Long-Lived Counters with Polylogarithmic Amortized Step Complexity

Alessia Milani, LaBRI

Joint work with A. M. Baig, D. Hendler and C. Travers
[DISC 2019]

Distributed computing

- *Distributed computing* : several computing entities (processes) *cooperate* to achieve a common goal.
- *Cooperate* requires processes to *communicate.*
- *Shared objects* provide a consistent interface for multiple processes to communicate.

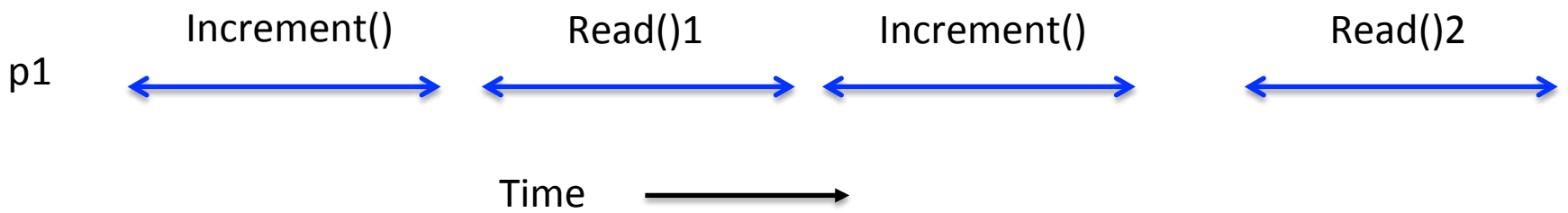
Linearizable Shared Objects

- Object type: finite set of operations and a sequential specification to describe the correct behavior.
- Several (sequential) processes perform concurrent operations on the object.
- An object implementation usually provides the illusion that these operations are applied sequentially (one after the other)
  - Linearizability [Herlihy and Wing, TOPLAS'90]

Counter Object Type

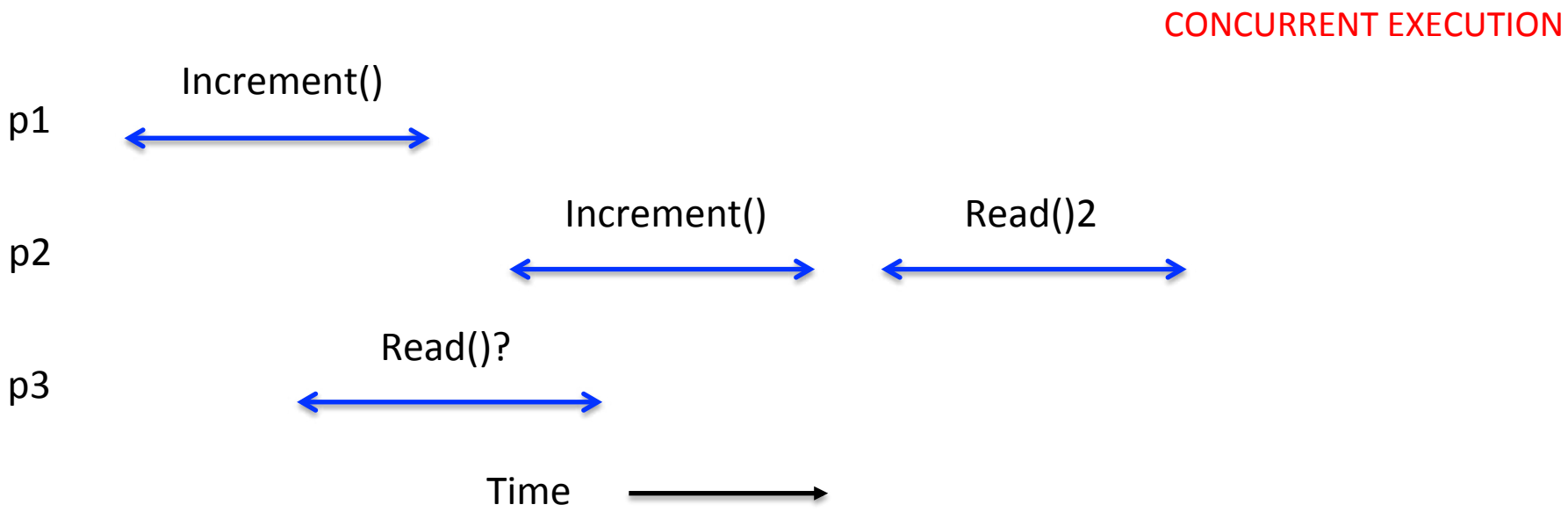
- Supports *Increment* and *Read* operations
- Sequential specification
  - An Increment increases the counter's value by 1 (initially 0)
  - A *Read* returns the number of previous *Increment* operations

SEQUENTIAL EXECUTION

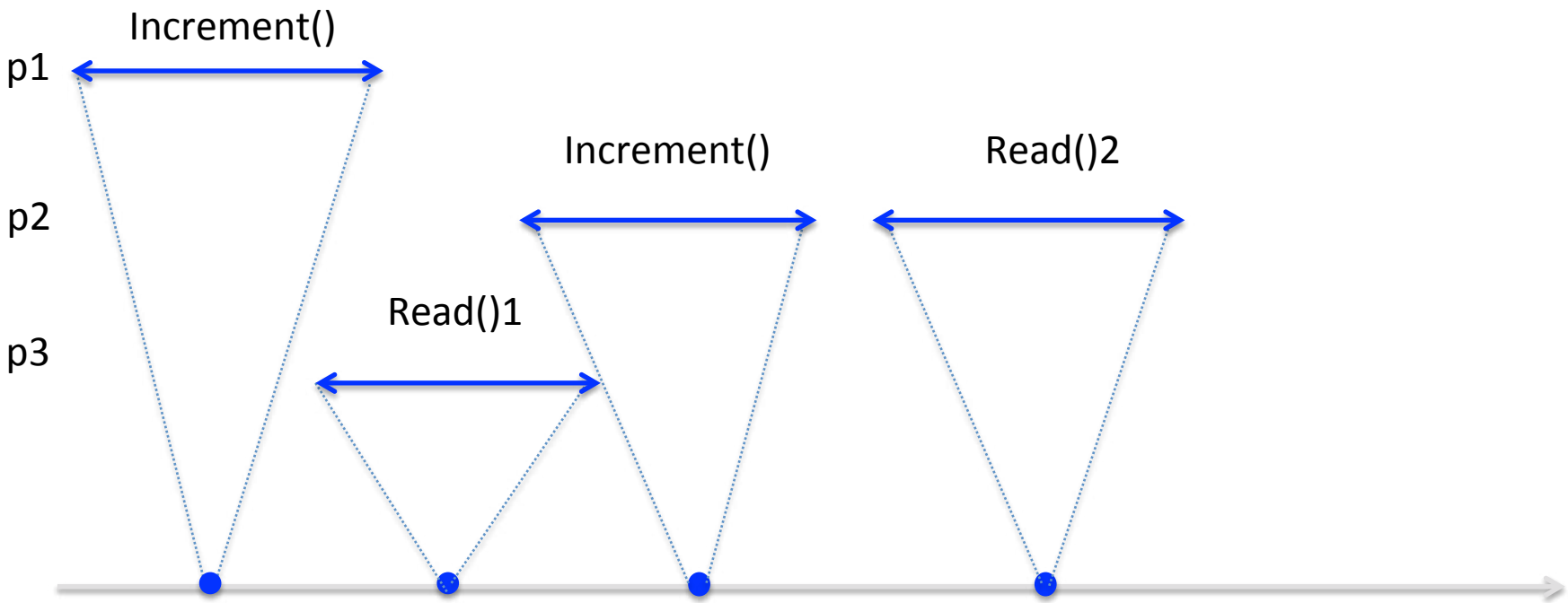


Shared Counter object

- *Increment* and *Read* operations are performed concurrently
  - A *Read* returns the number of *previous* *Increment* operations



Linearizable Counter



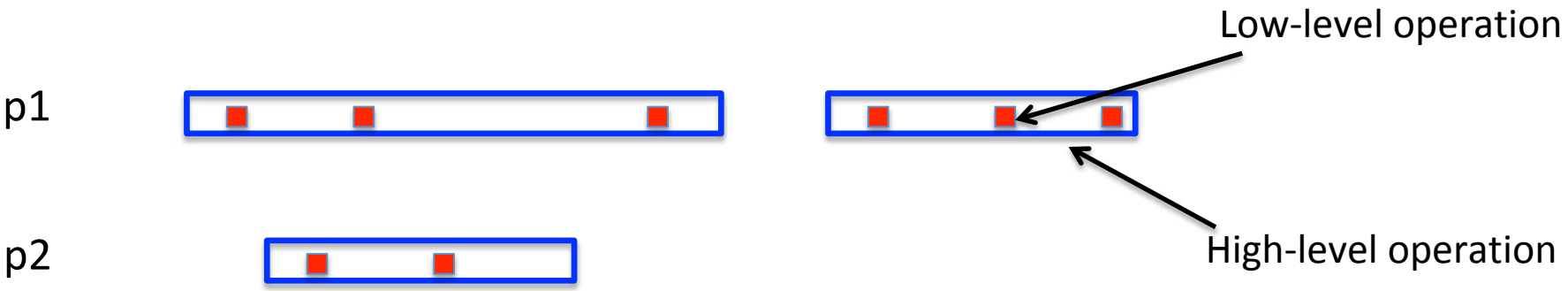
- Every concurrent execution H is equivalent to a sequential execution S that respects
  - the sequential specification of the object, and
  - the real time order of operations in H

We study the complexity of

- Linearizable wait-free shared counter implementations
  - Wait-free : all high-level operations finish in a finite number of low-level operations for any interleaving
- in a standard shared-memory model

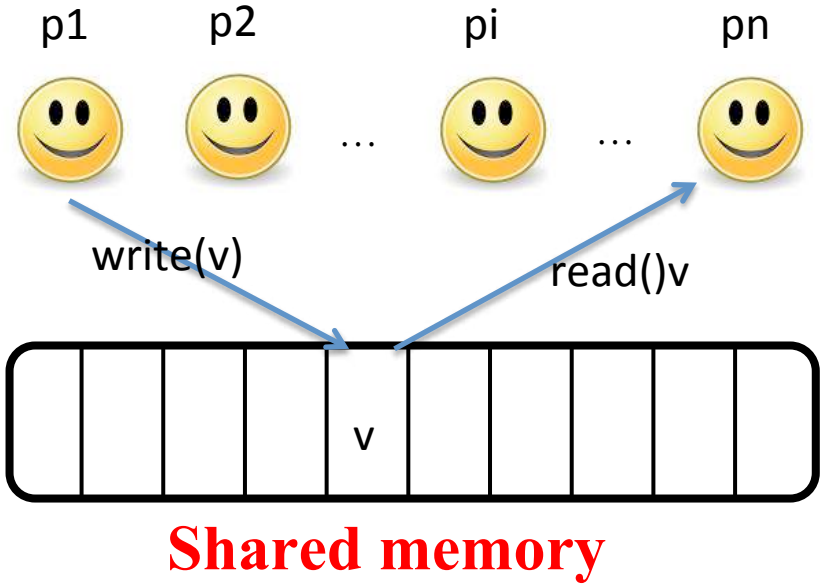
A Counter Implementation

- Provides the internal representation of the counter state
  - using other low-level shared objects
- And the algorithms which processes have to follow to perform (high-level) *Increment* and *Read* operations.



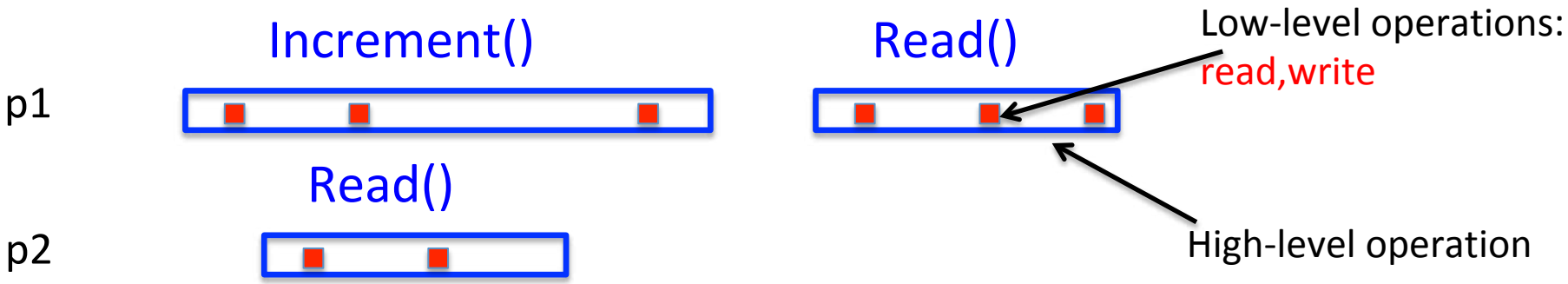
Model

- **n** asynchronous deterministic processes, unique IDs, may fail-stop
- Processes communicate using *read* and *write* operations on shared memory locations (called registers)
  - A write stores a new value in the register
  - A read returns the last value written



Implementation Complexity

The complexity of a high-level operation is the number of low-level operations (called steps) applied to perform it.



Counter Complexity Bounds

- O(n) steps counter — Inoue et al., IDWA '94
- Worst case complexity is in Ω(n) — Jayanti, Tan and Toueg, PODC '96
- Worst case complexity can be polylogarithmic in short executions! — Aspnes, Attiya and Censor-Hillel, JACM '12
  - Increment operations are O(min(logn logv,n))
  - Read operations are O(min(logv,n))
  - Where n is the number of processes and v is the counter's current value.
- Amortized step complexity is in Ω(n) in all known algorithms (for executions of arbitrary length)

Can we do better?

Our contribution

[Baig,Hendler, Milani,Travers-DISC 2019]

- We present a wait-free read/write counter algorithm with $O(\log^2 n)$ amortized step complexity
  - Amortized step complexity : worst case average number of accesses to low-level objects performed by high-level operations
- First wait-free counter with sub-linear amortized step complexity in executions of arbitrary length

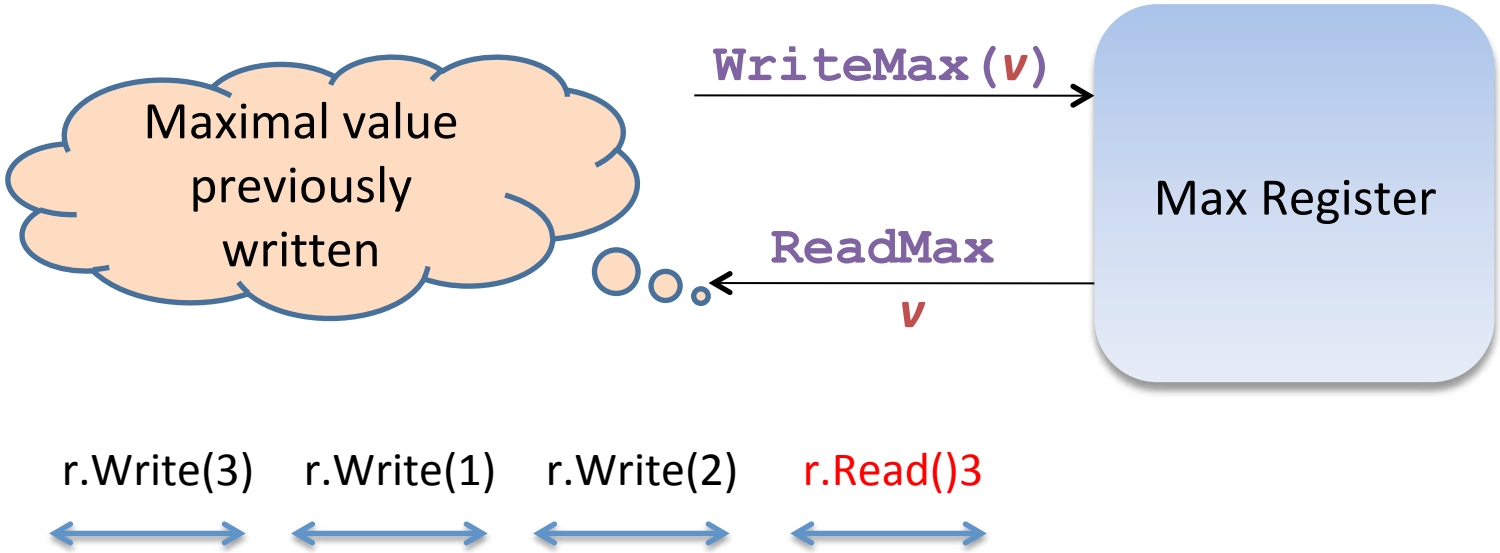
Our Linearizable Wait-free Counter

- Builds on the techniques used by Aspnes et al. to obtain a counter with polylogarithmic worst case complexity in short executions.
- Is based on a novel unbounded wait-free *max register* with logarithmic amortized step-complexity (under an assumption specified soon)

Max-register

Aspnes, Attiya and Censor-Hillel, JACM '12

- A max register r differs from a register since a read operation returns the maximal value written into the register (and not the last)

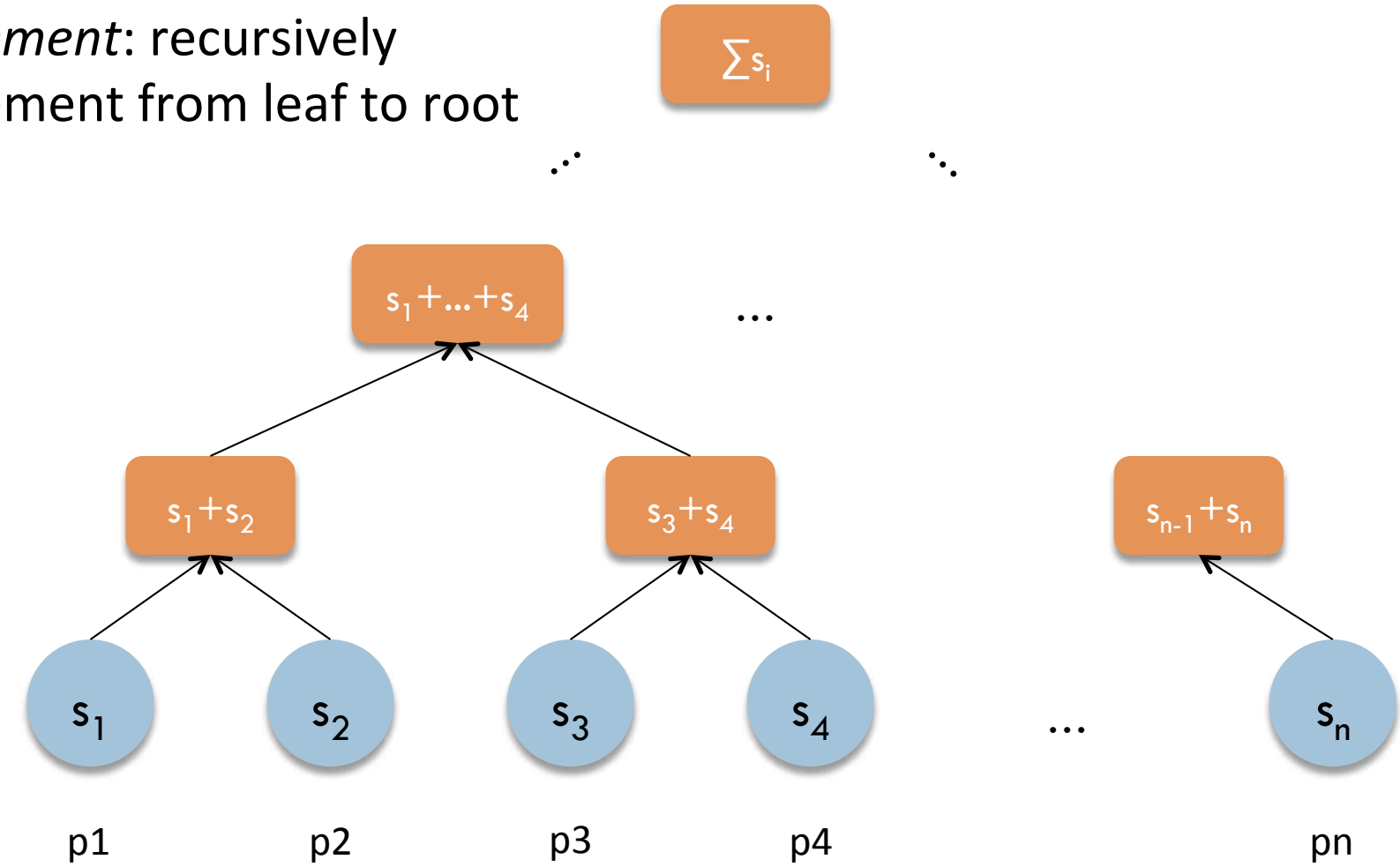


- Both WriteMax and ReadMax of value v take **O(min(log v, n))** steps (n number of processes)

Max-register-based Counter

Aspnes, Attiya and Censor-Hillel, JACM '12

Increment: recursively increment from leaf to root

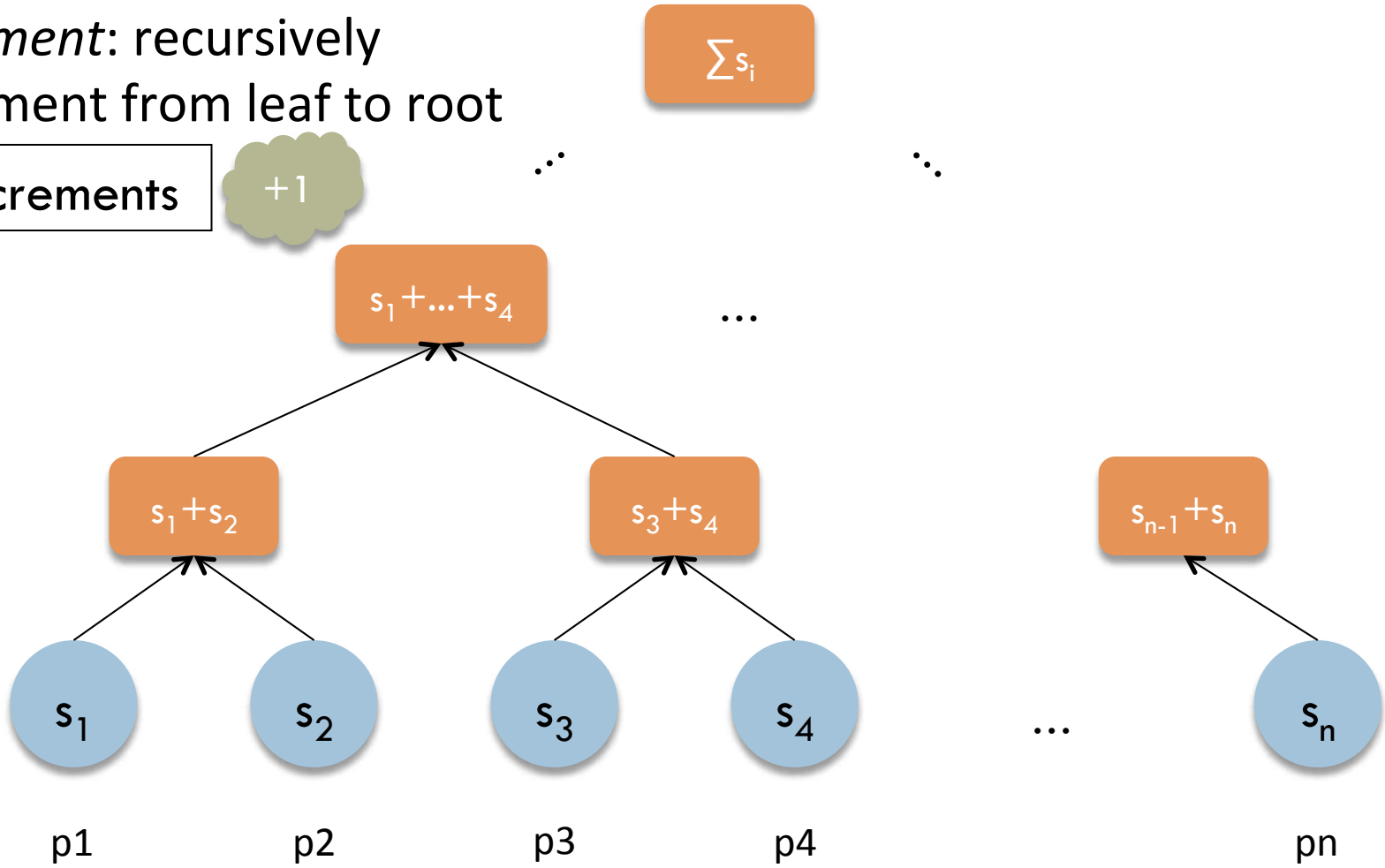


Max-register-based Counter

Aspnes, Attiya and Censor-Hillel, JACM '12

Increment: recursively increment from leaf to root

p_1 increments

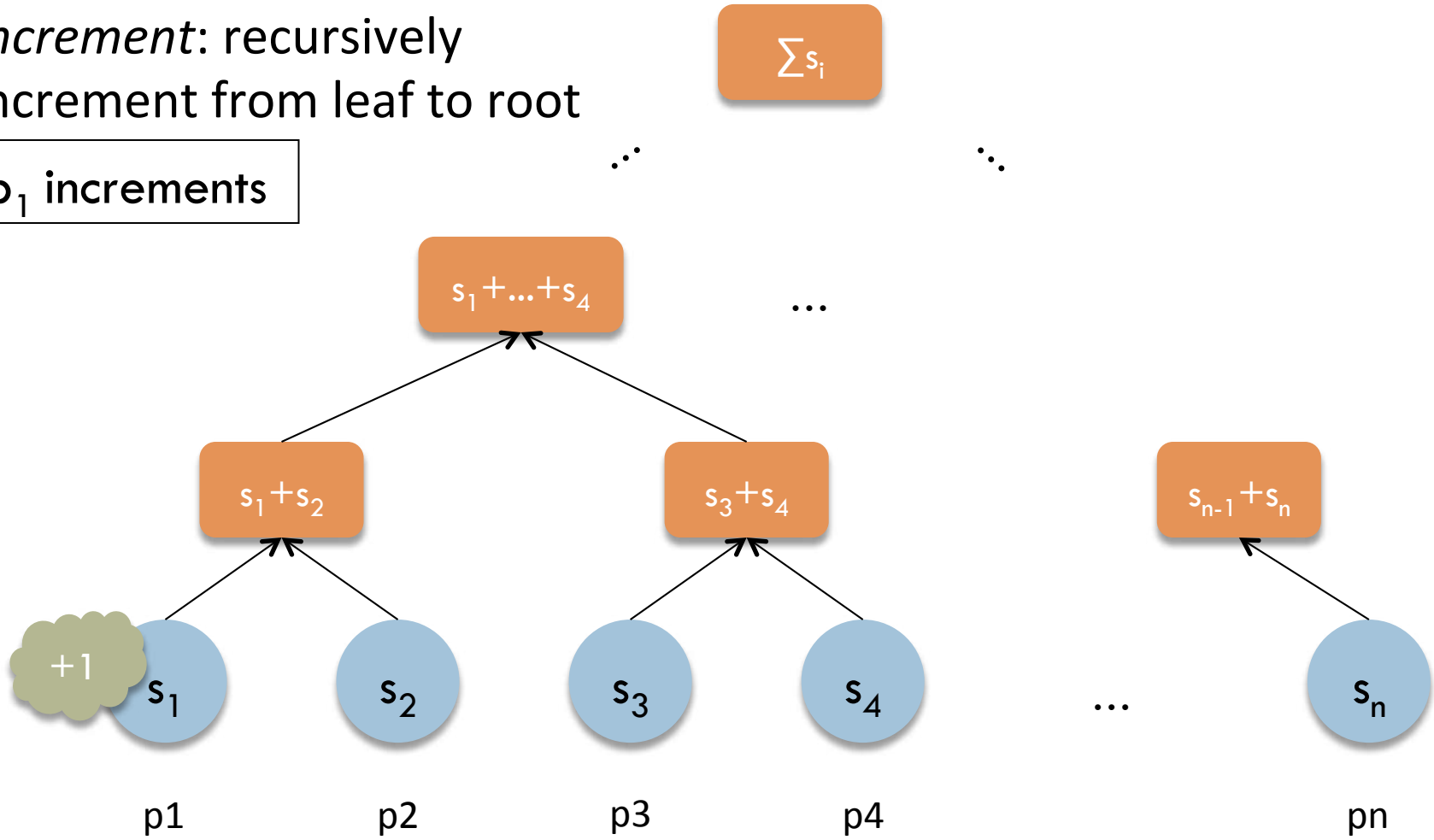


Max-register-based Counter

Aspnes, Attiya and Censor-Hillel, JACM '12

Increment: recursively increment from leaf to root

p_1 increments

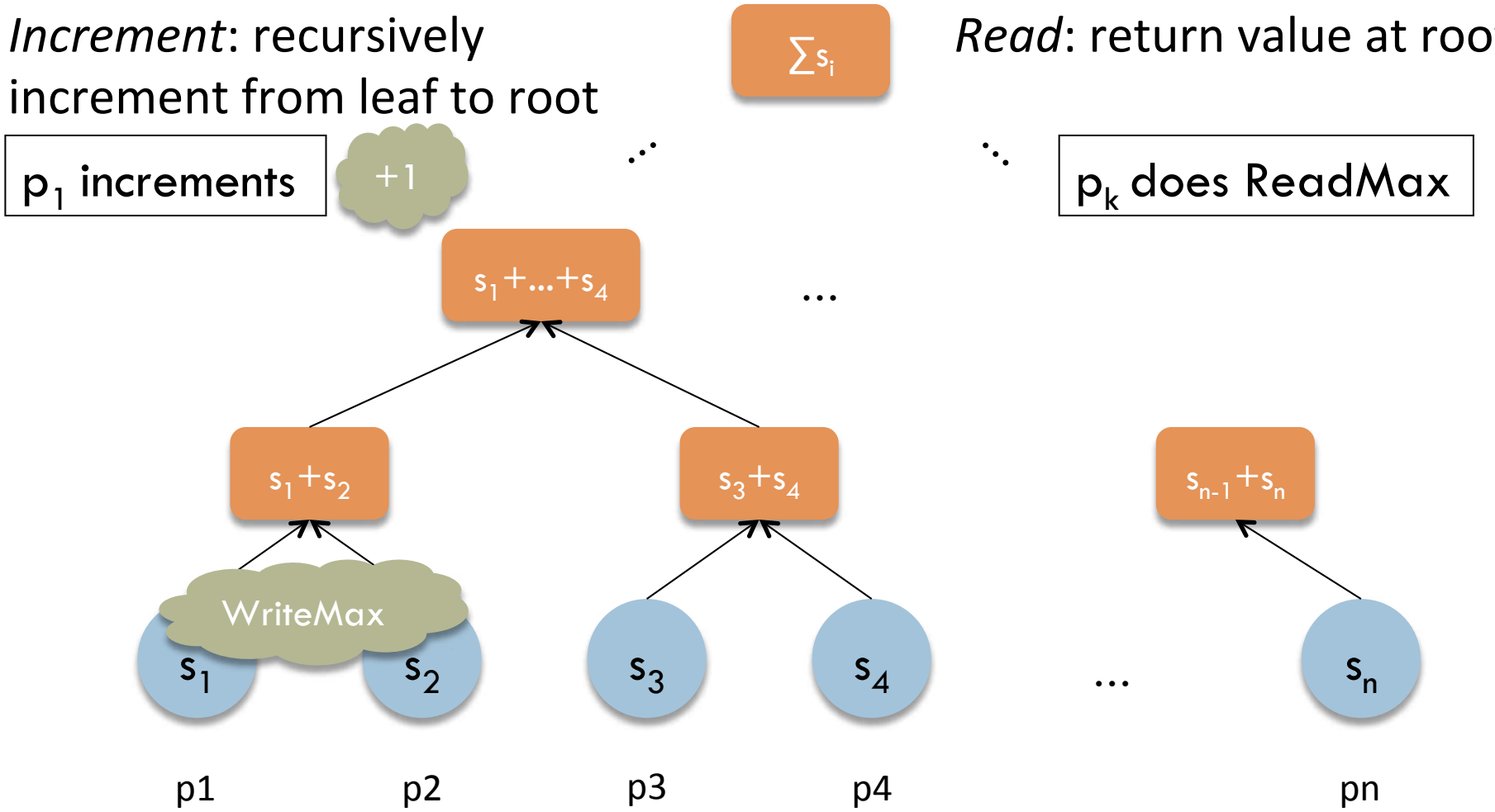


Max-register-based Counter

Aspnes, Attiya and Censor-Hillel, JACM '12

Increment: recursively increment from leaf to root

Read: return value at root

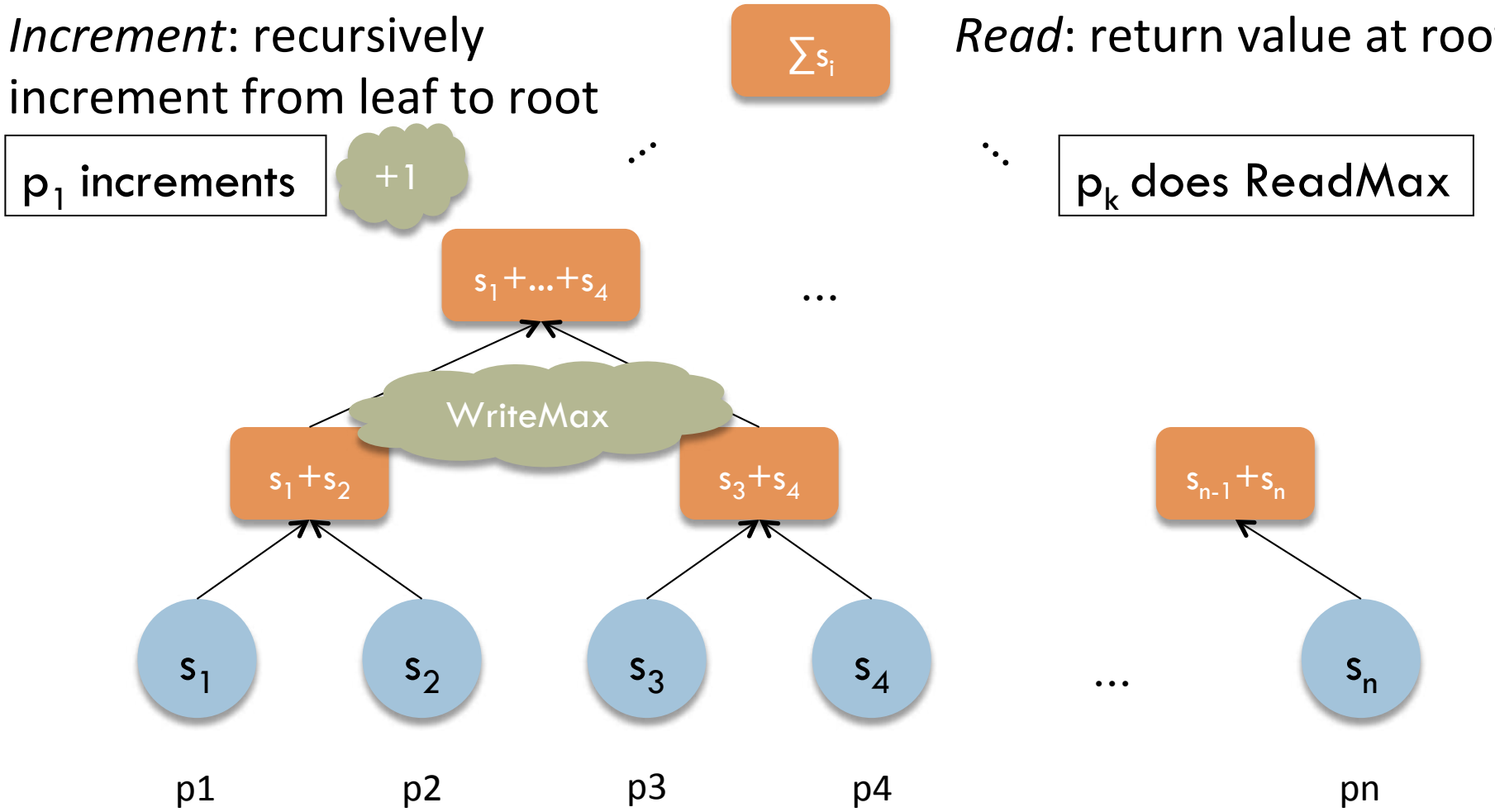


Max-register-based Counter

Aspnes, Attiya and Censor-Hillel, JACM '12

Increment: recursively increment from leaf to root

Read: return value at root

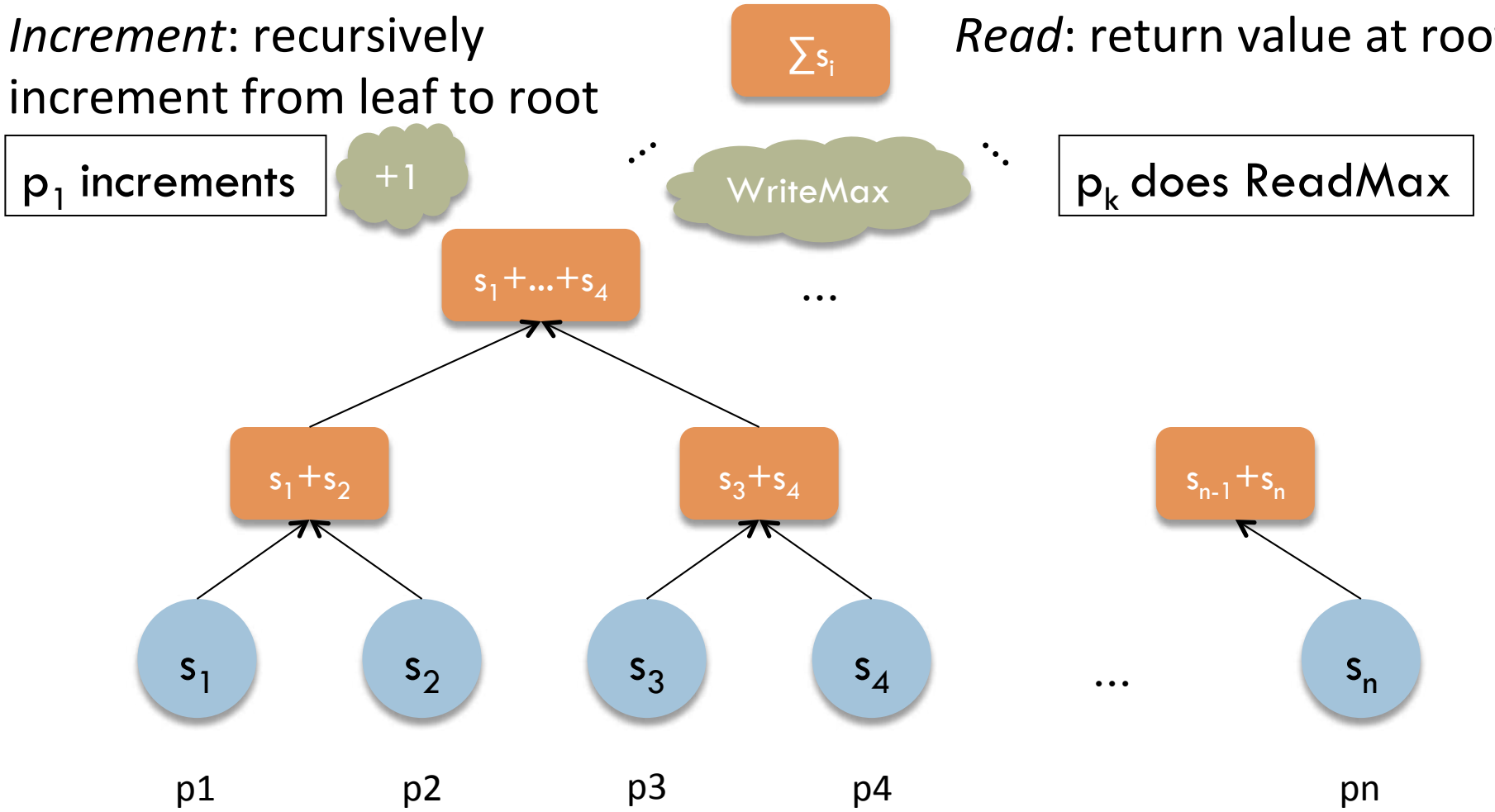


Max-register-based Counter

Aspnes, Attiya and Censor-Hillel, JACM '12

Increment: recursively increment from leaf to root

Read: return value at root

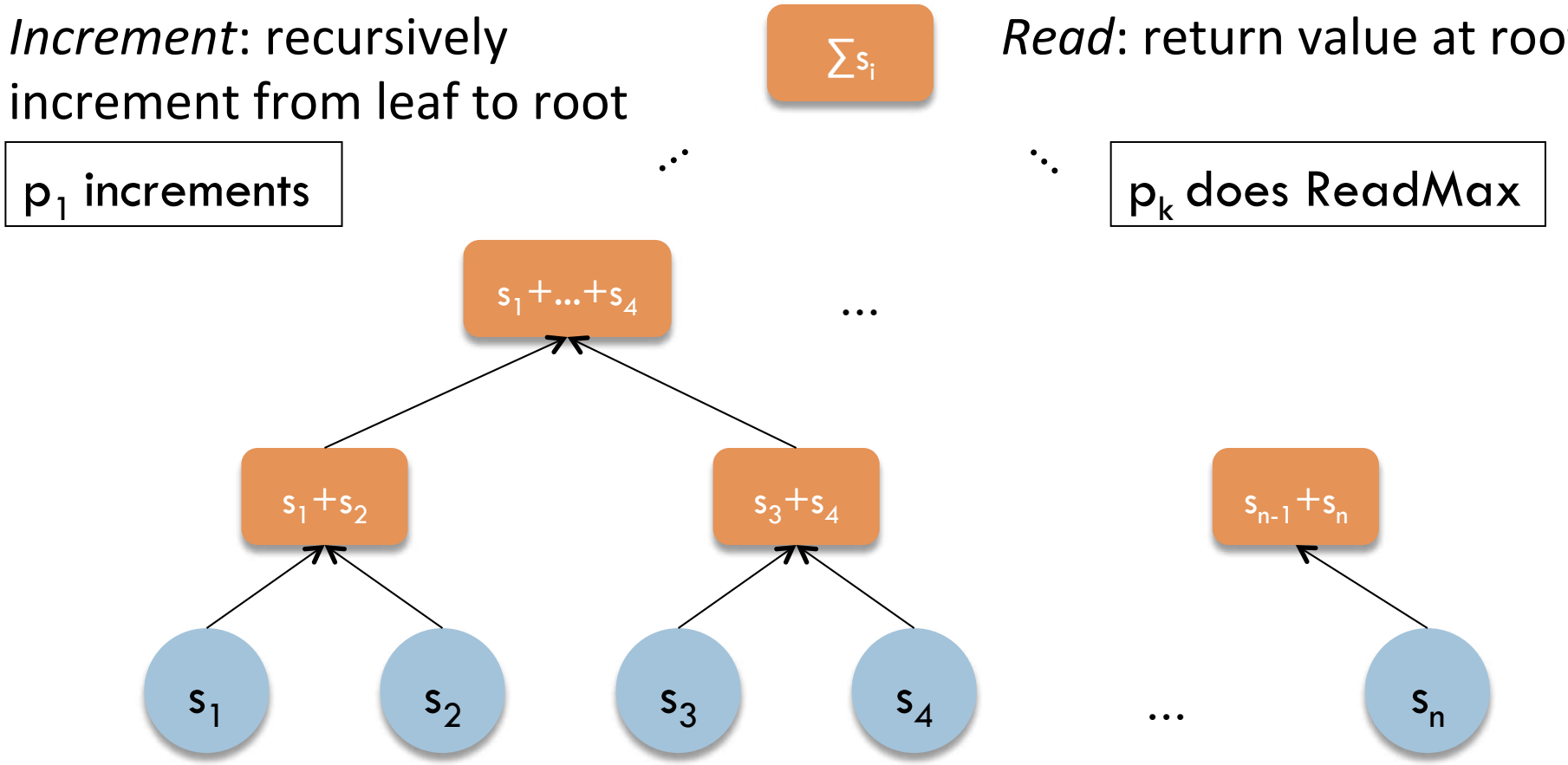


Max-register-based Counter

Aspnes, Attiya and Censor-Hillel, JACM '12

Increment: recursively increment from leaf to root

Read: return value at root



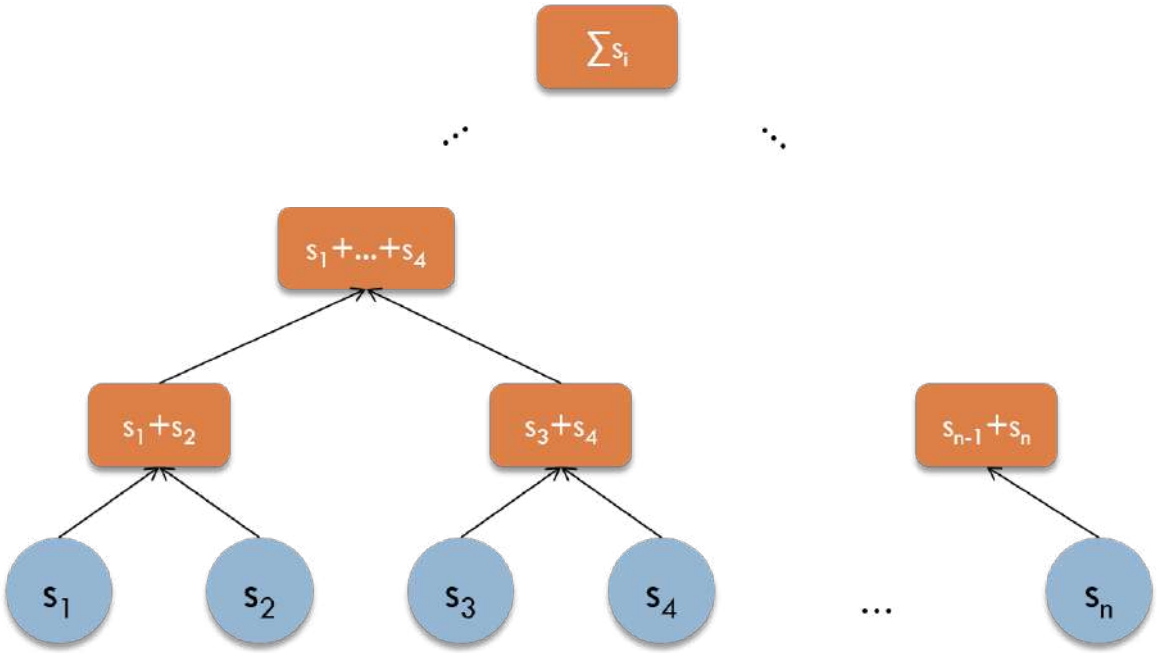
Increment : O(min(logn logv,n))

Read : O(min(logv,n))

Talk Outline

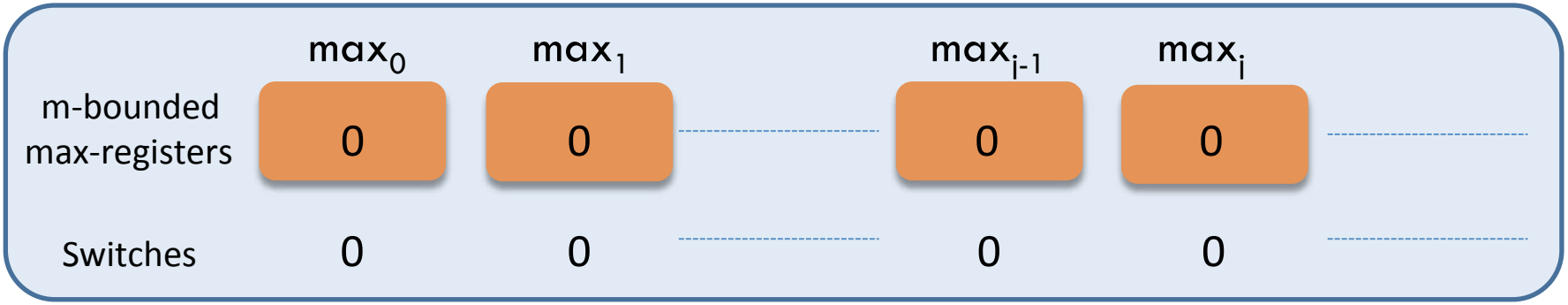
- Preliminaries
- The new algorithm
- Discussion

An observation: n-bounded increments



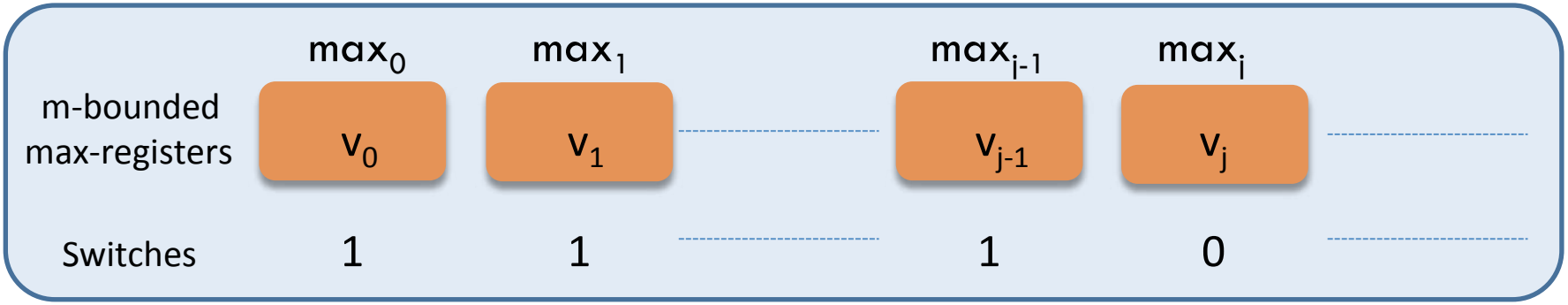
The value of any max register in the tree is never increased by more than *n*

A unbounded lock-free Max register



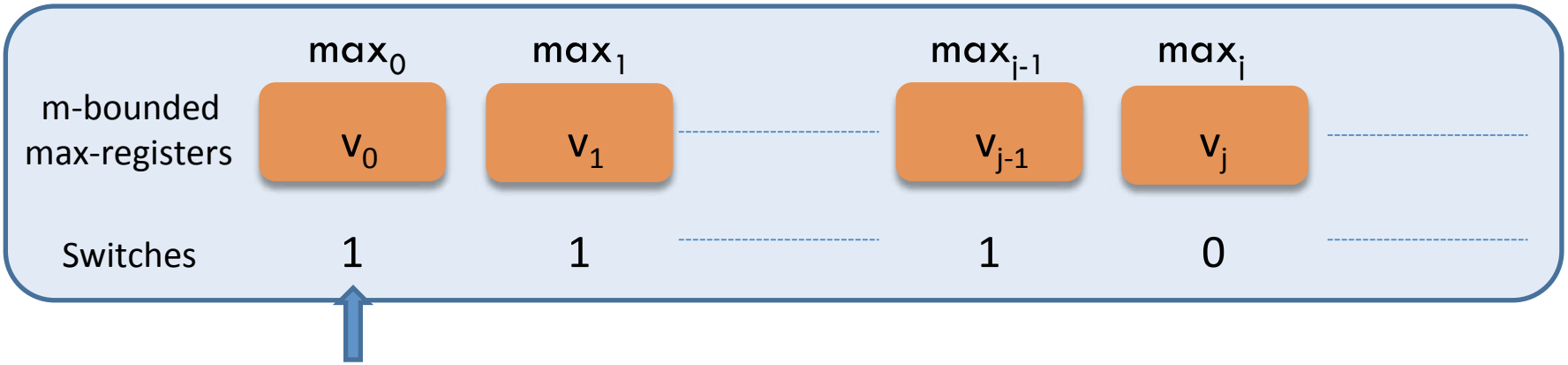
- An infinite array of m bounded max registers, max_j for all non-negative integers j
  - Each max register can store a value in [0,…,m-1]
  - Initially are all 0
- max_j is used to represents values in the range [mj,m(j+1) -1]
- An infinite number of 1-bit registers, $switches_j$ for all non-negative integers j
  - Initially are all 0

WriteMax(v) operation: Idea



- v=jm+v’ where v’<m for some non-negative integer j
- WriteMax(v’) in the low level max register max_j
- Set $switch_{j-1}$ to 1 to make it obsolete (greater values have been written)

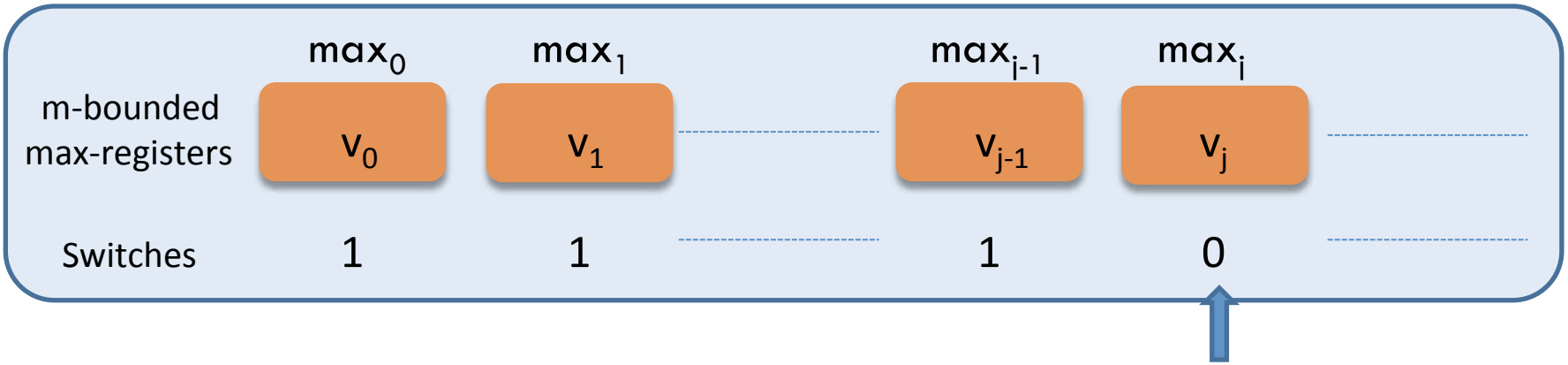
ReadMax operation : Idea



ReadMax() // by process i

Scan switches in increasing order to find a non-obsolete max_j

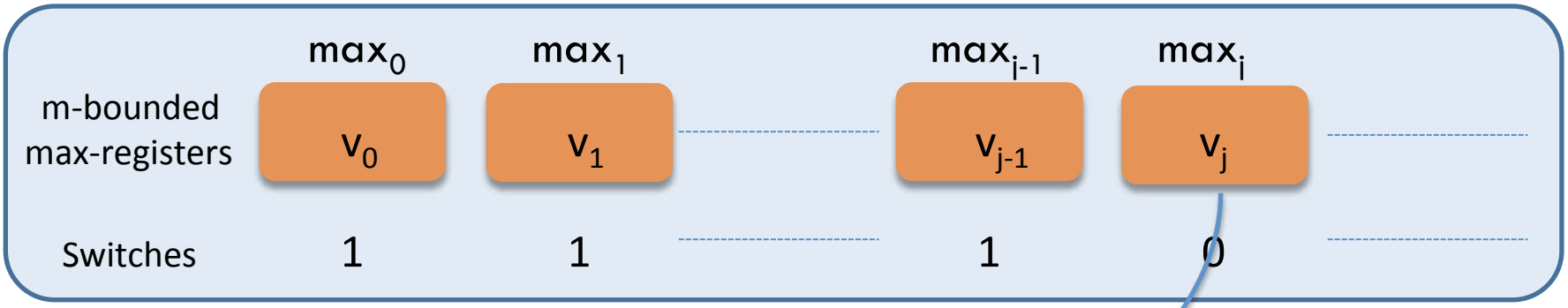
ReadMax operation : Idea



ReadMax() // by process i

Scan switches in increasing order to find a non-obsolete max_j

ReadMax operation : Idea



ReadMax() // by process i

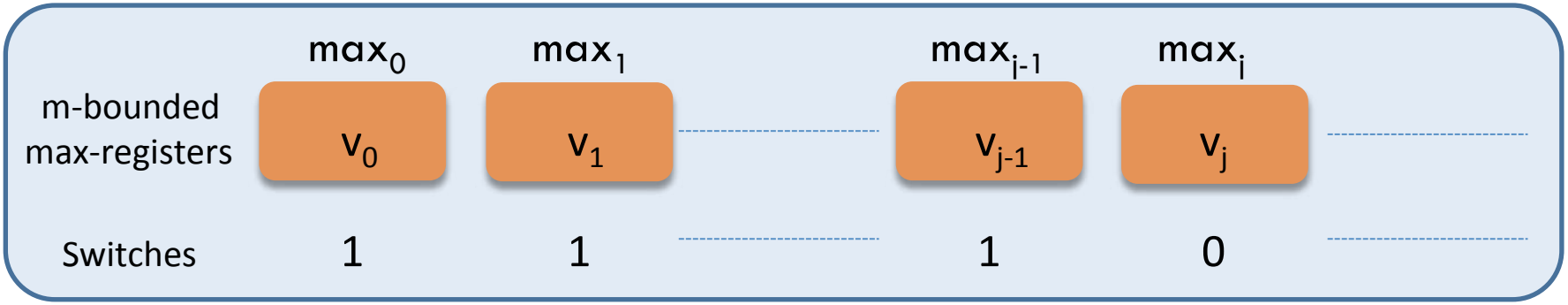
Scan switches in increasing order to find a non-obsolete max_j

If found such max_j

v' → *ReadMax*(max_j)

return j·m+v'

WriteMax operation: scenario #1

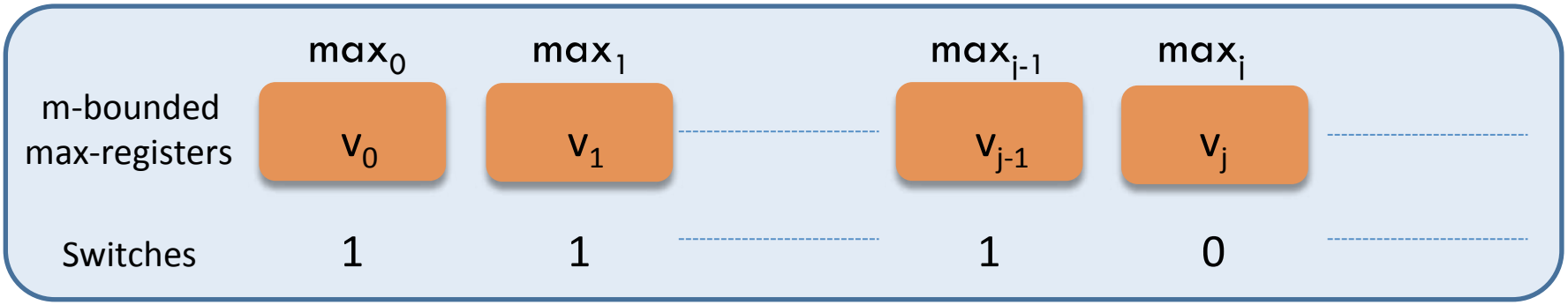


WriteMax(v) // by process i

v' ← v mod m

j ← ⌊v/m⌋

WriteMax operation: scenario #1



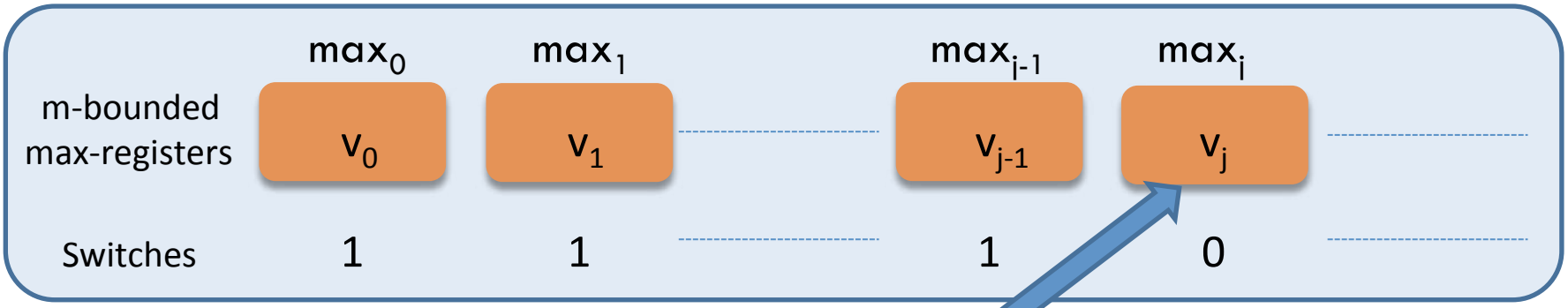
WriteMax(v) // by process i

v' ← v mod m

j ← ⌊v/m⌋

if switch$_j$=0

WriteMax operation: scenario #1



WriteMax(v) // by process i

v' ← v mod m

j ← ⌊v/m⌋

if switch$_j$=0

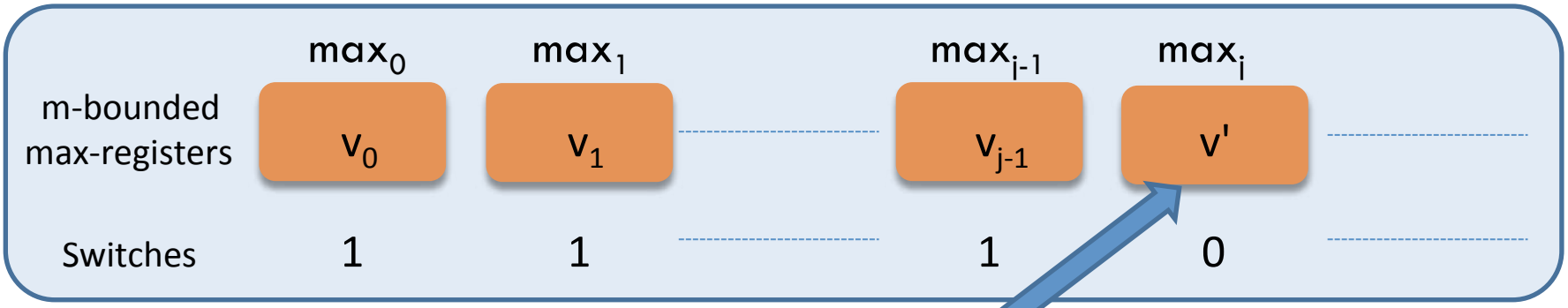
WriteMax(max$_j$,v')

last$_i$ ← max(j,last$_i$) // used by ReadMax operations

Old value: $j \cdot m + v_j$

New value: $j \cdot m + \max(v_j, v')$

WriteMax operation: scenario #1



WriteMax(v) // by process i

v' ← v mod m

j ← ⌊v/m⌋

if switch$_j$=0

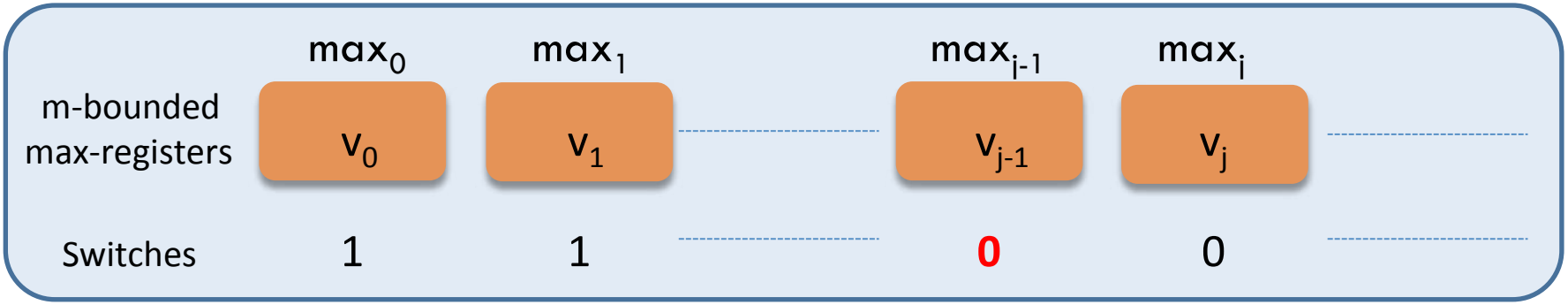
WriteMax(max$_j$,v')

last$_i$ ← max(j,last$_i$) // used by ReadMax operations

Old value: $j \cdot m + v_j$

New value: $j \cdot m + \max(v_j, v')$

WriteMax operation: scenario #2



WriteMax(v) // by process i

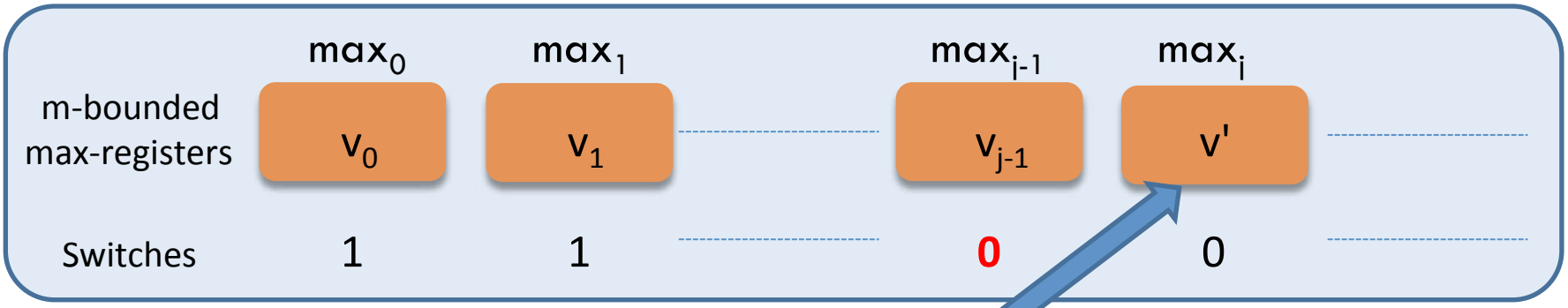
v' ← v mod m

j ← ⌊v/m⌋

if switch$_j$=0

WriteMax(max$_j$,v')

WriteMax operation: scenario #2



WriteMax(v) // by process i

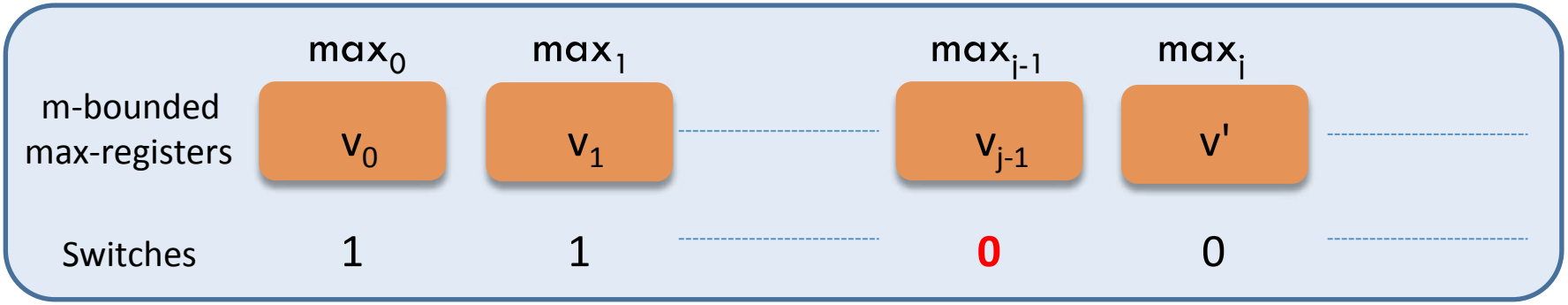
v' ← v mod m

j ← ⌊v/m⌋

if switch$_j$=0

WriteMax(max$_j$,v')

WriteMax operation: scenario #2



WriteMax(v) // by process i

v' ← v mod m

j ← ⌊v/m⌋

if switch$_j$=0

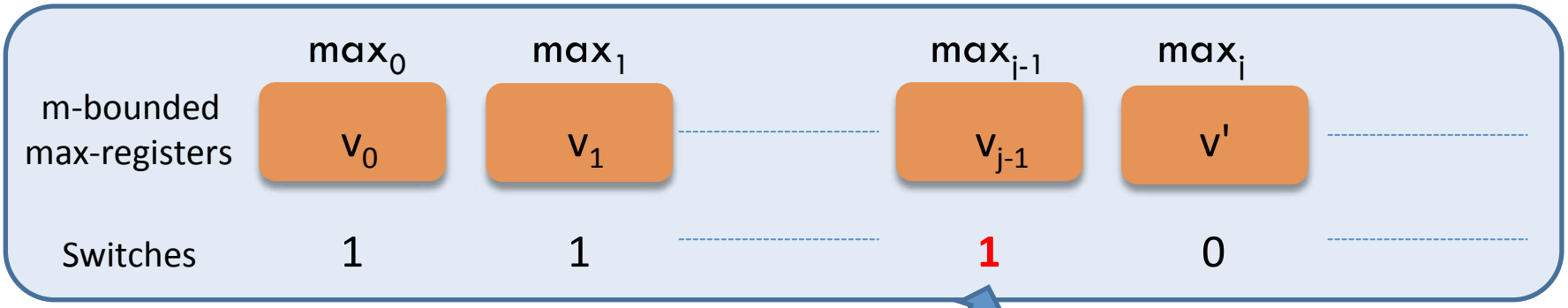
WriteMax(max$_j$,v')

if (**switch$_{j-1}$= 0**)

switch$_{j-1}$←1

last$_i$ ← max(j,last$_i$) // used by ReadMax operations

WriteMax operation: scenario #2



WriteMax(v) // by process i

v' ← v mod m

j ← ⌊v/m⌋

if switch$_j$=0

WriteMax(max$_j$,v')

if (**switch$_{j-1}$= 0**)

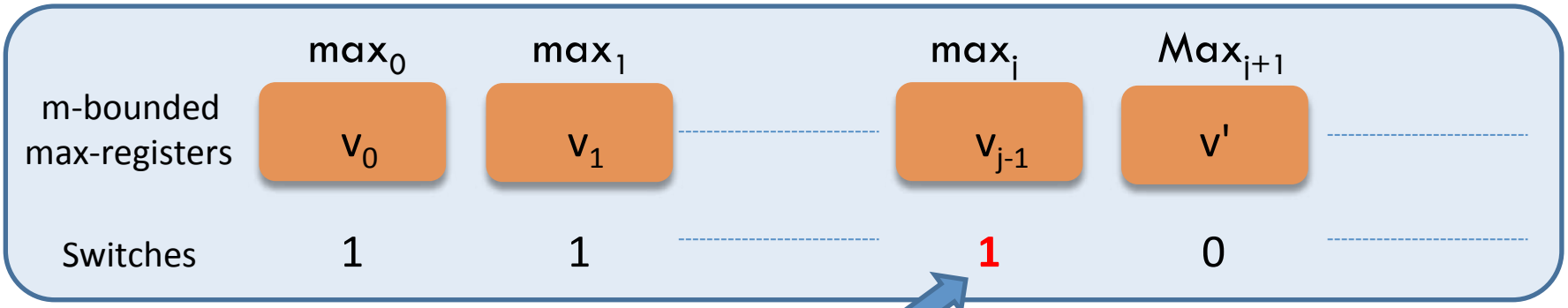
switch$_{j-1}$←1

last$_i$ ← max(j,last$_i$) // used by ReadMax operations

Old value: (j-1)·m + v$_{j-1}$

New value: j·m + max(v$_j$,v')

WriteMax operation: scenario #3



WriteMax(v) // by process i

v' ← v mod m

j ← ⌊v/m⌋

if switch$_j$=0

…

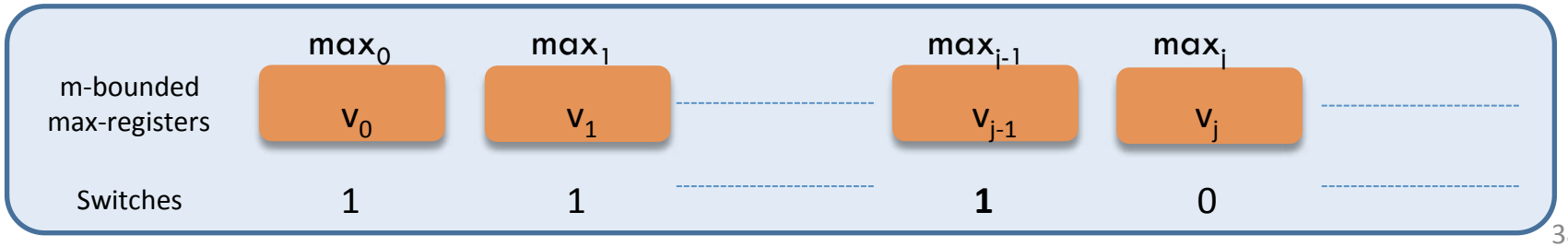
last$_i$ ← max(j,last$_i$) // used by ReadMax operations

Max register already obsolete,
no need to invoke *MaxWrite*

Unbounded Max register amortized step complexity

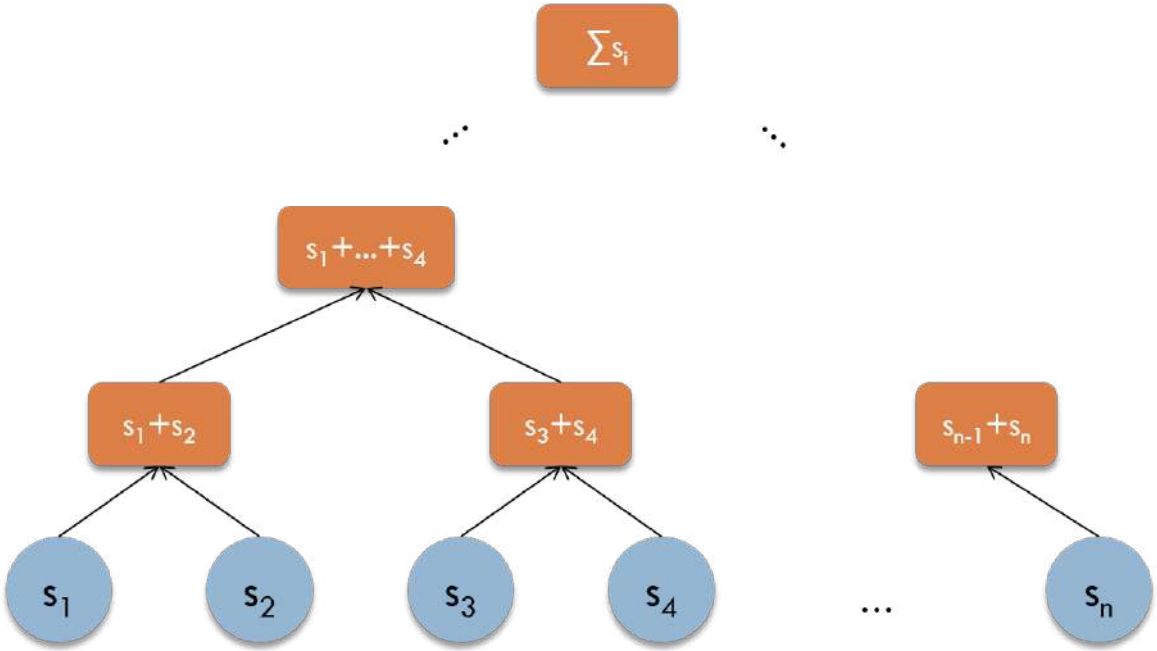
- Each high-level WriteMax performs at most a single low-level max_j.*WriteMax* operation ➔ O(logm) steps
- Each high-level ReadMax op performs at most a single low-level max_j.*ReadMax* operation ➔O(logm) steps
- By n-bounded increments assumption, and if $m \geq n^2$ all reads of the switch of an obsolete max_j can be amortized against at least n max_j.*WriteMax* operations performed on it

the amortized step complexity is O(logn)

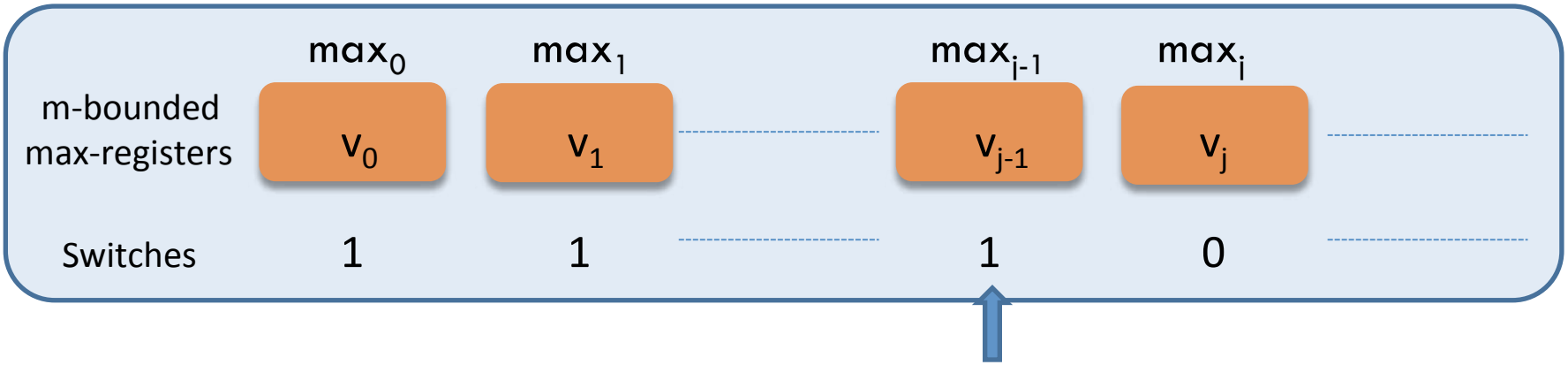


Counter amortized step complexity

Counter's amortized step complexity is $O(\log^2 n)$



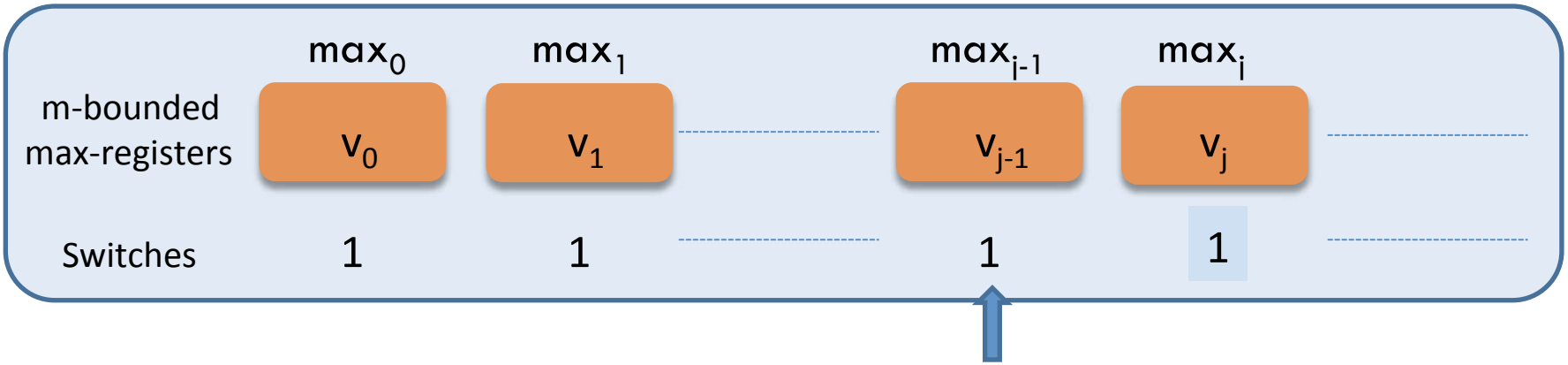
The wait-free algorithm : Idea



ReadMax() // by process i

Scan switches in increasing order to find a non-obsolete max_j

The wait-free algorithm : Idea



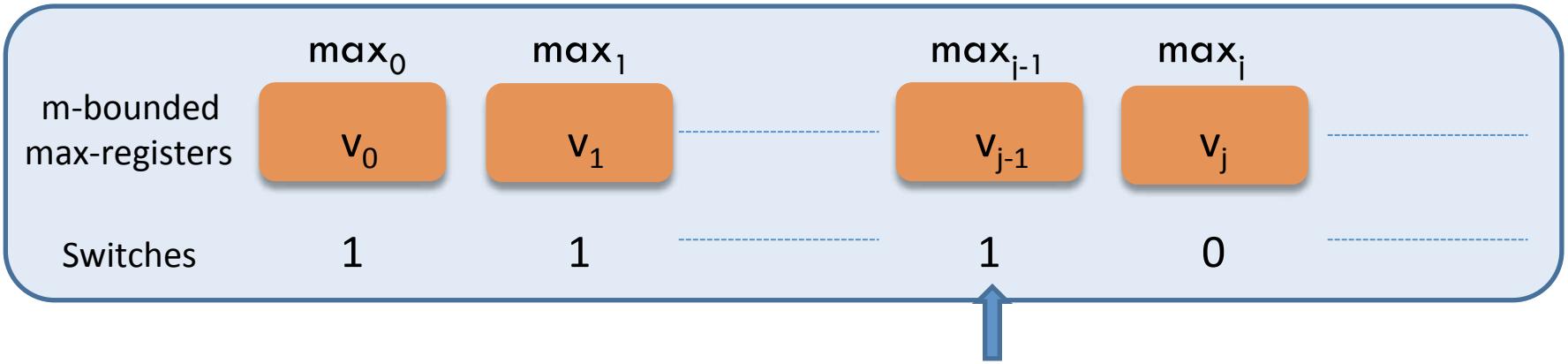
ReadMax() // by process i

Scan switches in increasing order to find a non-obsolete max_j

WriteMax() // by process j

Makes max_j obsolete before the ReadMax reads the corresponding switch.

The wait-free algorithm : Idea



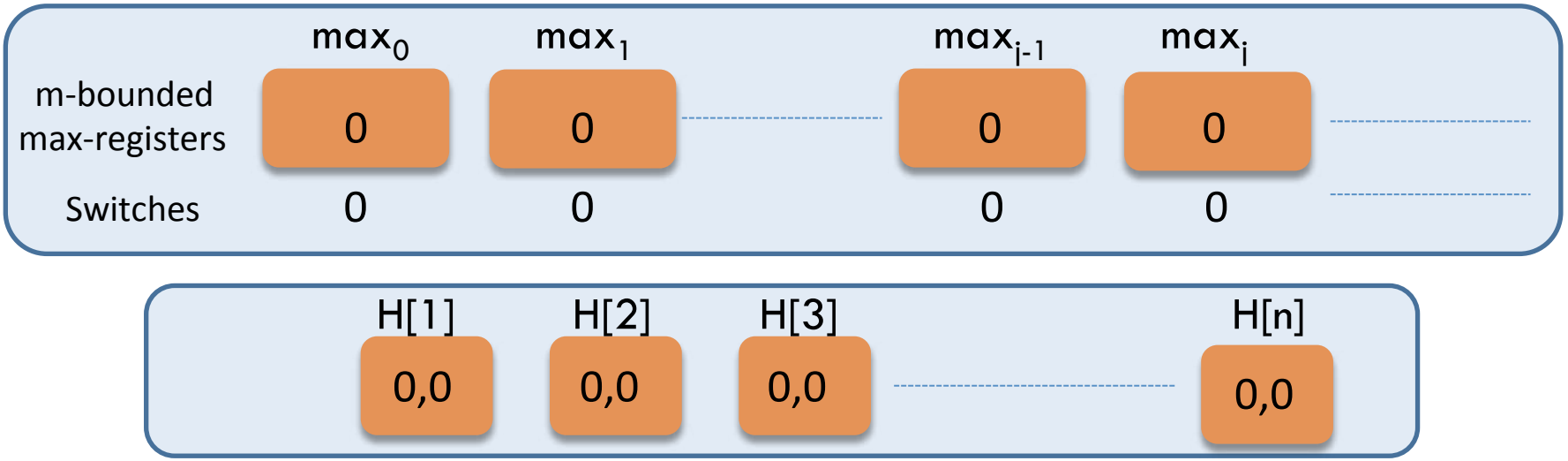
ReadMax() // by process i

Scan switches in increasing order to find a non-obsolete max_j

WriteMax() // by process j

1. Read max_j, compute the value for the ReadMax and store it into a register
2. Make max_j obsolete

The wait-free algorithm : helping mechanism



- A process i that makes a max-register obsolete, writes in the register H[i] the computed value of the max register and a sequence number
- Every O(n) steps, a ReadMax operation op reads all H[i]. It returns the value of a H[j] whose sequence number has been incremented at least twice during op.

Discussion

- We present the first wait-free read/write counter algorithm with sublinear amortized step complexity
  - Wait-free, amortized step complexity $O(\log^2 n)$
- Logarithmic lower bound ($\Omega(\log n)$) on amortized step complexity of counters and other objects

Attiya and Hendler, TPDS '10

- Is this bound tight?
- Space complexity is infinite. Can it be bounded?

Thank you.

Questions?