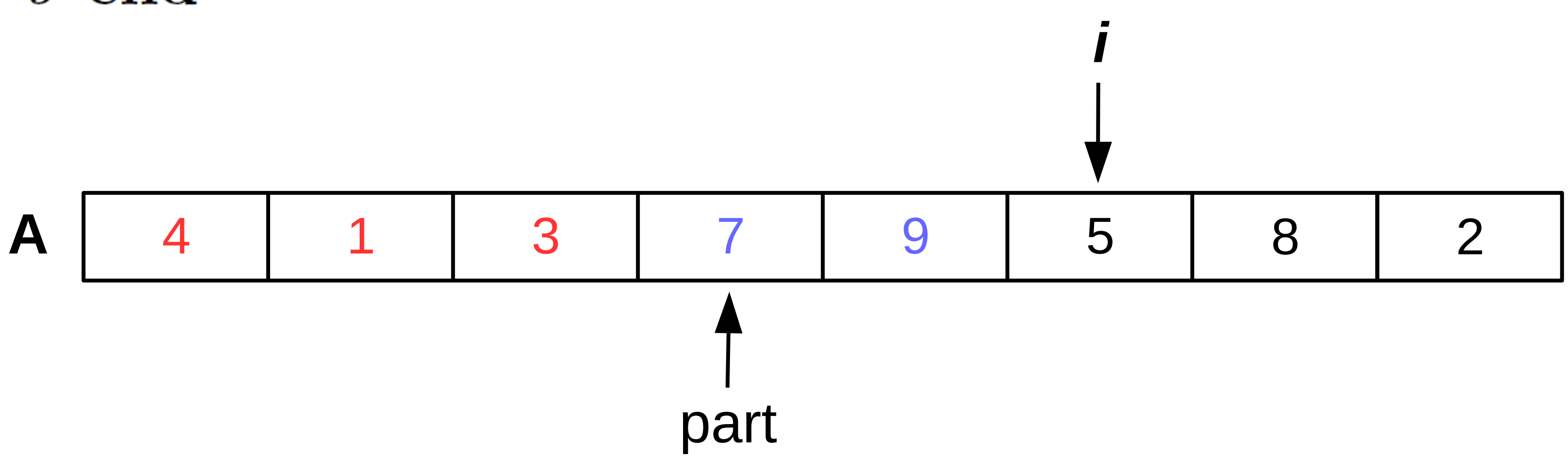
```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then    TRUE
        swap(A[i], A[part])
        part++
    end
end
```



Credits: Nodari Sitchinava

Partition elements

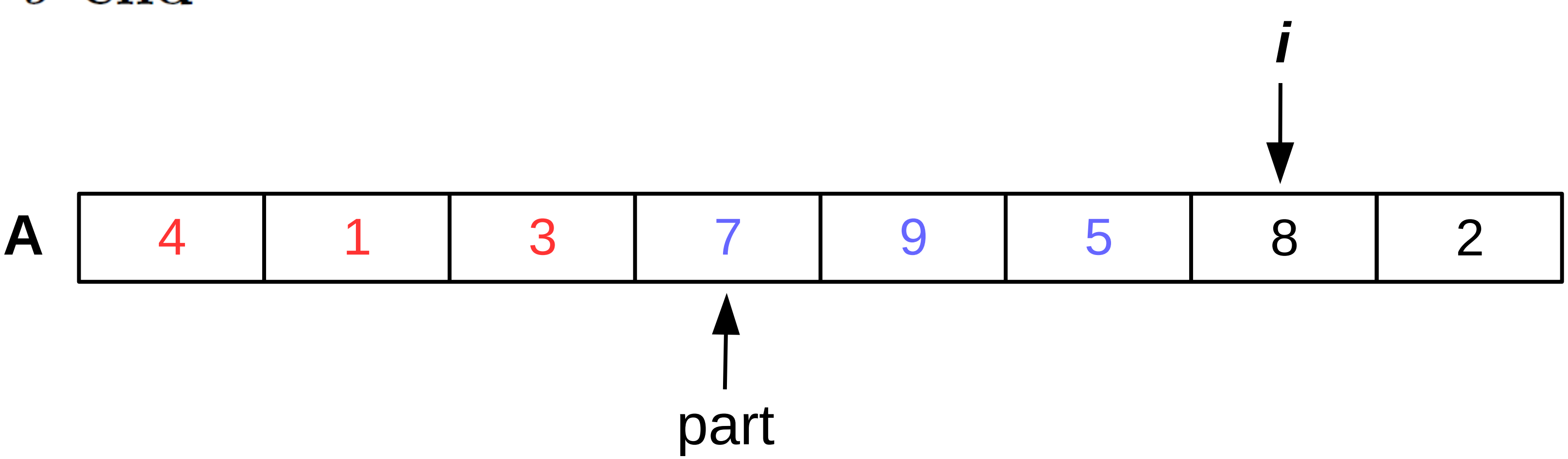
```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then     FALSE
        swap(A[i], A[part])
        part++
    end
end
```



Credits: Nodari Sitchinava

Partition elements

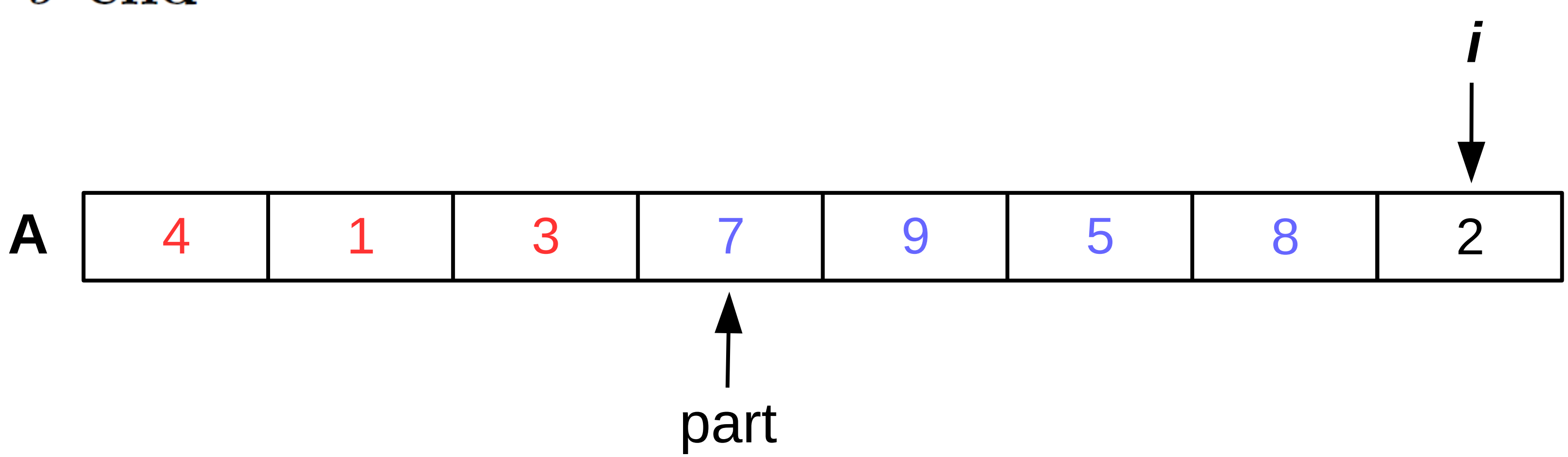
```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then    FALSE
        swap(A[i], A[part])
        part++
    end
end
```



Credits: Nodari Sitchinava

Partition elements

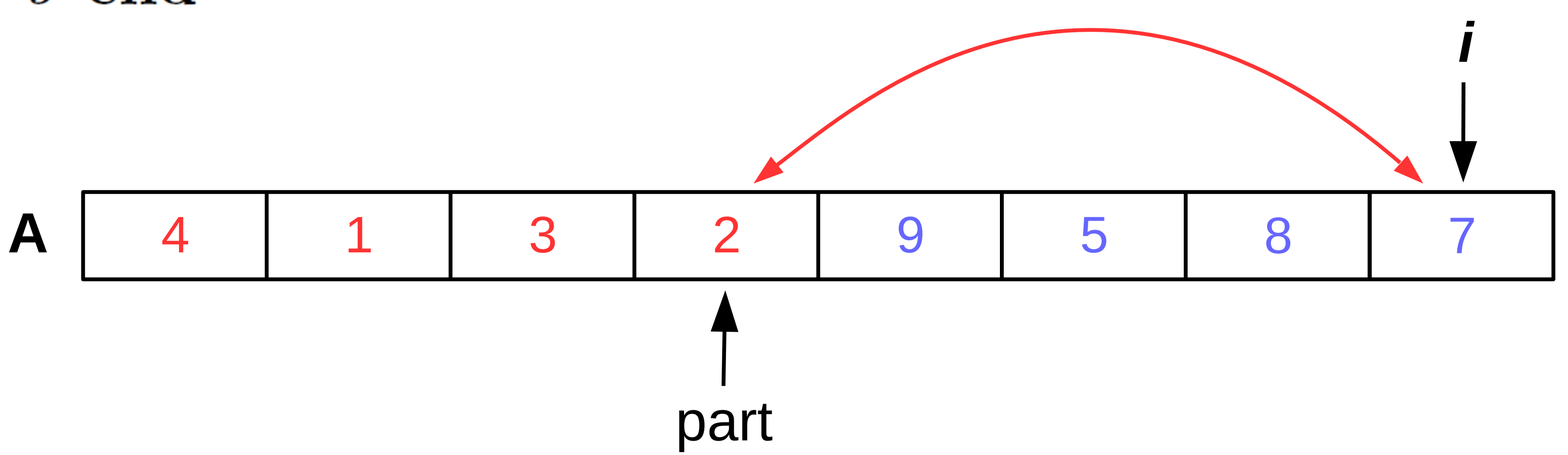
```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then        TRUE
        swap(A[i], A[part])
        part++
    end
end
```



Credits: Nodari Sitchinava

Partition elements

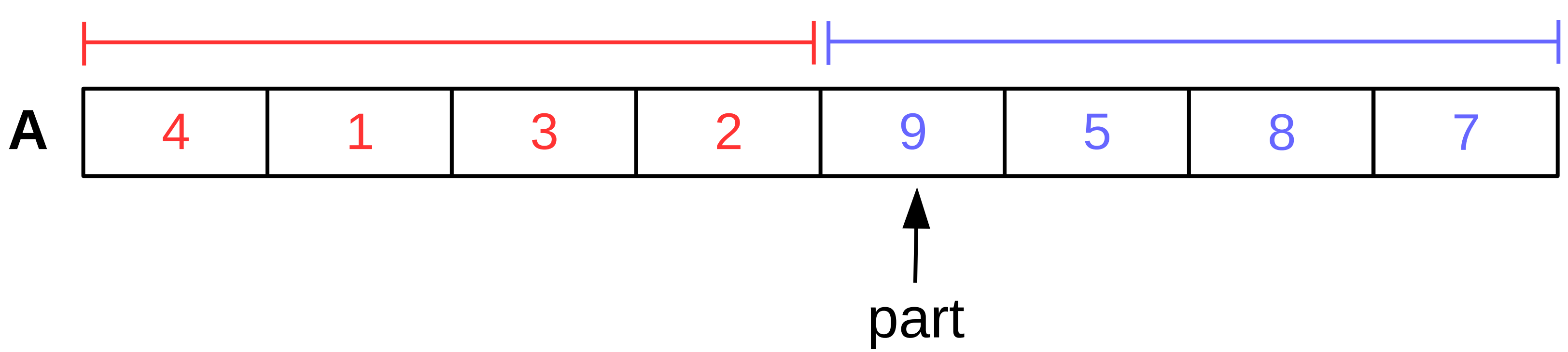
```
part = 1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then        TRUE
        swap(A[i], A[part])
        part++
    end
end
```



Credits: Nodari Sitchinava

Recurse

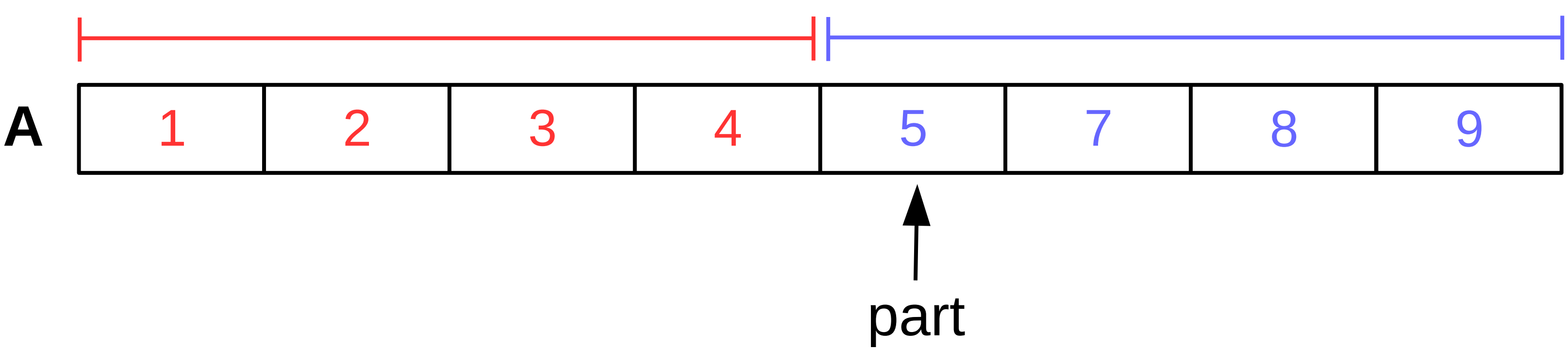
```
if part > 2 then
    Quicksort(A[0,···,part-1])
end
if part < n-1 then
    Quicksort(A[part,···,n-1])
end
```



Credits: Nodari Sitchinava

Recursion sorts sublists

```
if part > 2 then
    Quicksort(A[0,···,part-1])
end
if part < n-1 then
    Quicksort(A[part,···,n-1])
end
```



Credits: Nodari Sitchinava

How can we parallelize?

Quicksort$(A[0, \cdots, n-1])$

```
pivot = random(1 ··· n)
swap(A[0], A[pivot])
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then
        swap(A[i], A[part])
        part++
    end
end
if part > 2 then
    Quicksort(A[0,···,part-1])
end
if part < n-1 then
    Quicksort(A[part,···,n-1])
end
```

O(1)

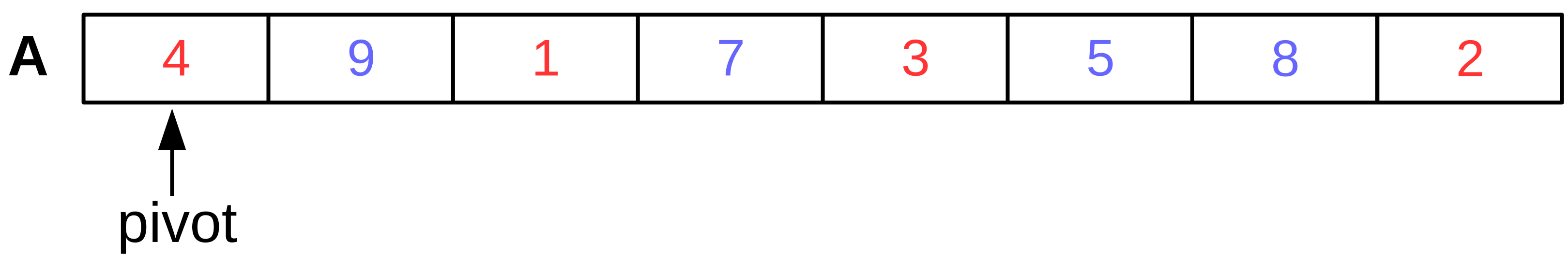
???

Parallel calls

Credits: Nodari Sitchinava

Parallel partition

- **Separate** all elements ≤ pivot

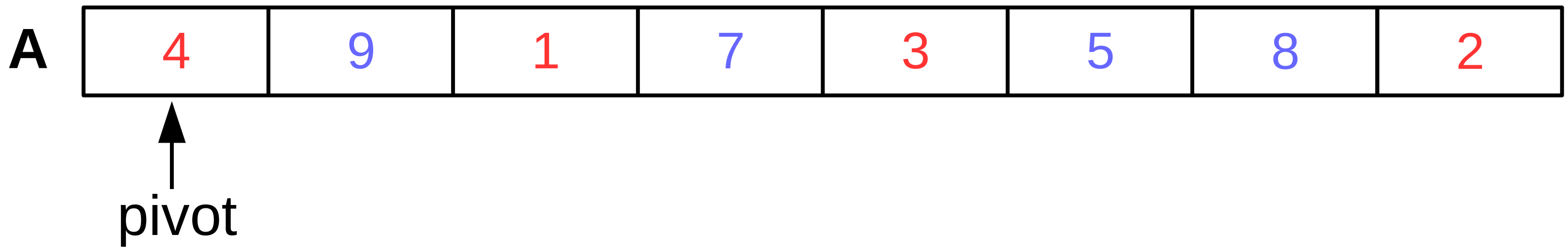


- How can we do this in parallel?

Credits: Nodari Sitchinava

Parallel partition

- **Separate** all elements ≤ pivot



- How can we do this in parallel?
  - Prefix sums!

Credits: Nodari Sitchinava

Parallel partition

- Create **B[i]** by comparing A[i] to pivot
  - 1 if A[i] ≤ A[0]
  - 0 otherwise

A	4	9	1	7	3	5	8	2
B	1	0	1	0	1	0	0	1

Credits: Nodari Sitchinava

Parallel partition

- Prefix sums on B

A	4	9	1	7	3	5	8	2

B	1	1	2	2	3	3	3	4

Credits: Nodari Sitchinava

Parallel partition

- Write each A[i] ≤ A[0] to array **C**
  - C[B[i]] = A[i]

A	4	9	1	7	3	5	8	2
B	1	1	2	2	3	3	3	4
C	4	1	3	2				

Credits: Nodari Sitchinava

Parallel partition

- Create **B‘** as opposite of **B**
  - B‘[i] = 1 if A[i] > A[0]
  - B‘[i] = 0 otherwise

A	4	9	1	7	3	5	8	2

B‘	0	1	0	1	0	1	1	0

C	4	1	3	2				

Credits: Nodari Sitchinava

Parallel partition

- Prefix sums on **B‘**

A	4	9	1	7	3	5	8	2
B‘	0	1	1	2	2	3	4	4
C	4	1	3	2				

Credits: Nodari Sitchinava

Parallel partition

- Write remaining elements to **C**
  - C[B[n-1] + B‘[i]] = A[i]

A	4	9	1	7	3	5	8	2
B‘	0	1	1	2	2	3	4	4
C	4	1	3	2	9	7	5	8

Credits: Nodari Sitchinava

Parallel quicksort analysis

- Each recursive call performs prefix sum
- Worst-case, pivot is always min or max:

$$W(n) = W(n-1) + O(n) = O(n^2)$$

- If we assume “good“ pivot is chosen:

$$W(n) = W(\frac{n}{2}) + O(n) = O(n \log n)$$

Credits: Nodari Sitchinava

Parallel quicksort analysis

- Assuming a “good“ pivot choice:

$$T(n) = T(\frac{n}{2}) + O(\log n)$$

$$\begin{aligned} &= \log n + \log \frac{n}{2} + \cdots + \frac{n}{n} \\ &= \log n) + (\log n - 1) + (\log n - 2) + \cdots + 1 \\ &= \frac{(\log n)(\log n + 1)}{2} = O(\log^2 n) \end{aligned}$$

Credits: Nodari Sitchinava

Issues with parallel quicksort

- Have to copy **A** to **C** => **not** in-place
  - O(n) extra space needed
- $O(\log^2 n)$ “average“ parallel runtime
- Recursive definition
  - Difficult to make iterative
  - Perform **many** small prefix-sums
    - Performance overhead

Credits: Nodari Sitchinava

Iterative solution

- What if we can combine recursive calls
  - One iteration for each level

- Separate recursive calls on partitions:

	P_0				P_1			P_2					P_3			
A	1	3	5	2	7	6	9	12	16	19	17	14	22	25	20	23

Credits: Nodari Sitchinava

Iterative solution

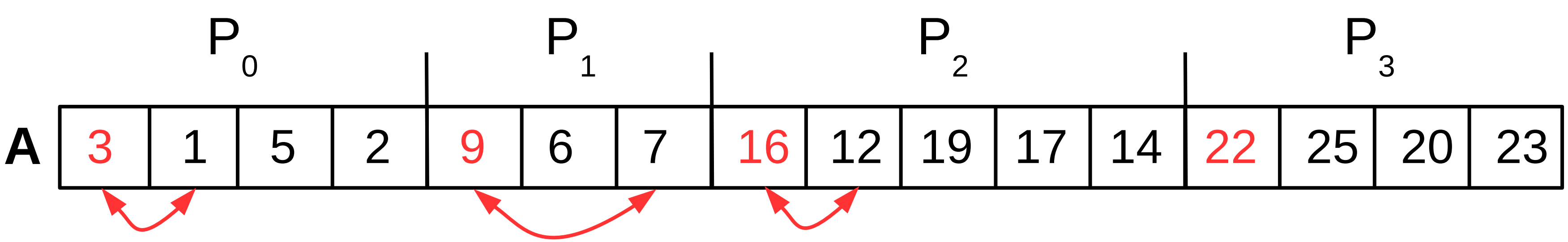
- Know size of partition i = $|P_i|$
- Find a pivot for each partition

	P_0				P_1			P_2					P_3			
A	1	3	5	2	7	6	9	12	16	19	17	14	22	25	20	23

Credits: Nodari Sitchinava

Iterative solution

- Know size of partition i = $|P_i|$
- Find a pivot for each partition
  - Move pivots to front



Credits: Nodari Sitchinava

Iterative solution

- Know size of partition i = $|P_i|$
- Find a pivot for each partition
  - Move pivots to front
- Compute **B**
  - Compare each to the pivot in its partition

	P_0				P_1			P_2					P_3			
A	3	1	5	2	9	6	7	16	12	19	17	14	22	25	20	23
B	1	1	0	1	1	1	1	1	1	0	0	1	1	0	1	0

Credits: Nodari Sitchinava

Iterative solution

- Want prefix sum **within** each partition:
- ***Segmented prefix sums***
  - Each partition is a separate **segment**

	P_0				P_1			P_2					P_3			
A	3	1	5	2	9	6	7	16	12	19	17	14	22	25	20	23
B	1	1	0	1	1	1	1	1	1	0	0	1	1	0	1	0

Credits: Nodari Sitchinava

Iterative solution

- Want prefix sum **within** each partition:
- ***Segmented prefix sums***
  - Each partition is a separate **segment**
  - Can combine into 1 operation...

	P_0				P_1			P_2					P_3			
A	3	1	5	2	9	6	7	16	12	19	17	14	22	25	20	23
B	1	2	2	3	1	2	3	1	2	2	2	3	1	1	2	2

Credits: Nodari Sitchinava

Segmented prefix sums

- Input array **A** and *flag bits* **F**
  - 1 if start of new segment
  - 0 otherwise
- Prefix sums, except sum **resets** when F[i]=1

A	3	1	4	1	5	2	1	3	4	0	2	6	1	0	3	4
F	0	0	0	1	0	0	0	0	0	1	0	1	1	0	0	0

Credits: Nodari Sitchinava

Segmented prefix sums

- Input array **A** and *flag bits* **F**
  - 1 if start of new segment
  - 0 otherwise
- Prefix sums, except sum **resets** when F[i]=1

A	3	1	4	1	5	2	1	3	4	0	2	6	1	0	3	4
F	0	0	0	1	0	0	0	0	0	1	0	1	1	0	0	0
C	3	4	8	1	6	8	9	12	16	0	2	6	1	1	4	8

Credits: Nodari Sitchinava

Partition with segments

- Create **F** with partition boundaries

	P_0				P_1			P_2					P_3			
A	3	1	5	2	9	6	7	16	12	19	17	14	22	25	20	23
B	1	1	0	1	1	1	1	1	1	0	0	1	1	0	1	0
F	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0

Credits: Nodari Sitchinava

Partition with segments

- Create **F** with partition boundaries
- Perform ***segmented prefix sums*** on **B** and **F**

	P_0				P_1			P_2					P_3			
A	3	1	5	2	9	6	7	16	12	19	17	14	22	25	20	23
B	1	2	2	3	1	2	3	1	2	2	2	3	1	1	2	2
F	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0

Credits: Nodari Sitchinava

Partition with segments

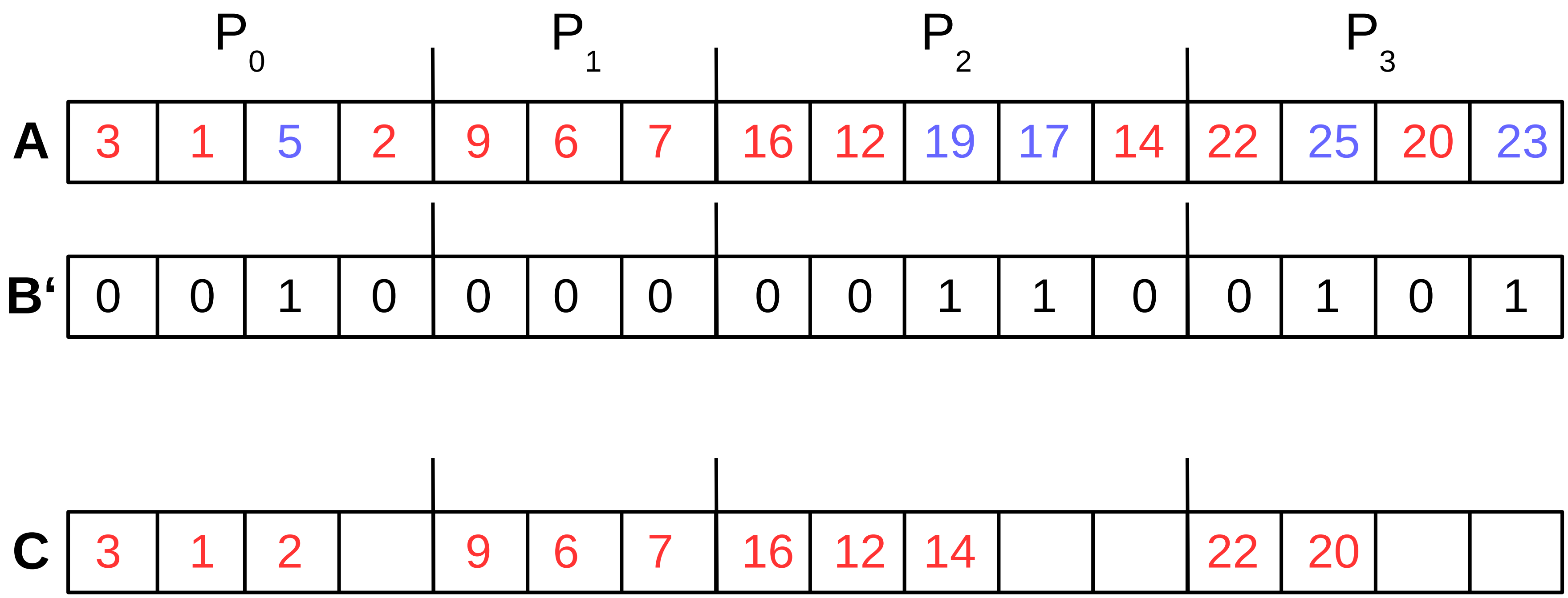
- Create **F** with partition boundaries
- Perform ***segmented prefix sums*** on **B** and **F**
- Copy **A[i]** into **C[B[i]]** (plus partition offsets)

	P_0				P_1			P_2					P_3			
A	3	1	5	2	9	6	7	16	12	19	17	14	22	25	20	23
B	1	2	2	3	1	2	3	1	2	2	2	3	1	1	2	2
C	3	1	2		9	6	7	16	12	14			22	20		

Credits: Nodari Sitchinava

Partition with segments

- Repeat for > pivots:
  - Build **B'**



Credits: Nodari Sitchinava

Partition with segments

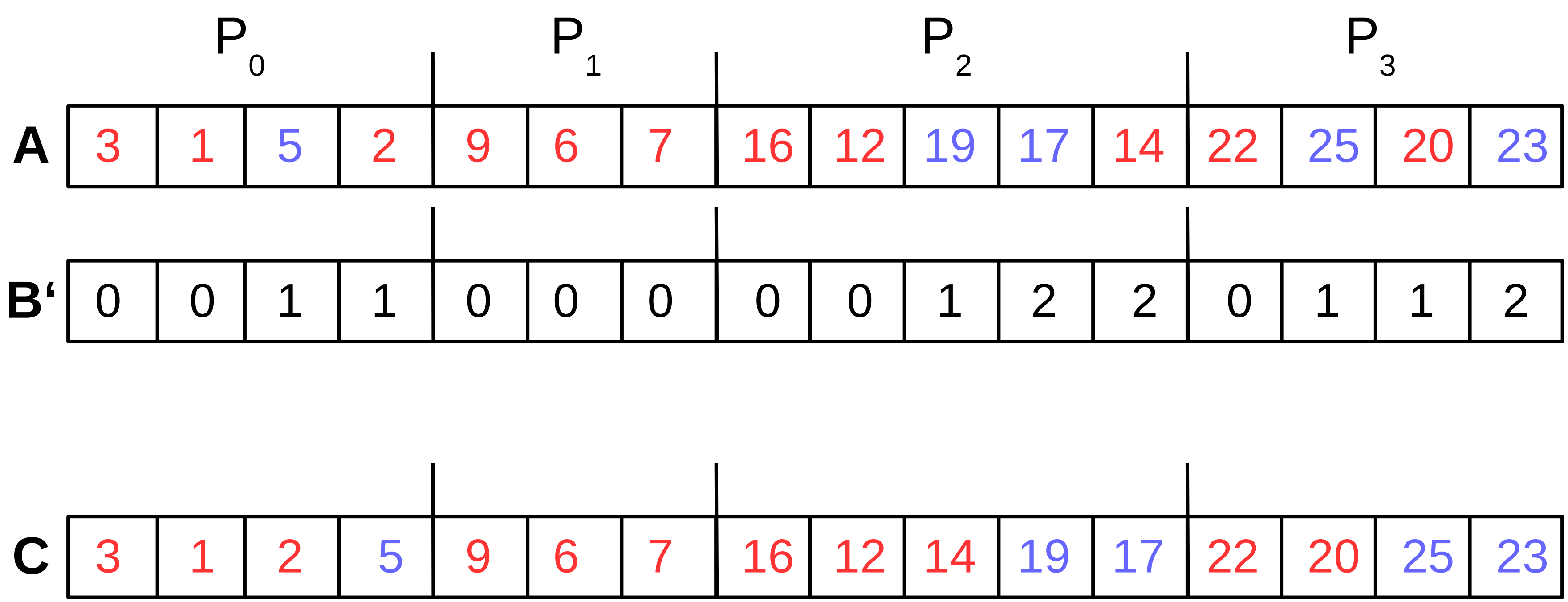
- Repeat for > pivots:
  - ***Segmented prefix sums*** on **B‘**

	P_0				P_1			P_2					P_3			
A	3	1	5	2	9	6	7	16	12	19	17	14	22	25	20	23
B‘	0	0	1	1	0	0	0	0	0	1	2	2	0	1	1	2
C	3	1	2		9	6	7	16	12	14			22	20		

Credits: Nodari Sitchinava

Partition with segments

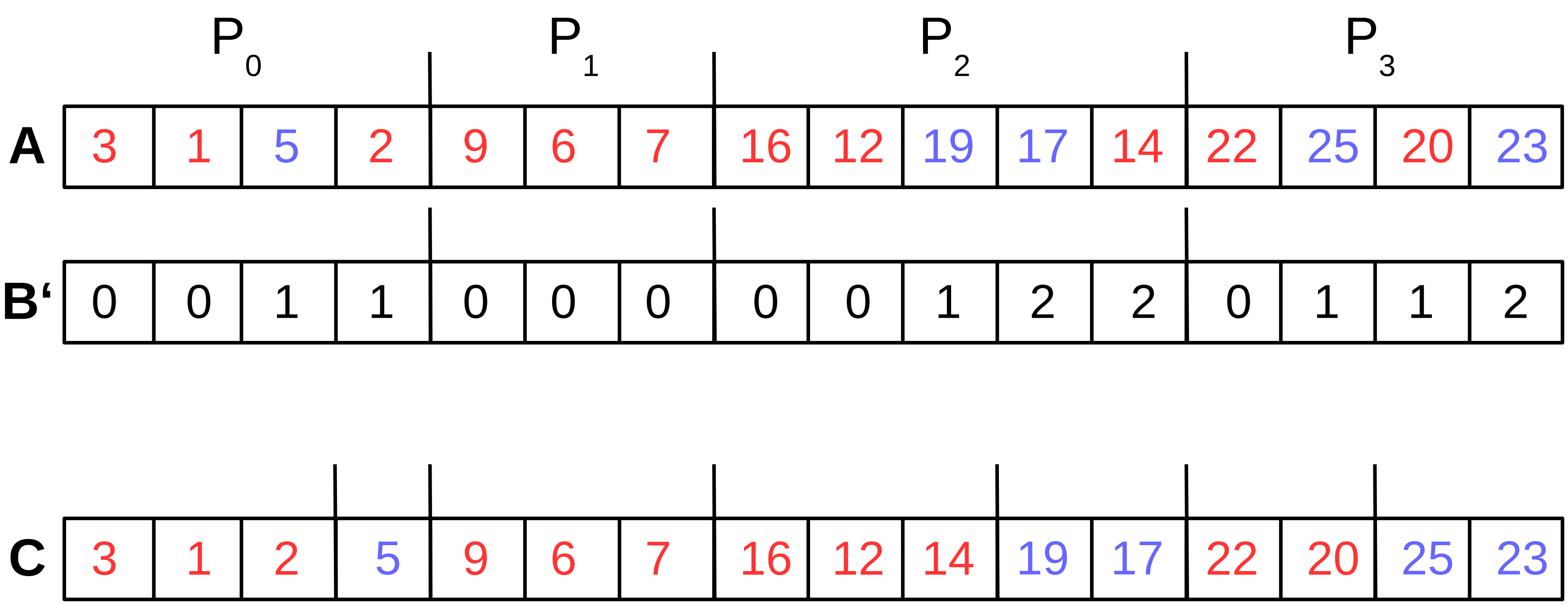
- Repeat for > pivots:
  - Copy remaining **A** values into **C**



Credits: Nodari Sitchinava

Partition with segments

- Ready for next iteration...



Credits: Nodari Sitchinava