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 IPS⁴O: 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

quicksort

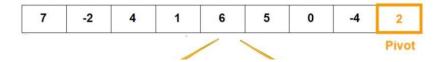
Strictly in-place

About 33,000 results (0.10 sec)

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

Quicksort review

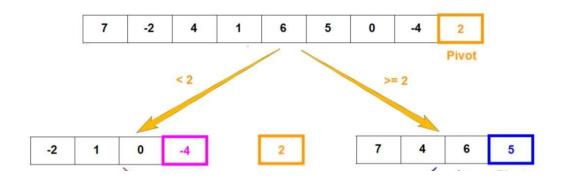
1: Choose Pivot



Quicksort review

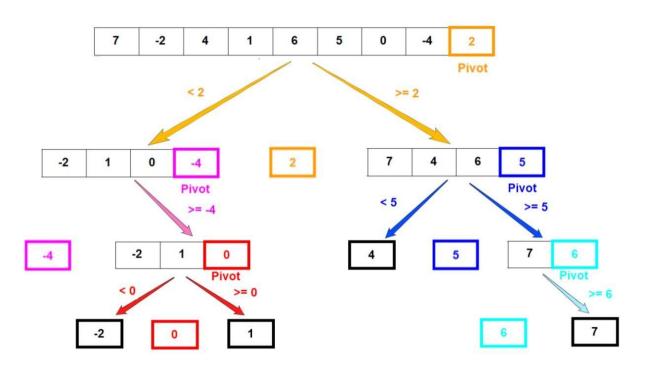
1: Choose Pivot

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 $s_0...s_{k-2}$ 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.

IPS⁴O

- 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

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

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

K = 3, a=2, ka - 1 elements

										_				
11	13	11	17	10	11	9	3	18	4	7	18	19	3	
														1

11	13	11	17	10	11	9	3	18	4	7	18	19	3
K = 3	3, a=2	., ka -	1 eler	nents									
11	13	11	17	10	11	9	3	18	4	7	18	19	3
11	3	7	11	10	17	9	13	18	4	11	18	19	3

11	13	11	17	10	11	9	3	18	4	7	18	19	3
K = 3	3, a=2	, ka -	1 eler	nents									
11	13	11	17	10	11	9	3	18	4	7	18	19	3
3	7	10	11	11	17	9	13	18	4	11	18	19	3

11	13	11	17	10	11	9	3	18	4	7	18	19	3
K = 3	3, a=2	, ka -	1 eler	nents									
11	13	11	17	10	11	9	3	18	4	7	18	19	3
3	7	10	11	11	17	9	13	18	4	11	18	19	3
K = 3	3, k - ´	l 1 splitt	ers (p	icked	equic	l listant	l ly)						
3	7	10	11	11	17	9	13	18	4	11	18	19	3

IPS⁴O Sampling

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

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

Create branchless decision tree, k buckets

7 11

Performance Hack: IPS⁴O 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}, \dots \rangle \text{ for } i := 1 \text{ to } n \text{ do } \text{ // locate each element}
j := 1 \text{ // current tree node } := \text{ root}
repeat \log k \text{ times } \text{ // will be unrolled}
j := 2j + (a_i > t_j) \text{ // left or right?}
j := j - k + 1 \text{ // bucket index}
|b_j| + + \text{ // count bucket size}
o(i) := j \text{ // remember oracle}
|b_1 \rangle b_2 \rangle b_3 \rangle b_4 \rangle b_5 \rangle b_6 \rangle b_7 \rangle b_8
```

Performance Hack: IPS⁴O 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}, \dots \rangle \text{ // for } i := 1 \text{ to } n \text{ do } \text{ // locate each element}
j := 1 \text{ // current tree node } := \text{ root}
repeat \log k \text{ times } \text{ // will be unrolled}
j := 2j + (a_i > t_j) \text{ // left or right?}
j := j - k + 1 \text{ // bucket index}
|b_j| + + \text{ // count bucket size}
o(i) := j \text{ // remember oracle}
|b_1 \rangle b_2 \rangle b_3 \rangle b_4 \rangle b_5 \rangle b_6 \rangle b_7 \rangle b_8
```

Performance Hack: IPS⁴O 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"

IPS4O

- 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

t = 2, split into *t* "stripes"

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

t = 2, split into *t* "stripes"

3	7	10	11	11	17	4
---	---	----	----	----	----	---

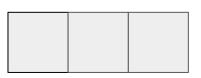
t = 2, split into *t* "stripes"

3	7	10	11	11	17	4
---	---	----	----	----	----	---

k=3, each thread has *k* "buffer blocks"

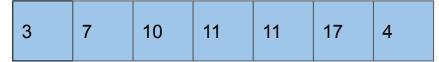


t = 2, split into *t* "stripes"

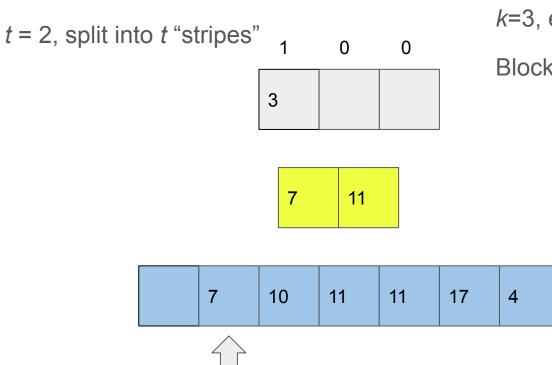


7 11

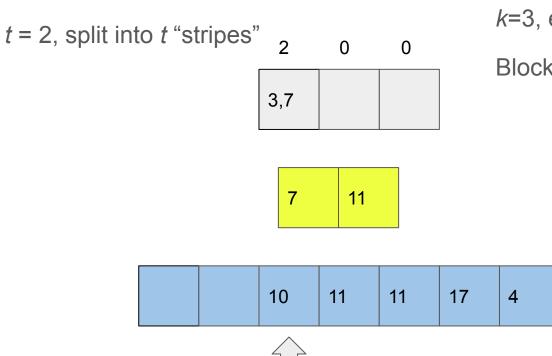
k=3, each thread has *k* "buffer blocks"



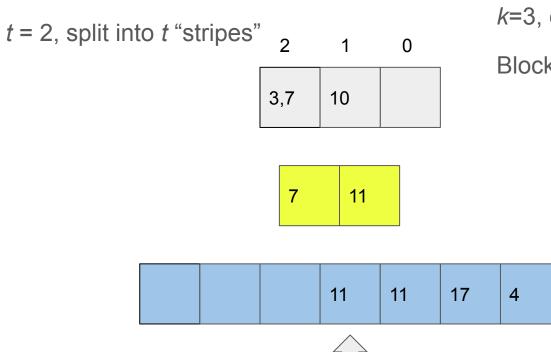




k=3, each thread has *k* "buffer blocks"



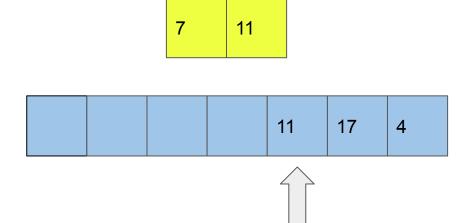
k=3, each thread has *k* "buffer blocks"



k=3, each thread has *k* "buffer blocks"

t = 2, split into t "stripes" $\begin{bmatrix} 2 & 2 & 0 \\ 3,7 & 11 \end{bmatrix}$

k=3, each thread has *k* "buffer blocks"

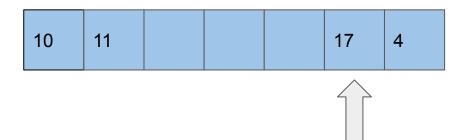


t = 2, split into t "stripes" $\begin{bmatrix} 2 & 3 & 0 \\ 3,7 & 11 \end{bmatrix}$

k=3, each thread has *k* "buffer blocks"

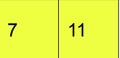
k=3, each thread has *k* "buffer blocks"

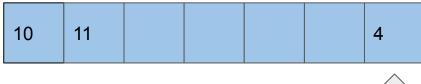
7 11



t = 2, split into t "stripes" $\begin{bmatrix} 2 & 3 & 1 \\ 3,7 & 11 & 17 \end{bmatrix}$

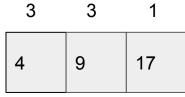
k=3, each thread has k "buffer blocks"



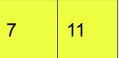


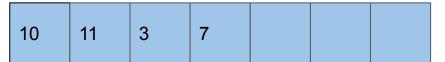


t = 2, split into *t* "stripes"



k=3, each thread has k "buffer blocks"



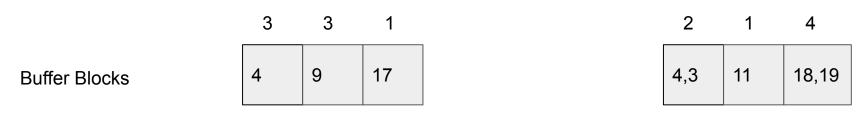




IPS4O

- 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

IPS⁴O Block Permutation

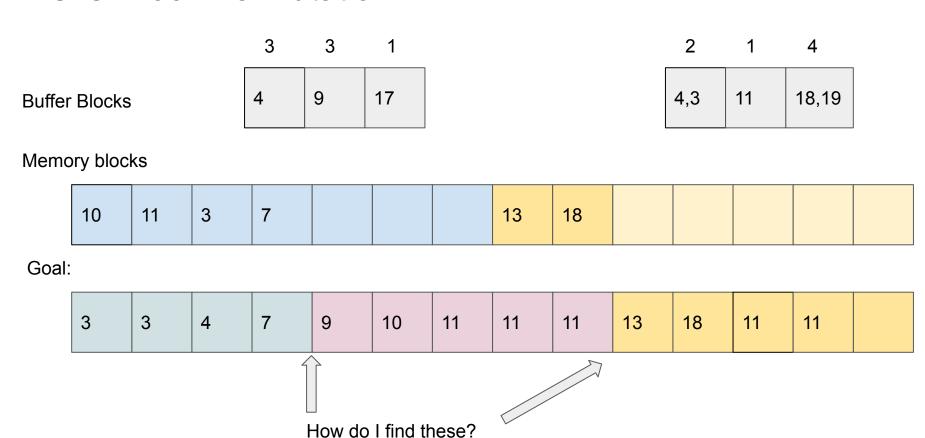


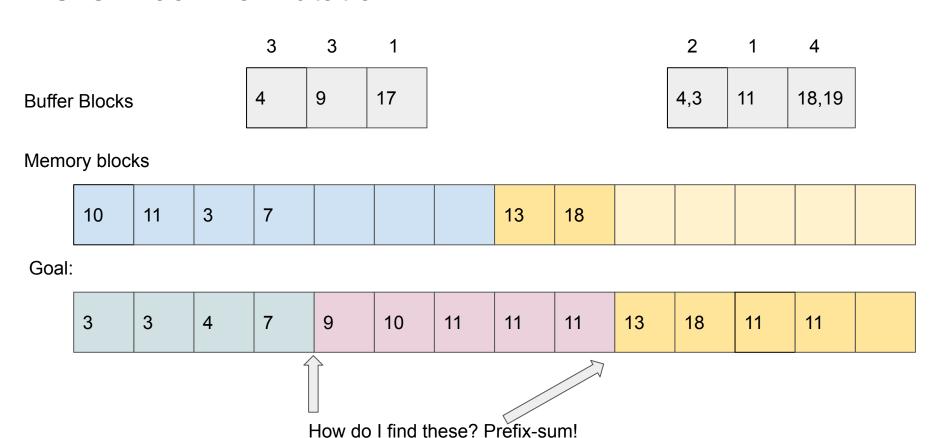
Memory blocks

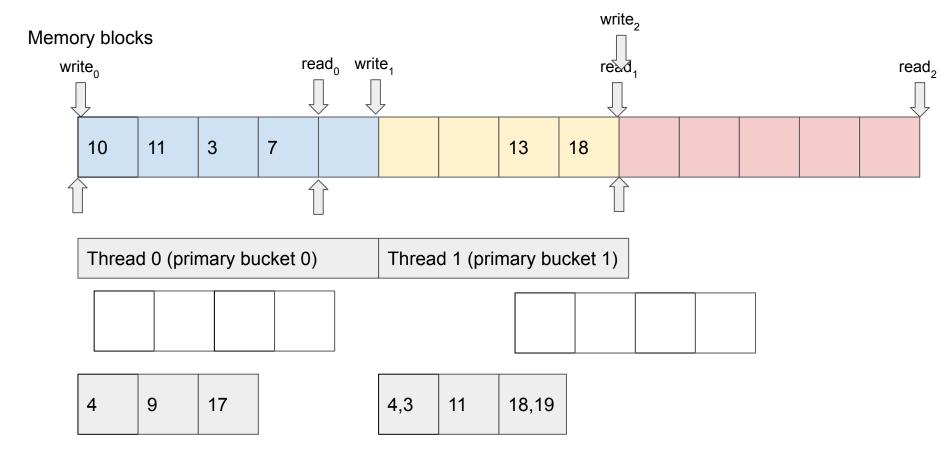
Goal:

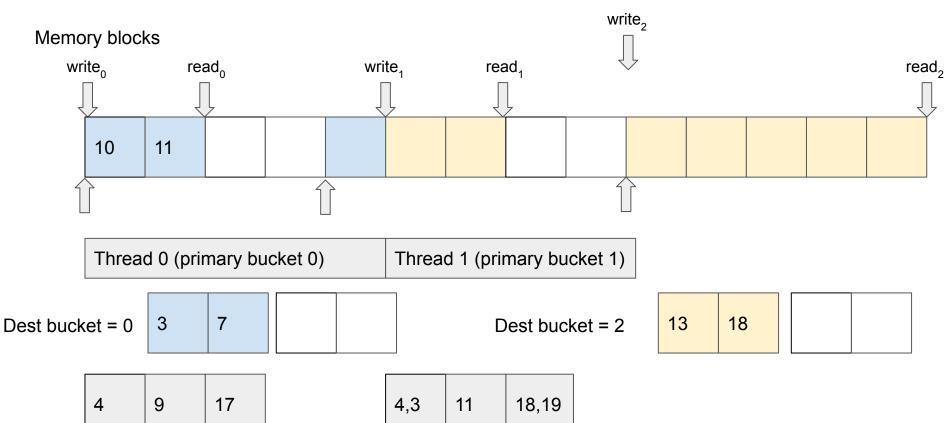
3	3	4	7	9	10	11	11	11	13	18	11	11		
---	---	---	---	---	----	----	----	----	----	----	----	----	--	--

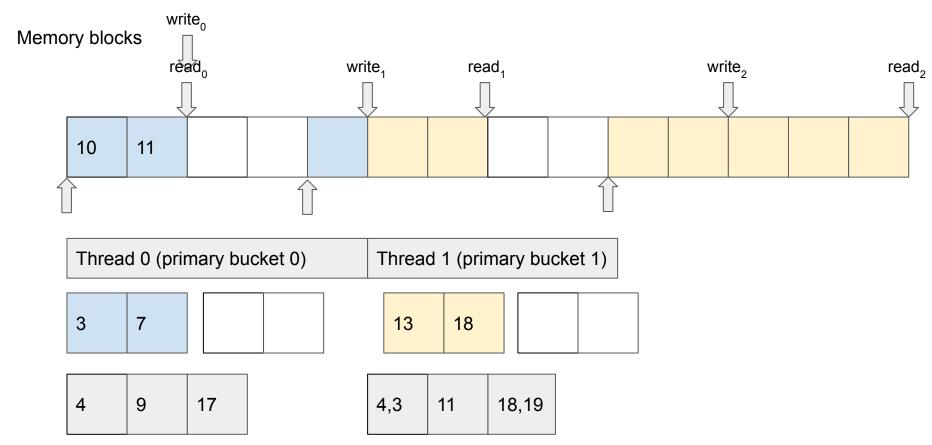
IPS⁴O Block Permutation

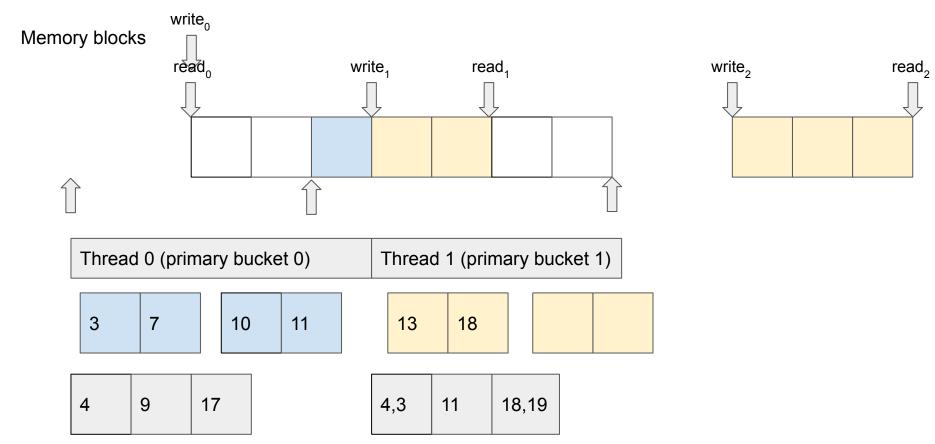


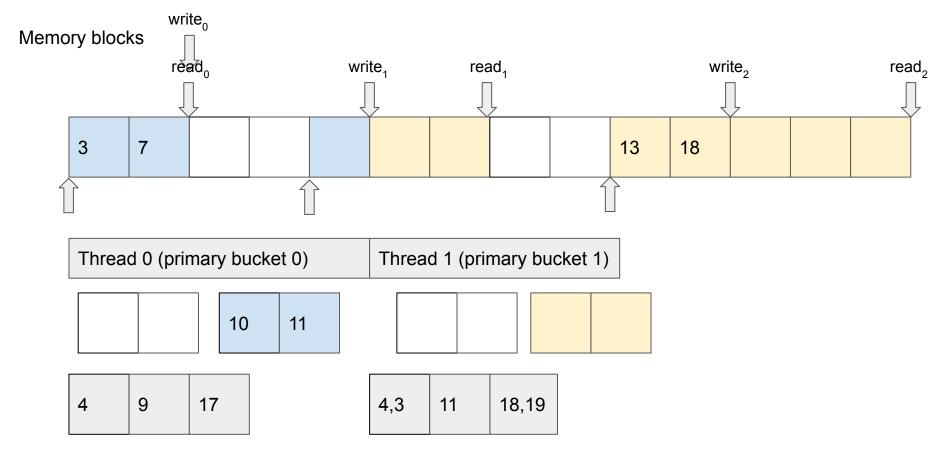








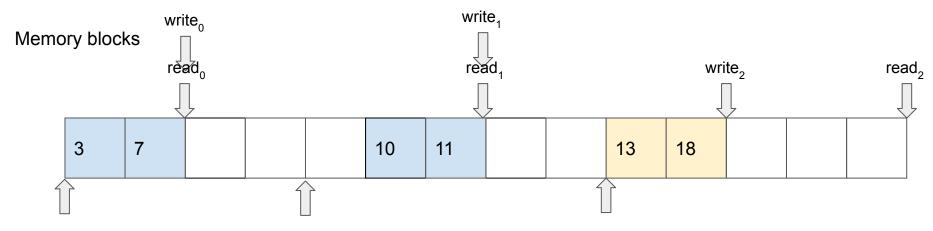




9

4

17

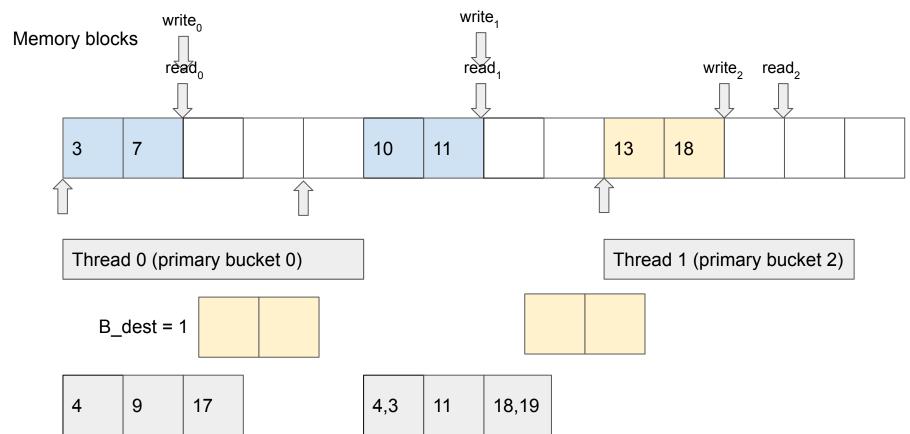


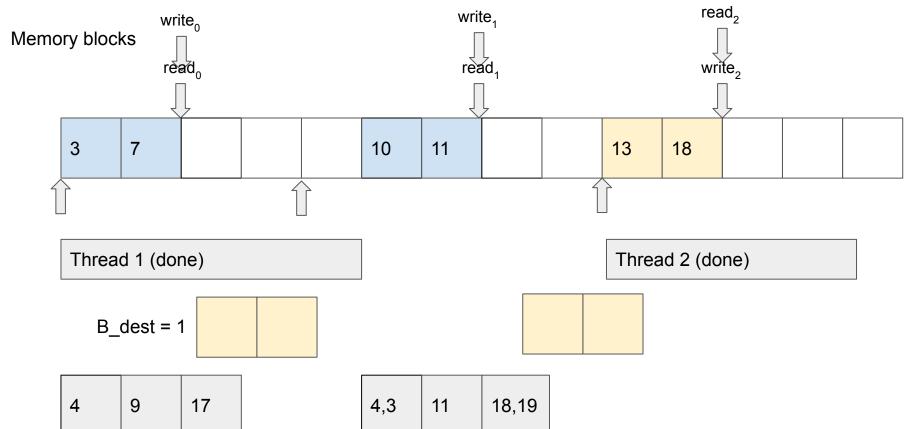
Thread 0 (prima	ary bucket 0)	Thread 1 (primary bucket 1)					
B_dest = 1							

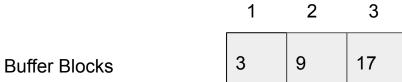
4,3

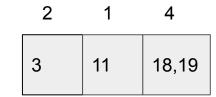
11

18,19







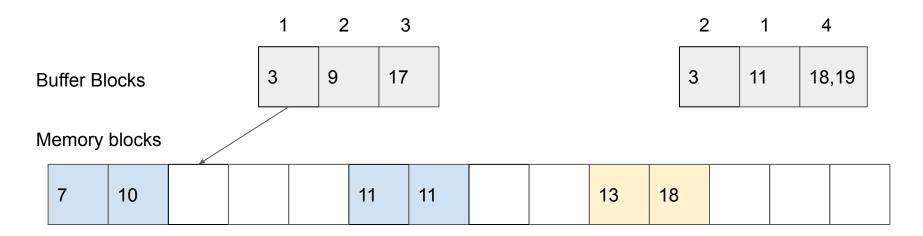


Memory blocks

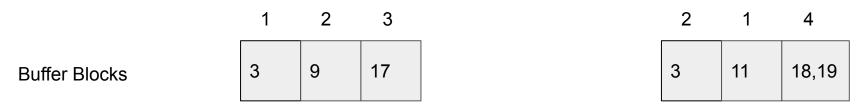
3	7				10	11			13	18			
---	---	--	--	--	----	----	--	--	----	----	--	--	--

IPS4O

- 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



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

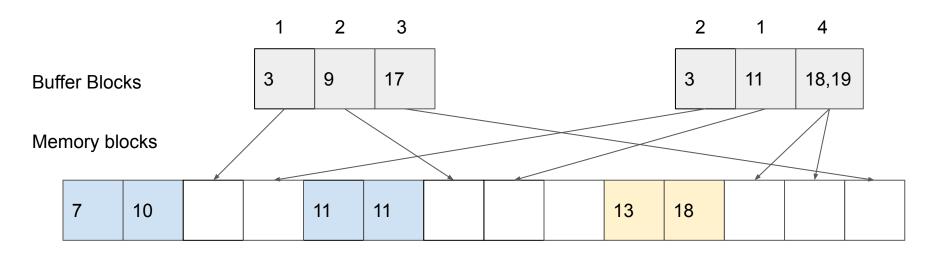


Memory blocks

7	10		11	11		13	18		

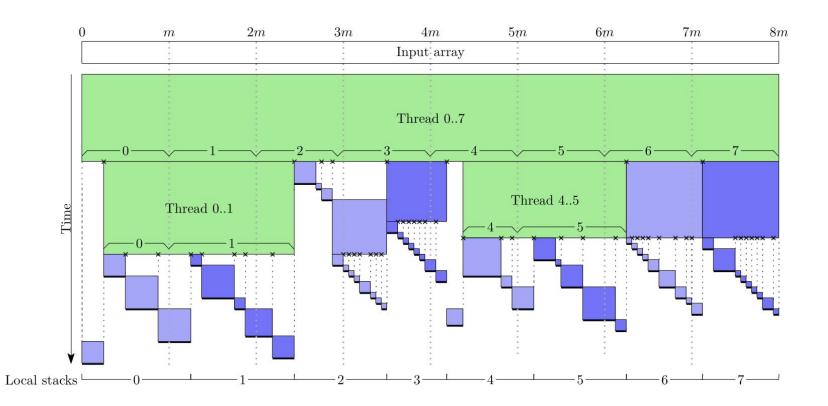
- 1. Bucket overlap
- 2. Partially filled buffers
- 3. Last bucket can be in swap buffer

IPS⁴O 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

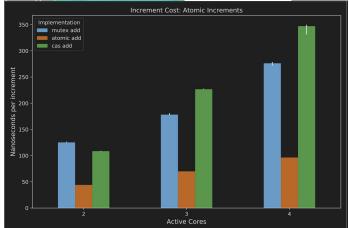
Why 128 bit CAS -- Read and write stored in 64-bit pointers, must be updated together

"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."

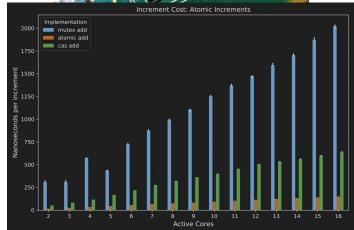
Performance Hack?: Implementation of pointer arithmetic

-- does this matter?





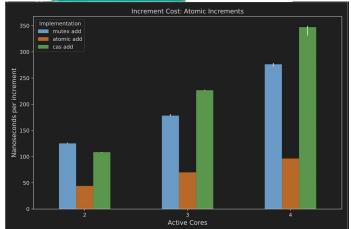




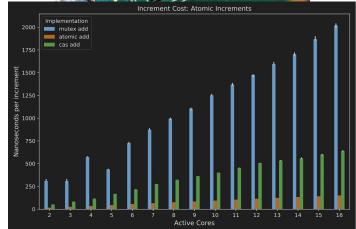
Performance Hack?: Implementation of pointer arithmetic

-- 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."

```
--- Perforeing iest CUMPLICE, SUPPORTS, MARCH, MATIYE - Fals8

CMake Error at extern/insed_journal/CMakeLists.txt:22 (message):

IPSAO_OPTIMIZE_FOR_NATIVE: ON

--- Parallel support of IPSAo disabled: ON

--- Perforeing Test TLX_CXX_HAS_CXX17 - Success

--- TLX_CMAKE_CXX_FLAGS: -Wshadow -Wold-style-cast -std=c++17 - g -W -Wall -Wextra -fPIC -Wdeprecated

CMake Error at extern/ipsZra_journal/CMakeLists.txt:24 (message):

IPSZRA_OPTIMIZE_FOR_NATIVE: ON

--- Parallel support of IpsZra disabled: ON

CMake Error at extern/psAo/CMakeLists.txt:22 (message):

PSAO_OPTIMIZE_FOR_NATIVE: ON

--- Parallel support of PSAo disabled: ON

CMake Error at extern/psAo/CMakeLists.txt:22 (message):

COLUB NOT find Denning C (missing) openNP_C_FLAGS OpenNP_C_UTB_NAMES)

Call Stack (most recent call first):

/opt/homebrew/Callar/cmake/3.25.1/share/cmake/Modules/FindOpenNP_cmake:S80 (find_package_handle_standard_args.

CMakeLists.txt:32 (find_package)
```

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