Theoretically-Efficient and Practical Parallel In-Place Radix Sorting

Omar Obeya
MIT CSAIL
obeya@mit.edu

Endrias Kahssay
MIT CSAIL
endrias@mit.edu

Edward Fan
MIT CSAIL
edwardf@mit.edu

Julian Shun
MIT CSAIL
jshun@mit.edu

ABSTRACT

Parallel radix sorting has been extensively studied in the literature for decades. However, the most efficient implementations require auxiliary memory proportional to the input size, which can be prohibitive for large inputs. The standard serial in-place radix sorting algorithm is based on swapping each key to its correct place in the array on each level of recursion. However, it is not straightforward to parallelize this algorithm due to dependencies among the swaps.

This paper presents Regions Sort, a new parallel in-place radix sorting algorithm that is efficient both in theory and in practice. Our algorithm uses a graph structure to model dependencies across elements that need to be swapped, and generates independent tasks from this graph that can be executed in parallel. For sorting n integers from a range r, and a parameter K, Regions Sort requires only $O(K \log r \log n)$ auxiliary memory. Our algorithm requires $O(n \log r)$ work and $O((n/K + \log K) \log r)$ span, making it work-efficient and highly parallel. In addition, we present several optimizations that significantly improve the empirical performance of our algorithm. We compare the performance of Regions Sort to existing parallel in-place and out-of-place sorting algorithms on a variety of input distributions and show that Regions Sort is faster than optimized out-of-place radix sorting and comparison sorting algorithms, and is almost always faster than the fastest publicly-available in-place sorting algorithm.

ACM Reference Format:
Omar Obeya, Endrias Kahssay, Edward Fan, and Julian Shun. 2019. Theoretically-Efficient and Practical Parallel In-Place Radix Sorting. In *31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19), June 22–24, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3323165.3323198

1 INTRODUCTION

Parallel integer sorting has received significant attention in the literature due to its efficiency over comparison sorting when the keys are integers. n integer keys in the range $[0, \ldots, r-1]$ can be sorted stably in $\Theta(n/\epsilon)$ work and $O((r^\epsilon + \log n)/\epsilon)$ span for a constant $0 < \epsilon < 1$ [44]. No linear-work polylogarithmic-span stable integer sorting algorithm is known, although integers in the range $[0, \ldots, n \log^{O(1)} n]$ can be sorted non-stably in $\Theta(n)$ work and $O(\log n)$ span [33]. Radix sorting is a form of integer sorting that repeatedly sorts on a constant number of bits of the keys. MSD (most significant digit) radix sorting starts from the higher-order bits, and recursively sorts each set of keys that share the same higher-order bits. LSD (least significant digit) radix sorting starts from the lower-order bits, sorting all keys based on a constant number of bits each time. LSD radix sorting requires each pass to be stable, while MSD radix sorting does not. Parallel radix sorting can be implemented in $(n \log r)$ work and $O(\log r \log n)$ span [9, 38, 46], and there are a variety of practical parallel implementations that have been developed [9, 12, 23, 24, 26, 31, 32, 36–38, 41–43, 45, 46].

Most existing parallel radix sorting algorithms require $\Omega(n)$ additional memory, which can be undesirable for large values of n. As such, there has been recent interest in designing fast *in-place* parallel sorting algorithms [3, 12], where the required auxiliary memory is sub-linear in the input size. Reducing the amount of additional memory required by an algorithm can enable larger inputs to be processed in the memory of a machine, or a machine with smaller RAM to be used, which can lead to considerable savings when renting cloud computing resources. Furthermore, a lower memory footprint can decrease the number of cache misses and page faults, which can in turn improve performance, especially in the parallel setting where memory bandwidth can be a bottleneck.

A standard serial in-place radix sorting algorithm is an MSD algorithm that swaps each key into its correct place on each level of recursion. The algorithm is non-stable, takes $O(n \log r)$ work, and uses $O(\log r \log n)$ additional memory to store per-bucket pointers and information for recursion. More details and pseudocode for this algorithm are presented in Section 2. The only existing parallel in-place radix sorting algorithm is PARADIS [12], which is based on partitioning the input among the available processors, and having each processor serially process its partition by swapping each key into its desired location. The algorithm is speculative in that not all keys may be sorted after each processor finishes its partition, and therefore additional iterations are needed. The authors show that PARADIS performs well in practice, but in the worst case, one level of recursion can take linear span, which is theoretically no better than the serial algorithm. We describe PARADIS in more detail in Section 2.

In this paper, we design Regions Sort, a parallel in-place radix sorting algorithm that is efficient both in theory and in practice. Like the serial in-place algorithm and PARADIS, our algorithm is a non-stable MSD radix sort. The algorithm uses a graph structure where vertices represent sub-arrays that will store keys of the same bucket, and an edge between vertex u and v with weight w represents a set of w keys that are in the sub-array corresponding to vertex u and need to be swapped into the sub-array corresponding to vertex v. Using the graph, we iteratively identify and execute

SPAA '19, June 22–24, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6184-2/19/06…$15.00
https://doi.org/10.1145/3323165.3323198

tasks in parallel, and modify the graph based on the executed tasks. We propose two strategies for processing the graph, one based on processing cycles and one based on processing 2-paths. Similar to the standard parallel MSD radix sorting algorithm, Regions Sort recursively sorts sub-arrays that share the same bucket in parallel, until all of the bits have been processed. We prove that the work of Regions Sort is $O(n \log r)$, which matches that of the serial in-place radix sorting algorithm. We also show that the span of Regions Sort is $O((n/K + \log K) \log r)$ where K is an algorithm parameter. The additional space required by Regions Sort is $O(K \log r \log n)$. Our algorithm is presented in Section 3, and performance optimizations are presented in Section 4.

In Section 5, we perform an extensive experimental evaluation of Regions Sort and compare it to several existing parallel sorting algorithms (out-of-place radix sort, in-place sample sort, out-of-place sample sort, and a sorting algorithm from the C++ Standard Template Library[1]) as well as a state-of-the-art serial in-place radix sorting algorithm. On a variety of input distributions using a 36-core machine with two-way hyper-threading, we are almost always faster than the fastest existing parallel in-place sorting algorithm IPS^{4}o [3] (ranging from 3.6x faster to 1.02x slower). In addition, Regions Sort is between 1.2–4.4x faster than an optimized parallel out-of-place radix sorting algorithm. We achieve between 19–65x parallel speedup on 36 cores with two-way hyper-threading. We present additional experiments comparing the impact of input size, key range, skewness of input distributions, and radix size on performance. The code for Regions Sort is publicly available at https://github.com/omarobeya/parallel-inplace-radixsort.

2 PRELIMINARIES

Notation. In this paper, we use n to denote the size of the input. For integer inputs in the range $[0, \ldots, r-1]$, we say that its range is r. We use P to denote the number of processors available for computation. For radix sorting algorithms, we use b to denote the number of bits (e.g., the size of the radix) it sorts at a time, and $B = 2^b$ is the number of possible values of the radix. We measure space in bits and assume that pointers take $O(\log n)$ space. For a graph, we use (i, j, w) to refer to a directed edge from vertex i to vertex j with weight w.

Work-Span Model. We analyze algorithms in the work-span model, where the ***work*** is the number of operations used by the algorithm and the ***span*** is the length of the longest sequential dependence in the computation [15, 22].

Parallel Primitives. The following parallel procedures are used in the paper. ***Prefix sum*** takes as input an array A of length n, an associative binary operator $\oplus$, and an identity element $\perp$ such that $\perp \oplus x = x$ for any x, and returns the array $(\perp, \perp \oplus A[0], \perp \oplus A[0] \oplus A[1], \ldots, \perp \oplus_{i=0}^{n-2} A[i])$ as well as the overall sum, $\perp \oplus_{i=0}^{n-1} A[i]$. Prefix sum can be done in $O(n)$ work and $O(\log n)$ span (assuming $\oplus$ takes $O(1)$ work) [22]. ***Merging*** takes two sorted arrays A_1 and A_2 and outputs a sorted array containing all elements in A_1 and A_2. Merging requires $O(|A_1| + |A_2|)$ work and $O(\log |A_1| + \log |A_2|)$ span [22]. The standard algorithms for prefix sum and merging use linear auxiliary space [22].

[1]We are unable to compare with PARADIS as their code is not open source and we could not obtain it from the authors. However, we perform a rough comparison with PARADIS based on numbers reported in the PARADIS paper [12].

Algorithm 1 Pseudocode for Serial In-Place Radix Sort

▷ *EXTRACT(a, level) extracts the appropriate bits from element a based on the radix size and current level.*

```
1:  procedure RadixSort(A, N, level)
2:      C = {0, ..., 0} ▷ length-B array storing per-bucket counts
3:      for a in A do
4:          C[extract(a, level)]++
5:      Compute prefix sum on C to get per-bucket start and end
    indices stored in H and T, respectively
6:      for i = 0 to B − 1 do
7:          while H[i] < T[i] do
8:              while extract(A[H[i]], level) ≠ i do
9:                  swap(A[H[i]], A[H[extract(A[H[i]], level)]++])
10:             H[i]++
11:     if level < maxLevel then
12:         for i = 0 to B − 1 do
13:             if C[i] > 0 then
14:                 RadixSort(A_i, C[i], level + 1)        ▷ A_i is the
    sub-array containing elements in bucket i
```

In-Place Algorithms. We call an algorithm ***in-place*** if it requires $o(n)$ additional memory for an input of size n. We assume that the number of processors P is sub-linear in n, otherwise it would not be possible to use all processors without using $\Omega(n)$ additional memory. We note that our definition differs from [4], which defines a parallel in-place algorithm to take $O(P \log n)$ additional memory.

Serial In-place Radix Sorting. Here we review the serial in-place radix sorting algorithm, also known as American Flag Sort [30]. The pseudocode is shown in Figure 1.

The algorithm is an MSD algorithm that maintains buckets where each bucket stores all keys with the same radix at the current level, and where buckets are ordered based on the value of the radix. Lines 2–4 computes the number of keys per bucket (i.e., a histogram). The EXTRACT() function computes the bucket of a key by extracting the appropriate bits based on the radix size and current level. Line 5 uses prefix sum to generate the starting and ending offsets in the array for each bucket (stored in arrays H and T). Lines 6–10 processes the positions one-by-one in each bucket. For each position x in bucket i, the algorithm repeatedly swaps $A[x]$ to its target bucket until the value in $A[x]$ belongs to bucket i. The next position is then processed until the end of the bucket is reached. After all buckets are processed, all keys are sorted with respect to the current radix, and the algorithm recurses on each bucket to sort by the next radix (Lines 11–14). Note that the sort on each level is not stable, and thus an LSD approach would not work.

The histogram and prefix sum take linear work. In Lines 6–10, each swap places at least one key in its correct location, leading to linear work across all buckets. The sum of the sizes of recursive calls is linear, and there are $O(\log_B r)$ levels of recursion. For sub-problems of size $O(B)$ we switch to a comparison sort which takes $O(B \log B)$ work. In the worst case, the calls to comparison sort incur an overall work of $O((n/B)B \log B) = O(n \log B)$. The rest of the work can be bounded by $O(n \log_B r)$. Therefore the total

work is $O(n(\log_B r + \log B))$, which is $O(n \log_B r)$ if we set B such that $\log^2 B = O(\log r)$, e.g., by setting B to a constant. The only auxiliary space required is for storing the bucket counts and offsets per level, pointers on the stack to keep track of recursion, and comparison sort on problems of size $O(B)$. The overall additional space is $O(B \log_B r \log n)$.

Parallelization. The histogram and prefix sum of Algorithm 1 can be implemented in parallel using $O(PB \log n)$ additional memory. The recursive calls can also be executed in parallel. However, the swaps on Lines 6–10 cannot be easily done in parallel as there are data dependences across keys as well as within a cycle of swaps. To distribute the keys to the correct buckets, existing out-of-place parallel MSD radix sort algorithms use $\Omega(n)$ additional memory to store the output array so that keys can be moved from the input to the output array in parallel without any dependences after computing a transpose of the histogram. This paper will present a novel parallel algorithm to distribute the keys in-place.

PARADIS Algorithm. Cho et al. [12] present the PARADIS algorithm for executing the distribution step in-place. To distribute the keys, PARADIS repeatedly executes two phases: the *permutation* and *repair* phases. The permutation phase divides each bucket equally across all processors. Each processor is responsible for swapping the keys in its sub-buckets into the correct place. However, it is not necessarily possible for all keys to be placed into the correct bucket, since a processor could own more keys that need to be swapped into a bucket than its sub-bucket size. The repair phase places all keys not yet in their correct bucket at the end of their current bucket. Each processor is responsible for repairing a subset of the buckets, which are assigned greedily to try to minimize the maximum load (however, each bucket is processed by only one processor). The two phases are then repeated on just the keys that are not yet in their correct buckets. PARADIS uses a custom load-balancing scheme to allocate processors to recursive calls.

The authors argue that the work complexity for processing each radix is $\Theta(Bn)$. The factor of B comes from the fact that the permutation and repair phases have to be repeated multiple times. Summed across all radices, the work is $\Theta(Bn \log_B r)$. The span per radix is $\Theta(Bn(1/P + w))$ where w is the maximum fraction of misplaced keys processed by a processor in the repair phase. The n/P term comes from the permutation phase, which splits the work equally among processors, and the nw term comes from the repair phase where in the worst case one processor has to do $\Theta(nw)$ work. The overall span is $\Theta(Bn(1/P + w) \log_B r)$ assuming that the number of processors assigned to each recursive problem is proportional to the problem size. In the worst case $w = \Theta(1)$ as the bucket sizes can be highly skewed. Thus, the worst case span is $\Theta(Bn \log_B r)$, giving no theoretical parallelism. The algorithm uses B buckets and $O(B)$ pointers per processor on each level, and so the space usage is $O(PB \log_B r \log n)$.[2]

3 PARALLEL IN-PLACE RADIX SORT

This section presents Regions Sort, our new parallel algorithm for in-place radix sorting. Table 1 serves as a reference for the terminology used. Our algorithm is an MSD algorithm, and the high-level structure is presented in Algorithm 2. The algorithm is parameterized by K, an initial number of blocks, and B, the number of possible values of the radix. Regions Sort consists of three main phases prior to recursion: local sorting, constructing the regions graph, and global sorting. For a problem of size N, the local sorting phase (Lines 4–6) partitions the input array into $K' = \lceil K \cdot (N/n) \rceil$ ***blocks***, where n is the initial input size, and sorts the blocks independently in parallel. The number of blocks K' is chosen to be proportional to the original K based on the sub-problem size in order to provide enough parallelism across sub-problems while ensuring that the work on blocks does not grow too much in lower levels of recursion. For simplicity, our discussion assumes that K' evenly divides N, although the algorithm and analysis work in the general case. The local sort uses the serial in-place distribution method presented in Lines 6–10 of Algorithm 1. We also keep track of the counts of each radix value in each block.

In the graph construction phase (Lines 7–8), we use prefix sums to generate the global bucket counts from the local bucket counts obtained from the local sorting phase, followed by another prefix

Term	Definition
Block	A contiguous sub-array of A. The k'th block is the sub-array $A[\frac{kN}{K'}, \ldots, \frac{(k+1)N}{K'} - 1]$.
Country	A contiguous sub-array of A that will contain all keys belonging to a certain bucket after it gets sorted by the current radix. Country i (also referred to as A_i) is the sub-array containing all keys belonging to bucket i after sorting.
Region	A contiguous sequence of keys belonging to the same bucket within the same block and the same country.

Table 1: Terminology.

Algorithm 2 Pseudocode for Regions Sort

```
1:  n = initial input size
2:  K = initial number of blocks
3:  procedure RadixSort(A, N, level)
4:      K' = ⌈K · N/n⌉
                                              ▷ Local Sorting Phase
5:      parfor k = 0 to K' − 1 do
6:          Do serial in-place distribution on A[kN/K', ..., (k+1)N/K' − 1]
                                              ▷ Graph Construction Phase
7:      Compute global counts array C from local counts, and per-
    bucket start and end indices
8:      Generate regions graph G
                                              ▷ Global Sorting Phase
9:      while G is not empty do
10:         Process a subset of non-conflicting edges in G in parallel
11:         Remove processed edges and add new edges to G if
    needed
                                              ▷ Recursion
12:     if level < maxLevel then
13:         parfor i = 0 to B − 1 do
14:             if C[i] > 0 then
15:                 RadixSort(A_i, C[i], level + 1)
```

[2]The work and space bounds we report differ from [12], as they assume that P, B, and r are constant, and that pointers take $O(1)$ space.

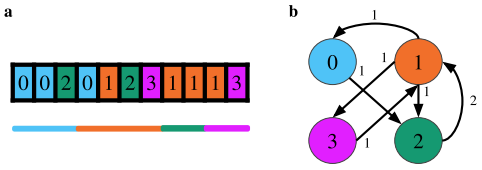

Figure 1: (a) shows the value of the current radix of keys in the input for a 2-bit radix ($B = 4$), and lines below corresponds to the countries (blue is country 0, orange is country 1, green is country 2, and purple is country 3). (b) shows the corresponding regions graph.

sum to obtain offsets for each bucket in the global array. We define a ***country*** to be a contiguous sub-array of the input array that would contain all of the keys belonging to a certain bucket after sorting. At this point, each block may have keys that are not located in its correct country. The ***regions graph*** will represent the source and destination countries of the misplaced keys. Each vertex in the regions graph corresponds to a country. There is a directed edge of weight w from vertex i to vertex j if w contiguous keys from country i should be moved to country j (these contiguous keys are referred to as a ***region***). Multiple edges can exist between two vertices since the regions from i to j can come from multiple blocks (a country can span multiple blocks). We do not include self-edges, as they represent regions that are already correctly placed. An example of a regions graph is shown in Figure 1. The regions graph satisfies the following lemma that will be useful in our proofs of correctness.

Lemma 1. *For each vertex in the regions graph, the sum of the weights of incoming edges for the vertex equals the sum of the weights of outgoing edges for the vertex.*

Proof. A country, represented by a vertex v in the regions graph, is defined to have size equal to the number of keys that belong in the country after sorting. Let $w_{v,\text{in}}$ and $w_{v,\text{out}}$ be the sum of weights of incoming and outgoing edges of v, respectively. By definition, there are $w_{v,\text{out}}$ misplaced keys in country v. Therefore there must be $w_{v,\text{out}}$ keys in other countries that belong to country v. By definition, the incoming edges represent keys in other countries that belong to country v. Therefore, $w_{v,\text{out}} = w_{v,\text{in}}$. □

In the global sorting phase (Lines 9–11), Regions Sort swaps keys in the array by identifying independent swaps that can be executed in parallel. An edge (i, j, w) in the regions graph corresponds to a swap of w keys from country i to j, and the swaps for multiple edges can be performed in parallel as long as none of the source and target memory locations overlap. Processing an edge may introduce a new edge in the regions graph, as a region can be moved to another country where it is still misplaced. Therefore, we need to update the graph by deleting the old edges and adding the new edges. We describe two strategies for using the regions graph to perform the global sorting in Section 3.1.

Finally, we perform the recursive calls of the radix sort on the next radix in parallel on Lines 12–15. For small problems sizes, we switch to comparison sort.

3.1 Global Sorting Strategies

In this section, we present two different methods for using the regions graph to perform global sorting.

3.1.1 Cycle Processing. Our first method is based on processing directed cycles in the regions graph. Observe that for a length-l cycle $(v_0, v_1, w_0), \ldots, (v_{l-1}, v_0, w_{l-1})$, if we move the first $w_{\min} = \min(w_0, \ldots, w_{l-1})$ keys in each of these corresponding regions from country v_i to country $v_{i+1 \bmod l}$ then $w_{\min}$ keys from each of these corresponding regions will have been moved to the correct country. We can then decrement the weight of all of the edges in the cycle by $w_{\min}$ and remove any edge with weight 0, and the resulting regions graph will be valid. Figure 2 shows an example of processing cycles for an input. Each of the $w_{\min}$ keys can be processed in parallel, but the swaps for a particular key have to happen serially due to data dependences (this is similar in spirit to Lines 8–9 of Algorithm 1). The algorithm iteratively processes cycles in the regions graph until no more edges are left, at which point the keys will be sorted. The following lemma shows that using cycle processing, Regions Sort will terminate when all keys are sorted.

Lemma 2. *Using cycle processing, Regions Sort will terminate when all keys are sorted.*

Proof. We argue that a non-empty regions graph contains at least one cycle. A regions graph that is non-empty means that there is at least one misplaced key in the array. By Lemma 1, each vertex with an incoming edge must have at least one outgoing edge. Consider the process where we start with a misplaced key e_0, find the key e_1 in e_0's target country that e_0 wants to swap with, and repeat with e_1 until we have seen two keys from the same country. After inspecting at most $B + 1$ vertices, we must have encountered two keys e_i and e_j from the same country. The edges in the regions graph corresponding to the keys $e_i, \ldots, e_j$ form a cycle.

This algorithm always makes progress, as there is always a cycle in a non-empty regions graph, and after processing a cycle, at least one edge in the cycle will have a weight of 0 and be deleted. Eventually, all edges will be deleted, which corresponds to a sorted array. □

We analyze the theoretical complexity of the cycle processing approach in Section 3.2.

3.1.2 2-path Processing. Our second method is based on processing 2-paths in the regions graph. For a 2-path with directed edges (v_0, v_1, w_0) and (v_1, v_2, w_1), we refer to v_0 as the ***source***, v_1 as the ***broker***, and v_2 as the ***sink***. Let R_0 be the region containing the keys that should be moved from v_0 to v_1, R_1 be the region containing the keys that should move from v_1 to v_2, and R_2 be the target region of the edge (v_1, v_2, w_1). Processing a 2-path involves swapping $w_{\min} = \min(w_0, w_1)$ from region R_0 with $w_{\min}$ consecutive keys in R_1. In the case that $v_0 \neq v_2$ (i.e., the 2-path is not a 2-cycle), as a result of the swap, $w_{\min}$ keys from R_1 that were destined for R_2 are moved to R_0, and so we add a new edge from v_0 to v_2 of weight $w_{\min}$. Figure 3 shows an example of processing a 2-path.

Processing a 2-path will decrease the weight of both edges in the path by $w_{\min}$, which deletes at least one of them from the graph (since it will have a weight of 0). If $v_0 \neq v_2$, a new edge from the

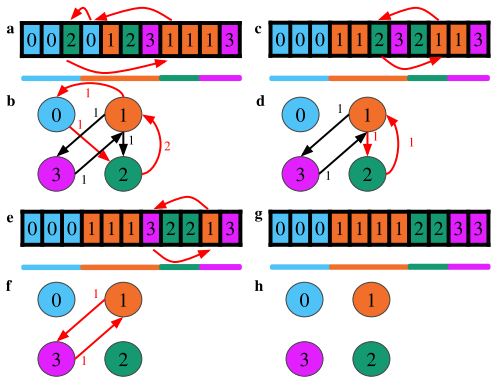

Figure 2: (a) shows the value of the current radix of keys in the input for a 2-bit radix ($B = 4$). At each step, the corresponding swaps for each cycle is shown in the array above the regions graph. (b) shows the corresponding regions graph of (a), and the cycle to process is highlighted in red. (c) and (d) show the array and regions graph after processing the highlighted cycle in (b). In particular, the edges $(0, 2, 1)$ and $(1, 0, 1)$ are deleted, and $(2, 1, 2)$ is modified to $(2, 1, 1)$. The next cycle to process is highlighted in (d). (e) and (f) shows the input and regions graph after processing the highlighted cycle in (d). After processing the remaining cycle in (f), we are left with the sorted array for the current level, shown in (g), and the regions graph with no edges, shown in (h).

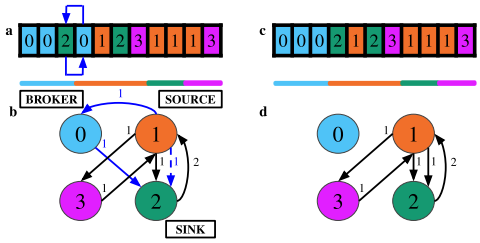

Figure 3: (a) and (b) show an example of processing a 2-path formed by the edges $(1, 0, 1)$ and $(0, 2, 1)$. Here vertex 1 is the source, vertex 0 is the broker, and vertex 2 is the sink. The result of processing this 2-path is shown in (c) and (d). The edges $(1, 0, 1)$ and $(0, 2, 1)$ are deleted, and a new edge $(1, 2, 1)$ is formed (the dashed edge in (b)).

source to the sink of weight $w_{\min}$ will be formed. Since the weight of the two path edges each decrease by $w_{\min}$ units, and the possible new edge has weight $w_{\min}$, the sum of weights in the regions graph decreases by at least $w_{\min}$. Therefore, Regions Sort will eventually terminate. Each of the $w_{\min}$ keys can be swapped in parallel. Our algorithm processes all 2-paths for a particular vertex acting as a broker before moving on to another vertex. Once all 2-paths for a particular vertex have been processed, it will have no more incident edges (since the sum of weights of incoming edges and outgoing edges must be the same by Lemma 1) and can be removed from the regions graph. Figure 4 illustrates how processing 2-paths can be used to sort an array. We analyze the complexity of this approach in Section 3.2.

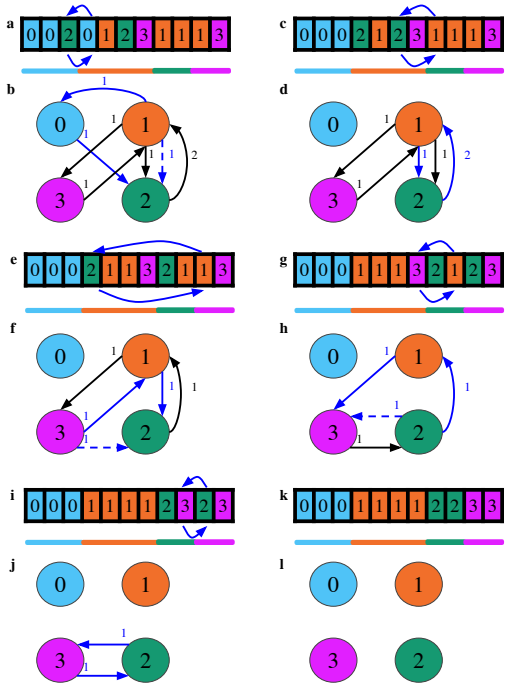

Figure 4: An illustration of global sorting with 2-path processing. At each step, the corresponding swap for the 2-path is shown in the array above the regions graph. In (b), we process the 2-path $(1, 0, 1), (0, 2, 1)$, and create the dashed edge $(1, 2, 1)$ resulting in (d). In (d), we process the 2-path $(1, 2, 1), (2, 1, 2)$, which does not add any new edges, resulting in (f). In (f), we process the 2-path $(3, 1, 1), (1, 2, 1)$, which creates the dashed edge $(3, 2, 1)$, resulting in (h). In (h), we process the 2-path $(2, 1, 1), (1, 3, 1)$, which creates the dashed edge $(2, 3, 1)$, resulting in (j). Finally, in (j), we process the remaining 2-path $(2, 3, 1), (3, 2, 1)$, and are left with an empty graph in (l), at which point the array is sorted.

3.2 Analysis

In this section, we analyze the theoretical complexity of Regions Sort in terms of work, span, and auxiliary space. We first analyze the complexity of the initial call before recursion, where $n = N$ and $K' = K$, and then analyze the complexity of the algorithm across all recursive calls.

3.2.1 Local Sorting. The local sorting phase requires $O(n + KB)$ work and $O(n/K)$ span since each block sorts n/K keys serially and keeps local counts of each radix per block. Using a work-stealing scheduler, such as Cilk's [10], up to P tasks can be executed in parallel by P processors. Each task requires a temporary variable to run the serial in-place distribution, as well as a constant number of pointers. Including the bucket pointers and local counts, the additional space used in the local sorting phase is $O((P + KB) \log n)$.

3.2.2 Graph Construction. In the graph construction phase, computing the global counts from the local counts and generating offsets per bucket uses prefix sums and takes $O(KB)$ work, $O(\log(KB))$ span, and $O((P + KB) \log n)$ additional space. The regions graph can be constructed by creating an edge (i, j, w) if w consecutive keys

of a block are in country i but should be moved to country j. We first compute the number of edges per block and perform a prefix sum to get the total number of edges and offsets for each block into an array. All edges can then be written to the array in parallel. The edges with the same source will be contiguous in memory, and this can be used to create the regions graph. The following lemma shows that the total number of edges in the regions graph is $O(KB)$.

Lemma 3. *The total number of edges in the regions graph is $O(KB)$.*

Proof. After sorting, each block has at most B subarrays containing the same radix value, as all keys with the same radix value are consecutive. Subarrays that do not cross a country boundary are their own region, and subarrays that cross country boundaries are partitioned into regions containing keys from one country. Since there are B countries, the number of country boundaries is $B-1$. Thus the total number of additional regions generated from partitioning is $B-1$. Regions that are not in the correct country form an edge in the regions graph, and thus the total number of edges is upper bounded by $K(B-1)+B-1=KB$. □

Therefore, for graph construction, the work is $O(KB)$, span is $O(\log(KB))$, and additional space is $O((P+KB)\log n)$. We discuss settings of K and B at the end of this section such that the total additional space is $o(n)$.

3.2.3 Global Sorting – Cycle Processing. For the cycle processing strategy, we find a cycle serially by choosing a starting vertex and following the first edge out of each vertex until a vertex v is visited twice along the path. The explored path starting at v and ending at v forms a cycle. This takes $O(B)$ work and $O(B)$ span since there are at most B vertices in the regions graph. We then decrease the weight of all edges in the cycle by the minimum weight, and remove any edges with weight 0 (by incrementing the pointer to the vertex's first edge). This can be done in $O(B)$ work and $O(1)$ span. We delete at least one edge every time we process a cycle, so the overall work and span spent in finding cycles and updating the graph is $O(KB^2)$. The keys being swapped when processing a cycle can all be processed in parallel, and each one does at most $O(B)$ swaps. Each key is processed exactly once, and so the total work for swapping is $O(n)$ and the total span is $O(KB^2)$. The $O(KB^2)$ work term for processing cycles can be charged to keys being swapped since each edge in a cycle causes at least one key to be swapped, and is therefore dominated by the $O(n)$ work term. The auxiliary space needed for swapping is $O(P\log n)$ since at most P swaps can happen at once, each of which requires a temporary variable.

3.2.4 Global Sorting – 2-path Processing. For the 2-path processing strategy, we process all incident edges for a vertex in parallel before moving on to the next vertex. We first describe how to process 2-paths one-by-one. Each swap places at least one key in the correct location, and so there are a total of $O(n)$ swaps needed. Each 2-path can be extracted in $O(1)$ work. We look at the first incoming and first outgoing edge to extract a 2-path. We then delete one of the two edges by incrementing the pointer to the start of the incoming or outgoing edge list for the current vertex, and decrement the weight of the other edge (possibly deleting it as well). A new edge is possibly added to the end of the sink vertex's edge list. The number of edges incident per broker is upper bounded by $O(KB)$, and so the total work is $O(KB^2)$, but this work can be charged to keys being swapped since each 2-path processed swaps at least one key.

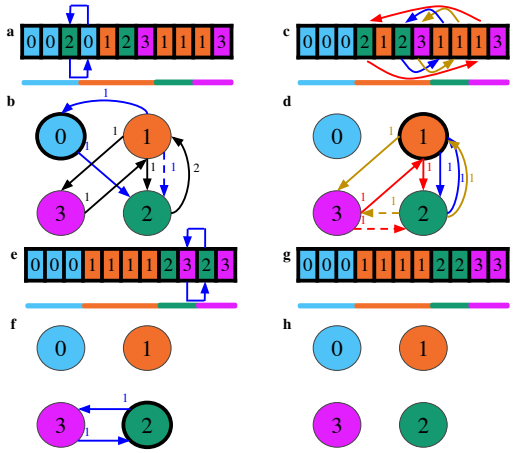

Figure 5: An illustration of how 2-paths incident on a broker are processed in parallel. The current broker being processed is circled in bold. At each step, the corresponding swap for each 2-path is shown in the array above the regions graph using the same color as the 2-path. In (b), we process the single 2-path with vertex 0 as broker, which is $(1,0,1),(0,2,1)$, and create the dashed edge $(1,2,1)$ resulting in (d). In (d), we process all three 2-paths with vertex 1 as broker in parallel. The three 2-paths are $(1,2,1),(2,1,2)$, which does not add any new edges; $(3,1,1),(1,2,1)$, which creates the dashed edge $(3,2,1)$; and $(2,1,1),(1,3,1)$, which creates the dashed edge $(2,3,1)$. This results in (f). Finally, in (f), we process the single 2-path with vertex 2 as broker, which is $(2,3,1),(3,2,1)$, and are left with an empty graph in (h), at which point the array is sorted.

We now describe a solution for 2-path processing with lower span. We can process all 2-paths with a common broker in parallel, and we can use prefix sums and merging to calculate appropriate offsets so that all swaps operate on disjoint memory locations. In this case, an edge may participate in multiple 2-paths (it splits its weight among the multiple paths). We find all of the 2-paths with vertex v as a broker by creating two arrays I and O, where I stores the weights of the incoming edges and O stores the weights of the outgoing edges. We then compute the prefix sums of the weights in each of I and O, giving arrays I' and O'. Next, we merge I' and O' together (ties are broken by giving higher priority to edges in I'). Now each incoming edge in I' can be matched to one or more outgoing edges in O' to form 2-paths, and vice versa. In particular, in the merged output, each edge in I' is matched to the closest edge from O' to its right, and each edge in O' is matched to the closest edge from I' to its left (which again can be computed using prefix sums). We can use the result of the prefix sums to determine exactly which keys in the two regions are used to form the 2-path and then execute all of the swaps independently in parallel. The new edges formed by processing the 2-paths can be stored in an output array and sorted at the end using a parallel integer sort to insert them into the regions graph. An example of processing all 2-paths on a broker in parallel is shown in Figure 5.

The work for prefix sums and merging for one vertex is proportional to the number of edges incident on the selected broker vertex, which is upper bounded by $O(KB)$. The span is $O(\log(KB))$ for prefix sums and merging. The range of the edge values are in $[0, \ldots, (B-1)^2]$, and so we can perform the integer sort in $O(KB)$ work, $O(B + \log(KB))$ span, and linear additional space [44]. Summed across the B vertices, the work is $O(KB^2)$ and span is $O(B(\log(KB) + B))$. Again, the $O(KB^2)$ work term can be charged to keys being swapped and is dominated by $O(n)$. The auxiliary space needed for the prefix sums, merge, and integer sort is $O(KB \log n)$. Each swap places at least one key in the correct location, and so there are a total of $O(n)$ swaps needed to sort the array for the current radix, leading to $O(n)$ work. As with cycle processing, the auxiliary space needed for swapping is $O(P \log n)$.

3.2.5 Overall Algorithm. We now analyze the complexity of our algorithm across all recursive calls. After the local sorting, graph construction, and global sorting phases, our algorithm recurses on each of the buckets (countries) in parallel using the next radix. For a range r and number of possible values of the radix B, the total number of levels of recursion is $O(\log_B r)$. The work for swapping the keys in global sorting is $O(n)$ per level. We now need to bound the extra work incurred by local sorting and graph processing operations in global sorting.

We assume that the original input size n is at least αKB for a constant α to be determined later. If the original input is smaller than αKB, we pad the input with dummy keys (which will be discarded at the end of computation) until it is of size αKB. For a sub-problem of size $N < B$, we switch to a parallel comparison sort [13] which contributes $O(n \log B)$ work overall, similar to the serial algorithm. We now argue that for a sub-problem of size $N \geq B$, the extra $O(K'B)$ work term for constructing the regions graph and managing buckets in local sorting can be charged to swapping the keys. Let this extra work equal $\beta K'B$, which we want to upper bound by γN, for constants β and γ. Recall that we set $K' = \lceil K \cdot N/n \rceil$. Consider two cases, $K' = 1$ and $K' > 1$. If $K' = 1$, then we are essentially running the serial in-place algorithm, which incurs $O(N + B) = O(N)$ work. Otherwise if $K' > 1$, then $K' \leq 2K \cdot N/n$ and we need $\beta(2K \cdot N/n)B \leq \gamma N$. This holds true if $n \geq 2\beta KB/\gamma$, and so we set $\alpha = 2\beta/\gamma$. Therefore, the work for graph processing and local sorting can be charged to swapping the keys, and the total work bound is $O((n + KB)(\log_B r + \log B))$.

The span per level for local sorting is $O(n/K)$ since the total number of blocks across sub-problems is at least K. The span for global sorting of sub-problems, which are all executed in parallel, is upper bounded by the span of the top-level since the number of blocks per sub-problem is at most K. The span for the calls to parallel comparison sort is $O(\log B)$ [13].

We now bound the auxiliary space across recursive calls. An additional $O(PB \log_B r \log n)$ space is needed to keep track of bucket pointers per level and pointers in the stack for recursion. The calls to parallel comparison sort require $O(PB \log n)$ space overall. When run serially, at most one local sorting, graph construction, or global sorting phase can happen at any time, leading to $O(KB \log n)$ space. A straightforward space bound of $O(PKB \log n)$ when using P processors can be obtained by using Cilk's space bound [10]. However, a more careful analysis gives a tighter bound, as shown in the following lemma.

Lemma 4. *The auxiliary space usage required by local sorting, graph construction, and global sorting across all recursive calls is* $O((K + P)B \log n)$.

Proof. For the worst case space analysis of local sorting and regions graphs, we assume that each processor is working on a task for which different auxiliary memory is allocated (we only need to consider the space for P such tasks since the space at higher levels of recursion has already been freed). Let N_i be the size of the sub-problem for the task that processor i is working on. The total number of blocks across the P tasks is $\sum_{i=1}^{P} \lceil K \cdot N_i/n \rceil \leq P + K \sum_{i=1}^{P} N_i/n$. At any point during the computation, each key can be involved in only one sub-problem because the algorithm recurses only when it is done processing the key at the current level. Thus, we have $\sum_{i=1}^{P} N_i \leq n$ and $\sum_{i=1}^{P} N_i/n \leq 1$. Therefore, the total number of blocks across the active tasks is bounded by $K + P$, and the overall space used for local sorting, graph construction, and global sorting is $O((K + P)B \log n)$. □

At the beginning of the algorithm, we compute the range r by finding the value of the maximum key. This can be done by splitting the array into K sub-arrays, and in parallel across sub-arrays, computing the maximum of each sub-array serially. Finally, we use a prefix sum on the K results to get the overall maximum. This requires $O(n)$ work, $O(n/K + \log K)$ span, and $O(K \log n)$ space, which does not increase our overall bounds. The following theorems summarize the complexity of the two variants of our algorithm using the cycle processing and 2-path processing strategies as a function of n, K, B, P, and r.

Theorem 5. *Using cycle processing, Regions Sort takes* $O((n + KB)(\log_B r + \log B))$ *work,* $O((n/K + KB^2) \log_B r)$ *span, and* $O((PB \log_B r + KB) \log n)$ *auxiliary space.*

Theorem 6. *Using 2-path processing, Regions Sort takes* $O((n + KB)(\log_B r + \log B))$ *work,* $O((n/K + B(\log(KB) + B)) \log_B r)$ *span, and* $O((PB \log_B r + KB) \log n)$ *auxiliary space.*

Our algorithm sets K to be $\Theta(P)$ for the initial call to radix sort, and B to be a constant. With these parameter settings, our 2-path variant takes $O(n \log r)$ work, $O((n/P + \log P) \log r)$ span, and $O(P \log r \log n)$ additional space. The algorithm achieves linear parallel speedup for $P = O(n/\log n)$ although we need $P = o(n/(\log n \log r))$ to have an in-place algorithm. The bounds are summarized in the following theorem.

Theorem 7. *Using 2-path processing with* $K = \Theta(P)$ *and* $B = \Theta(1)$*, Regions Sort takes* $O(n \log r)$ *work,* $O((n/P + \log P) \log r)$ *span, and* $O(P \log r \log n)$ *additional space.*

A processor-oblivious parallel in-place algorithm can be obtained by setting $K = o(n/(\log n \log r))$, and the value of K trades off between span and auxiliary space. For example, $K = \Theta(n/(\log n \log r)^2)$ gives an algorithm with $O(n \log r)$ work, $O(\log^2 n \log^3 r)$ span and $O(P \log r \log n + n/(\log n \log^2 r))$ auxiliary space.

4 IMPLEMENTATION AND OPTIMIZATIONS

Implementation. We implemented the two variants of Regions Sort using Cilk Plus [27], which provides a provably-efficient work-stealing scheduler [10]. We found the 2-path variant to always be faster due to avoiding the overhead of cycle finding, and being able to effectively use the early recursion optimization that we will describe. To represent the regions graph in the 2-path variant, we use two edge lists per vertex, one for the incoming edges and one for the outgoing edges. This makes it easy to match incoming and outgoing edges to form 2-paths with a particular vertex as the broker. Each edge is stored only on the vertex that will be processed first instead of on both endpoints. Newly created edges are added to the end of an edge list. To represent the regions graph in the cycle processing variant, we only store outgoing edges of vertices and there are no new edges that get created.

The remainder of this section describes optimizations for both the 2-path and cycle processing variants to improve performance.

Processing 2-cycles. When processing 2-paths that are not cycles, after performing a swap, only one of the keys will be in the correct country. However, if we process a 2-cycle, then both keys will be in the correct country after swapping. Furthermore, no new edge has to be created in the regions graph. Therefore, processing as many 2-cycles as possible will reduce the overall work. Our algorithm first finds and processes all 2-cycles on a vertex before processing 2-paths that are not cycles. Although there is additional work needed to determine 2-cycles, we found that the overall work decreases since a swap places two keys in the correct country instead of one, and no new edges have to be formed in the regions graph.

Parallelization Across Cycles and 2-paths. In our cycle processing algorithm, we process all cycles for a vertex v before moving to the next vertex. We first find all of the cycles for vertex v serially, but allow edges to have their weight split across multiple cycles, so that all of v's outgoing edges will be contained in one or more cycles. Then we execute the swaps associated with all of the cycles in parallel. Finally, we remove v from the graph. While this optimization does not improve the worst-case span (cycle-finding is still done serially), it provides more parallelism during global sorting.

For the 2-path processing variant, our implementation finds the 2-paths on a broker using a serial merging algorithm, as we found that the number of edges was too small to benefit from parallelism. However, after finding all of the 2-paths on the broker vertex, we execute the swaps for all of them in parallel.

Early Recursion. After processing all of the 2-paths or cycles associated with a vertex, there will be no more swaps in and out of the corresponding country. Therefore, we can recursively call radix sort on the keys in the country before we finish processing 2-paths or cycles of other vertices. This increases the available parallelism and improves overall performance. This optimization fuses together Lines 9–11 and Lines 12–15 of Algorithm 2.

Country Sorting. There is more work in the countries with more keys, and with early recursion it is beneficial to recurse on the larger countries first to generate more available parallelism. Therefore, we sort the countries by size and process 2-paths or cycles from countries in decreasing order of size. The edges in the regions graph will only be stored on the vertex corresponding to the larger country, as that country will be processed first. For the 2-path variant, this optimization improves performance significantly in skewed distributions, where the country sizes vary widely. For the the cycle processing variant, the optimization does not help as much, since most of the work is done on the first vertex anyway regardless of whether or not we sort the countries.

Coarsening. Our algorithm includes multiple levels of coarsening. When n is less than 32, we switch to serial insertion sort. Otherwise, when $K = 1$ or when n is smaller than 20000, we use Ska Sort, an optimized serial in-place radix sort [40]. To reduce overheads of parallelism, when executing parallel loops, we use a parallel for-loop (`cilk_for`) when the number of iterations is greater than 4000, and otherwise we use a serial for-loop.

5 EVALUATION

Experimental Setup. We evaluate the performance of Regions Sort on an AWS EC2 `c5.18.xlarge` instance with two 18-core processors with hyper-threading (Intel Xeon Platinum 8124M at 3.00 GHz with a 24.75 MB L3 cache), for a total of 36 cores and 72 hyper-threads. The system has a total of 144 GB of RAM. We compile our code using g++ version 7.3.1 (which has support for Cilk Plus), and we used the `-O3` optimization flag.

The datasets we use include inputs of only integer keys as well as inputs of integer key-value pairs. We test on both 4-byte and 8-byte keys/values (inputs with larger values can be sorted by storing a pointer to the value). Our inputs unif(x, y) are inputs of length y with uniform random integers in the range x. Our inputs zipf$_\theta(x, y)$ are inputs of length y drawn from a Zipfian distribution of range x with parameter θ [17]. We also test on several degenerate inputs of length 10^9: an all equal input (allEqual(10^9)), an input with $10^{9/2}$ copies of $10^{9/2}$ distinct values placed equally apart ($\sqrt{n}$Equal(10^9)), a sorted input (sorted(10^9)), and an almost sorted input with $10^{9/2}$ random keys out of place (almostSorted(10^9)).

Unless otherwise specified, we use an 8-bit radix ($B = 256$) and an initial value of $K = 5000$ for the 2-path variant and $K = P$ for the cycle processing variant, which we found to give the best performance overall. The high value of K benefits the 2-path variant of our algorithm because it allows the blocks to more easily fit in cache and helps Cilk load balance work among processors more effectively. The cycle processing variant benefits from these effects well, but in practice it does not improve for high values of K because the length of cycles becomes too large, which degrades performance.

Other Algorithms. The only existing parallel in-place radix sort algorithm we are aware of is PARADIS [12], but there is no publicly-available code, and we were unable to obtain the code from the authors. Therefore, we perform a rough comparison based on the results reported in their paper.

We compare against the following sorting implementations: the state-of-the-art serial in-place radix sorting algorithm Ska Sort [40], a parallel in-place sample sort IPS^{4}o by Axtmann et al. [3], a parallel out-of-place radix sort RADULS [24, 25], a parallel out-of-place radix sort from the Problem Based Benchmark Suite (PBBS) [38], a parallel out-of-place sample sort from PBBS, and a parallel out-of-place comparison sorting algorithm from MCSTL [39].

	Ska Sort	2-path Regions Sort			Cycle Regions Sort			IPS^{4}o			RADULS			PBBS Radix Sort			PBBS Sample Sort			MCSTL Sort		
Input	T_1	T_1	T_{36h}	SU	T_1	T_{36h}	SU	T_1	T_{36h}	SU	T_1	T_{36h}	SU	T_1	T_{36h}	SU	T_1	T_{36h}	SU	T_1	T_{36h}	SU
unif(10^9, 10^9)	23.02	14.19	**0.49**	28.96	19.48	0.72	27.06	33.52	0.93	36.04	27.79	1.79	15.53	14.40	0.62	23.23	81.41	1.81	44.98	100.36	2.54	39.51
unif(10^9, 10^9)-pairs	34.59	20.05	**0.84**	23.87	27.66	1.31	21.11	139.44	3.01	46.33	41.11	3.49	11.78	28.00	1.23	22.76	92.16	2.34	39.38	116.94	4.27	27.39
unif(2^{63}, 10^9)	38.12	32.27	1.00	32.27	92.12	1.42	64.87	35.99	1.15	31.30	15.15	**0.78**	19.42	42.60	1.50	28.40	136.00	2.93	46.41	103.12	4.62	22.32
unif(2^{63}, 10^9)-pairs	54.91	40.05	1.78	22.50	119.80	2.37	50.55	144.22	3.32	43.44	23.12	**1.67**	13.84	59.60	2.96	20.14	148.00	3.31	44.71	119.50	7.71	15.50
zipf$_{0.75}$(10^9, 10^9)	22.02	14.81	**0.55**	26.93	20.05	0.70	28.64	29.66	0.82	36.17	28.86	1.93	14.95	14.70	0.68	21.62	77.57	1.74	44.58	94.52	2.54	37.21
zipf$_{0.75}$(10^9, 10^9)-pairs	33.04	20.61	**0.98**	21.03	29.32	1.32	22.21	139.30	3.03	45.97	43.81	3.64	12.04	36.50	1.33	27.44	92.71	2.57	36.07	120.09	4.58	26.22
allEqual(10^9)	6.60	6.77	**0.20**	33.85	6.63	0.22	30.14	2.32	0.29	8.00	30.29	1.87	16.20	17.70	0.88	20.11	22.84	0.77	29.66	13.65	0.84	16.25
$\sqrt{n}$Equal(10^9)	15.07	11.43	0.43	26.58	12.56	0.66	19.03	13.71	**0.42**	32.64	28.47	2.28	12.49	16.70	0.60	27.83	39.22	1.36	28.84	30.85	1.23	25.08
sorted(10^9)	14.54	14.51	**0.31**	46.81	16.36	0.38	43.05	24.60	0.62	39.68	30.25	1.67	18.11	14.70	0.60	24.50	42.20	1.25	33.76	19.65	0.94	20.90
almostSorted(10^9)	18.47	16.41	**0.36**	45.58	17.30	0.39	44.36	25.16	0.62	40.58	28.55	2.27	12.58	16.60	0.60	27.67	43.70	1.25	34.96	22.62	2.62	8.63

Table 2: Serial times (T_1) and parallel (T_{36h}) times (seconds), as well as the parallel speedup (SU) on a 36-core machine with two-way hyper-threading. The fastest parallel time for each input is bolded.

Running Times. Table 2 shows the single-thread, 72-thread, and parallel speedup of all of the algorithms on our inputs.

We see that our 2-path processing strategy always outperforms the cycle processing variant. The two variants of Regions Sort achieve a parallel speedup of between 19–65x on 36 cores with hyper-threading. While we use Ska Sort as a base case, our single-threaded time for 2-path processing is usually faster than the highly-optimized serial Ska Sort. Although Regions Sort incurs more work, our local sorting phase has better temporal locality because we split the input into blocks which can more easily fit into cache.

IPS^{4}o is the fastest publicly-available parallel in-place sorting algorithm, and the 2-path variant of Regions Sort achieves a 1.1–3.6x speedup over it in parallel across all inputs, except for $\sqrt{n}$Equal(10^9) on which we are about 1.02x slower. IPS^{4}o, described more in Section 6, is a comparison sorting algorithm, which is more general than our algorithm, and incurs more work to determine the bucket that a key belongs in. The two variants of Regions Sort as well as IPS^{4}o use a negligible amount of auxiliary memory compared to the original input size (less than 5%). PBBS sample sort and MCSTL sort are almost always slower than IPS^{4}o and Regions Sort, and require auxiliary space proportional to the input size. Again, PBBS sample sort and MCSTL sort are comparison sorting algorithms, which are more general. Compared to PBBS radix sort, which is a parallel out-of-place radix sorting algorithm, the 2-path variant of Regions Sort is between 1.2–4.4x faster in parallel. Compared to RADULS, another out-of-place radix sorting algorithm, we are between 1.3x slower to 9.4x faster. We are slightly slower than RADULS on the inputs with 8-byte key ranges, and we believe that RADULS has been optimized for very large key ranges.

Since the PARADIS code is not publicly-available, we perform a rough comparison based on the numbers reported in their paper [12]. On an input of random 8-byte key-value pairs of size 10^9, PARADIS reports a running time of approximately 6 seconds using 16 cores of two Intel Xeon E7-8837 processors running at 2.67GHz. On 16 cores without hyper-threading on our machine, the 2-path variant of Regions Sort takes about 3 seconds on unif(2^{63}, 10^9)-pairs. We do not know the range of the keys PARADIS used, and so we chose a very large range for our experiments. We believe that even accounting for differences in machine specifications, Regions Sort would still be faster. Furthermore, Regions Sort has much stronger theoretical guarantees than PARADIS.

Parallel Scalability. Figures 6 and 7 show the speedup of the algorithms over the single-threaded running time of 2-path as a function of thread count for unif(10^9, 10^9) and zipf$_{0.75}$(10^9, 10^9), respectively. The algorithms all obtain reasonable parallel speedup as the thread count increases, except RADULS which gets worse at higher thread counts. We see that the 2-path variant of Regions Sort outperforms all of the other algorithms on all thread counts.

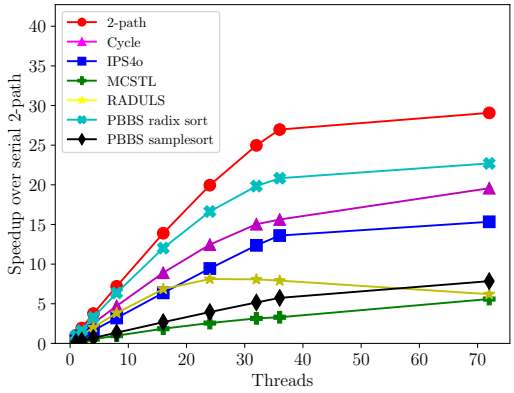

Figure 6: Speedup over single-thread performance of 2-path vs. thread count for unif(10^9, 10^9).

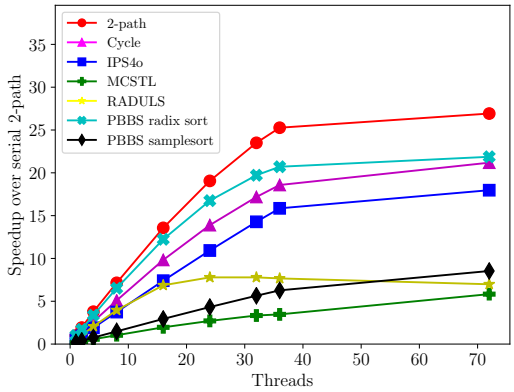

Figure 7: Speedup over single-thread performance of 2-path vs. thread count for zipf$_{0.75}$(10^9, 10^9).

Input Size Scalability. Figure 8 shows the performance of all of the algorithms on uniform random inputs of varying size, with a range of 10^8. For the 2-path variant in this experiment, we decreased K proportionally with the input size. As expected, the running times of all of the algorithms increase with the input size. The 2-path variant is always the fastest on all of the different input sizes.

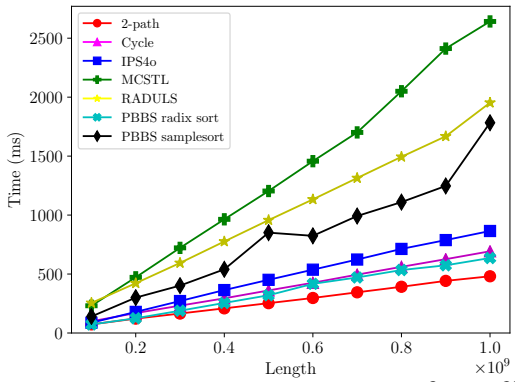

Figure 8: 36-core running time vs. input size n for unif$(10^8, n)$.

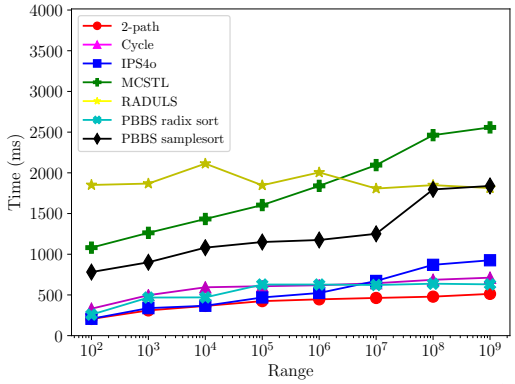

Figure 9: 36-core running time vs. range r for unif$(r, 10^9)$.

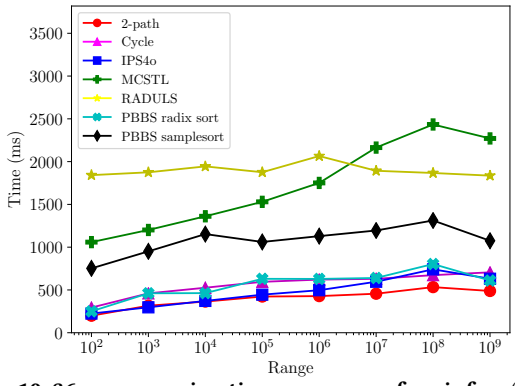

Figure 10: 36-core running time vs. range r for zipf$_{0.75}(r, 10^9)$.

Effect of Key Range. In Figures 9 and 10, we plot the running time of the algorithms as a function of the range of the input keys on uniform random inputs. As expected, the running times of the algorithms increase with key range. However, the increase for both variants of Regions Sort as well as PBBS radix sort and RADULS is sub-linear in $\log r$, the worst-case number of levels of recursion for radix sort, because the algorithms switch to comparison sort when the input size is small enough.

Effect of Skewness in Zipfian Distributions. In Figure 11, we show the performance of the algorithms on zipf$_\theta(10^9, 10^9)$ with varying θ parameter. A larger θ parameter increases the skew of the distribution. We observe that the performance of both variants of Regions Sort is robust across different θ values as the number

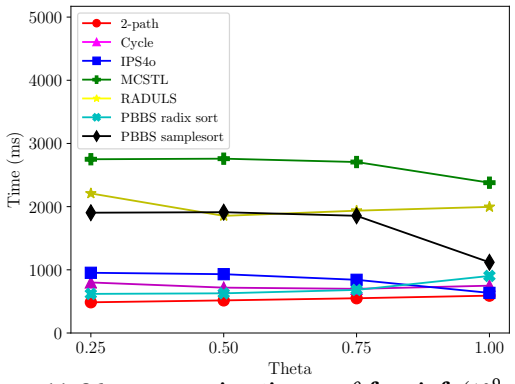

Figure 11: 36-core running time vs. θ for zipf$_\theta(10^9, 10^9)$.

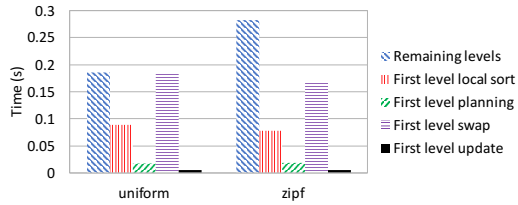

Figure 12: Breakdown of 36-core running time for the 2-path variant of Regions Sort on unif$(10^9, 10^9)$ and zipf$_{0.75}(10^9, 10^9)$.

of levels of recursion depends mostly on the range, which is fixed in this experiment. The comparison sorting algorithms perform better when the skew is higher because having fewer unique keys reduces the number of recursive calls needed. The 2-path variant of Regions Sort still performs the best across different θ values.

Breakdown of Running Time. In Figures 12, we show the breakdown of running time for the 2-path variant of Regions Sort on the unif$(10^9, 10^9)$ and zipf$_{0.75}(10^9, 10^9)$ inputs. We break down the running time into various phases of the first level of recursion and group together the time for the remaining levels. For these two inputs, there are four levels of recursion since we use an 8-bit radix. We see that the first level of radix sort takes a significant fraction of the total time (over half for unif$(10^9, 10^9)$ and almost half for zipf$_{0.75}(10^9, 10^9)$). The remaining levels of recursion are relatively faster since the problem size decreases and is more likely to fit in cache. Within the first level, local sorting and performing the swaps associated with 2-paths dominate the running time. Finding the 2-paths (labeled as "First level planning") and updating the graph (labeled as "First level update") take very little time (under 10% of the first level time).

Effect of Radix Size. Figures 13 and 14 shows the impact of varying the number of bits of the radix on the performance of the two variants of Regions Sort on unif$(10^9, 10^9)$ and zipf$_{0.75}(10^9, 10^9)$, respectively. We achieve the best performance on 8-bit radices, but the performance on 6-bit and 7-bit radices is almost as good. For smaller radices, more passes are needed before the problem size fits into cache, which leads to worse performance. Increasing the number of bits beyond 8 makes it so that the buckets during local sorting are less likely to fit in L1 cache (32 KB on our machine), which causes performance degradation.

Effect of K. Figures 15 and 16 show the running time for different values of K for the 2-path variant of Regions Sort on unif$(10^9, 10^9)$

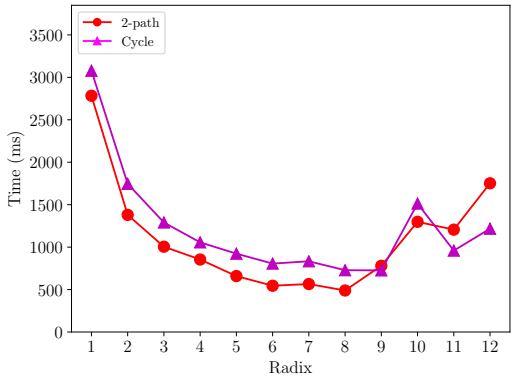

Figure 13: 36-core running time vs. radix size for unif$(10^9, 10^9)$.

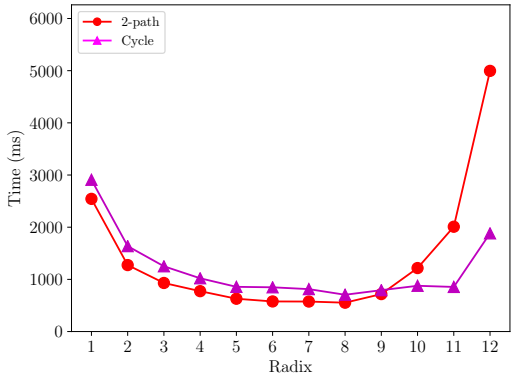

Figure 14: 36-core running time vs. radix size for zipf$_{0.75}(10^9, 10^9)$.

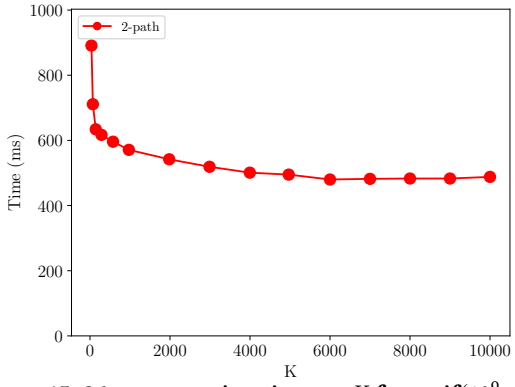

Figure 15: 36-core running time vs. K for unif$(10^9, 10^9)$.

and zipf$_{0.75}(10^9, 10^9)$, respectively. We see that the running time for large values of K is much lower than for small values of K. For small values of K, the local sorts in the first level are less likely to fit in cache, which increases the overall running time.

6 ADDITIONAL RELATED WORK

Axtmann et al. propose a parallel in-place sample sorting algorithm called IPS^{4}o [3]. In sample sort, a sample of keys is sorted and pivots are chosen from the sorted sample to determine bucket boundaries. The remaining keys are inserted into the buckets, and each bucket is recursively sorted. In IPS^{4}o, each processor maintains k buffers of size b, where k is the number of buckets and b is a constant. In the first phase, each processor scans a subarray of n/P keys, placing each key into a buffer corresponding to its bucket; whenever a buffer becomes full, the b keys in the buffer (which we call a chunk) are copied to the beginning of the subarray for that processor. The counts of the number of keys belonging to each bucket are maintained, and a prefix sum across all processors gives the total number of keys per buckets and determines bucket boundaries. In the second phase, each processor moves chunks that are out of place from buckets in its subarray to its target bucket, using a swap buffer of size b. If the source chunk swaps with a chunk in the target bucket, the target chunk is placed in the swap buffer and the processor continues to swap it; otherwise it processes the next chunk in its own subarray. Atomics are needed to increment pointers and implement readers-writer locks on buckets as they can be read and written by multiple processors. The amount of auxiliary space needed by IPS^{4}o is $O(bkP \log n)$. In contrast to IPS^{4}o, our algorithm does not require locks or swap buffers. We compare with the performance of IPS^{4}o in Section 5. We note that our regions graph idea could also be used inside a sample sorting algorithm, and it would be an interesting direction for future work to implement such an approach.

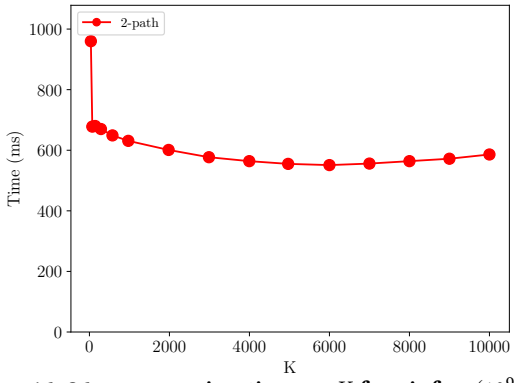

Figure 16: 36-core running time vs. K for zipf$_{0.75}(10^9, 10^9)$.

There has been significant work in the literature on parallel integer sorting. With the exception of PARADIS [12], these algorithms all require $\Omega(n)$ additional space. A standard linear-work and sublinear span stable integer sorting algorithm is described in [44]. Rajasekaran and Reif describe a linear-work polylogarithmic-span integer sorting algorithm for integers in the range $[n \log^{O(1)} n]$, although the algorithm is non-stable [33]. Several super-linear work and polylogarithmic-span stable integer sorting algorithms have been described [1, 5, 18, 19, 29, 34]. However, obtaining a stable linear-work polylogarithmic-span integer sorting algorithm has been a long-standing open problem.

There has also been significant prior work on parallel algorithms for comparison sorting (e.g., [2, 6–8, 11, 13, 14, 16, 20, 21, 28, 35, 38, 42, 47]). In terms of implementations, IPS^{4}o is one of the most recent and has been shown to achieve state-of-the-art performance [3].

Berney et al. present parallel in-place algorithms for constructing implicit search tree layouts from sorted sequences [4]. Our work is complementary in that we can efficiently generate sorted sequences of integers in-place, which can be used as inputs to their algorithms. One of their approaches uses cycles to represent a series of swaps.

The cycles in their approach are determined statically, while in our cycle processing algorithm, the cycles are determined dynamically.

7 CONCLUSION

We have introduced Regions Sort, a novel parallel in-place radix sorting algorithm that is work-efficient and highly-parallel. Our experiments show that Regions Sort achieves good scalability and almost always outperforms existing parallel in-place sorting algorithms. Future work includes optimizing our algorithm for NUMA characteristics to improve scalability, designing a work-efficient in-place radix sorting algorithm with both polylogarithmic span and polylogarithmic space, and using our ideas to design other parallel in-place algorithms.

Acknowledgments. This research was supported by DARPA SDH Award #HR0011-18-3-0007, Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA, DOE Early Career Award #DE-SC0018947, and NSF CAREER Award #CCF-1845763.

REFERENCES

[1] Susanne Albers and Torben Hagerup. 1997. Improved Parallel Integer Sorting without Concurrent Writing. *Information and Computation* 136, 1 (1997), 25 – 51.
[2] Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. 2015. Practical Massively Parallel Sorting. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 13–23.
[3] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. 2017. In-place Parallel Super Scalar Samplesort (IPS^{4}o). In *European Symposium on Algorithms*.
[4] K. Berney, H. Casanova, A. Higuchi, B. Karsin, and N. Sitchinava. 2018. Beyond Binary Search: Parallel In-Place Construction of Implicit Search Tree Layouts. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1070–1079.
[5] P. C. P. Bhatt, K. Diks, T. Hagerup, V. C. Prasad, T. Radzik, and S. Saxena. 1991. Improved deterministic parallel integer sorting. *Information and Computation* 94, 1 (1991), 29–47.
[6] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally Deterministic Parallel Algorithms Can Be Fast. In *ACM SIGPLAN Symposium on Proceedings of Principles and Practice of Parallel Programming (PPoPP)*. 181–192.
[7] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2010. Low Depth Cache-oblivious Algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 189–199.
[8] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2016. Parallelism in Randomized Incremental Algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 467–478.
[9] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. 1991. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 3–16.
[10] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748.
[11] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. 2008. Efficient implementation of sorting on multi-core SIMD CPU architecture. In *International Conference on Very Large Data Bases (VLDB)*. 1313–1324.
[12] Minsik Cho, Daniel Brand, Rajesh Bordawekar, Ulrich Finkler, Vincent Kulandaisamy, and Ruchir Puri. 2015. PARADIS: An Efficient Parallel Algorithm for In-place Radix Sort. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1518–1529.
[13] Richard Cole. 1988. Parallel Merge Sort. *SIAM J. Comput.* 17, 4 (Aug. 1988), 770–785.
[14] Richard Cole and Vijaya Ramachandran. 2017. Resource Oblivious Sorting on Multicores. *ACM Trans. Parallel Comput.* 3, 4, Article 23 (March 2017).
[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3. ed.)*. MIT Press.
[16] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. 2006. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *ACM SIGMOD International Conference on Management of Data*. 325–336.
[17] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-record Synthetic Databases. In *ACM SIGMOD International Conference on Management of Data*. 243–252.
[18] Torben Hagerup. 1987. Towards optimal parallel bucket sorting. *Information and Computation* 75, 1 (1987), 39 – 51.
[19] Torben Hagerup. 1991. Constant-time Parallel Integer Sorting. In *ACM Symposium on Theory of Computing (STOC)*. 299–306.
[20] David R. Helman, David A. Bader, and Joseph JaJa. 1998. A Randomized Parallel Sorting Algorithm with an Experimental Study. *J. Parallel and Distrib. Comput.* 52, 1 (1998), 1 – 23.
[21] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. 2007. AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors. In *International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 189–198.
[22] J. Jaja. 1992. *Introduction to Parallel Algorithms*. Addison-Wesley Professional.
[23] Daniel Jiménez-González, Juan J. Navarro, and Josep-L. Larrba-Pey. 2001. Fast Parallel In-memory 64-bit Sorting. In *International Conference on Supercomputing (ISC)*. 114–122.
[24] Marek Kokot, Sebastian Deorowicz, and Agnieszka Debudaj-Grabysz. 2017. Sorting Data on Ultra-Large Scale with RADULS. In *Beyond Databases, Architectures and Structures*. 235–245.
[25] Marek Kokot, Sebastian Deorowicz, and Maciej Długosz. 2018. Even Faster Sorting of (Not Only) Integers. In *Man-Machine Interactions 5*. 481–491.
[26] Shin-Jae Lee, Minsoo Jeon, Dongseung Kim, and Andrew Sohn. 2002. Partitioned Parallel Radix Sort. *J. Parallel Distrib. Comput.* 62, 4 (April 2002), 656–668.
[27] Charles E. Leiserson. 2010. The Cilk++ concurrency platform. *The Journal of Supercomputing* 51, 3 (2010).
[28] Hui Li and Kenneth C. Sevcik. 1994. Parallel Sorting by Over Partitioning. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 46–56.
[29] Yossi Matias and Uzi Vishkin. 1991. On parallel hashing and integer sorting. *Journal of Algorithms* 12, 4 (1991), 573–606.
[30] Peter M. McIlroy, Keith Bostic, and M. Douglas McIlroy. 1993. Engineering Radix Sort. *Computing Systems* 6, 1 (1993), 5–27.
[31] Duane Merrill and Andrew Grimshaw. 2011. High Performance and Scalable Radix Sorting: A Case Study of Implementing Dynamic Parallelism for GPU Computing. *Parallel Processing Letters* 21, 02 (2011), 245–272.
[32] Orestis Polychroniou and Kenneth A. Ross. 2014. A Comprehensive Study of Main-memory Partitioning and Its Application to Large-scale Comparison- and Radix-sort. In *ACM SIGMOD International Conference on Management of Data*. 755–766.
[33] S. Rajasekaran and J. H. Reif. 1989. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.* 18, 3 (1989), 594–607.
[34] Rajeev Raman. 1990. The Power of Collision: Randomized Parallel Algorithms for Chaining and Integer Sorting. In *Foundations of Software Technology and Theoretical Computer Science*. 161–175.
[35] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. 2011. TritonSort: A Balanced Large-scale Sorting System. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 29–42.
[36] N. Satish, M. Harris, and M. Garland. 2009. Designing efficient sorting algorithms for manycore GPUs. In *IEEE International Parallel Distributed Processing Symposium (IPDPS)*. 1–10.
[37] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. 2010. Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. In *ACM SIGMOD International Conference on Management of Data*. 351–362.
[38] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: the Problem Based Benchmark Suite. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 68–70.
[39] Johannes Singler, Peter Sanders, and Felix Putze. 2007. MCSTL: The Multi-core Standard Template Library. In *Euro-Par*. 682–694.
[40] Malte Skarupke. 2016. I Wrote a Faster Sorting Algorithm. https://probablydance.com/2016/12/27/i-wrote-a-faster-sorting-algorithm/.
[41] Andrew Sohn and Yuetsu Kodama. 1998. Load Balanced Parallel Radix Sort. In *International Conference on Supercomputing (ISC)*. 305–312.
[42] E. Solomonik and L. V. Kalé. 2010. Highly scalable parallel sorting. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–12.
[43] Elias Stehle and Hans-Arno Jacobsen. 2017. A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs. In *ACM SIGMOD International Conference on Management of Data*. 417–432.
[44] Uzi Vishkin. 2010. Thinking in Parallel: Some Basic Data-Parallel Algorithms and Techniques.
[45] Jan Wassenberg and Peter Sanders. 2011. Engineering a Multi-core Radix Sort. In *Euro-Par*. 160–169.
[46] Marco Zagha and Guy E. Blelloch. 1991. Radix sort for vector multiprocessors. In *ACM/IEEE Conference on Supercomputing (SC)*. 712–721.
[47] K. Zhang and B. Wu. 2012. A Novel Parallel Approach of Radix Sort with Bucket Partition Preprocess. In *IEEE International Conference on Embedded Software and Systems*. 989–994.