Engineering In-place (Shared-memory) Sorting Algorithms

By Axtmann, Witt, Ferizovic, Sanders

Reviewed by Kasra Mazaheri

Definitions and Foundation

- **Strictly In-place Sorting:** constant additional memory
- **In-place Sorting:** logarithmic additional memory
- **PEM Model:** threads are assumed to have private cache with atomic operations (e.g., *fetch_and_add*)
- **Samplesort:** k-way generalization of quicksort, where the input is divided into k smaller subproblems based on k-1 pivots
- **Super Scalar Samplesort (S4o):** a variant of Samplesort based on *branchless decision trees*

Related Work and State of the World

- **Quicksort and its Variants:** industry standard, in-place
  - Parallel Quicksort by Tsigas [74]
  - Branchless execution by Edelkamp [23] (BlockQuicksort)
- **Samplesort:** not in-place but with better parallelism and cache-efficiency
- **Super Scalar Samplesort (S4o):** avoids branch misprediction, much faster
- **Radix Sort:** in-place, parallel, limited data types
  - Parallel Radix Sort but with high-contention (SkaSort)
  - Parallel Radix Sort by Orestis [64]
- **QuickMergesort (QMSort):** strictly in-place, non-stable sorting by Edelkamp [24]

IPS4o Algorithm

1. For small arrays, use a base sorting algorithm.
2. Otherwise, k-way partition the array *somehow reasonably* .
3. Solve the subproblems recursively.

IPS4o Algorithm

Partitioning process:

1. **Sampling:** draw k *splitters* to partition the array and find *bucket boundaries*.

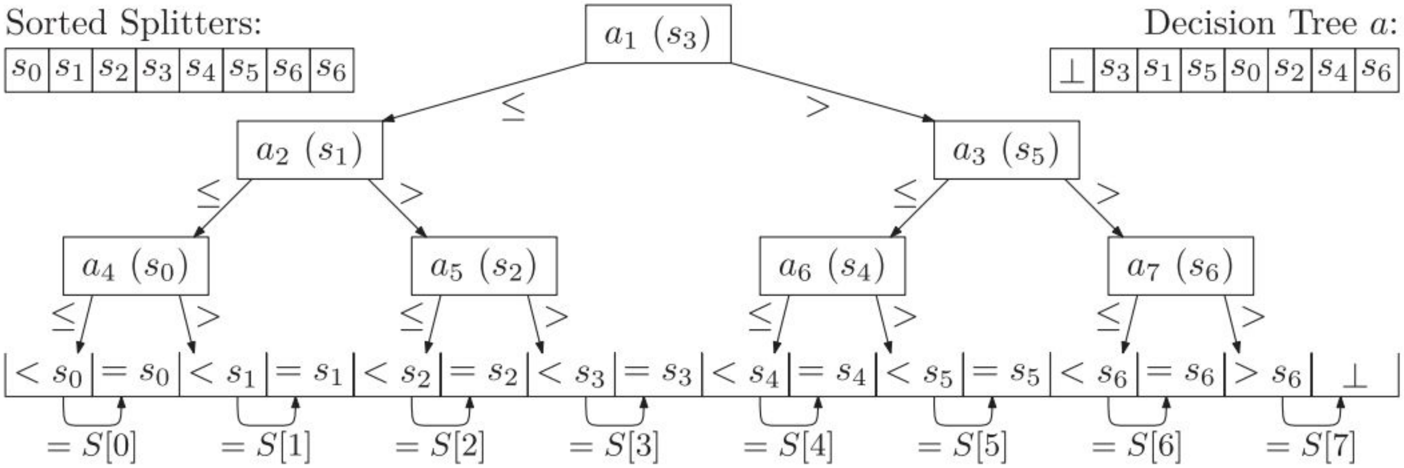
2. **Classification:** group elements into *blocks* based on their bucket using *local buffers* for parallel processing.

3. **Block Permutation:** rearrange blocks into their correct order using atomic operations for thread coordination.

4. **Cleanup:** handle *wildcards*, i.e., elements crossing bucket boundaries.

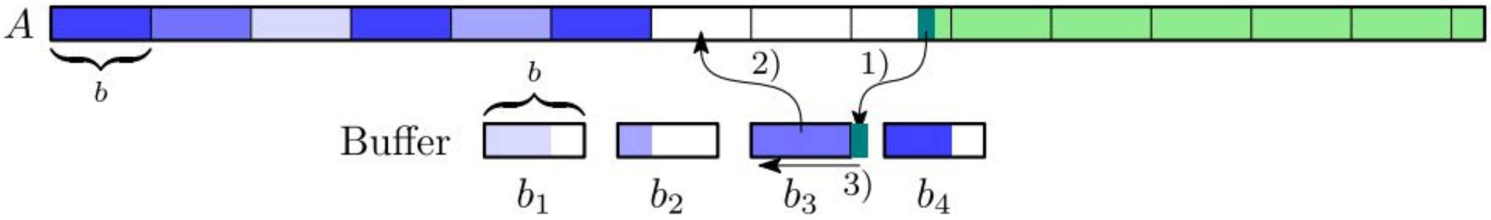
IPS4o Algorithm: Sampling

1. Sample αk - 1 elements and sort them. **α** is the **oversampling** factor.
2. Choose k-1 splitters equidistantly from the sorted sample and remove duplicates.
3. Build the decision tree (as done in S4o), potentially with *equality buckets*.



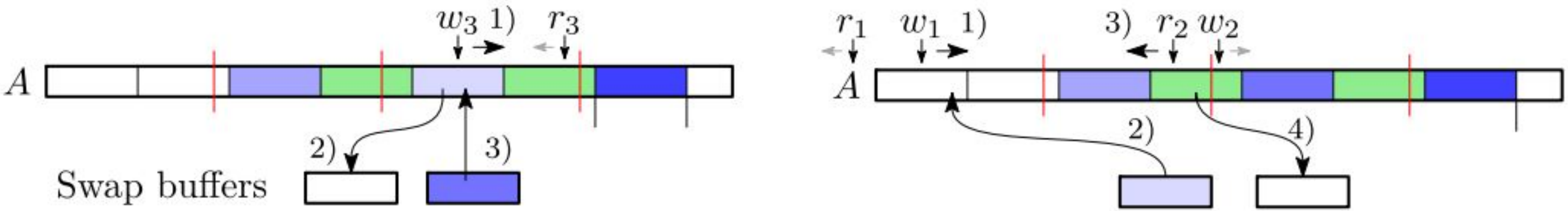
IPS4o Algorithm: Classification

1. The input is interpreted as blocks of size ***b***, and is divided into ***t*** stripes for parallel processing. Each thread then has an array of ***k*** *buffer blocks* of size ***b***.
2. Using the search tree, each thread **classifies** each element into the buffer block corresponding to its bucket.
3. If full, the buffer block is written to the stripe, then the element is placed in the buffer. This way, blocks in the memory will belong to the same bucket.
4. Bucket sizes are maintained for threads, and aggregated in the end to compute boundaries for buckets.



IPS4o Algorithm: Block Permutation

- Essentially, now swap blocks one-by-one to place them in the correct bucket.
- To allow parallelism, use atomic read and writes pointers for each bucket.
- The cost for these pointers are offsetted by using a **large** block size.

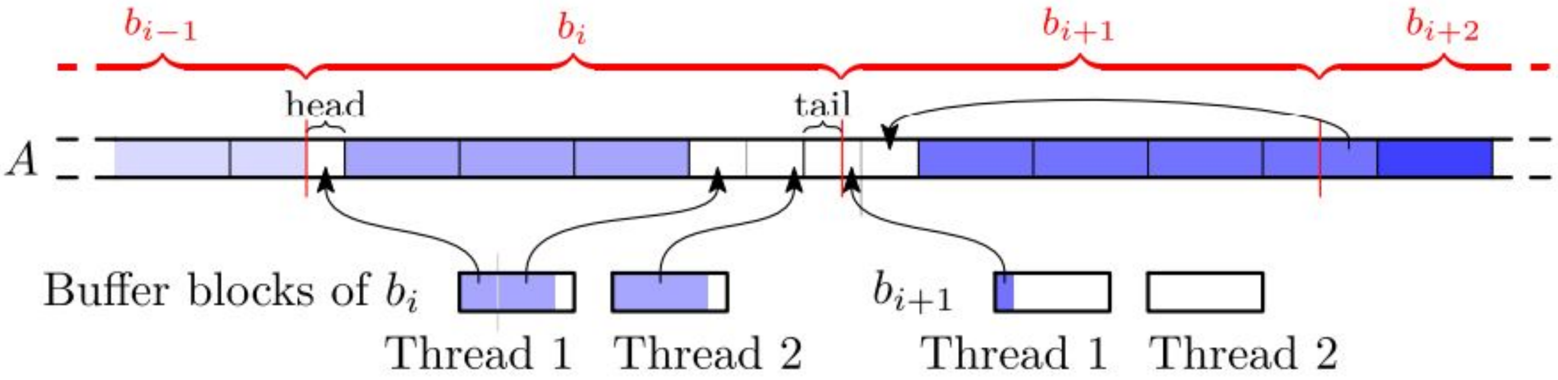


(a) Swapping a block into its correct position.

(b) Moving a block into an empty position, followed by re-filling the swap buffer.

IPS4o Algorithm: Clean up

- Since the algorithm processes elements in blocks, it’s possible to have misplaced elements in blocks that span multiple buckets.
- There might also be *leftover* elements in the classification buffers.
- Essentially, carefully move these elements one at a time.



IPS4o Task Scheduler

Components:

- **Static load balancing:** to evenly divide the resources amongst task.
- **Dynamic rescheduling:** to utilize idle threads.

Complexity Analysis

- **Memory Requirement:**
  - Theoretically, IPS4o can use as little as $O(kb)$ additional memory per thread (for k buffer blocks, as well as 2 swap blocks). This follows the same logic as the *strictly in-place quicksort* [22].

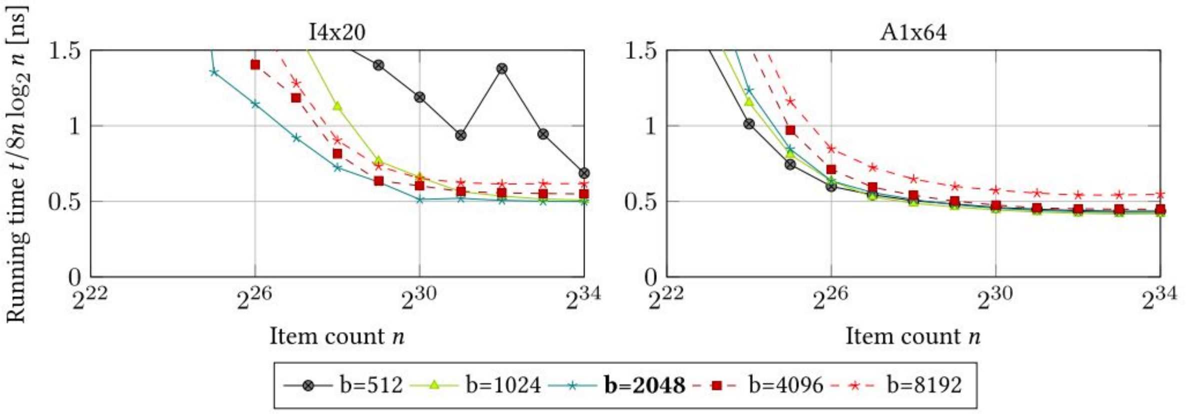
  - Practically, IPS4o uses local stacks, resulting in $O(k(b + t \log (n/n0)))$ additional memory per thread.

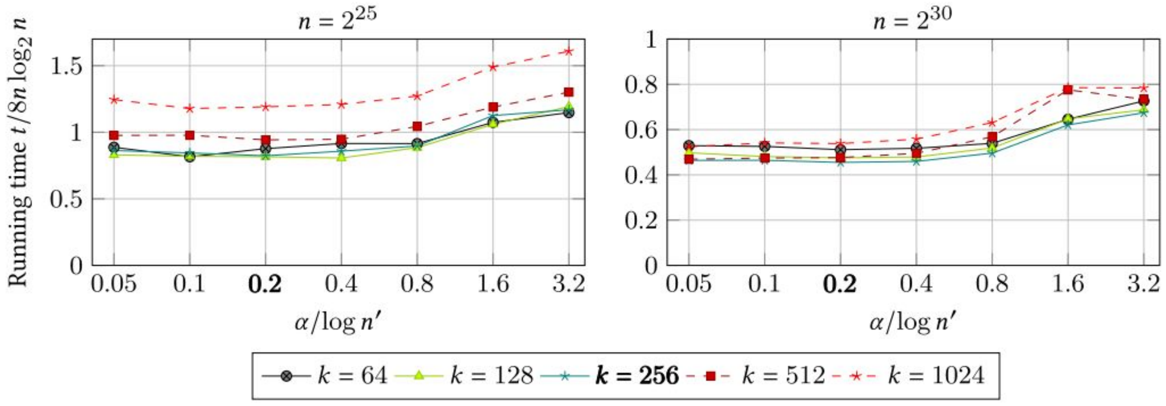
- **Work Complexity:** IPS4o has total work of $O(n \log n)$ with probability 1-4/n.

Tuning *b, k, a*

Results:

- **Block size** of ***b=2048*** showed good performance across most inputs and machines.
- **Bucket size** of ***k=256*** showed the best performance.
- **Oversampling size** of ***a=0.2 log n'*** works best in practice.





Results

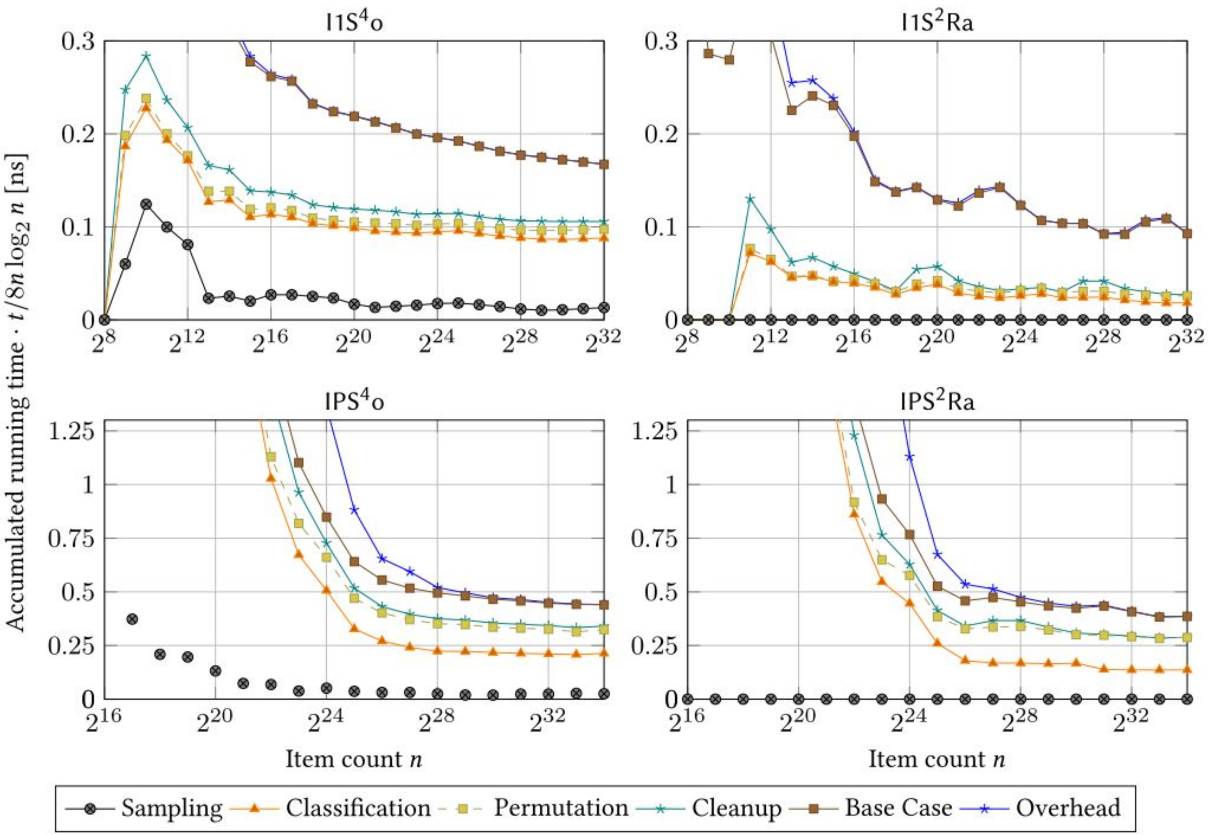
Running Time across Phases of the Algorithm

About the data:

- **uint64** values
- **Uniform** Distribution

Notably:

- **The permutation phase** takes considerably more (about 11-20x) time for parallel implementations, due to memory bottlenecks.
- **The overhead** remains a significant part of the running time, especially for smaller inputs.



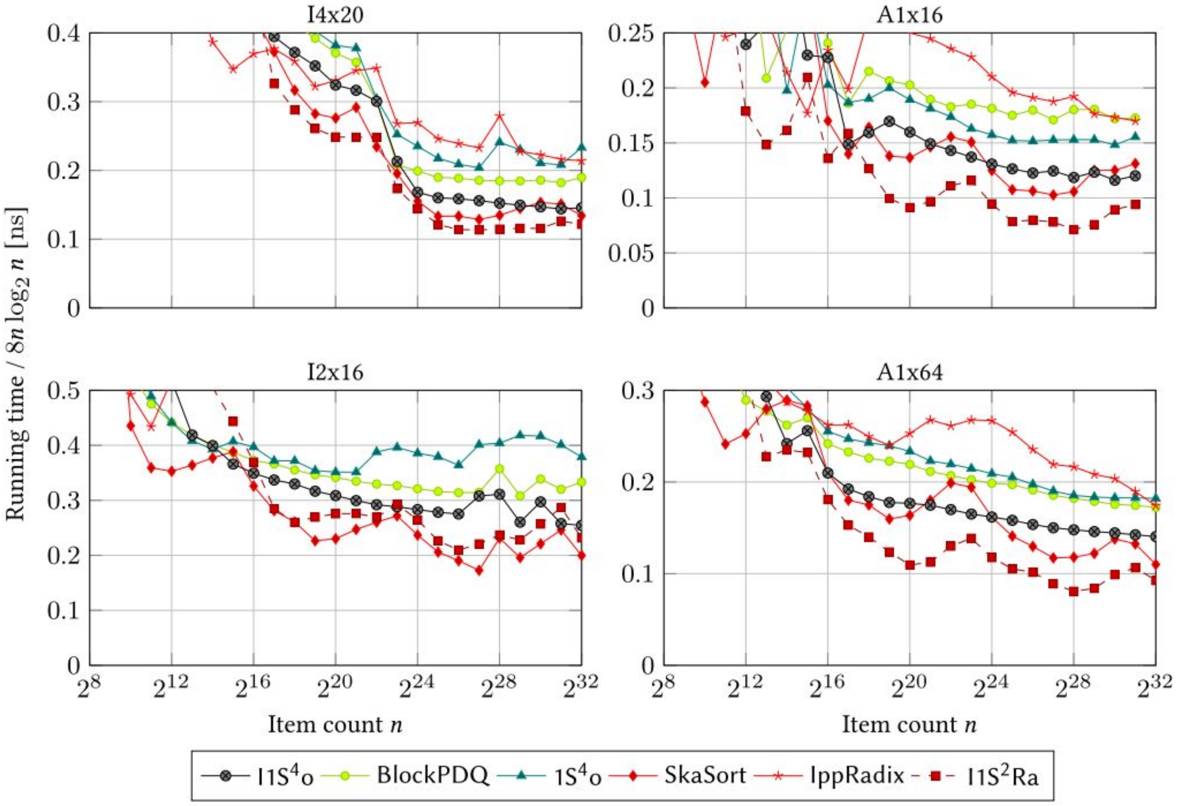
Results

Sequential Algorithms

About the data:

- **uint64** values
- **Uniform** Distribution

DualPivot, *std::sort* and *QMSort* not displayed as they their running times exceeded the plot.

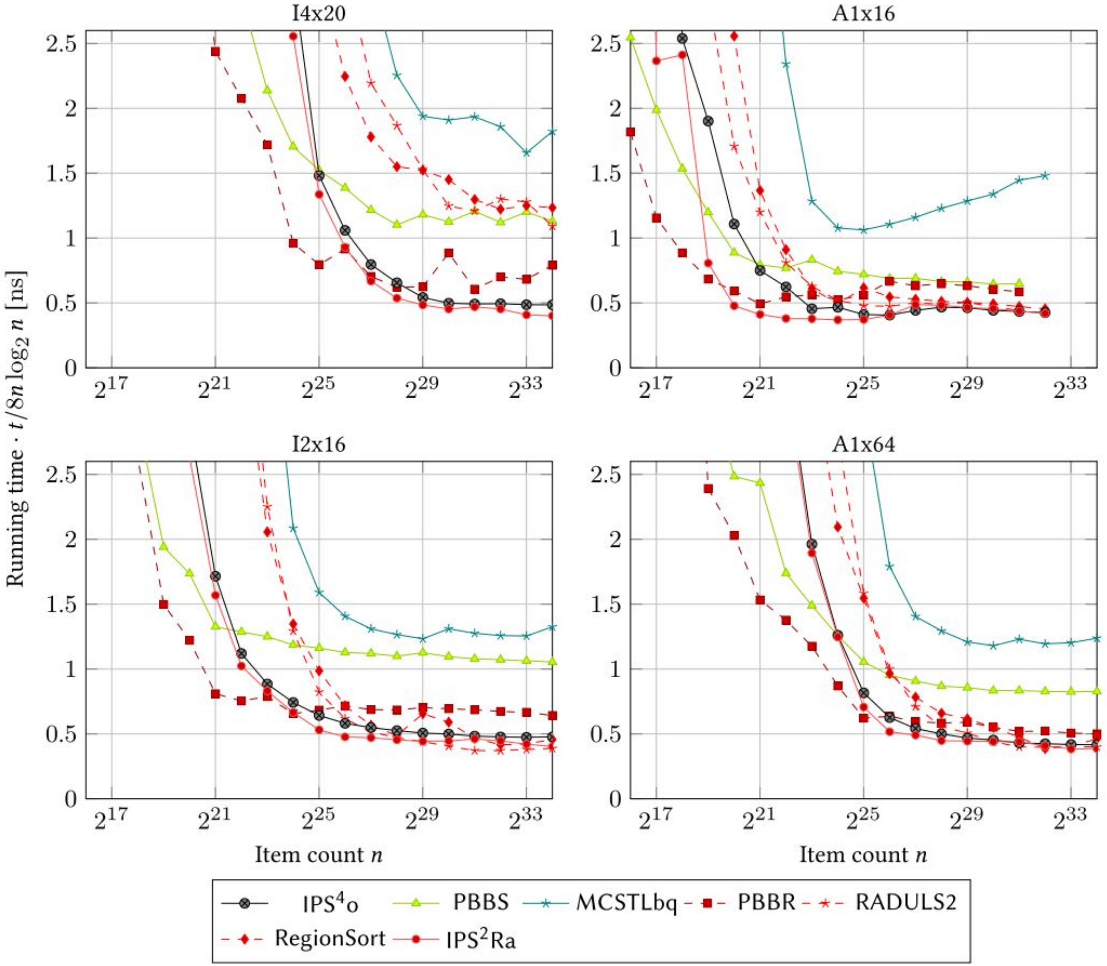


Results

Parallel Algorithms

About the data:

- **uint64** values
- **Uniform** Distribution



Results

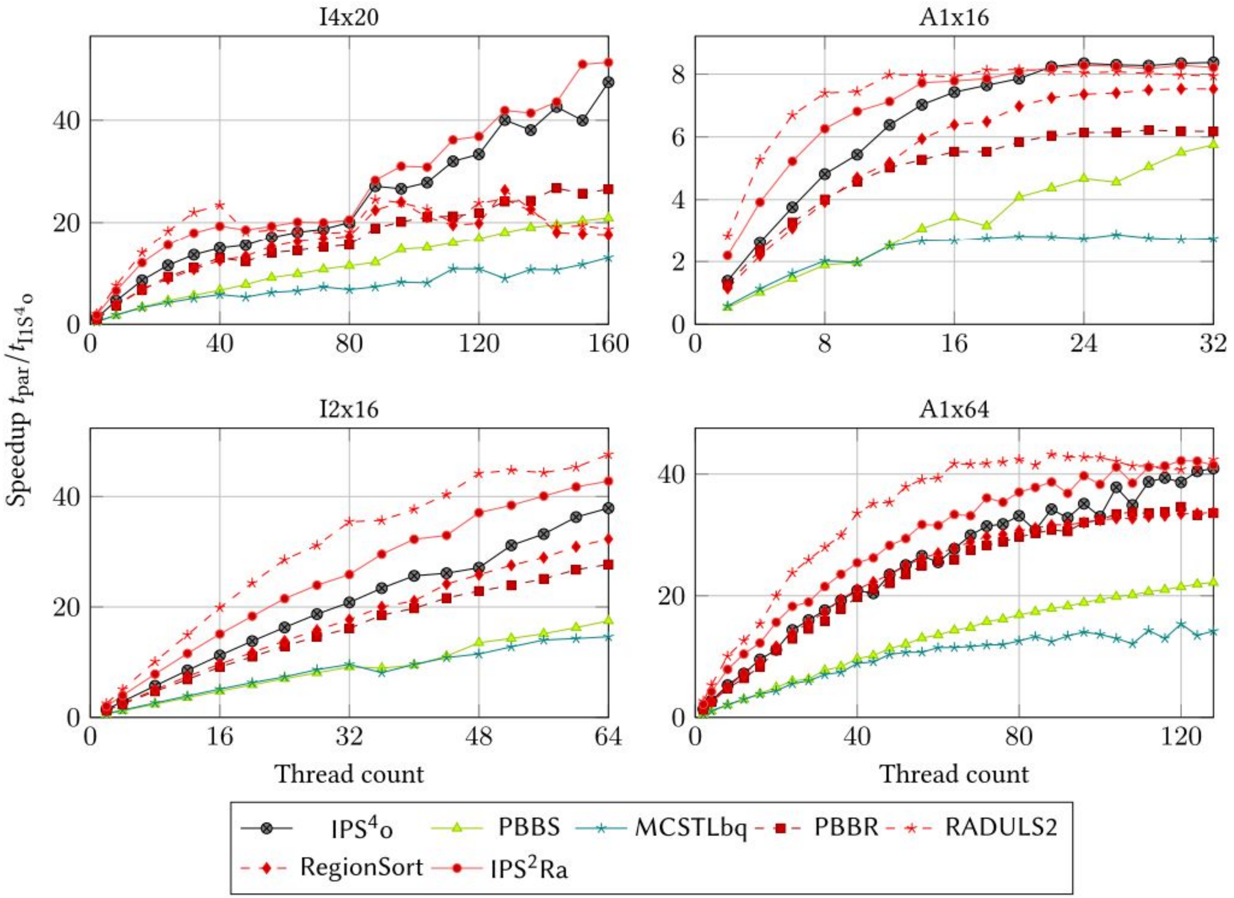
Parallel Speed-up

About the data:

- **2^30 elements**
- **uint64** values
- **Uniform** Distribution

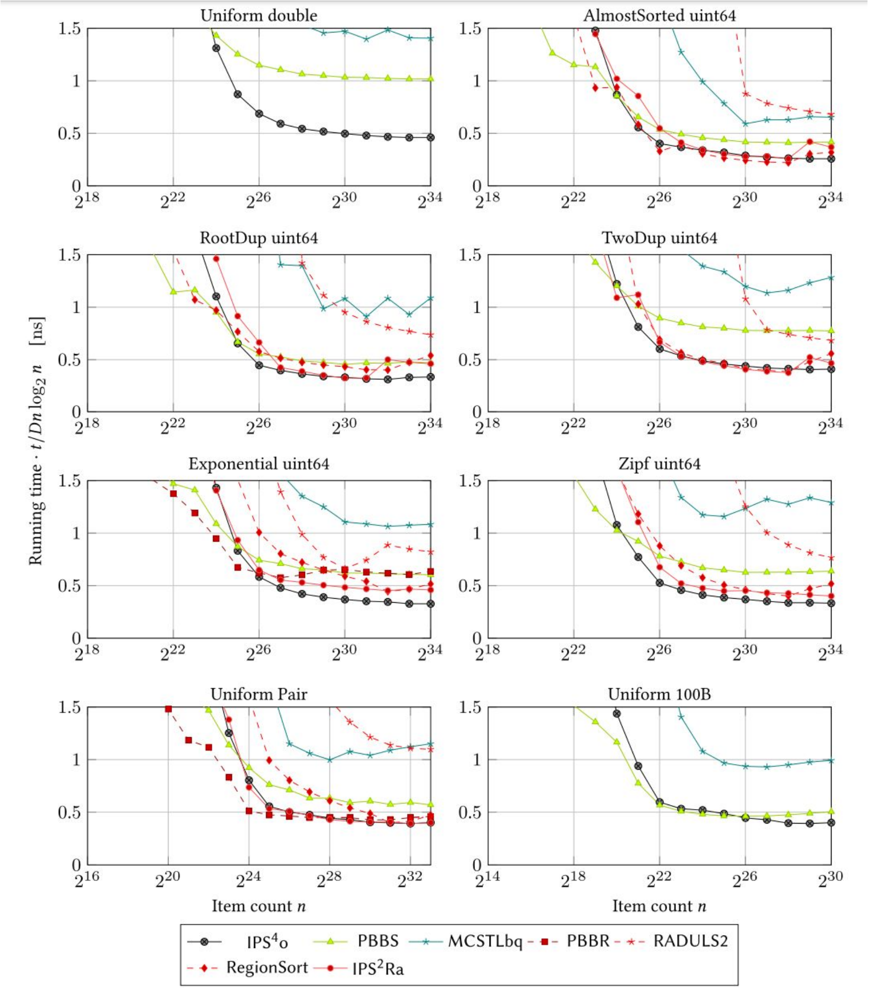
Notably:

- **RADULS2** shows better parallelism initially and gradually slows down.
- **IPS2Ra** starts with better parallelism and converges to IPS4o.



Results

Even more plots…



Future Work

- Better special case handling
  - Small datasets
  - Almost sorted input
  - Datasets with highly duplicated keys

- Theoretical work to reduce span

- SIMD portability to improve sampling phase

Evaluation

- **Strengths**
  - Extensive benchmarking across various data types, input sizes, and distributions and against a wide range of competitive sorting algorithms.
  - Superior performance of the IPS4o and IPS2Ra algorithms over existing parallel and sequential sorting methods in most tested scenarios.
  - Detailed exploration of future work and potential improvements.
  - Comprehensive discussion on both theoretical aspects and practical enhancements, showcasing the depth of the research.
- **Weaknesses**
  - The extensive length and highly detailed content can at time be overwhelming, potentially obscuring the main contributions and findings.
  - Parts of the paper, especially those with intricate optimization strategies and detailed comparisons, could be streamlined or moved to appendices to enhance readability.