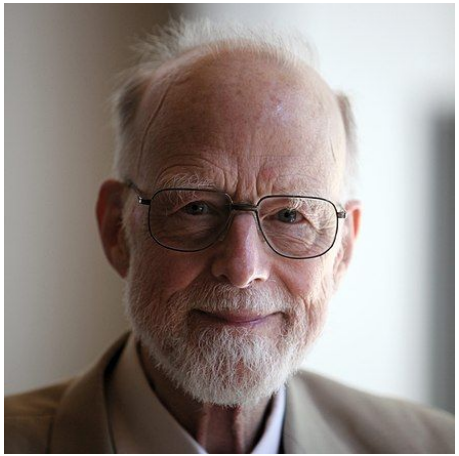
In place shared-memory sorting algorithms

Chris Rinard

Quicksort History:

Invented in 1951 by Tony Hoare

Architecture of the time is in a museum now



TL;DR

- This paper presents IPS^{4}O: a parallel, in-place version of samplesort
- At the time of writing, Quicksort and variants are the predominantly used sorting alg

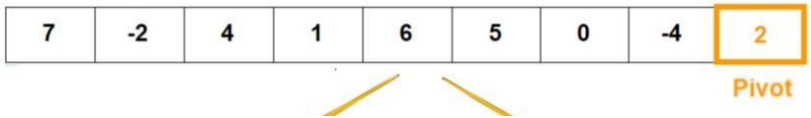
“You have to outperform quicksort in every respect in order to replace it”

Improvements to quicksort

- Strictly in-place
- 2-3 pivots (20% better than single pivot)
- Parallel Quicksort (Tsigas, Zhang)
- Samplesort

Quicksort review

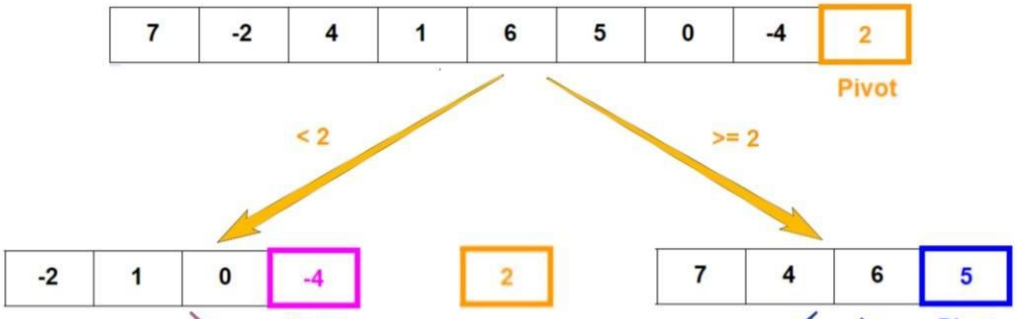
1: Choose Pivot



Quicksort review

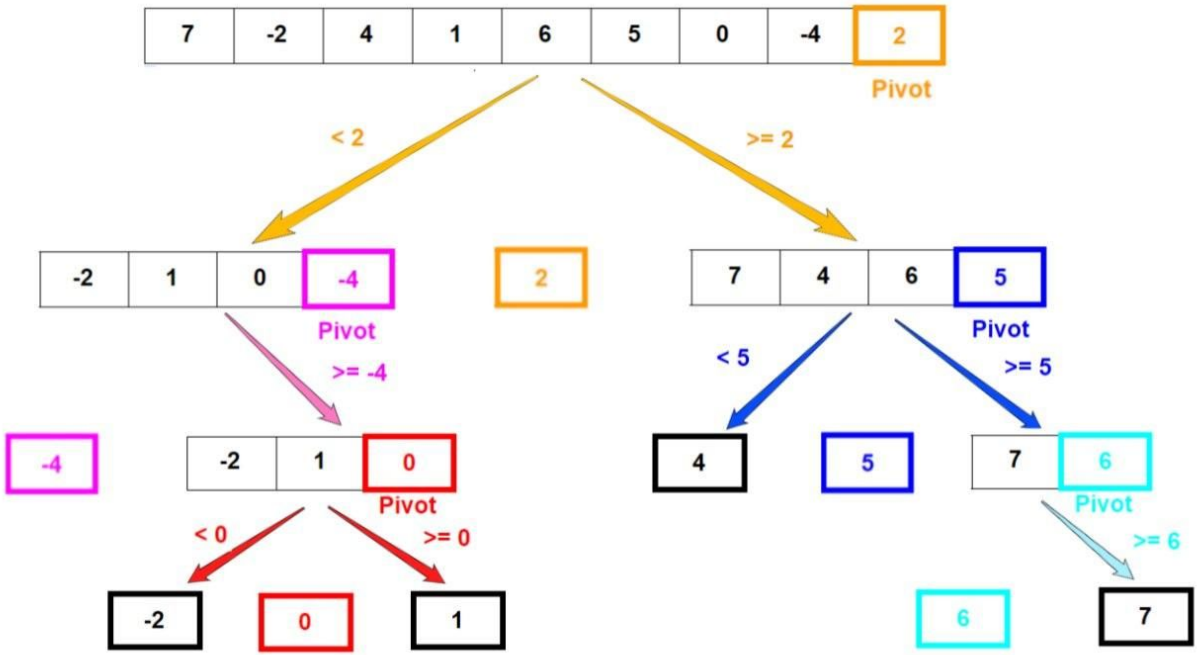
1: Choose Pivot

2: Put pivot at its correct sorted position, all smaller elements before pivot, and all greater elements after pivot



Quicksort review

3: Quicksort the smaller and larger elements (left and right)



Samplesort

Basic idea: k-way Quicksort

3 Phases + recursion

1. Sampling
2. Classification
3. Distribution
4. Recurse

Sampling

1. Sample a * k - 1 randomly sampled inputs into array *S*.
2. Sort *S*
3. Pick splitters $s_0 \ldots s_{k-2}$ from *S*

Classification

1. For each element, find bucket index, and keep track of bucket size (*e* in b_i if $s_{i-1} < e <= s_i$).
2. Classify each element of the input into correct bucket
3. Find memory locations of boundaries

Distribution

1. Copy elements from input array into buckets.

IPS^{4}O

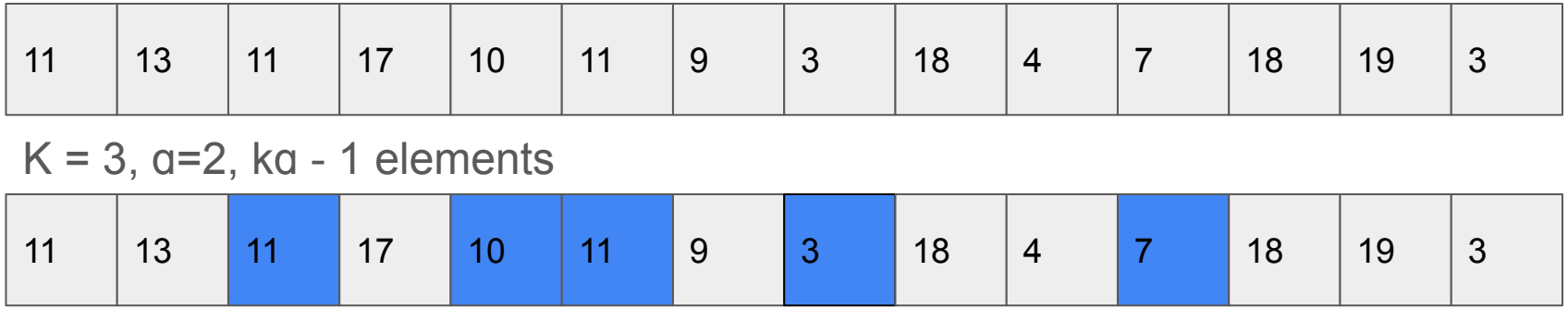
4 Stages + recursion:

1. Sampling: bucket boundaries
2. Classification: Group input into blocks (in block, every elem in same bucket)
3. Permutation: Globally order blocks
4. Cleanup: Clean up partially filled or crossing blocks

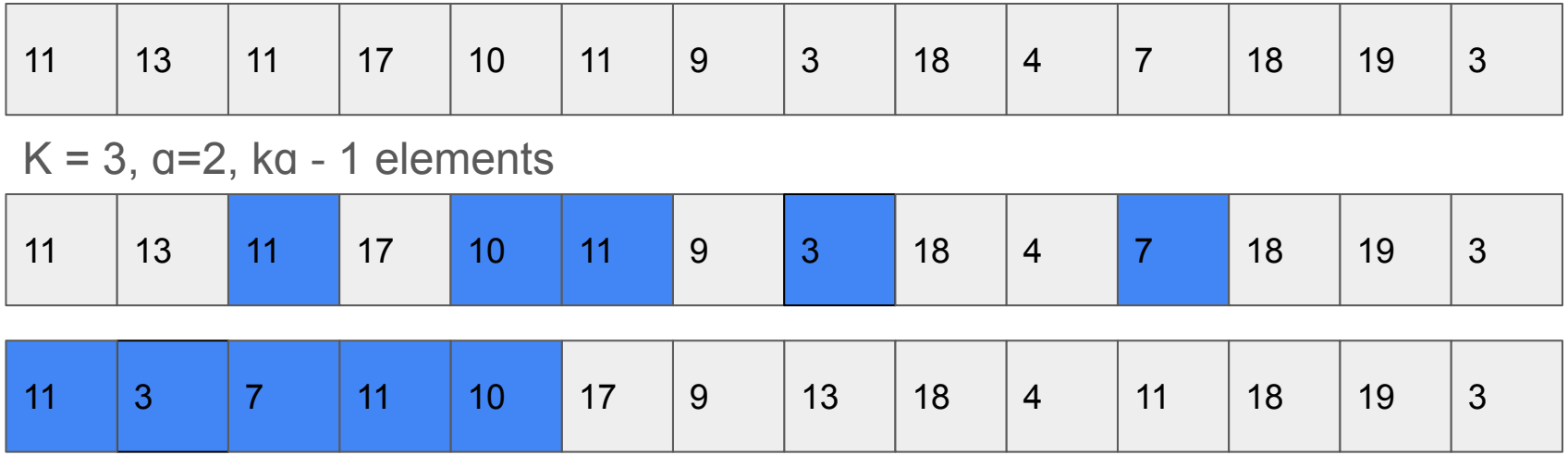
IPS^{4}O Sampling phase

11	13	11	17	10	11	9	3	18	4	7	18	19	3

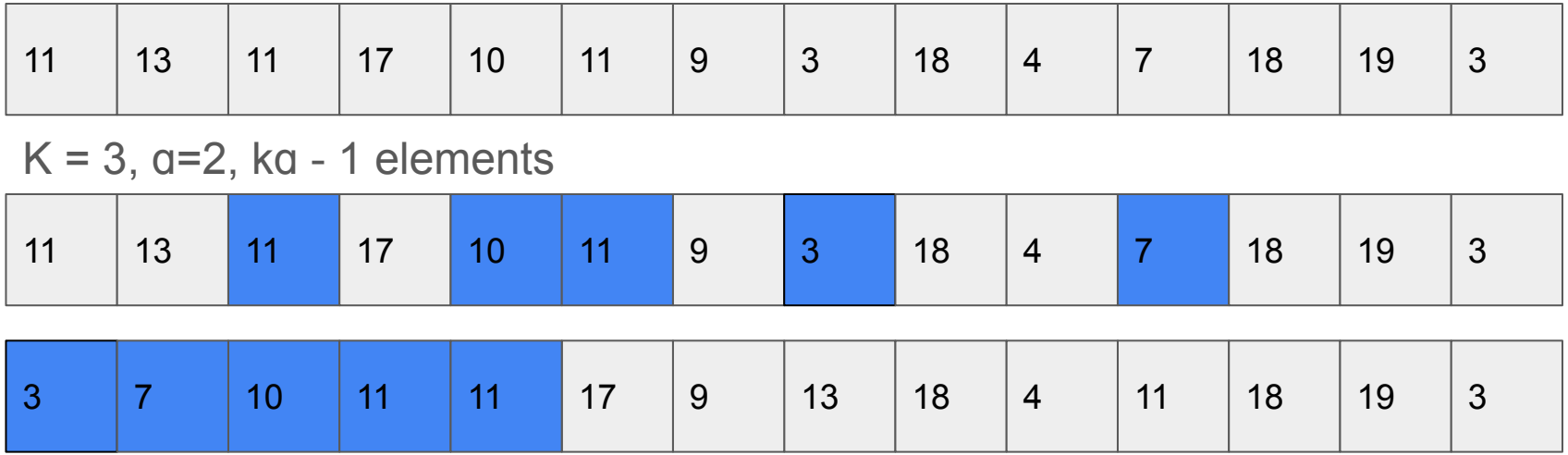
IPS^{4}O Sampling phase



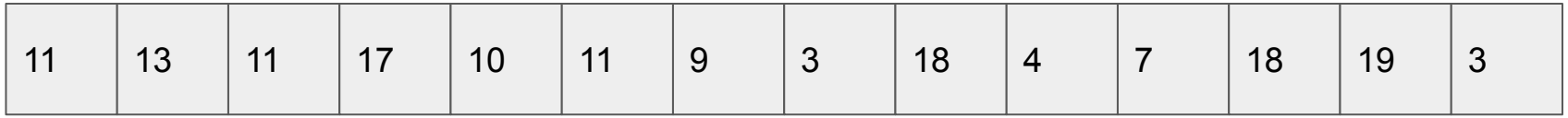
IPS^{4}O Sampling phase



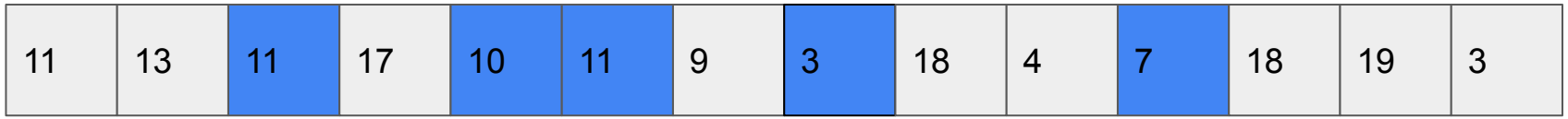
IPS^{4}O Sampling phase

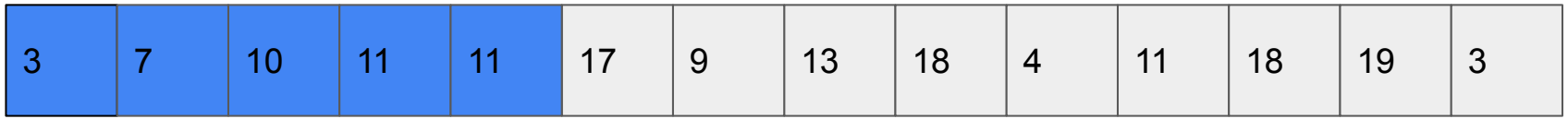


IPS^{4}O Sampling phase

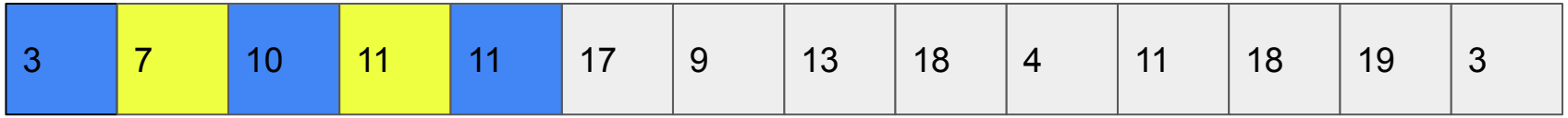


K = 3, α=2, kα - 1 elements



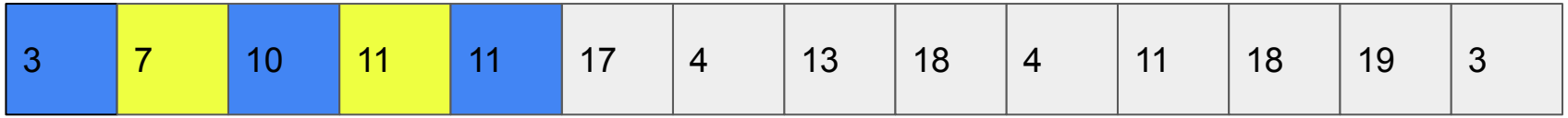


K = 3, k - 1 splitters (picked equidistantly)

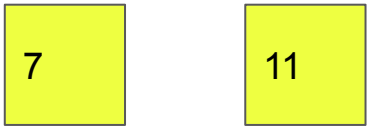


IPS^{4}O Sampling

K = 3, k - 1 splitters (picked equidistantly)



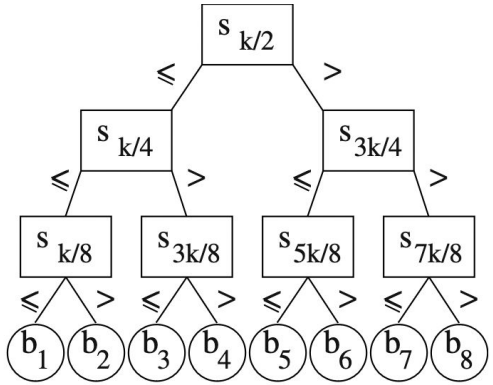
Create branchless decision tree, k buckets



Performance Hack: IPS^{4}O bucket structure (branchless decision tree)

- Eliminates branch mispredictions: use of a = (<>) ? b : c, easy to store

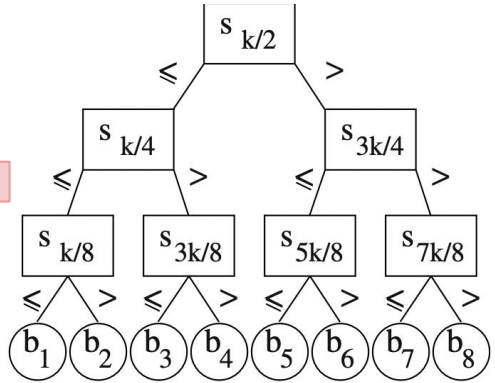
$t := \langle s_{k/2}, s_{k/4}, s_{3k/4}, s_{k/8}, s_{3k/8}, s_{5k/8}, s_{7k/8}, \ldots \rangle$ //
for $i := 1$ **to** n **do** // locate each element
  $j := 1$ // current tree node := root
  repeat $\log k$ *times* // will be unrolled
    $j := 2j + (a_i > t_j)$ // left or right?
  $j := j - k + 1$ // bucket index
  $|b_j|{+}{+}$ // count bucket size
  $o(i) := j$ // remember oracle



Performance Hack: IPS^{4}O bucket structure (branchless decision tree)

- ~~Eliminates branch mispredictions: use of a = (<>) ? b : c, easy to store~~
- Better than this, you can unroll the loop

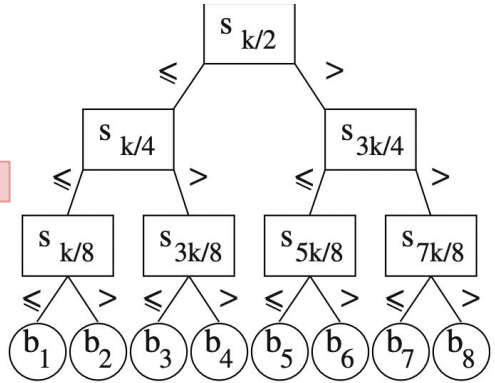
$t := \langle s_{k/2}, s_{k/4}, s_{3k/4}, s_{k/8}, s_{3k/8}, s_{5k/8}, s_{7k/8}, \ldots \rangle$ //
for $i := 1$ **to** n **do** // locate each element
  $j := 1$ // current tree node := root
  repeat $\log k$ *times* // will be unrolled
    $j := 2j + (a_i > t_j)$ // left or right?
  $j := j - k + 1$ // bucket index
  $|b_j|{+}{+}$ // count bucket size
  $o(i) := j$ // remember oracle



Performance Hack: IPS^{4}O bucket structure (branchless decision tree)

- ~~Eliminates branch mispredictions: use of a = (<>) ? b : c, easy to store~~
- Better than this, you can unroll this loop
- In practice, authors note “up to 2x faster than std::sort”

$t := \langle s_{k/2}, s_{k/4}, s_{3k/4}, s_{k/8}, s_{3k/8}, s_{5k/8}, s_{7k/8}, \ldots \rangle$ //
for $i := 1$ **to** n **do** // locate each element
  $j := 1$ // current tree node := root
  repeat $\log k$ *times* // will be unrolled
    $j := 2j + (a_i > t_j)$ // left or right?
  $j := j - k + 1$ // bucket index
  $|b_j|{+}{+}$ // count bucket size
  $o(i) := j$ // remember oracle



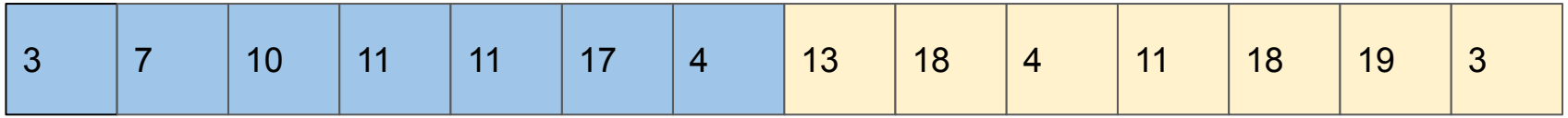
IPS^{4}O

4 Stages:

1. ~~Sampling: bucket boundaries~~
2. Classification: Group input into blocks (in block, every elem in same bucket)
3. Permutation: Globally order blocks
4. Cleanup: Clean up partially filled or crossing blocks

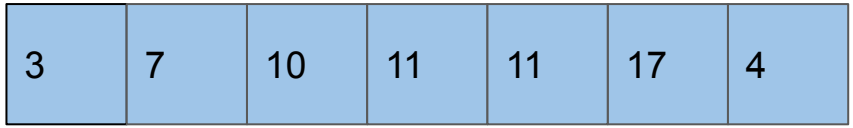
IPS^{4}O Classification

t = 2, split into *t* “stripes”



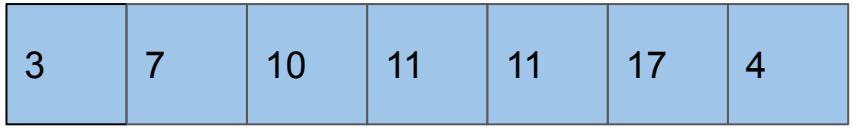
IPS^{4}O Classification

t = 2, split into *t* “stripes”



IPS^{4}O Classification

t = 2, split into *t* “stripes”



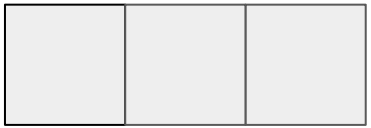
k=3, each thread has *k* “buffer blocks”

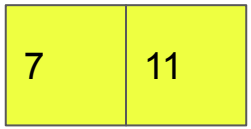
IPS^{4}O Classification

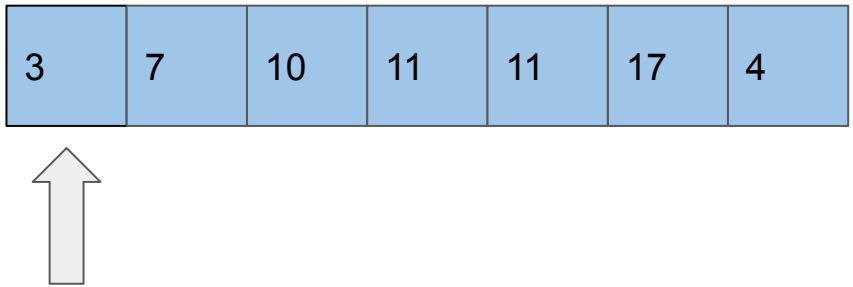
t = 2, split into *t* “stripes”

k=3, each thread has *k* “buffer blocks”

Block size limited -- for this case = 2

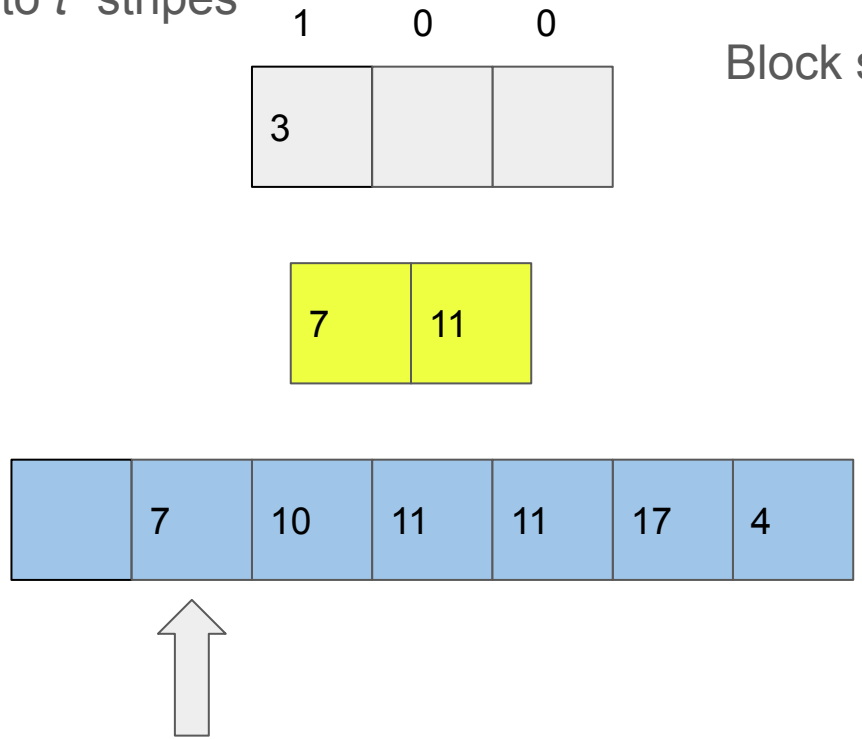






IPS^{4}O Classification

t = 2, split into *t* “stripes”

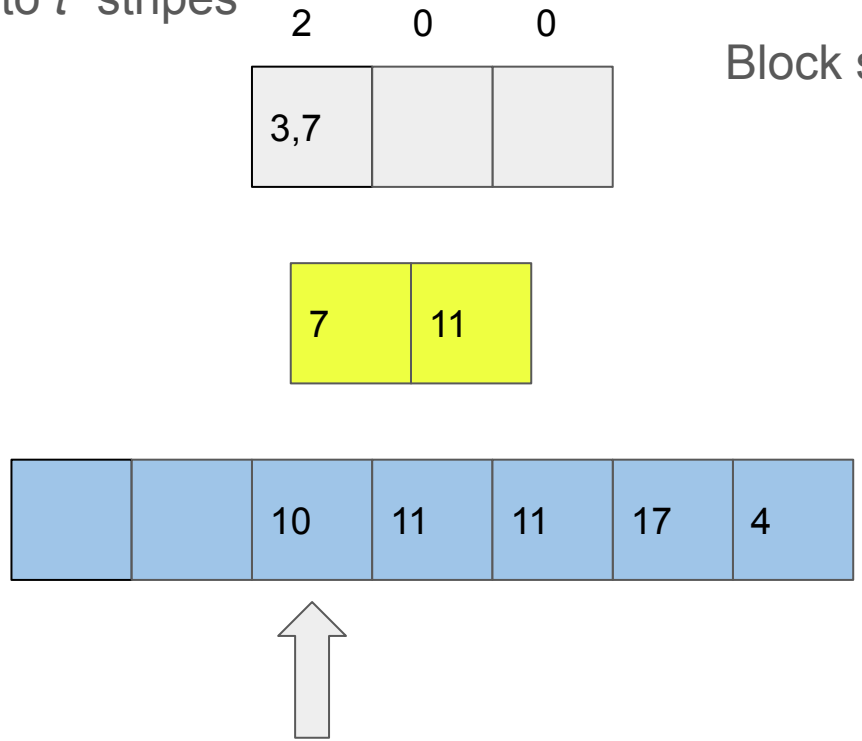


k=3, each thread has *k* “buffer blocks”

Block size limited -- for this case = 2

IPS^{4}O Classification

t = 2, split into *t* “stripes”

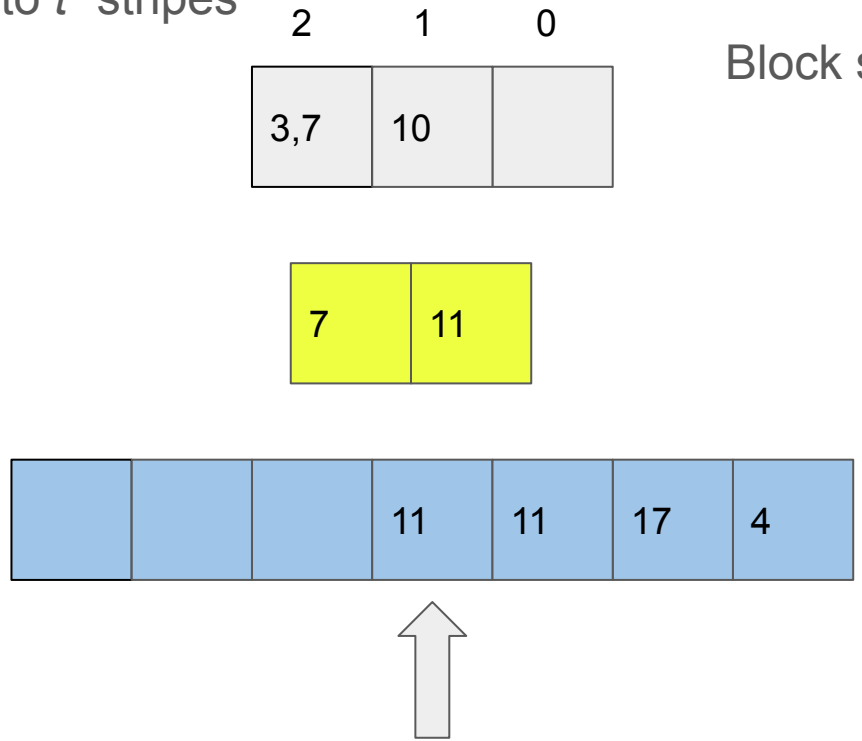


k=3, each thread has *k* “buffer blocks”

Block size limited -- for this case = 2

IPS^{4}O Classification

t = 2, split into *t* “stripes”

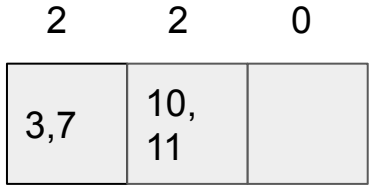


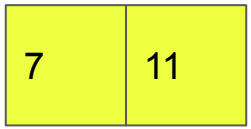
k=3, each thread has *k* “buffer blocks”

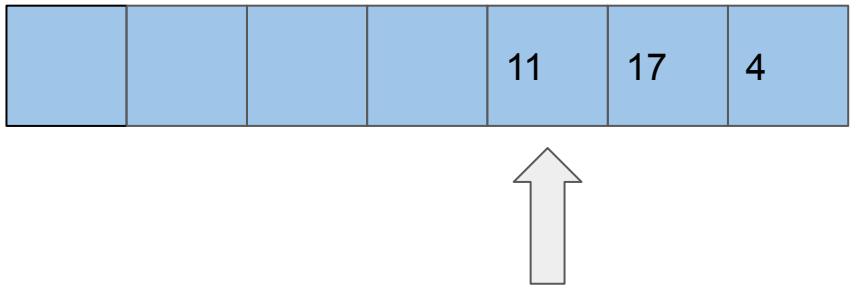
Block size limited -- for this case = 2

IPS^{4}O Classification

t = 2, split into *t* “stripes”







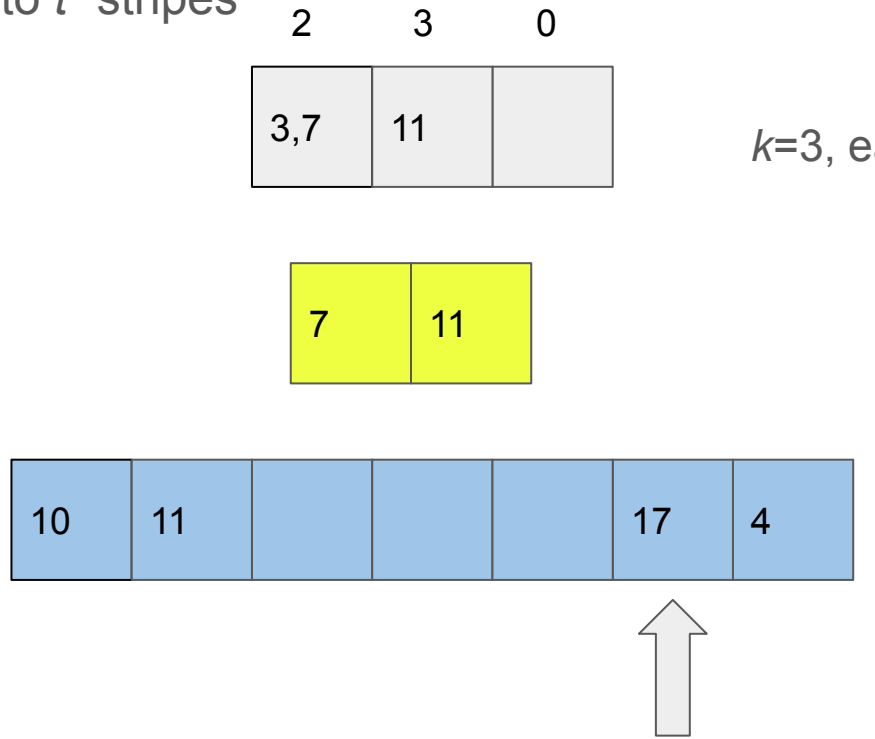
k=3, each thread has *k* “buffer blocks”

Block size limited -- for this case = 2

IPS^{4}O Classification

t = 2, split into *t* “stripes”

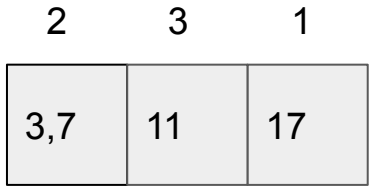
k=3, each thread has *k* “buffer blocks”



k=3, each thread has *k* “buffer blocks”

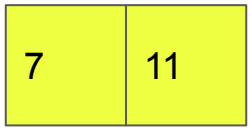
IPS^{4}O Classification

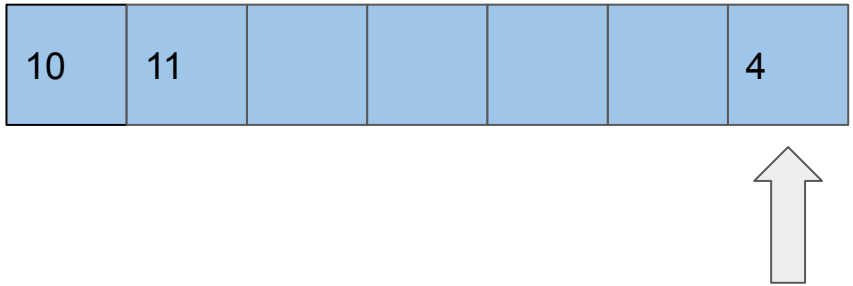
t = 2, split into *t* “stripes”



k=3, each thread has *k* “buffer blocks”

Block size limited -- for this case = 2



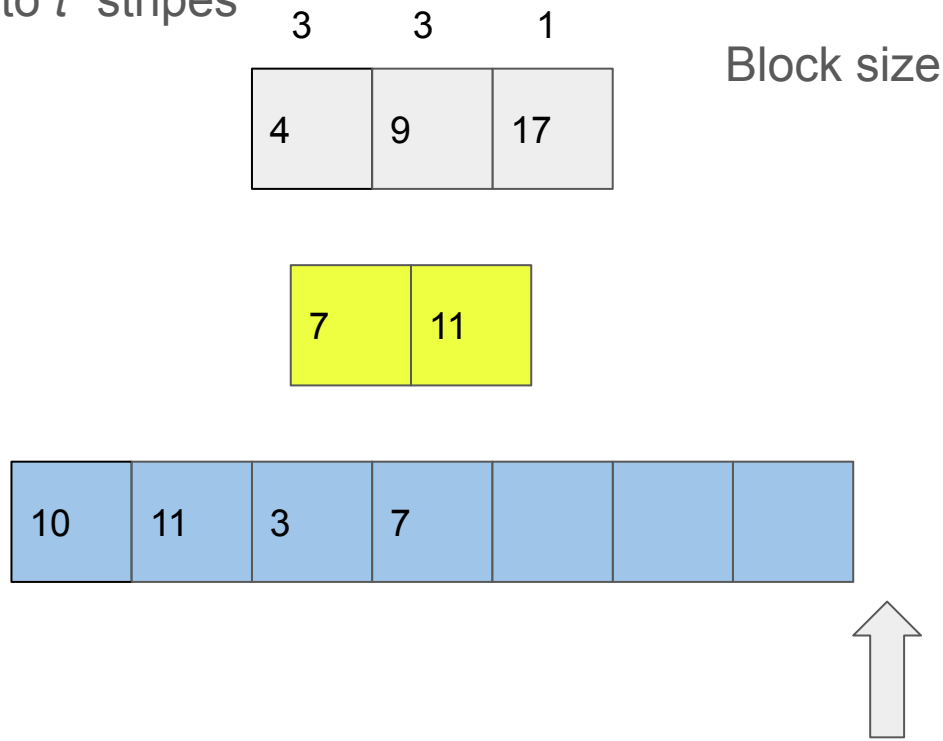


IPS^{4}O Classification

t = 2, split into *t* “stripes”

k=3, each thread has *k* “buffer blocks”

Block size limited -- for this case = 2

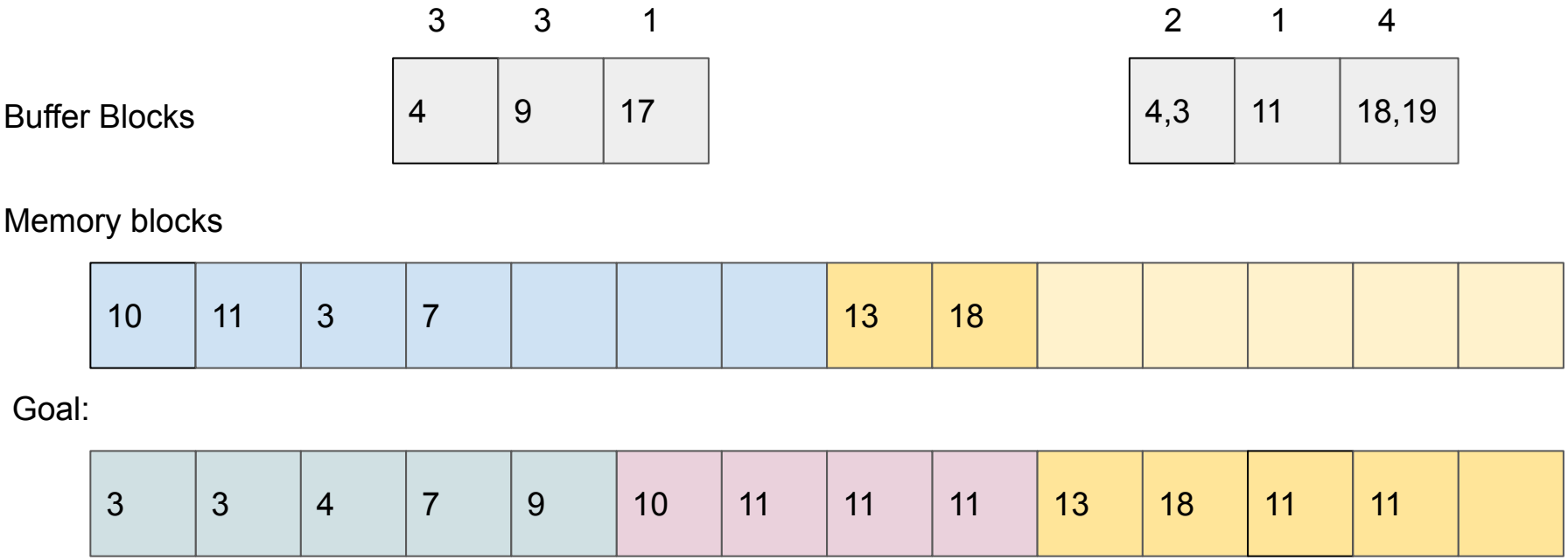


IPS^{4}O

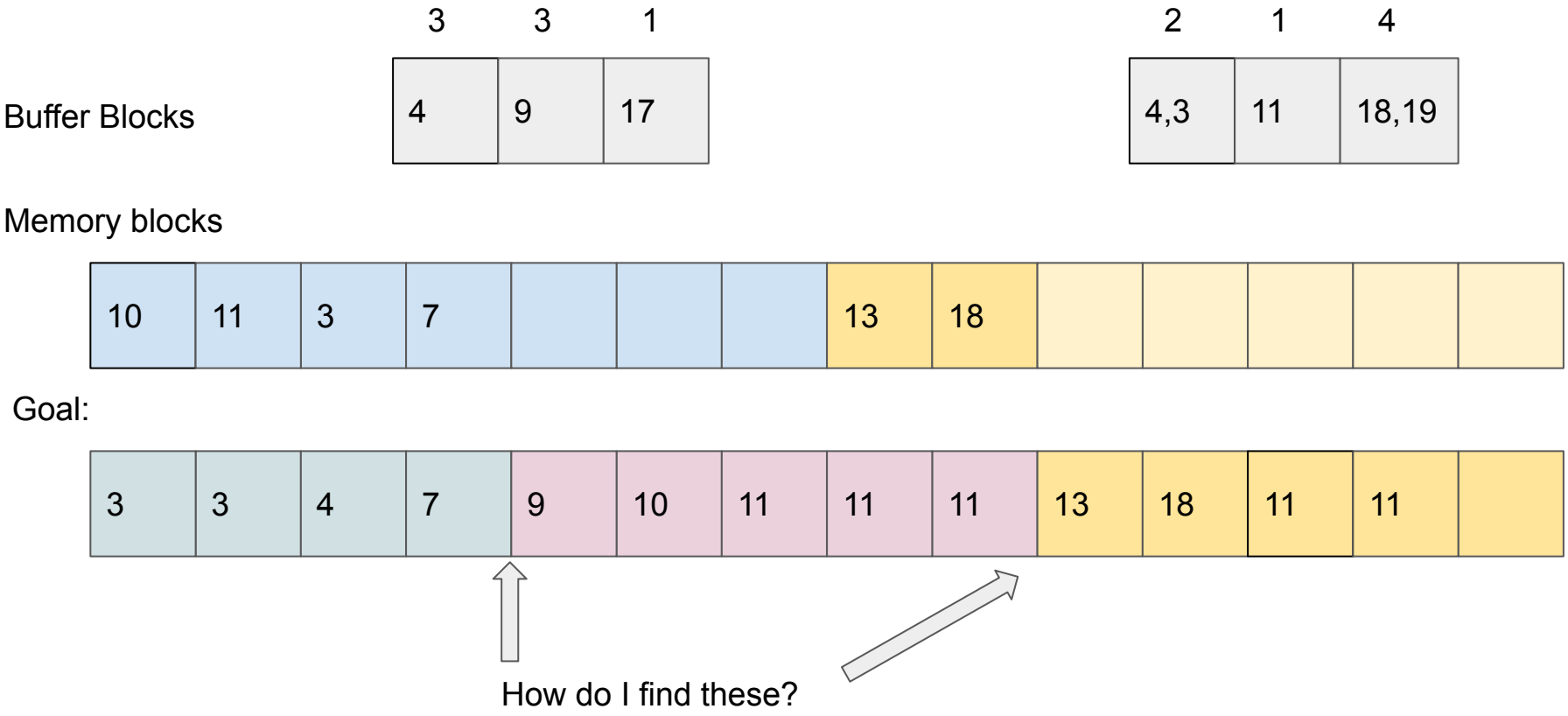
4 Stages:

1. ~~Sampling: bucket boundaries~~
2. ~~Classification: Group input into blocks (in block, every elem in same bucket)~~
3. Permutation: Globally order blocks
4. Cleanup: Clean up partially filled or crossing blocks

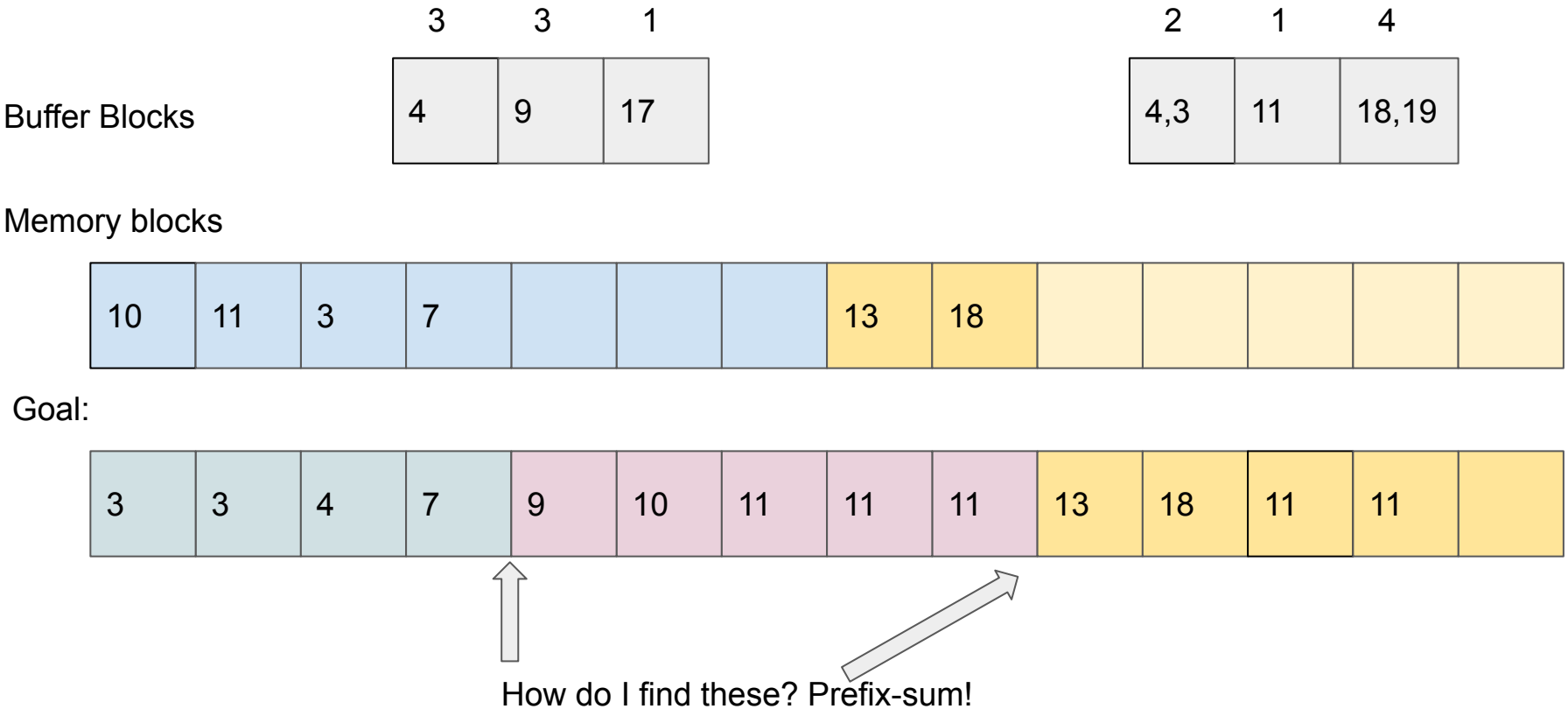
IPS^{4}O Block Permutation



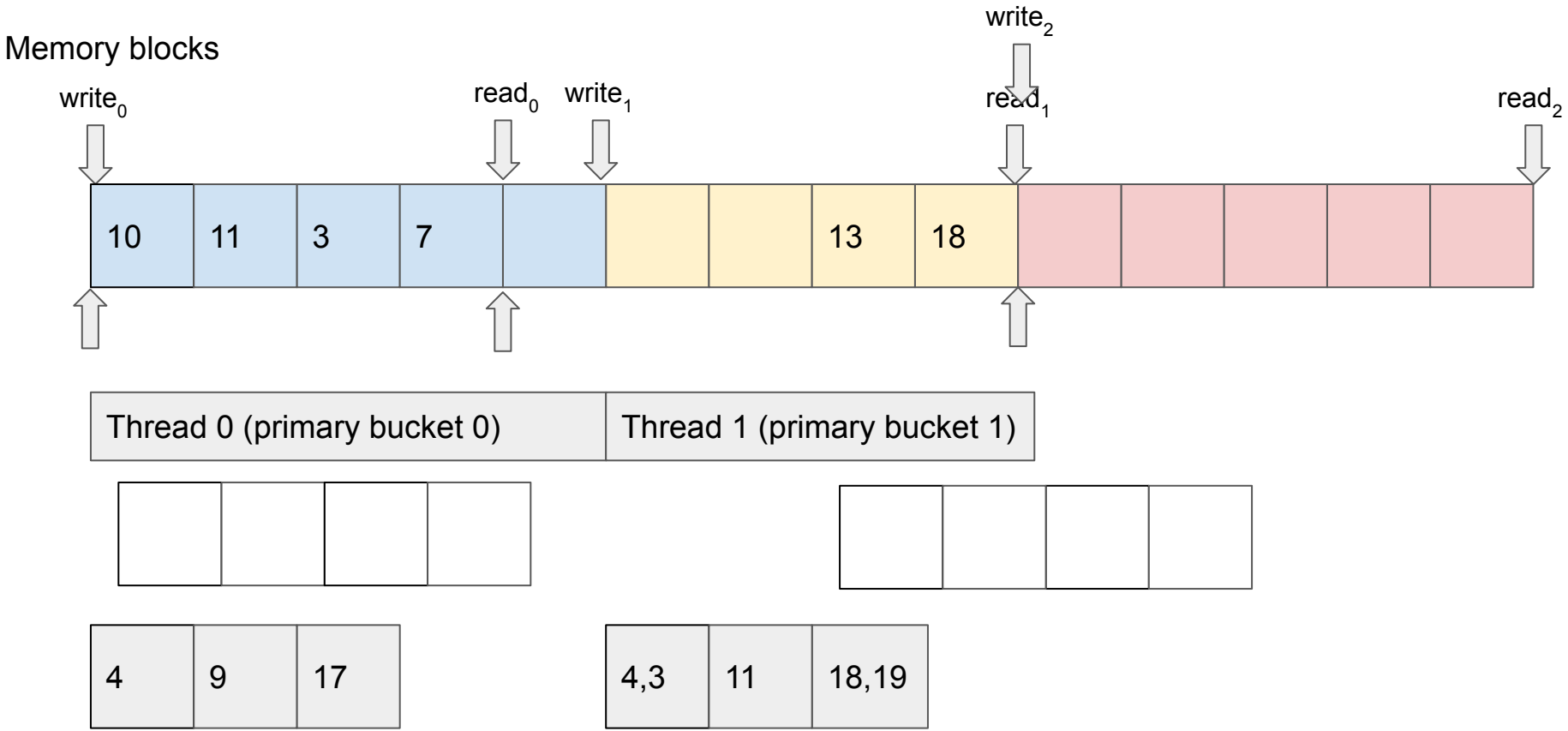
IPS^{4}O Block Permutation



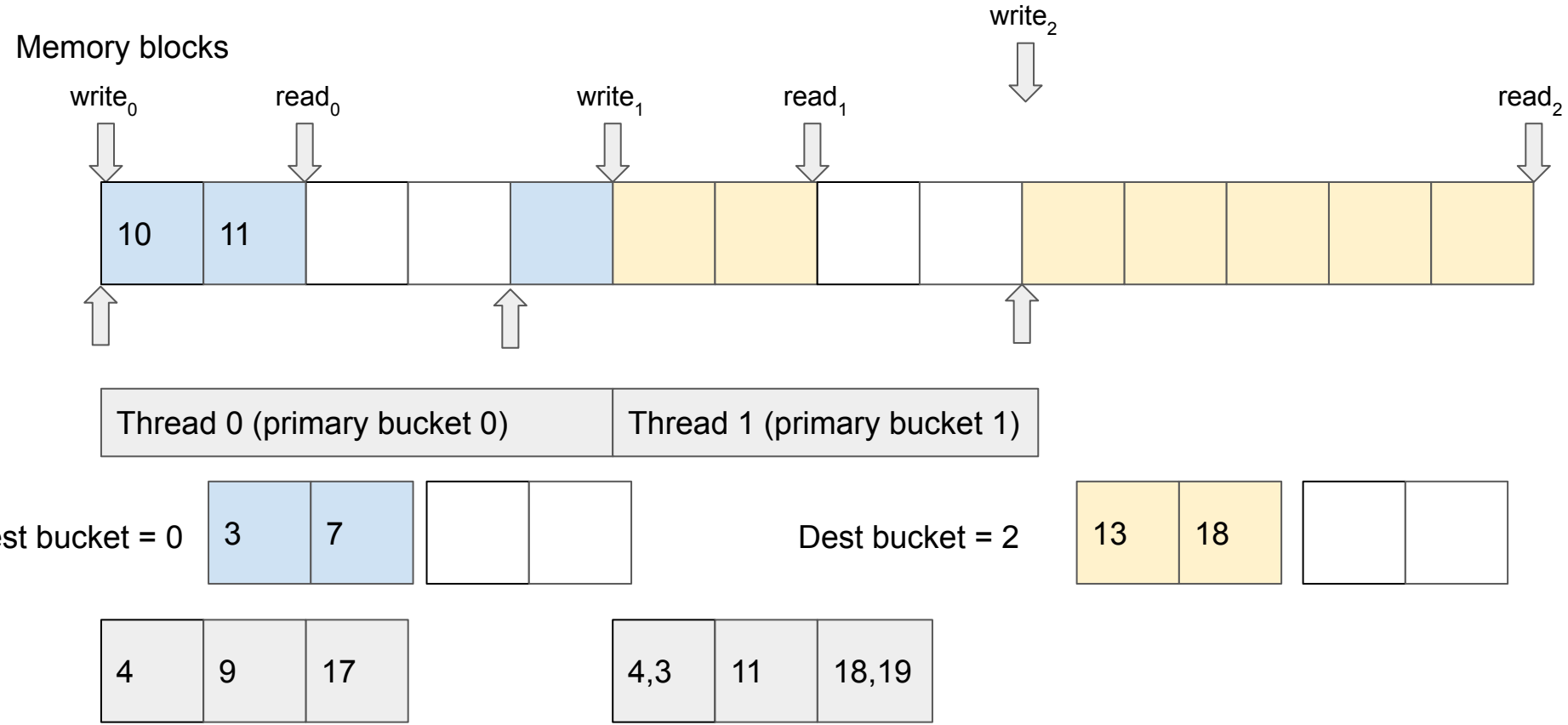
IPS^{4}O Block Permutation



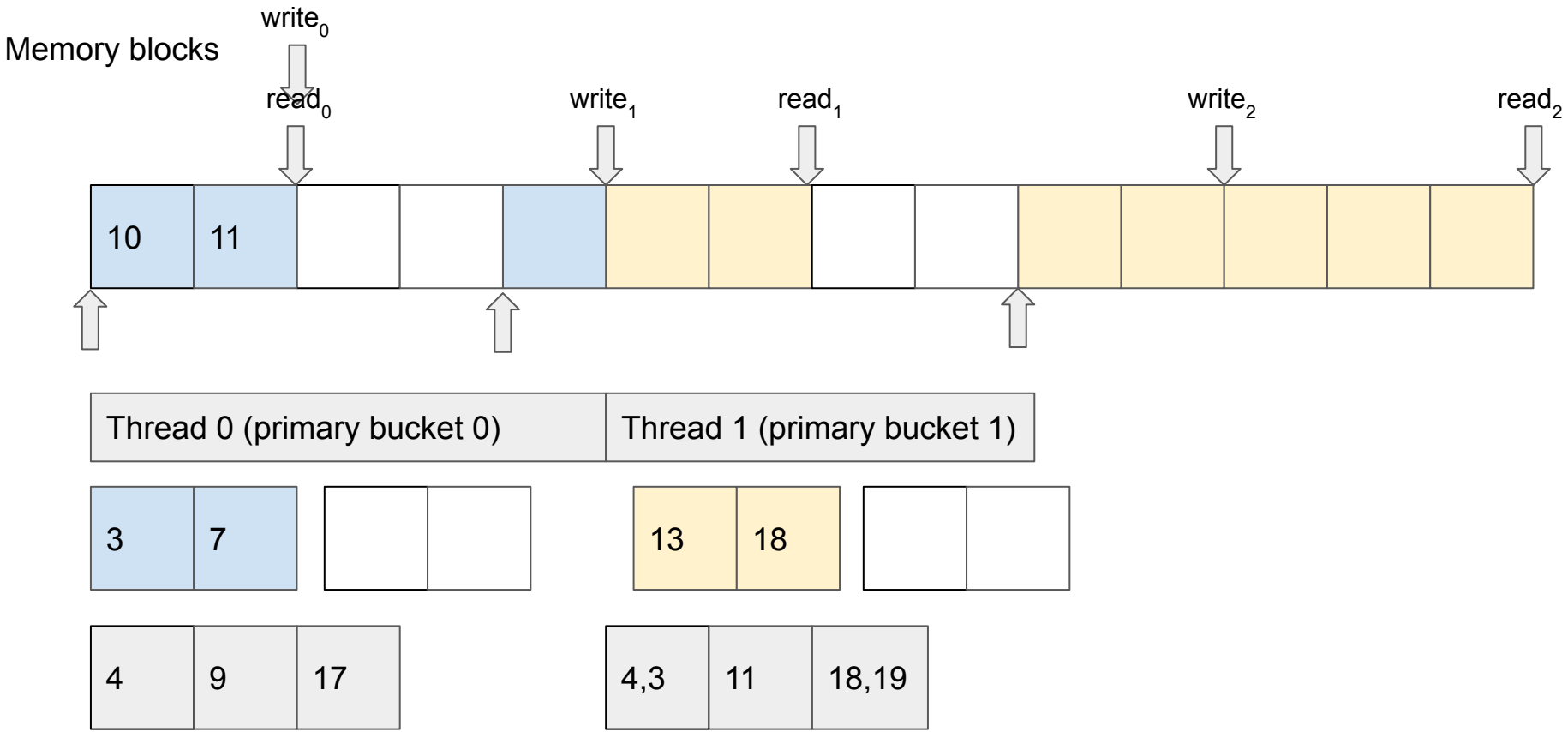
IPS^{4}O Block Permutation



IPS^{4}O Block Permutation

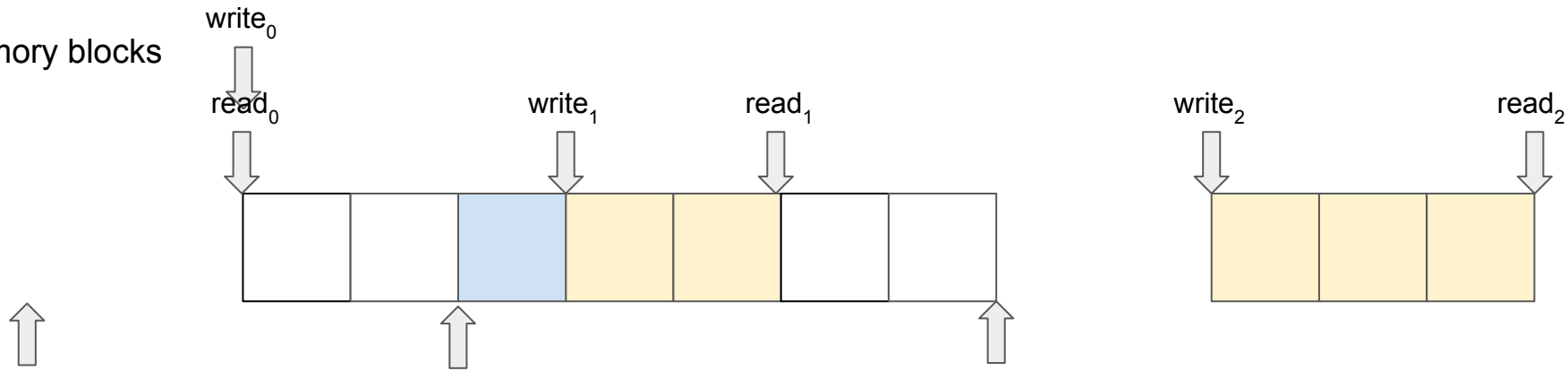


IPS^{4}O Block Permutation



IPS^{4}O Block Permutation

Memory blocks

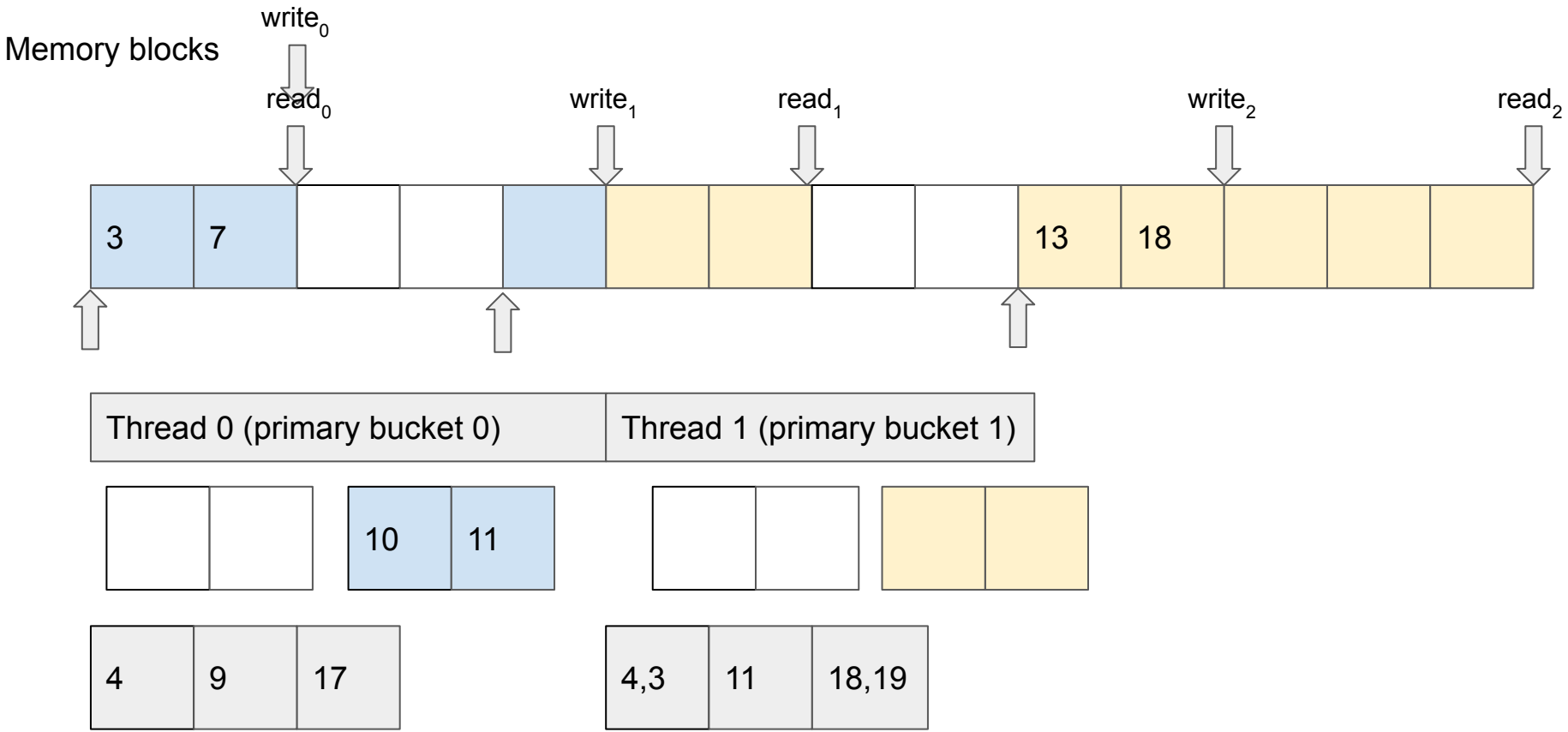


Thread 0 (primary bucket 0)		Thread 1 (primary bucket 1)	
3	7	10	11
13	18		

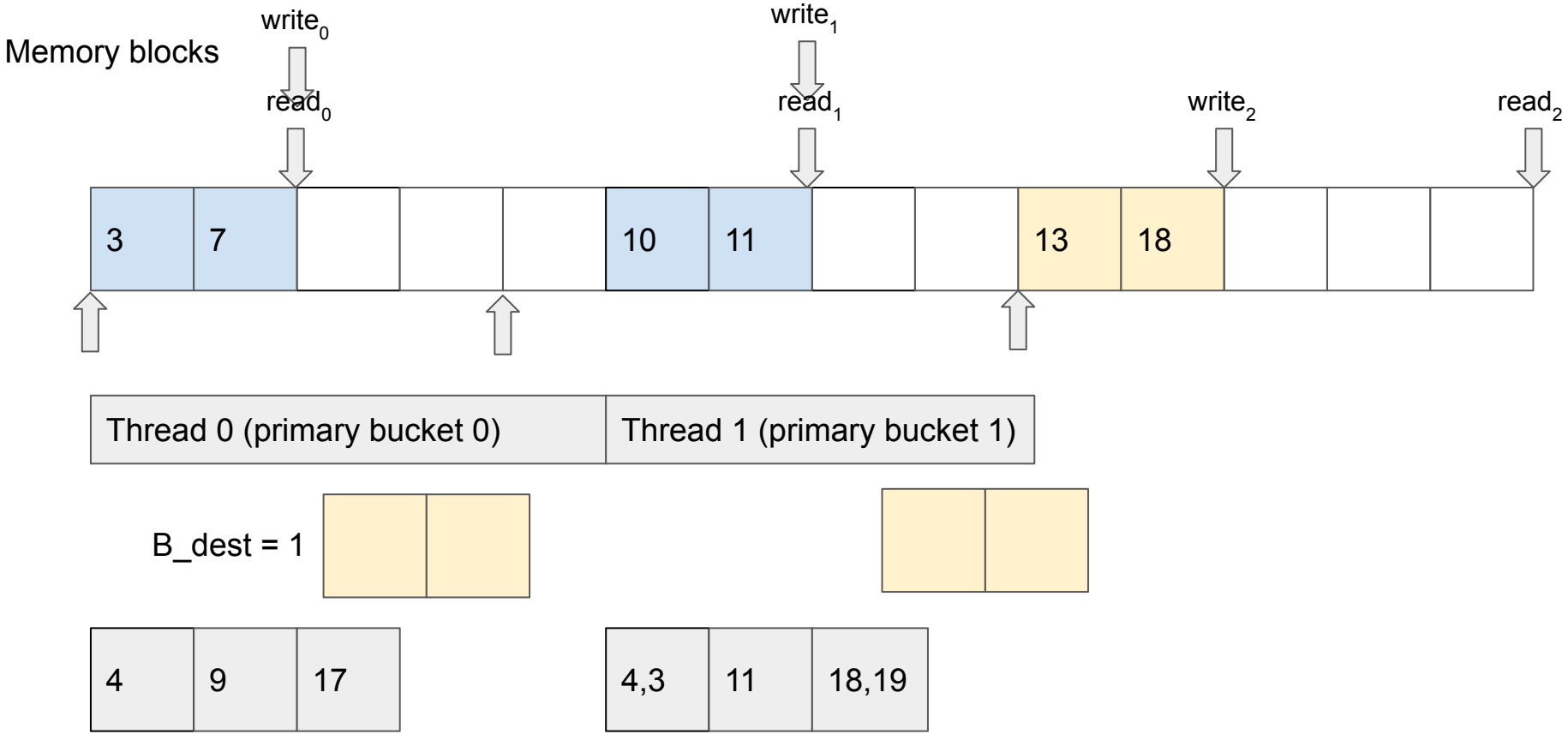
4	9	17

4,3	11	18,19

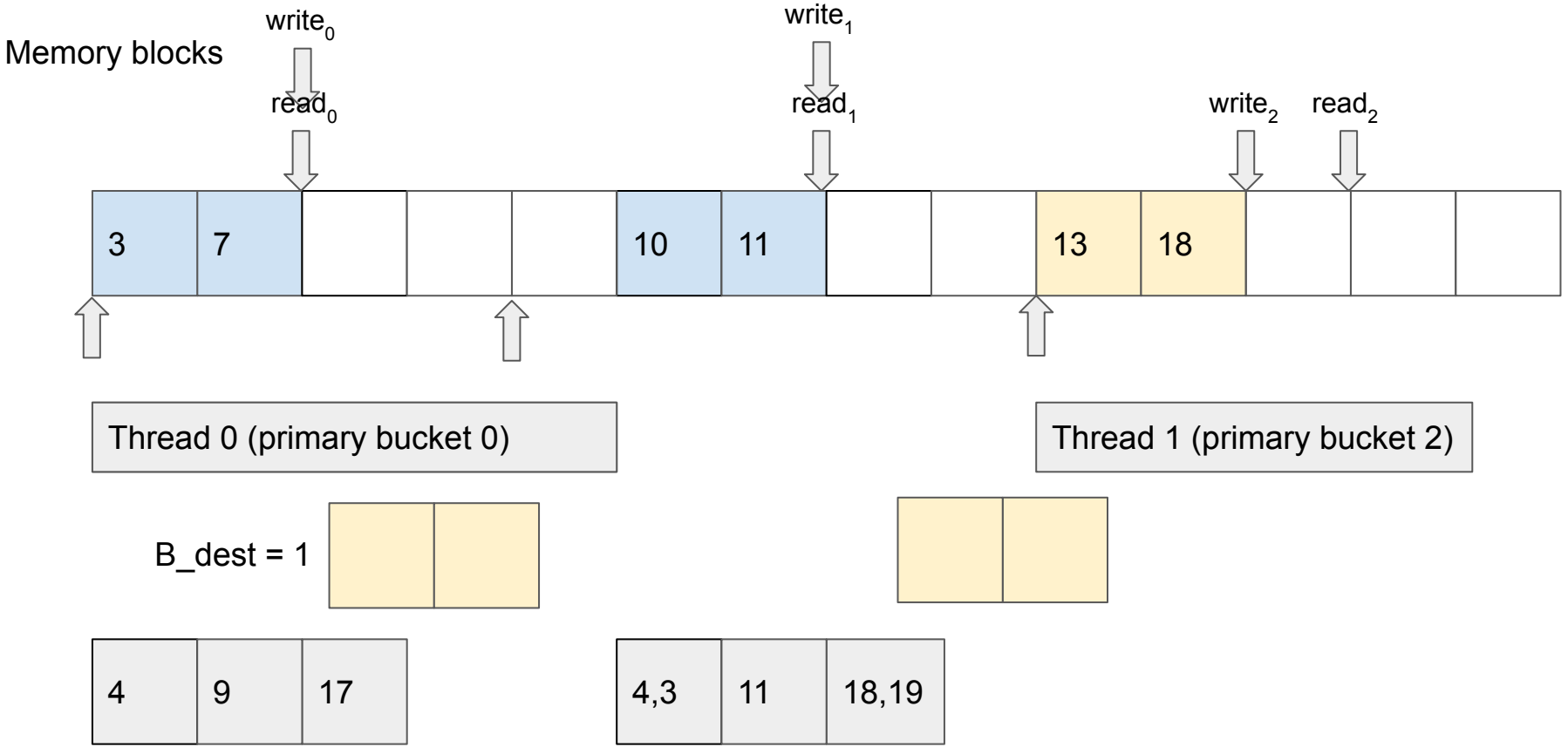
IPS^{4}O Block Permutation



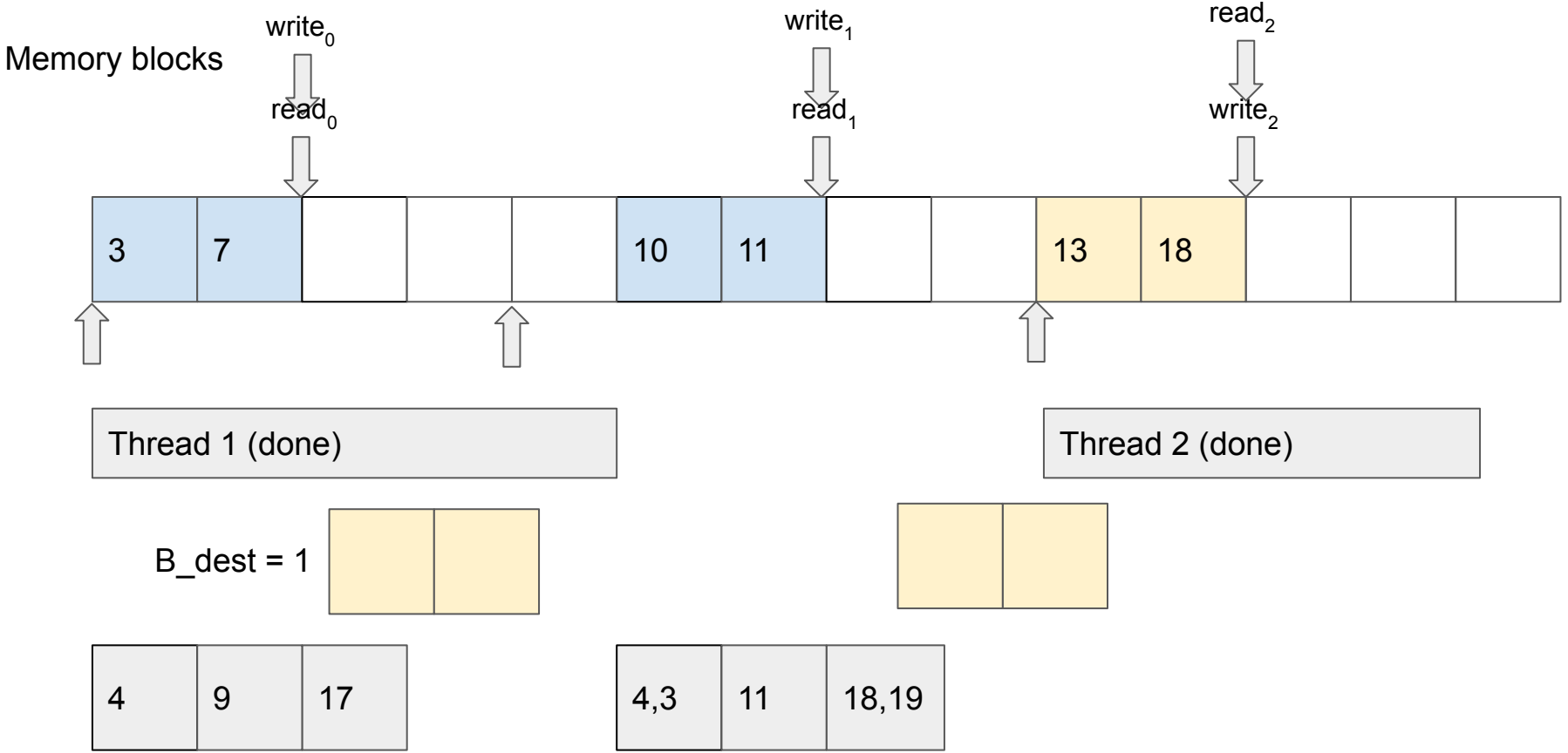
IPS^{4}O Block Permutation



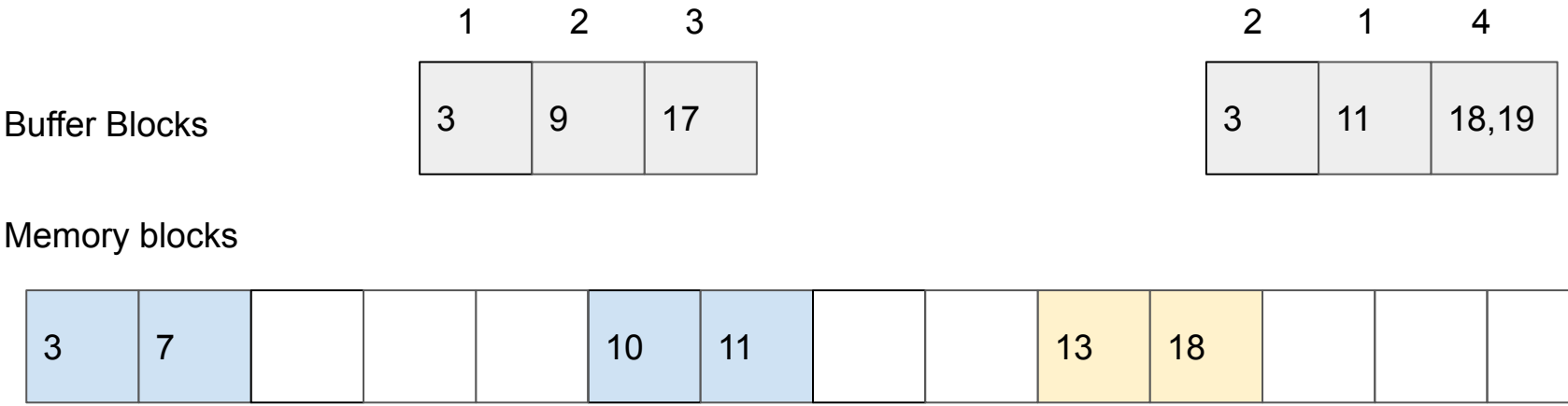
IPS^{4}O Block Permutation



IPS^{4}O Block Permutation



IPS^{4}O Block Permutation

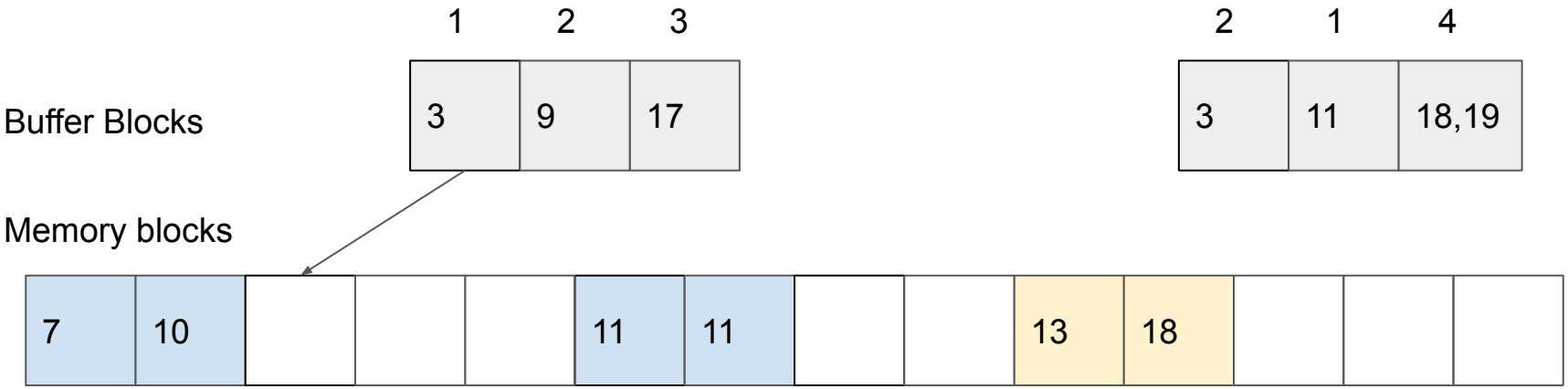


IPS^{4}O

4 Stages:

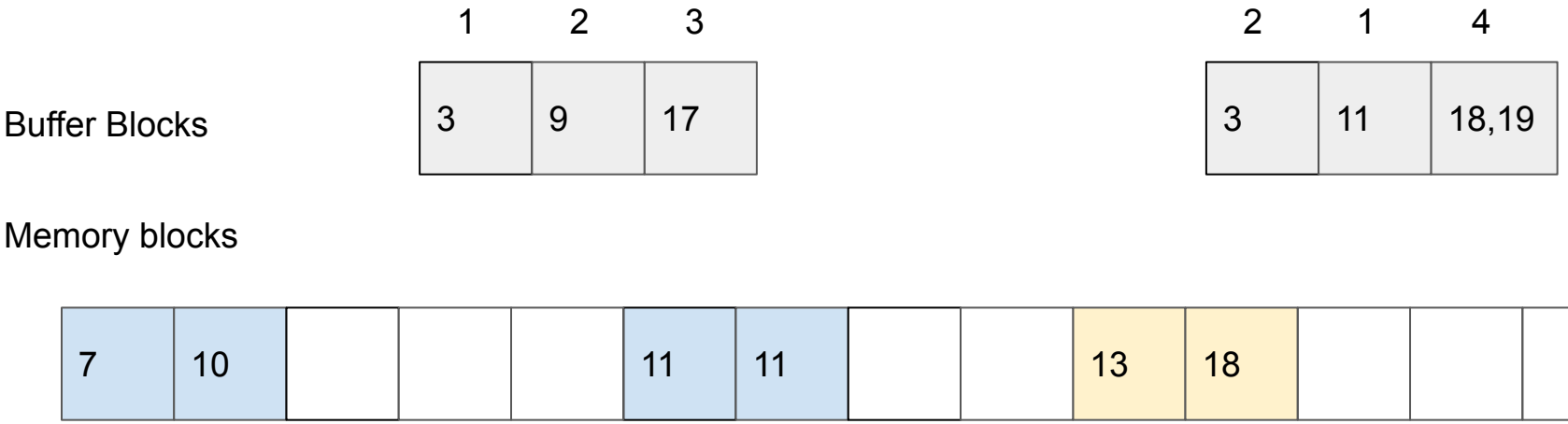
1. ~~Sampling: bucket boundaries~~
2. ~~Classification: Group input into blocks (in block, every elem in same bucket)~~
3. ~~Permutation: Globally order blocks~~
4. Cleanup: Clean up partially filled or crossing blocks

IPS^{4}O Block Permutation



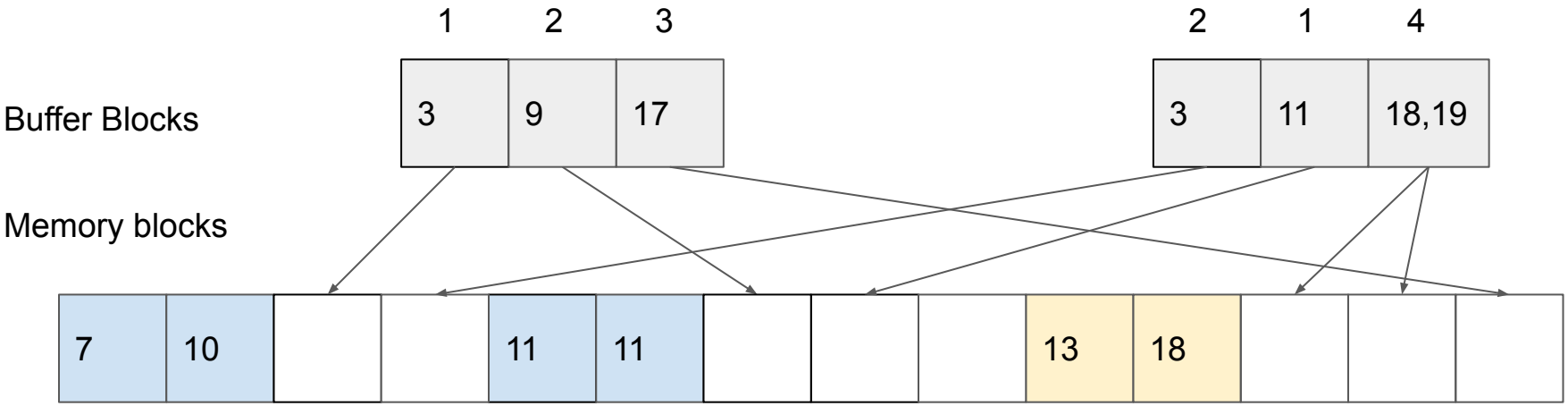
What's wrong with this array? (Yes this is a question)

IPS^{4}O Block Permutation



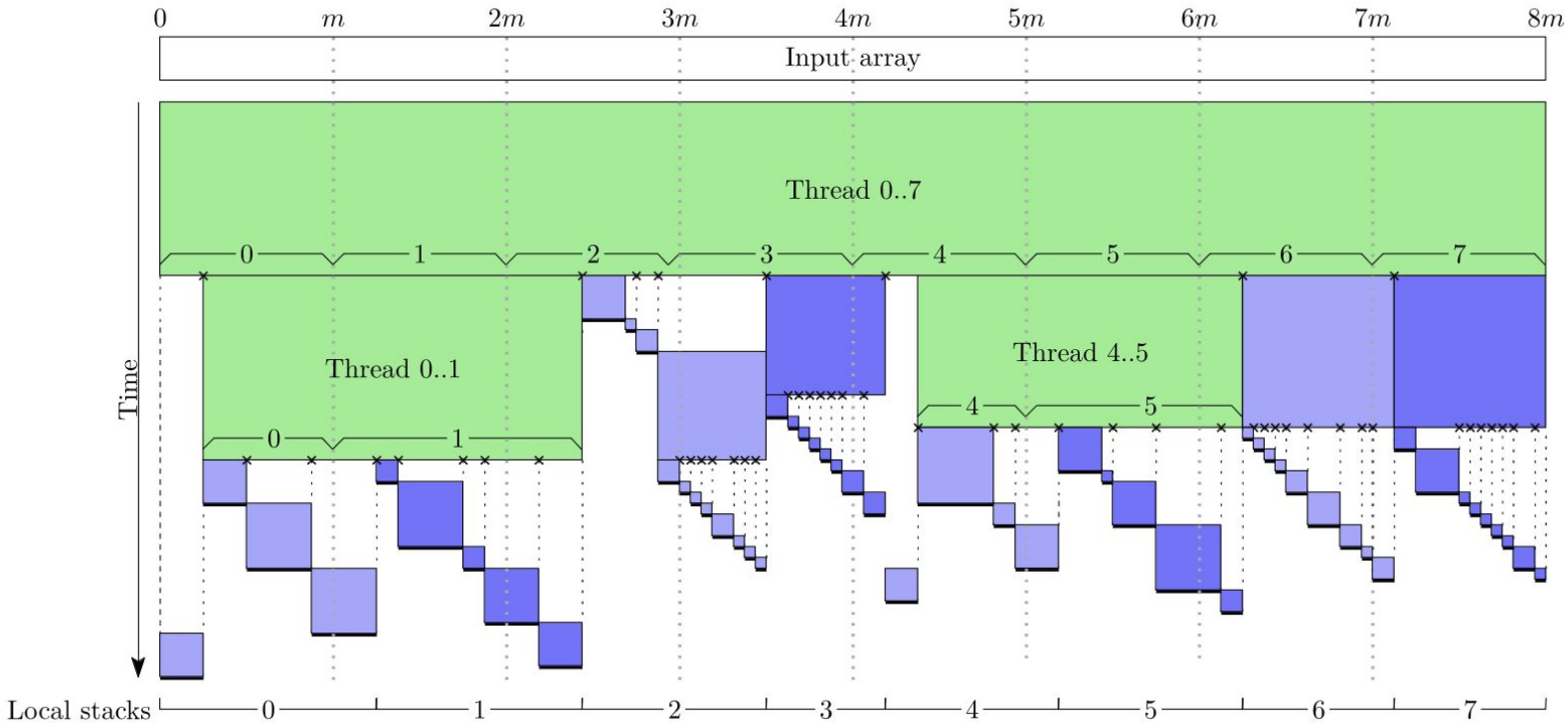
1. Bucket overlap
2. Partially filled buffers
3. Last bucket can be in swap buffer

IPS^{4}O Cleanup



1. Bucket overlap
2. Partially filled buffers
3. Last bucket can be in swap buffer

Recursion structure



Performance Hack: Implementation of pointer arithmetic

128-bit CAS instructions (if libatomic supports these), Mutex otherwise

Why 128 bit CAS?

Performance Hack: Implementation of pointer arithmetic

128-bit CAS instructions (if libatomic supports these), Mutex otherwise

Why 128 bit CAS -- Read and write stored in 64-bit pointers, must be updated together

> “The measurements reported in this paper were performed using somewhat non-portable implementations that use a 128-bit compare-and-swap instruction specific to x86 architectures (see also Section 6). Our portable variants currently use locks that incur noticeable overheads for inputs with only very few different keys. Different approaches can avoid locks without noticeable overhead but these would lead to more complicated source code.”

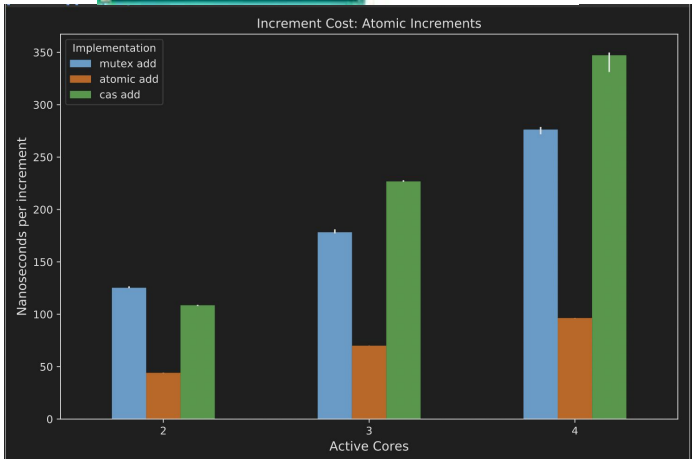
Performance Hack: Implementation of pointer arithmetic

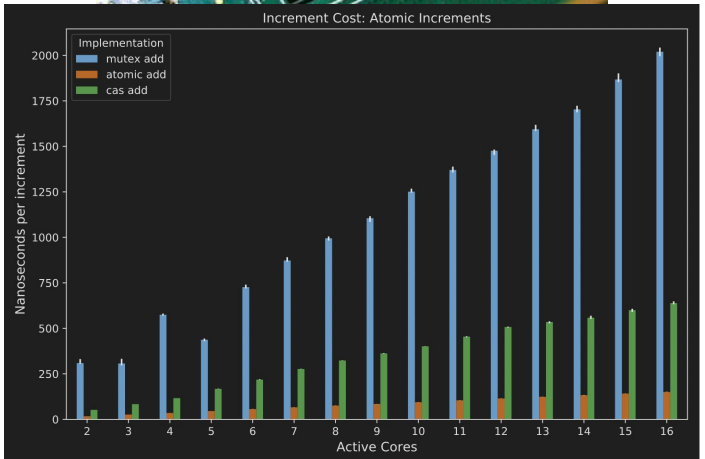
128-bit CAS instructions (if libatomic supports these), Mutex otherwise

Why 128 bit CAS -- Read and write stored in 64-bit pointers, must be updated together

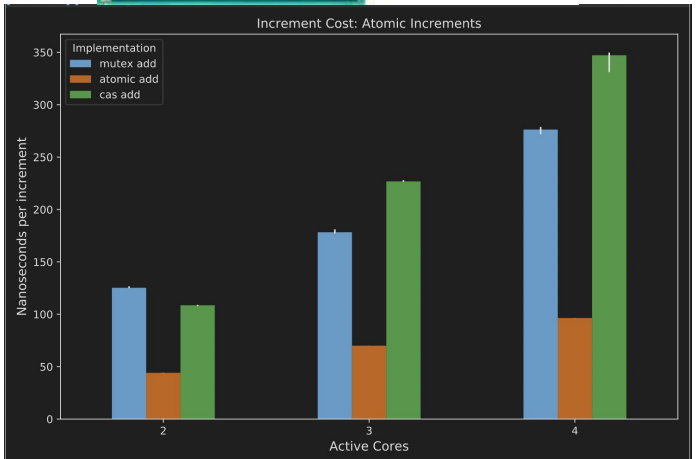
> “The measurements reported in this paper were performed using somewhat non-portable implementations that use a 128-bit compare-and-swap instruction specific to x86 architectures (see also Section 6). Our portable variants currently use locks that incur noticeable overheads for inputs with only very few different keys. Different approaches can avoid locks without noticeable overhead but these would lead to more complicated source code.”

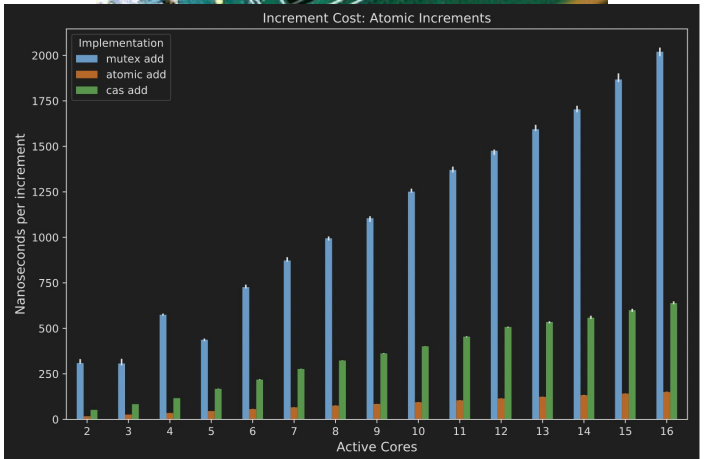
Performance Hack?: Implementation of pointer arithmetic -- does this matter?





Performance Hack?: Implementation of pointer arithmetic -- does this matter: It depends

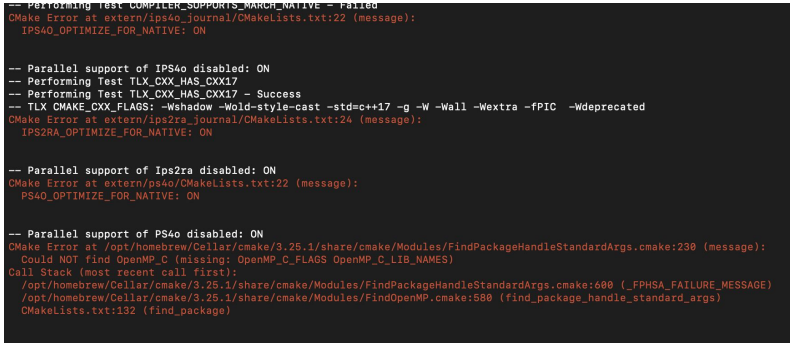




Performance and Portability Bugs

Try it yourself: https://github.com/ips4o/ips4o-benchmark-suite

“For the `run.sh` command, you need an installation of the Intel® Integrated Performance Primitives (IPP) as well as Cilk Plus. For Cilk Plus, you require a compiler supporting the Cilk Plus C++ language extension or you need provide your own Cilk Plus library which you add to the CMakeLists.txt file.”



Summary: What it takes to publish a paper on sorting these days

1. Incremental improvement on algorithm
2. Portable
3. 30 pages of analysis
4. Involved runtime analysis
5. Write your own scheduler
6. I/O analysis
7. *Branch mispredict analysis*
8. Base case optimization