Relaxing the Counting Requirement for Least Significant Digit Radix Sorts

Stuart Thiel, Larry Thiel, and Greg Butler

Concordia University,
1455 De Maisonneuve Blvd. W., Montreal, Quebec, Canada H3G 1M8.
sthiel@encs.concordia.ca, lhthiel@gmail.com, gregb@encs.concordia.ca

Abstract. Least Significant Digit Radix Sort is a classical distribution sort that makes use of an initial counting pass in its common array-based implementation. In Fast Radix Sort we implement an internal sort that avoids the initial counting step and estimates bin sizes. The dealing pass adapts to errors in estimating bin sizes by using an overflow bin. The experimental results demonstrate a consistent advantage of 4–8% in performance on large data sets across a variety of input distributions.

1 Introduction

It is estimated that upwards of 20% of server time is spent on sorting data [6, 24]. A reduction in server load would translate to a reduction in energy consumption and heat production, saving money for large server facilities. Improving performance for "typical applications" [21, p.708] of sorting that use interval data would deliver such a reduction in server load.

Radix Sorts perform better than Quicksort when they both can be used[7, 16]. Radix Sorts offer $\Theta(n)$ time-complexity, whereas comparison-based sorts have an average time-complexity $\Theta(n \log n)$[7, 25]. The difference in performance between comparison-based and distribution-based algorithms is immediately identifiable in empirical studies on these algorithms [10, 15].

Traditional comparison-based sorting algorithms work directly with ordinal inputs. A subset of ordinal data is interval data, such as integers or strings, where distribution-based sorting algorithms show improvements on running time; distribution sort algorithms are generally Radix Sorts [21, p702–724][16, 3].

Radix Sorts break up elements of an input into digits, or collections of contiguous bits, processing each digit in turn. Sequential processing of the same digit for each element of an input is referred to as a pass. Every pass, each element is placed into a new location, referred to as a bucket. The process of such placements is often called "dealing", a reference to the card-sorting technique that uses Radix Sort. After each pass of dealing, the next digit is considered until all digits have been used and all elements have been dealt into their final, sorted positions [16, 21].

Existing array-based Least Significant Digit (LSD) Radix Sort implementations defensively count elements to avoid collisions during dealing. This initial count reads through all input, considering each element, before beginning the dealing passes which result in a sorted array. Can one avoid the defensive step of initial counting and achieve an improvement in performance?

We introduce Fast Radix, a variation on the internal, array-based Least Significant Digit Radix Sort algorithm that does not take an initial counting step, instead benefiting from the statistical likelihood that the least significant bits will be uniformly distributed when estimating initial bucket sizes for the corresponding digit. Fast Radix resolves bucket overflow only when it occurs, improving performance on uniform distributions as well as other statistical distributions that are traditionally more problematic for Radix Sort variants.

Experiments on machines with a traditional memory hierarchy model considered input arrays with sizes of up to 100 million across both uniform and normal distributions of varying ranges and standard deviations, as well as pre-sorted data and data with varying degrees of structure added. Tests were performed on unsigned 32-bit integers as well as on unsigned 64-bit integers. Traditional LSD Radix Sort[14, p.170], a diverting Most Significant Digit (MSD) Radix Sort [4] and CC-Radix [10] were tested on the same inputs to provide comparison.

Fast Radix reliably outperformed all other algorithms on uniform and normal distributions. While the point at which improvements became apparent was dependent on parameters such as integers size, distribution and digit size, Fast Radix generally showed its improvement in input sizes around 100 thousand, and improvements leveled off in the range of 7–9% for 32-bit integers. For 64-bit integers Fast Radix became reliably better at around 2–5 thousand, and improvements leveled off at between 4–8%.

The rest of this paper is structured as follows: Related background literature is considered, focusing on internal distribution sorts, scoping this paper to internal integer distribution sorts for hierarchical memory models, while acknowledging some approaches outside of that scope. The distribution sorts considered in this paper are described, followed by a description of Fast Radix. The details of the performance experiments are given in terms of software, hardware, measurement approaches and data sets considered. A sample of the results are shown highlighting Fast Radix's improvements. These results are discussed further, examining some of the considerations leading to these results and the choice to focus on a specific subset of the results. The paper then concludes with a summary.

2 Background

Modern literature on Radix Sort focuses on either the implementation of Radix Sort on parallel architectures or on its application for sorting strings [1, 3, 11, 12, 16, 21]. In this century there is less work on single-processor integer sorting: Birkeland's Doctoral thesis considered processing of large data volumes [5], Jimenez-Gonzalez et al. provided a new variant on Radix Sort [10] (implemented as an internal, single-processor sort) and Li et al. considered tuning sorting libraries at installation time, specifically tuning for integer sorting [15]. Most other literature on single-processor sorting is from the mid-1900s.

CC-Radix is Jimenez-Gonzalez et al's cache-aware Radix Sort variant [10] that has, as drawback, that its suggested implementation can save extra reads against the input data at the cost of storing a large number of buffers for each bucket's initial counts [15]. CC-Radix is referenced for considerations of memory hierarchies [17, 23] and, though a single-processor algorithm, for consideration of parallel processing applications [20, 19]. As Fast-Radix proposes an improvement on the order of that offered by CC-Radix for the specific hardware[8] they used, it can be relevant to modern sorting outside the immediate single-processor domain.

Table 1: A Comparison of Properties of Radix Sort Variants

Algorithm	*Indexing Cost*	*Locality*	*Diversion*	*In-place Variant*
LSD Radix Sort	good	bad	no	no
MSD Radix Sort with Diversion	bad	good	yes	yes
CC-Radix	bad–average	good	yes	no
Fast Radix	good	bad	no	no

The established technique to avoid the counting step is to use list-based approaches to LSD Radix Sort, though this requires additional overhead. We have been unable to find any approaches that consider skipping initial counting on single-processor array-based approaches. Wassenberg and Sanders propose skipping initial counting by allocating to each bucket the size of the initial input, using an external sorting approach with the advantage of "multiple terabytes [of space that] are available on 64-bit systems"[25], but this is not feasible for current internal sorting environments. Table 1 summarizes relevant properties of the Radix Sort algorithms used in our comparison.

The reference to CC-Radix here should not be confused with other work by Jimenez-Gonzalez et al. [9] and subsequent work by Peter Sanders et al. [25, 2] on parallel sorting algorithms using "Communication Conscious" Radix Sorts and variants.

```
Input: input, buffer and counters
    in
    $buf \leftarrow initializearray[sizeof(in)]$
    $low \leftarrow initializearray[10]$
    $high \leftarrow initializearray[10]$
Output: sorted input
    in
 1: $DetermineBucketCounts$
 2: for $i = 0$ to $N - 1$ do
 3:     $low[1 + (in[i] \,\%\, 10)]{+}{+}$
 4:     $high[1 + (in[i] \,/\, 10)]{+}{+}$
 5: end for
 6: $DetermineBucketIndices$
 7: for $i = 2$ to $9$ do
 8:     $low[i] \leftarrow low[i] + low[i - 1]$
 9:     $high[i] \leftarrow high[i] + high[i - 1]$
10: end for
11: $SortToBuffferOnLeastSignificantDigits$
12: for $i = 0$ to $N - 1$ do
13:     $buf[low[(in[i] \,\%\, 10)]{+}{+}] \leftarrow in[i]$
14: end for
15: $SortToInputOnMostSignificantDigits$
16: for $i = 0$ to $N - 1$ do
17:     $in[high[(buf[i] \,/\, 10)]{+}{+}] \leftarrow buf[i]$
18: end for
```

Fig. 1: Two-Pass Radix Sort Pseudocode

2.1 LSD Radix Sort

The classical LSD Radix Sort described in Knuth [14, p.170] is an obvious candidate for comparison, as modern implementations have left it relatively unchanged [21, p.707].

After an initial counting pass, the initial dealing pass in LSD Radix Sort considers the least significant digit of each element, dealing each more significant digit in subsequent passes. The number of passes that occur is the integer size divided by the digit size. For example, an 8-bit digit would require four dealing passes for a 32-bit integer.

This LSD approach can be demonstrated by considering the pseudocode of a two-pass LSD Radix Sort as shown in Figure 1, which uses the traditional four parts: Initial frequency counts are made for all bit ranges (lines 2–5). These counts are transformed into indices (lines 7–10). Elements are dealt to a buffer (lines 12–14). Elements are dealt back into the original input (lines 16–18).

Given that LSD Radix Sort must consider each element in the same order every time, placing elements in their buckets in the order of occurrence ensures that this algorithm is stable. In fact, the algorithm relies on this stability [21, p.706]. Sedgewick also points out that LSD Radix Sorts has a "sweet-spot" for short, fixed-length keys, as is the case with 32 or 64-bit integers [21, 724].

One disadvantage is that array-based LSD Radix Sorts cannot be implemented in-place. As Sedgewick and Wayne highlight, the success of the LSD Radix algorithm is dependent on its stabillity [21, p.706]. For an in-place variant, moving one element into its target bucket would require moving another element out, and so forth, creating a cycle similar to that of in-place MSD Radix Sort implementations [5]. As this cycle is processed out of order of the occurrence of elements, there is no way to ensure that like elements are not moved out of stable order.

2.2 MSD Radix Sort with Diversion

Most Significant Digit (MSD) Radix Sort, is still considered an effective approach [7, 4, 5]. The MSD Radix Sort implemented by Birkeland is a traditional implementation that makes use of diversion to a lesser sorting algorithm when buckets are small [5, p.25]. Birkeland's implementation can be readily switched to a non-in-place version similar to Sedgewick and Wayne's implementation [21, p.712] for suitable comparison with LSD variants.

MSD Radix Sort, just as LSD Radix Sort, breaks up an element into digits and considers each digit in turn. Unlike LSD Radix Sort, MSD Radix Sort starts with the most significant digit. This provides an opportunity for diversion, leading to the variation that we will consider.

Diversion involves switching to a low-overhead sorting algorithm on small input in order to achieve a performance improvement. A diversion threshold is an empirically determined value for what constitutes "small" on a given platform for a given initial sorting algorithm and a given diversion algorithm [16].

MSD Radix Sort must count occurrences just as in LSD Radix Sort. Upon calculating indices, MSD Radix Sort deals to buckets as LSD Radix Sort does. Where LSD Radix Sort treats the newly filled buckets as a new input array and process them all at once, MSD Radix Sort considers each bucket in turn, recursively sorting it alone on the next most significant digit. Within each recursive step, new counts must be made and new indices calculated [21, p.710–712].

At the end of each pass, all elements within a given bucket are ordered with respect to adjacent buckets. In the last pass, diversion aside, each bucket will contain either zero or one elements, hence that section of the data is guaranteed sorted. When diversion is applied to a bucket, it is also guaranteed sorted. When the last bucket is sorted, the whole array must therefore be sorted [21, p.710–712].

The advantage of LSD Radix Sort over the Most Significant Digit (MSD) Radix Sort is that the process of converting occurrence counts into indices is relative to the sum of the bucket size for each pass. A digit size of 16 bits would yield the 65536 indices being calculated twice. With a small digit size, such as 8 bits, one would only calculate 256 indices four times.

If early diversion is achieved, MSD Radix Sorts can be very fast. If diversion cannot be achieved, the effort of adjusting indices for MSD Radix Sort would be on the order of the size of the key being considered. With 32-bit integers, this is $\sim$4 billion operations. For 64-bit integers, lack of diversion can quickly lead to using all resources on a machine.

While the in-place variant is not evaluated in this paper, MSD Radix Sort can be implemented in that fashion at the cost of stability [5, 16]. When space is a consideration, this can outweigh time advantage.

2.3 CC-Radix

Within the past few decades, the only new internal Radix Sort that we have found is Jimenez-Gonzalez et al's CC-Radix, which provides a cache-aware approach that merges aspects of both MSD and LSD Radix Sorts [10, 15].

CC-Radix seeks to apply MSD Radix Sort passes until a bucket is small enough to either divert, as in the MSD Radix Sort considered above, or switch to LSD Radix Sort for that bucket. The LSD Radix Sort passes can skip the high-order digits already put in place during the prior MSD Radix Sort passes [10, 15].

Most of the advantages attributed to this approach apply to external and parallel applications, specifically better cache and Translation Lookaside Buffer (TLB) hits, but also the opportunity to benefit from diversion and the reduced number of index calculations that LSD Radix Sort diversion provides [10, 15].

The disadvantages are that it is a slightly more complex algorithm, and that the process of calculating indices for each bucket when applying the LSD technique can be costly. While one can trade space to count occurrences as done in MSD passes, the calculation of the indices must still be done for each bucket [15]. It also shares the disadvantage of MSD Radix sorts when timely diversion cannot be achieved.

3 Fast Radix Sort

This section presents the Fast Radix sorting algorithm and the correctness of the algorithm.

3.1 The Algorithm

Fast Radix takes an input array of integer elements, where each element will be considered as being composed of several digits based on a fixed digit size. For each such digit, a set of buckets will be created, exactly as is done for LSD Radix Sort. An additional set of buckets will be created to track overflow.

Bucket positions for the first dealing pass will be estimated based on the statistical likelihood of a uniform distribution for the least significant digit, leading to a the buffer array slightly larger than the initial input array to allow each bucket to be the same size.

While the first dealing pass is performed, counts of occurrence for each other digit will be tallied. During this first pass, any elements that overflow the estimated bucket sizes will be dealt back to the beginning of the input array, where there is necessarily room. Counts of occurrence for the least significant digit will be tallied for overflow elements.

```
Input: input, buffer and counters
    in
    /* n-extra space to deal into on odd passes */
    $buf \leftarrow initializearray[sizeof(in)]$
    /* bucket positions for the low and high digits */
    $low \leftarrow initializearray[0xffff + 2]$
    $high \leftarrow initializearray[0xffff + 2]$
    /* bucket positions into overflow buffer */
    $over \leftarrow initializearray[0xffff + 3]$
    /* overflow space to deal into if there is overflow */
    $overb$
Output: sorted input
    in
 1: $InitializeBucketOffsets$
 2: $estimate \leftarrow (N \gg 16) + 1$
 3: for $i = 0$ to (0xFFFF-1) do
 4:     $low[i] \leftarrow i * estimate$
 5: end for
 6: $SortToBuffferOnLeastSignificantDigits$
 7: $overflowOffset \leftarrow 0$
 8: for $i = 0$ to $N - 1$ do
 9:     $high[1 + (in[i] \gg 16)]$++
10:     $target \leftarrow (in[i]$ & $0xFFFF)$
11:     if low[target] == (target+1)*estimate then
12:         $in[overflowOffset$++$] \leftarrow in[i]$
13:         $over[target + 2]$++
14:     else
15:         $buf[low[target]$++$] \leftarrow in[i]$
16:     end if
17: end for
18: $DetermineBucketIndices$
19: for $i = 3$ to $0xFFFF$ do
20:     $over[i] \leftarrow over[i] + over[i + 1]$
21: end for
22: for $i = 2$ to $(0xFFFF - 1)$ do
23:     $high[i] \leftarrow high[i] + high[i + 1]$
24: end for
25: $ProcessOverflow$
26: $overb \leftarrow initializearray[overflowOffset]$
27: for $i = 0$ to $overflowOffset$ do
28:     $overb[over[1 + (in[i]$ & $0xFFFF)]$++$] \leftarrow in[i]$
29: end for
30: $SortToInputOnMostSignificantDigits$
31: for $i = 0$ to 0xFFFF do
32:     for $j = (estimate * i)$ to (low[i]-1) do
33:         $in[high[buf[j] \gg 16]$++$] \leftarrow buf[j]$
34:     end for
35:     for $j = over[i]$ to (over[i+1]-1) do
36:         $in[high[over[j] \gg 16]$++$] \leftarrow overb[j]$
37:     end for
38: end for
```

Fig. 2: Two-Pass Fast Radix Pseudocode

After the first dealing pass, the tallied counts are converted into bucket positions as in LSD Radix Sort. An overflow buffer is allocated, and the overflow elements are dealt into it, freeing up the initial input array.

The second pass deals from the buffer array, stopping at bucket boundaries to deal from corresponding overflow buckets. Any gaps between estimated buckets are skipped. As bucket positions are now known, there is no overflow during this pass.

When this pass completes, the input array contains elements exactly as they would be after two passes of LSD Radix Sort. All subsequent passes perform exactly as in LSD Radix Sort, leaving the sorted elements in the input array.

We can compare Fast Radix to the LSD Radix Sort shown in Figure 1, though it is broken into more parts. An $\Theta(n)$ initial counting pass through the input is completely removed and replaced with a $\Theta(1)$ initialization based on an estimate of expected uniform distribution for low-order bits (lines 2–5). The subsequent initial dealing pass has the responsibility of performing latter counting passes (line 9) and checking for overflow before each deal (line 11). In the event of overflow, dealing is to an overflow buffer at the beginning of the input array and processing book-keeping (lines 12–13). The high pass deal is then processed similarly to the conventional method, except that the dealing is from two sources, the regular buffer and the overflow buffer.

The Fast Radix algorithm shown in Figure 2 uses two passes and a radix of 16 bits. The radix is a power of two with shift and mask operators allowing efficient computation of the low and high-order digits.

The difference between Fast Radix and LSD Radix Sort is that the initial counting pass is omitted and replaced with a check for overflow and processing the overflow. The check for overflow is much less intensive than the whole extra counting-pass performed by LSD Radix Sort, and the occurrence of overflow is rare in all but pathological cases, this difference should demonstrate a consistent improvement across a variety of distributions.

Theorem 1. *The Fast Radix sorting algorithm correctly sorts input.*

Proof. Given that, during its first dealing pass, Fast Radix moves the same elements in the same order, into either an estimated storage area or an overflow storage area, and given that these paired storage areas together correspond exactly to the concept of bucket described in LSD Radix Sort in terms of subsequent sequential access of elements contained in them, and given that all subsequent dealing passes perform the same way as in LSD Radix Sort, then if the original LSD Radix Sort algorithm is correct, Fast Radix must also be correct. □

4 Performance Experiments

Tests were performed with custom written c++ code and the standard utility std::sort. Tests were performed on a high end computational machine with large cache and an overclocked CPU. A framework was developed to support the scripting of tests against varying distributions and input sizes having varying properties, allowing for repeatable results.

32-bit integers were tested, but we focus on the more contemporary 64-bit integer data sets. Uniform distributions were evaluated for multiple ranges, including the maximum range for the integer size used. Normal distributions were tested with standard deviations of $\frac{1}{3}2^{63}$, 2^{51}, 2^{30} and 2^{10}. The choice of standard deviations attempted to capture the range of distributions used in Li et al. [15], but applied for 64-bit integers.

Pre-ordered inputs consisting of integers 1–n were also tested. To test expected pathological cases for Fast Radix, the previously discussed uniform and normal distributions were tested with structure applied, such as having only even integers, or only integers that were multiples of 10.

Software Used The utility std::sort was the Quicksort comparison-based reference against the distribution-based sorts. All other algorithms used were coded by Stuart Thiel based on the literature already cited. All algorithms were tuned; in the case of MSD Radix Sorts optimal thresholds for best dealing with large input sizes were used; in the case of CCRadix, LSD Radix Sort diversion threshold was tuned based on reported L1 cache size for the hardware platform and confirmed with empirical testing, diversion thresholds were empirically tested and we selected one that offered much better performance for small values with no noticeable impact for large input sizes.

Sorting algorithms were implemented to support 8-bit and 16-bit digits with 4-bit digits being identified as performing poorly and so not considered here [22]. 8-bit digits performed better for all Radix Sort implementations, so we will focus on those implementations.

Hardware Platform Experiments were run on a Hypertec Systems Kronos with an Intel i7 3960X 3.3Ghz processor that was overclocked to 4.8Ghz, with 32GB of DDR3(17000) RAM. This system made use of the traditional memory heirarchy model, being composed of three layers of cache aside from main memory.

Organization of Runs and Measurements The high resolution clock from the standard `<chrono>` library was used to get the μs-accurate timings. The standard `<random>` library was used for uniform and normal distributions [18]. The normal distribution code was re-written for 64-bit integers as what appears to be an error in broad distributions at high resolution appeared to be in the production library; the re-write using the same polar approach popularized by Knuth [13, p.122] and used in both the standard `<random>` library and the Java equivalent.

Timings before and after each sort were kept and the difference recorded. To reduce the noise of initial array allocation, for each algorithm a different system process was used and it started with the largest value of n, repeating run-count times on different input each time before stepping down and repeating.

Each algorithm was tested against the same distributions and inputs and the timings were recorded. Results for each size of input for a given algorithm and distribution were averaged to further reduce data noise.

Tests were run up to 200 million in size, but results stabilized readily around 100 million, so those values will be reported. Stepping was varied depending on range, with fewer steps being recorded for large input sizes. At each step, the average of all runs was reported. 5 runs were performed for ranges below 100 million, with a step of 10 million. 10 runs were performed for ranges below 1 million, with a step of 100 thousand. 100 runs were performed for ranges below 10 thousand, with a step of 1 thousand.

The system appeared to “warm-up” on initial runs, and thus if we wanted runs with inputs of size 100 million, we would start gathering data at size 120–150 million. This led to repeatable results at the target range, whereas the “warm-up” range occasionally gave varying results.

The configuration attempted to minimize overhead while allowing bulk tests to be run. Timing tests were run on a system without a graphical desktop and with no other users logged in, further reducing overhead.

The seeds used in all tests were generated in advance and stored in a binary file, allowing the same seeds to be used for each algorithm for a given distribution, maximum n, run-count and step size. Each run at each value of n uses a distinct seed. This provides variance between each run, while running each algorithm on the same inputs. All results for a given algorithm and distribution were output to a binary file with enough information to duplicate the test.

Datasets A test-harness was developed in C++ to support μs-accurate wall-clock timing of sorting for combinations of algorithms, distributions and input sizes. Sorting algorithms that implemented a "Sorter" interface could be tested against inputs produced by classes implementing a "Generator" interface. Command-line parameters indicated which algorithm to run tests on, details for the input generator and further details regarding input sizes and how many times to run.

Table 2: Average Runtimes of 32-bit Algorithms in Microseconds for Various n with Uniform Distributions

	Digit Size	*1K*	*10K*	*100K*	*1M*	*10M*	*100M*
Quicksort	N/A	27	322	3981	47269	550409	6331597
MSD Radix	8-bit	11	107	918	12390	174132	1563888
CC-Radix	8-bit	11	125	901	13752	106527	962810
LSD Radix	8-bit	10	67	761	7738	91237	914170
Fast Radix	8-bit	12	67	724	7252	84392	846983

4.1 Experiment Results

The results in this section will show selected timing results and comparisons against the LSD Radix Sort baseline. While MSD Radix Sort and CCRadix Sort show good performance with some inputs, LSD Radix Sort performed well in all data sets, justifying the choice of LSD Radix Sort as the baseline against which all other results will be considered.

Table 3: Speed of sorting algorithms for inputs of 100 million, normalized against 8-bit digit Radix Sort.

	Normal				Uniform		
64-bit algorithms	2^{10}	2^{30}	2^{51}	$\frac{1}{3}2^{63}$	2^{16}	2^{31}	$2^{64}-1$
Fast Radix	108.12%	106.20%	105.03%	104.16%	106.93%	106.16%	104.05%
Quicksort	50.90%	33.31%	36.80%	37.85%	42.63%	32.67%	37.72%
MSD Radix	SEGF	68.68%	78.01%	134.80%	SEGF	66.44%	139.85%
CCRadix	SEGF	73.79%	39.91%	108.37%	64.64%	76.80%	108.91%

Results are presented in Tables 2 and Table 3. Table 2 shows the runtimes in microseconds of algorithms considered in this study, at various input sizes and for a uniform distribution across the full range of 32-bit integers. Table 3 shows the performance of evaluated algorithms on an input size of 100 million. The performance is given as a percentage of speed relative to the LSD Radix Sort baseline to better highlight improvements. Thus Fast Radix processed 100 million integers 8.12% faster than LSD Radix Sort for a normal distribution with a standard deviation of 2^{10}.

Table 3 uses the notation SEGF where execution exhausted resources and caused a segmentation fault. This happened only with MSD Radix Sort and CCRadix, and encompasses both traditional segmentation faults and the OS crashing with a memory shortage error. Results for CCRadix at 2^{30} and 2^{51} had such errors that were avoided by adjusting the start of the tests, thus using different seeds. Implementations using custom stacks and iterations are known to mitigate these problems, but the underlying issue is well known. The current implementations highlight where these algorithms perform better than Fast Radix.

5 Discussion

Testing under C++ gave fairly consistent timing results, even with random seeds. Once algorithms started doing better given the size of input, the lead appeared to be maintained. Table 2 highlights the general trend, including Quicksort.

Experimental results showed that 8-bit digit Radix Sorts performed better than larger digit sizes, so we focus on those sorts. Both MSD Radix Sort and CC-Radix performed better at uniform and very broad normal distributions, but ran very slowly, or in some cases used too many resources to complete on narrower distributions or in uniform distributions across narrow ranges. Table 3 shows areas where MSD Radix Sort and CCRadix sort failed, with SEGF representing segmentation faults or other crashes due to resource exhaustion. Even mitigating some of the excess memory use, it still identifies the expected and well-documented performance problems of those approaches in adverse distributions; it is worth noting their excellent performance with optimal distributions.

Impact of Distributions The decision to use uniform distributions was based on the frequency with which it is used in the literature [7, 10, 15, 22]. Normal distributions better reflect integer data that is more likely to be a specific value and differs from that value because of a number of variables. To that end, normal distributions are often considered as more realistic for certain types of data. However, they do not appear as often in the Radix Sort literature, though Li et al. [15] did use them with "standard deviations of sizes $4^n * 512$ with n ranging from 0 to 8", or powers of 2 between 2^{10} and 2^{18}. Our normal distributions covered a broader range of distributions for application against 64-bit integers.

Pre-ordered data was tested, but neither helped nor hindered Fast Radix with respect to LSD Radix Sort, with results varying according to the underlying distribution.

Structured inputs, such as having only even integers, led to Fast Radix performing marginally worse than LSD Radix Sort, though certain structures with 32-bit integer sizes showed unexpected small performance gains over LSD Radix Sort, with results appearing to oscillate. In general this behaved similarly to the results of having a very narrow distribution but did not seem a likely occurrence in practice, and this research did not consider the small unexpected gains, save to note them here.

Impact of Cache Hits Cache hits were noticeable, particularly when tuning the MSD Radix and CCRadix algorithms. However, once tuned, cache advantage conferred by narrow distributions seemed to greatly improve all the Radix Sorts considered, leaving Fast Radix still ahead of the next nearest competitor given 32-bits, but allowing Fast Radix to be surpassed for certain broad distributions given 64-bit integers.

Cache hits were also the primary consideration for the 8-bit digit size as 16-bit digit implementations had many more cache misses.

Issues with Choice of Datasets 32-bit integers were commonly used in the literature [10, 15], though in part because 32-bit was the architecture in the 90s where much of the literature is from, with integer data types in Java and C++ correspondingly being 32 bits. Re-implementing in 64-bit and using 64-bit datasets reflects modern technology, and better highlights some of the advantages of the alternate algorithms considered, but also highlights some catastrophic results on less favorable distributions.

6 Conclusion

Fast Radix performs better than its competitors in all but two situations. First, when input sizes are very small or distributions are very narrow. Second, when inputs are large and distributed fairly uniformly across a wide range.

As shown in Tables 3, CCRadix and MSD Radix performed well in broad uniform and normal distributions, in the case of MSD Radix, performing exceptionally well, but with catastrophic performance on narrower distributions. Fast Radix consistently outperformed LSD Radix in all cases, and its improvement was reliably apparent given input sizes of 1000. Fast Radix's performance gains over LSD Radix Sort were in the 4–8% range.

This paper demonstrates that improvements can still be made on a well established algorithm, reinforcing the Engineering adage that something is not complete when there is nothing left to add, but when there is nothing left to take away.

7 Acknowledgements

The authors would like to thank Fine Arts Research Facilities, the Department of Computer Science abd Software Engineering, Concordia University, the Natural Science and Engineering Research Council of Canada (NSERC), and the Canada Foundation for Innovation (CFI), for their support in terms of both finance and equipment.

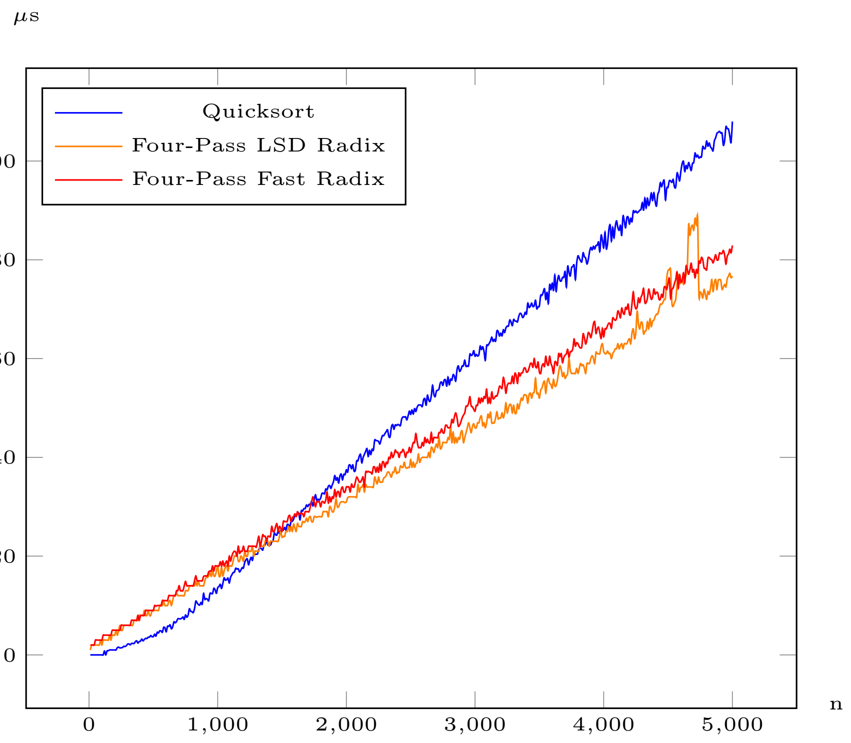

Fig. 3: Standard Sort, LSD Radix Sort and Fast Radix working with “real” event-timing data from one of KitFox’s tests (5878 records, 215 unique).

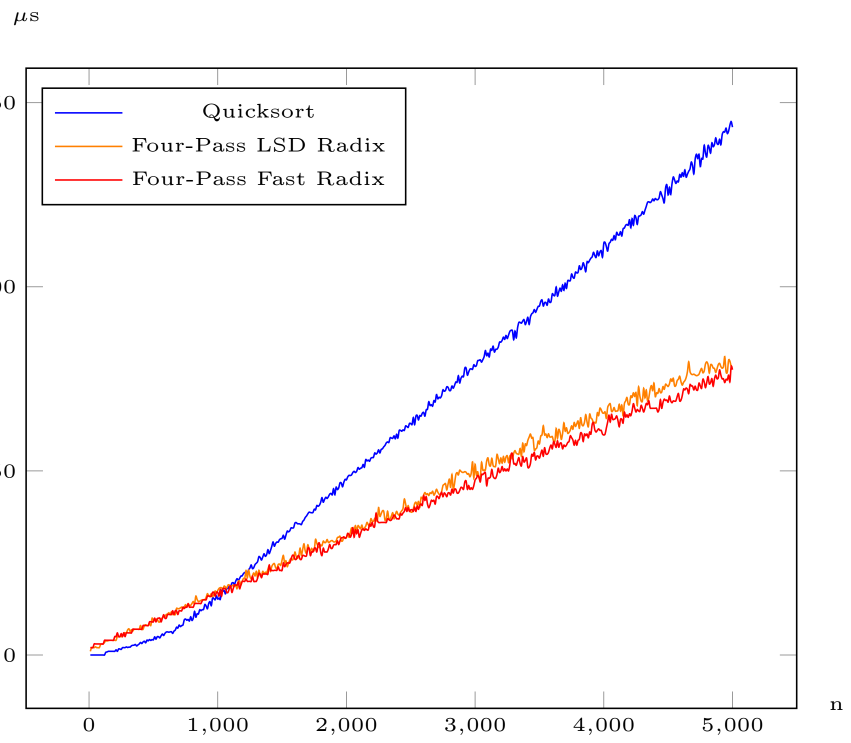

Fig. 4: Standard Sort, LSD Radix Sort and Fast Radix working with “real” isbn data from montreal’s public libraries.

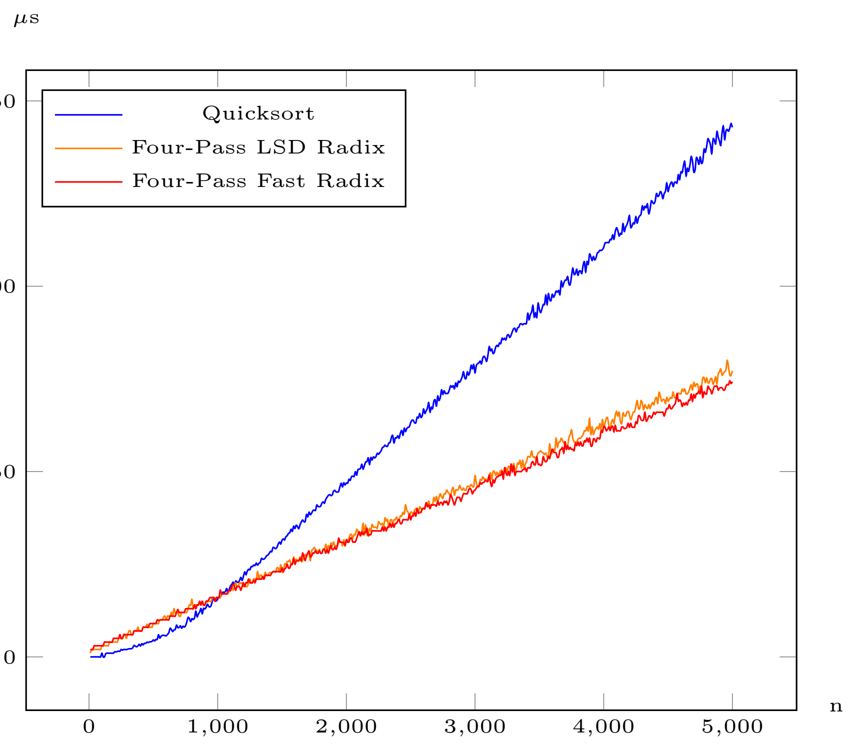

Fig. 5: Standard Sort, LSD Radix Sort and Fast Radix working with "real" customer data: all mobile numbers registered with a small Bahamas telco (13722 customers).

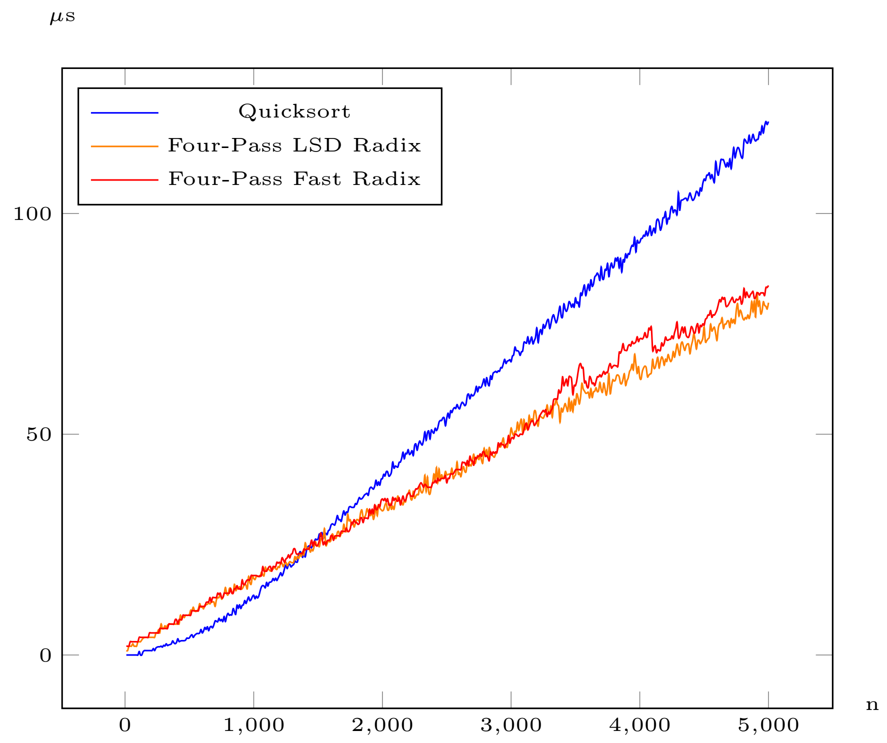

Fig. 6: Standard Sort, LSD Radix Sort and Fast Radix working with “real” customer data: all call data on a specific date for some sort of small Quebec phone company (36k numbers, 4177 unique).

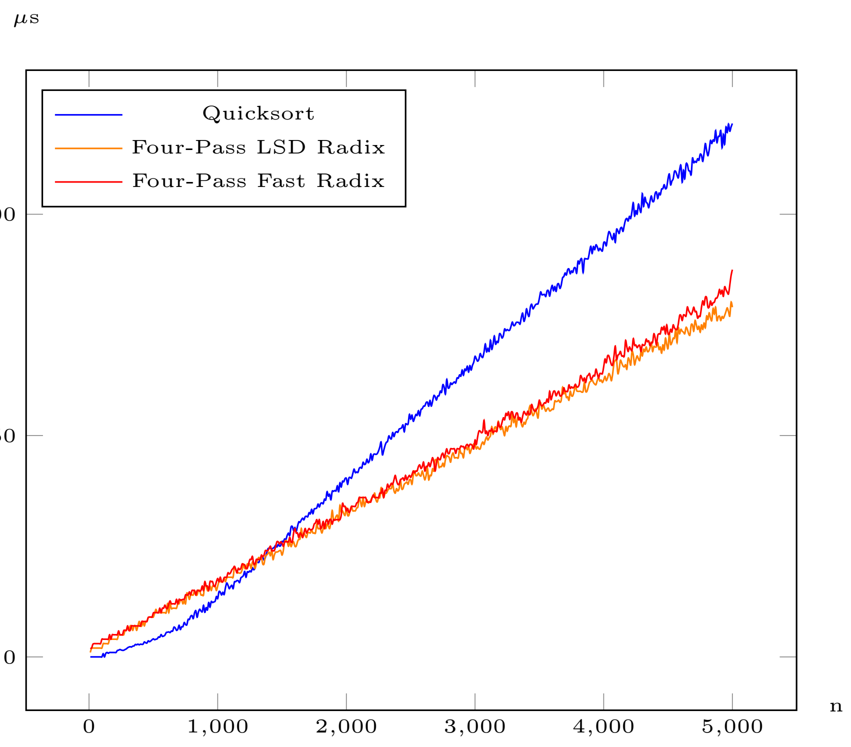

Fig. 7: Standard Sort, LSD Radix Sort and Fast Radix working with “real” customer data: all call data on a specific date for some sort of small Quebec phone company (84k numbers, 15955 unique).

References

1. Andersson, A.: Faster deterministic sorting and searching in linear space. In: Proceedings, 37th Annual Symposium on Foundations of Computer Science, 1996. pp. 135–141 (1996)
2. Beckmann, A., Meyer, U., Sanders, P., Singler, J.: Energy-efficient sorting using solid state disks. Sustainable Computing: Informatics and Systems pp. 151–163 (2011)
3. Bentley, J.L., Sedgewick, R.: Fast algorithms for sorting and searching strings. In: Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 360–369. Philadelphia, PA, USA (1997)
4. Benton, S.C.: Rethinking sorting. J. Comput. Sci. Coll. 15(3), 161–166 (2000)
5. Birkeland, O.R.: Searching large data volumes with MISD processing. Ph.D. thesis, Norwegian University of Science and Technology, Department of Computer and Information Science (2008)
6. Chen, S., Reif, J.: Using difficulty of prediction to decrease computation: fast sort, priority queue and convex hull on entropy bounded inputs. In: Proceedings, 34th Annual Symposium on Foundations of Computer Science, 1993. pp. 104–112 (1993)
7. Davis, I.J.: A fast radix sort. The Computer Journal 35(6), 636–642 (1992)
8. Jiménez-González, D., Guinovart, E., Larriba-Pey, J.L., Navarro, J.J.: Sorting on the SGI Origin 2000: Comparing MPI and Shared Memory Implementations. In: Proceedings. XIX International Conference of the Chilean. pp. 209–215 (1999)
9. Jiménez-González, D., Larriba-Pey, J.L., Navarro, J.J.: Communication Conscious Radix Sort. In: Proceedings of the 13th international conference on Supercomputing. pp. 76–82. ACM (1999)
10. Jiménez-González, D., Navarro, J., Larriba-Pey, J.: CC-Radix: a cache conscious sorting based on Radix sort. In: Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on. pp. 101–108 (2003)
11. Kärkkäinen, J., Rantala, T.: Engineering Radix Sort for Strings. Lecture Notes in Computer Science, vol. 5280, pp. 3–14 (2009)
12. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. Journal of the ACM 53(6), 918–936 (2006)
13. Knuth, D.E.: The Art of Computer Programming, 3rd edn., vol. 2. Seminumerical Algorithms (1998)
14. Knuth, D.E.: The Art of Computer Programming, volume 3: Sorting and Searching. Addison Wesley, 2nd edn. (1998)
15. Li, X., Garzarán, M.J., Padua, D.: A dynamically tuned sorting library. In: IEEE International Symposium on Code Generation and Optimization. pp. 111–122 (2004)
16. Mcllroy, P.M., Bostic, K., Mcllroy, M.D.: Engineering Radix Sort. Computing systems 6(1), 5–27 (1993)
17. Meyer, U., Sanders, P., Sibeyn, J.: Algorithms for memory hierarchies: advanced lectures. Springer-Verlag (2003)
18. Musser, D.R., Saini, A.: The STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library. Addison Wesley (1995)
19. Rashid, L., Hassanein, W.M., Hammad, M.A.: Analyzing and enhancing the parallel sort operation on multithreaded architectures. The Journal of Supercomputing 53(2), 293–312 (2010)
20. Satish, N., Kim, C., Chhugani, J., Nguyen, A.D., Lee, V.W., Kim, D., Dubey, P.: Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. pp. 351–362. ACM (2010)

21. Sedgewick, R., Wayne, K.: Algorithms. Addison-Wesley, 4th edn. (2011)
22. Shaffer, C.A.: Data Structures and Algorithm Analysis. PDF/Shaffer, 3.2.0.10 (C++ Version) edn. (2013)
23. Sinha, R., Zobel, J., Ring, D.: Cache-efficient string sorting using copying. Journal of Experimental Algorithmics 11, 1–2 (2007)
24. Wagar, B.: System for MSD radix sort bin storage management (1995), US Patent 5,440,734
25. Wassenberg, J., Sanders, P.: Engineering a multi-core radix sort. In: Euro-Par 2011 Parallel Processing, pp. 160–169. Springer (2011)