In-place Super Scalar Samplesort

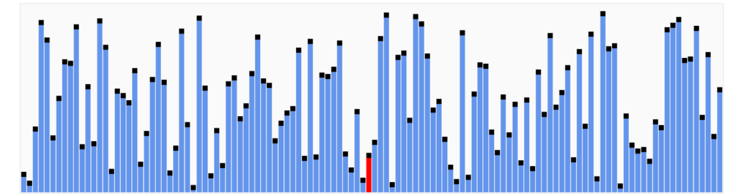
Tim Kralj

Outline

- Quicksort
- Super Scalar Samplesort
- In-place Super Scalar Samplesort (IPS^{4}o) Analysis
- Results
- Further work/Questions

Quicksort

- Finds pivots in the array
- Recursively sorts the sides of the pivots
- Expected runtime: $o(n \log n)$
- Almost in-place
- Parallelizable
- Small amount of code



commons.wikimedia.org/wiki/File:Quicksort.gif

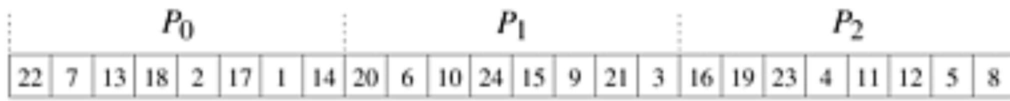
```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi - 1 do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

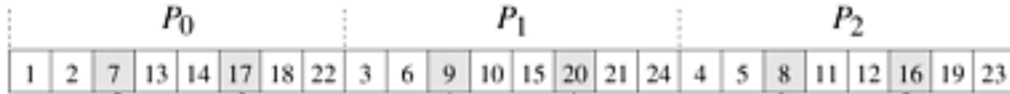
https://en.wikipedia.org/wiki/Quicksort

Super Scalar Samplesort

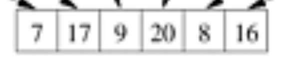
- Cache efficient
- Non-in-place
- Instructional Parallelism
- Avoids branch mis-predictions
- Uses k buckets and t threads



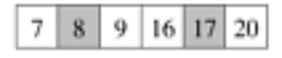
Initial element distribution



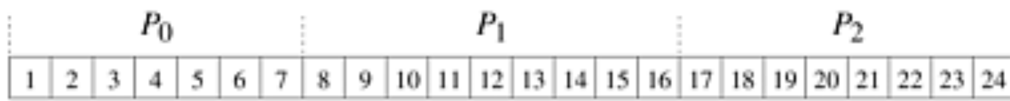
Local sort & sample selection



Sample combining



Global splitter selection



Final element assignment

Super Scalar Samplesort Algorithm

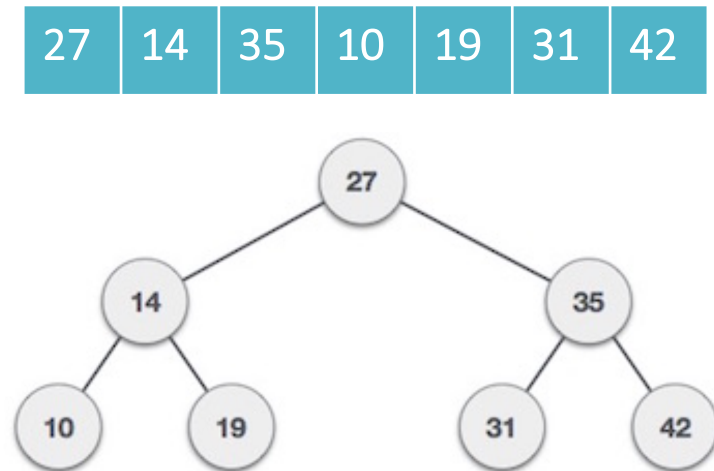
- Like quicksort, with multiple pivot points chosen
- Sampling
- Local Classification
- Block Permutation
- Cleanup

sampleSort($A[1..n], k, p$)
  if $n/k <$ threshold then smallSort(A) // if average bucket size is below a threshold switch to e.g. quicksort
  // Step 1 //
  select $S = [S_1, \ldots, S_{p(k-1)}]$ randomly from A // select samples
  sort S // sort sample
  $[s_0, s_1, \ldots, s_{p-1}, s_p] \leftarrow [-\infty, S_k, S_{2k}, \ldots, S_{(p-1)k}, \infty]$ // select splitters
  // Step 2 //
  for each $a \in A$
    find j such that $s_{j-1} < a \leq s_j$
    place a in bucket b_j
  // Step 3 and concatenation //
  return concatenate(sampleSort(b_1), ..., sampleSort(b_k))

Sampling

- Determines the bucket boundaries
- Sample in front of input array (keep in-place)
- sort $\alpha k \;\; - 1$ randomly sampled input elements
- Pivots picked equidistantly from sorted sample
- Elements stored in a binary search tree
- The left successor of a_i is a_{2i} and its right successor is a_{2i+1}

Binary Search Tree



https://www.tutorialspoint.com/data_structures_algorithms/binary_search_tree.htm

Local Classification

- Groups input blocks such that all elements in each block belong to the same bucket
- Write buffer to array A when full
- Block A gets t stripes of equal size-one for each thread
- Each element in stripe classified into one of the k buckets

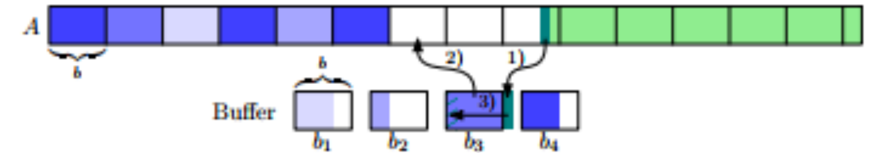


Figure 1 Local classification. Blue elements have already been classified, with different shades indicating different buckets. Unprocessed elements are green. Here, the next element (in dark green) has been determined to belong to bucket b_3. As that buffer block is already full, we first write it into the array A, then write the new element into the now empty buffer.

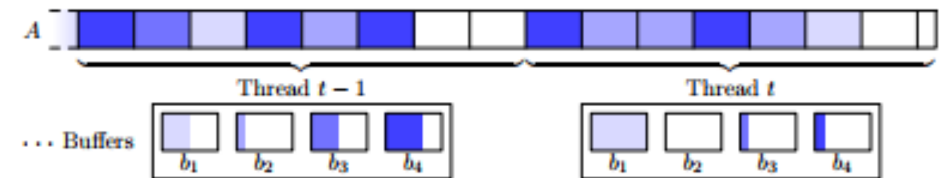


Figure 2 Input array and block buffers of the last two threads after local classification.

Block Permutation

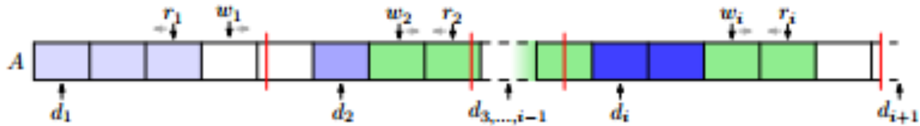
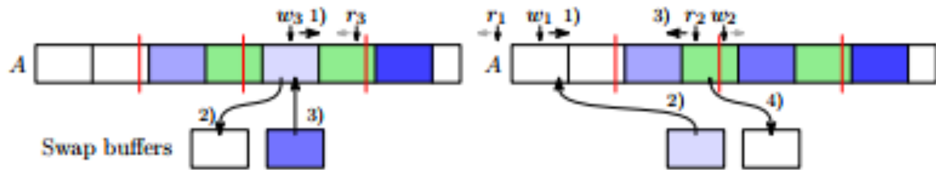


Figure 3 Invariant during block permutation. In each bucket b_i, blocks in $[d_i, w_i)$ are already correct (blue), blocks in $[w_i, r_i]$ are unprocessed (green), and blocks in $[\max(w_i, r_i + 1), d_{i+1})$ are empty (white).



(a) Swapping a block into its correct position.

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

Figure 4 Block permutation examples.

- Rearrange block in input array
- Perform prefix sum compute exact boundaries of buckets
- Allocate a single overflow block instead of writing to final block
- Invariant: each bucket has correct blocks, processed blocks, empty blocks
- Threads swap 2 blocks at a time until sorted

Cleanup

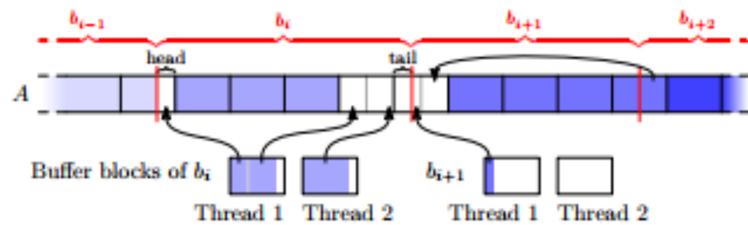
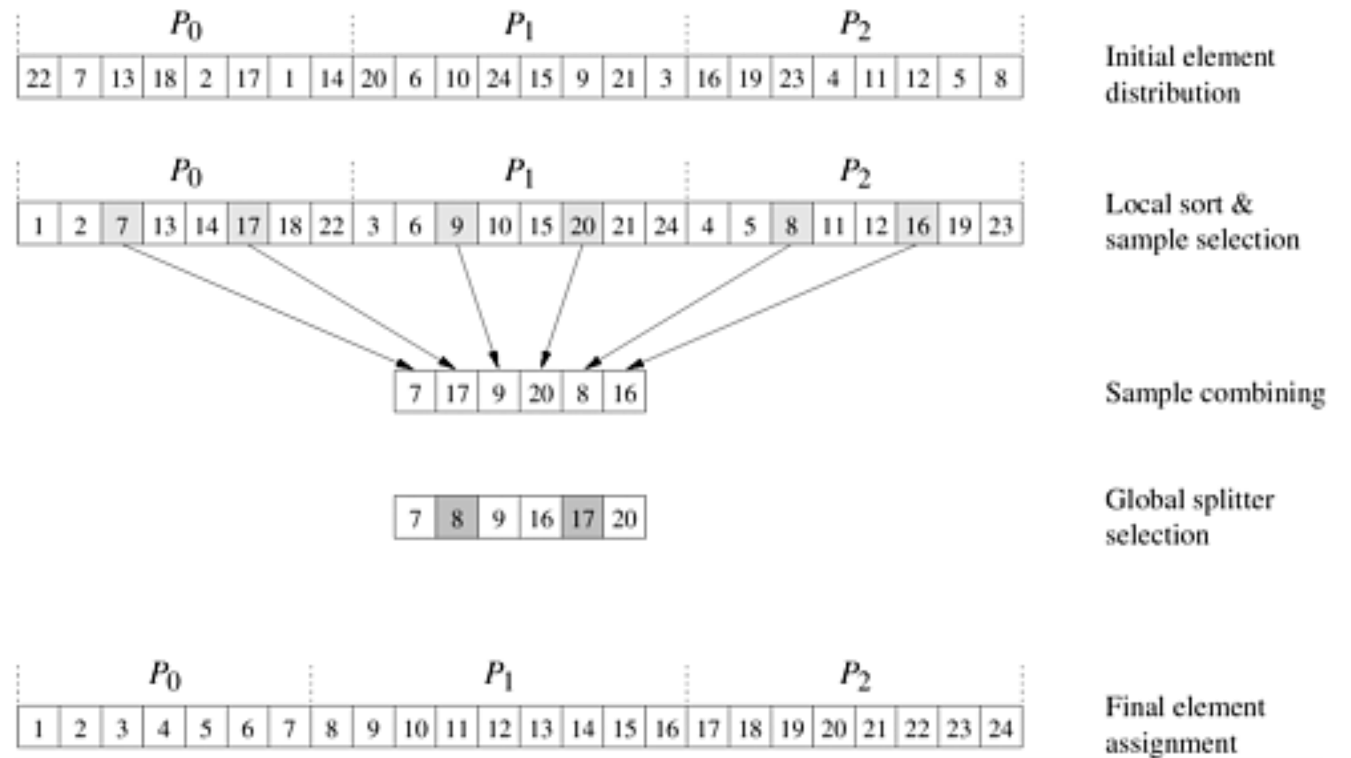


Figure 5 An example of the steps performed during cleanup.

- Blocks may span bucket boundaries
- May have split heads and tails
- Threads look at heads/tails from left to right
- Re-arrange when necessary
- Difficult implementation
- Similar to many 172 projects with bit swapping

Bringing it Together:



Identical Keys

- Many inputs of same key into many levels of recursion
- Turn such inputs into “easy” instances by introducing separate buckets for elements identical to pivots
- Keys occurring more then n k times are likely to become pivots
- Single addition comparison to find whether element goes to equality bucket

Making S^3 in place

```
i := 1                                                      -- first element of current bucket
j := n + 1                                                  -- first element of next bucket
while i < n do
    if j − i < n_0 then smallSort(a, i, j − 1);   i := j    -- base case
    else partition(a, i, j − 1)                             -- partition first unsorted bucket
    j := searchNextLargest(A[i], A, i + 1, n)               -- find beginning of next bucket
```

- Mark beginning of each bucket by storing largest element in first entry
- Find next larger element signals end of bucket
- log time with exponential/binary search

Other Details

- Coarsening recursion with insertion sort for small input
- Adaptive number of buckets for lower levels so don't have many buckets of few elements

- Tuning parameters:
- $k = 256$ buckets
- $\alpha = 0.2 \log(n)$ oversampling
- $\beta = 1$ overpartitioning
- $n_0 = 16$ base case-switch to insertion
- $b =$ 2KiB block size

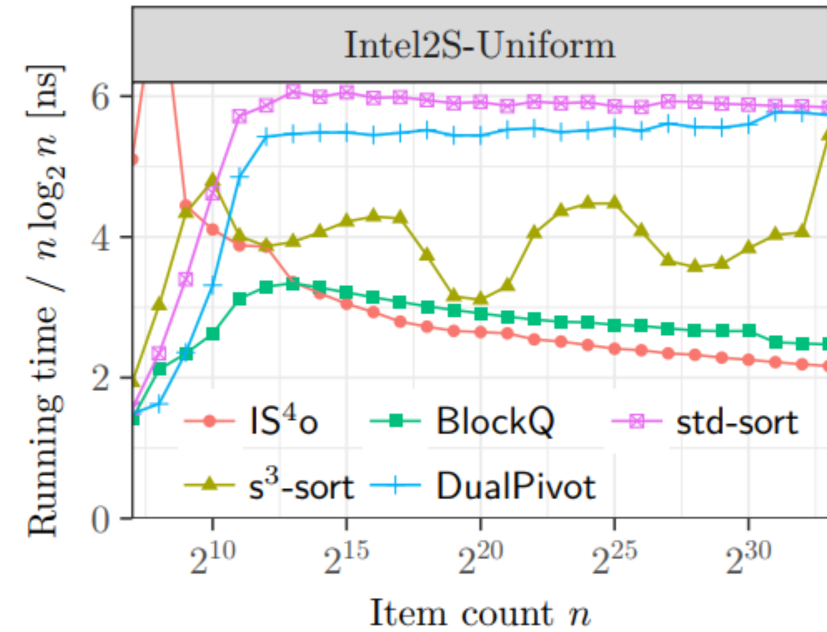
IPS^{4}o Analysis

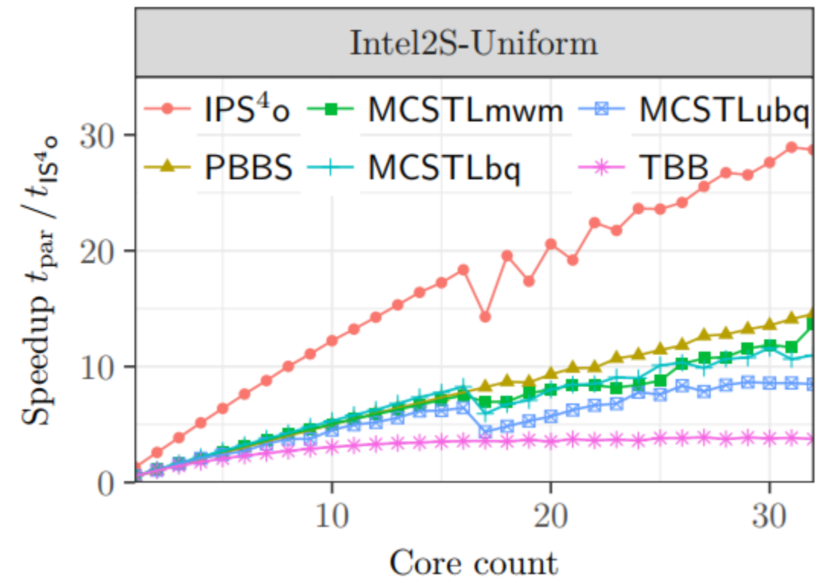
- Assuming: $\mathrm{b} = \theta(tB)$ (buffer block size)
- $M = \Omega(ktB)$
- Base Case size: $n_0 = O(M)$
- $n = \Omega(\max(\mathrm{k}, \mathrm{t})\mathrm{t}^2\mathrm{B})$
- I/O-complexity: $O\left(\frac{n}{tB}\log_k\left(\frac{n}{n_0}\right)\right)$ (with high probability)
- Additional Space: $O(kbt + \log_k \frac{n}{n_0})$
- Usually first term dominates, need to remove $\log(n)$ term for in-place

- Variables:
- t: number of threads
- M: private cache size of thread
- B: block size for main memory access

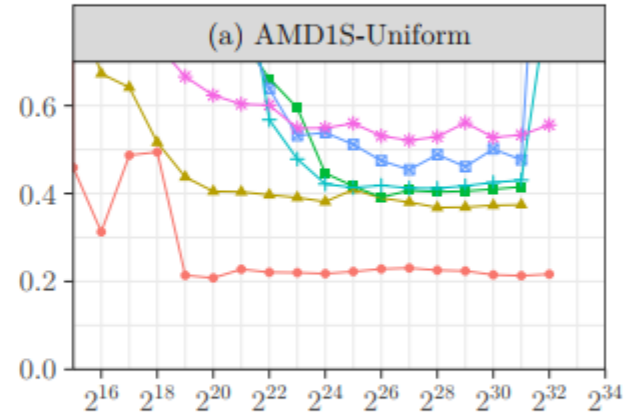
Results

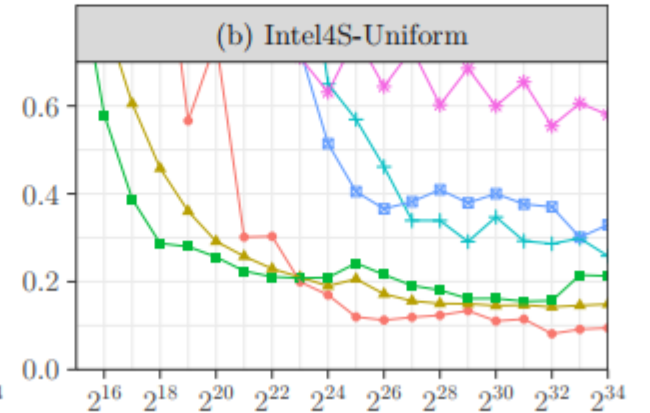
- Sequential: IS⁴o is faster by a factor of about 1.2
- Parallel: faster by factor 2.5 on almost all algorithms
- On inputs of 2^{10} to 2^{32}

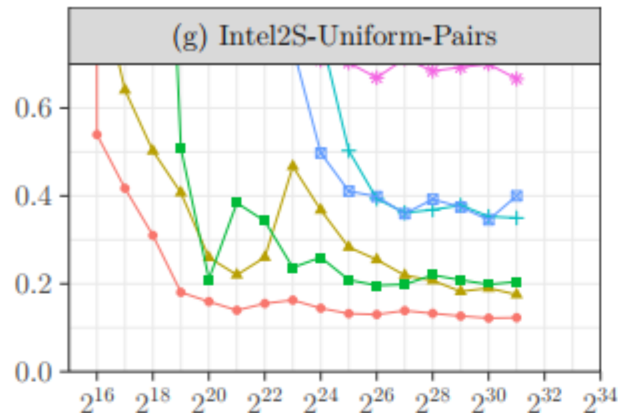


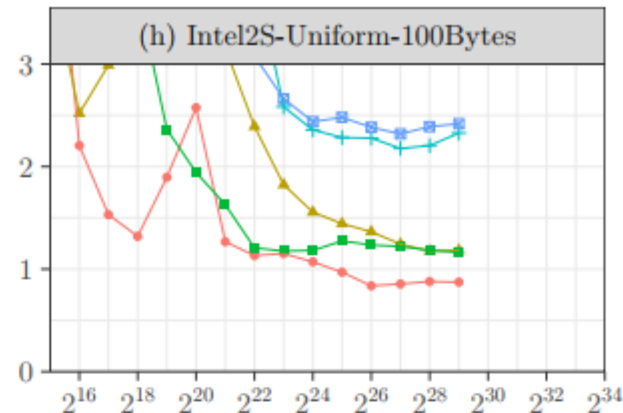


More Results







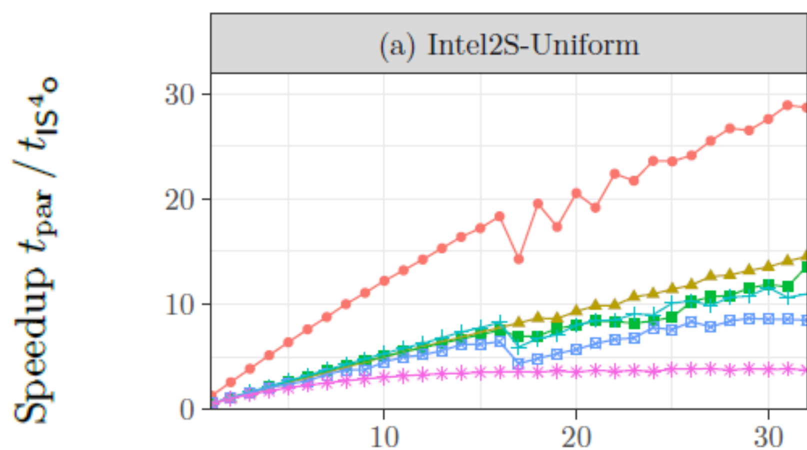


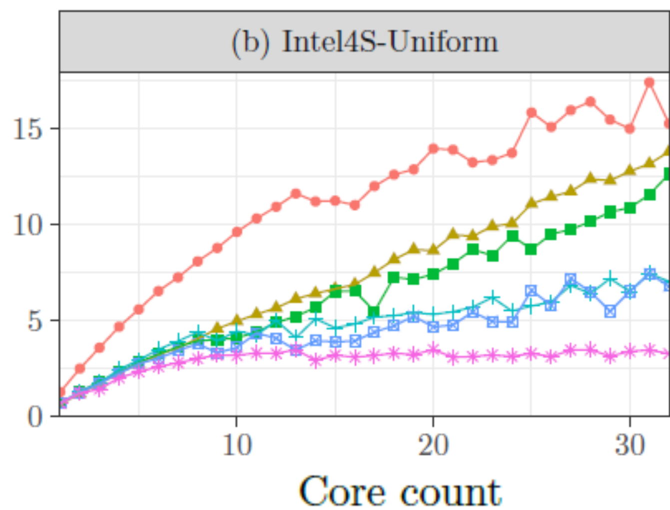
Item count n

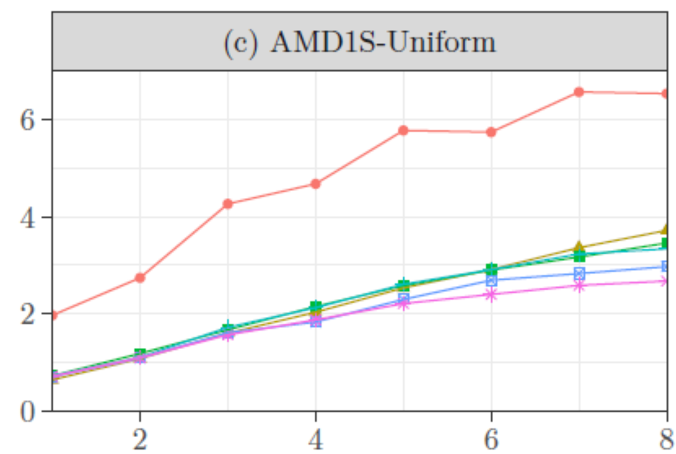
IPS^{4}o PBBS MCSTLmwm MCSTLbq MCSTLubq TBB

- Can lag behind other algorithms on smaller inputs
- Beats competition on large inputs

Multi-Core Results







IPS⁴o — MCSTLmwm — MCSTLubq

PBBS — MCSTLbq — TBB

Further Work/Questions

- Possibility of reducing the amount of code
- Reducing time for sequentially sorting large objects
- Fine Tuning parameters
- Adoption into a language?
    - Java has dual-pivot quicksort
    - C++ uses multi-way merge for parallel
- Viable alternate to Quicksort (over 50 years old)?