PARADIS: A PARALLEL IN-PLACE RADIX SORT ALGORITHM

ROLLAND HE
rhe@stanford.edu

Abstract. Radix sort is an efficient non-comparative integer sorting algorithm which can be generalized to floating point numbers and strings of characters. It is able to achieve a theoretical linear runtime complexity; moreover, in-place radix sort is able to do this, while only using constant memory. In this paper, we discuss a few of the innovations which have helped parallelize in-place radix sort, which have most recently led to PARADIS, an efficient parallel in-place radix sort algorithm. Experimental results show that PARADIS performs quite well

1. Introduction

Radix sort has long been known and used as a sorting method, dating back to 1887 with the work of Herman Hollerith on tabulating machines [5]. Its attractiveness comes from its simplicity, as well as its theoretical worst-case linear runtime complexity of $O(kn)$, where k is a constant representing the maximum number of digits for any given number in the data. As a result, radix sort is able to beat the runtimes of any comparison-based sorting method [6]. Unfortunately, the constant k factor often ends up being non-negligible in practice, and can often exceed $\log n$. Therefore, standard radix sort falls short in many applications.

In recent years, with the rise of parallel computing, many advancements have been made to parallelize radix sort [2][3][4]. In addition, the ever-increasing amounts of data available has resulted in greater memory demands; therefore, an in-place version of radix sort was also developed. However, combining these two ideas of parallelization and in-place sort turns out to be difficult for due to the inherently sequential nature of standard in-place radix sorting.

Most recently, in 2014, researchers at IBM have developed PARADIS, an efficient parallel in-place radix sort algorithm [1]. PARADIS introduces two main improvements over previous radix sort algorithms. First, it introduces a speculative permutation/repair strategy, which is able to efficiently parallelize the in-place permutation process of radix sort. In addition, the algorithm also implements a distribution-adaptive load balancing technique to balance the work done by different processors during recursive calls.

2. Background and Prior Results

2.1. **LSD Radix Sort.** We will continue adopting the convention of using k to represent the maximum number of digits in any element of the data. In addition, we will assume a radix of 10 for representational simplicity – we can easily extend our results to any radix value. The main idea behind radix sort is to use k passes of the data, one for each digit position, and sort all the data based on the digit at that position in a single pass. Since we only have to sort single digits, we can

instead place each element in one of 10 buckets, with each bucket corresponding to a certain digit – this allows us to avoid the $O(\log n)$ barrier of comparison sorts [6]. Below is pseudocode for the naive radix sort algorithm: Since the algorithm

Algorithm 1 Naive LSD radix sort algorithm

```
function naiveRadixSort(data, k)
    buckets = list of 10 buckets
    for i from 1 to k do
        for elem in data do
            d = i'th least significant digit of elem
            Place elem in buckets[d]
        end for
    end for
    Replace data with the elements from buckets (in the same order)
    return data
end function
```

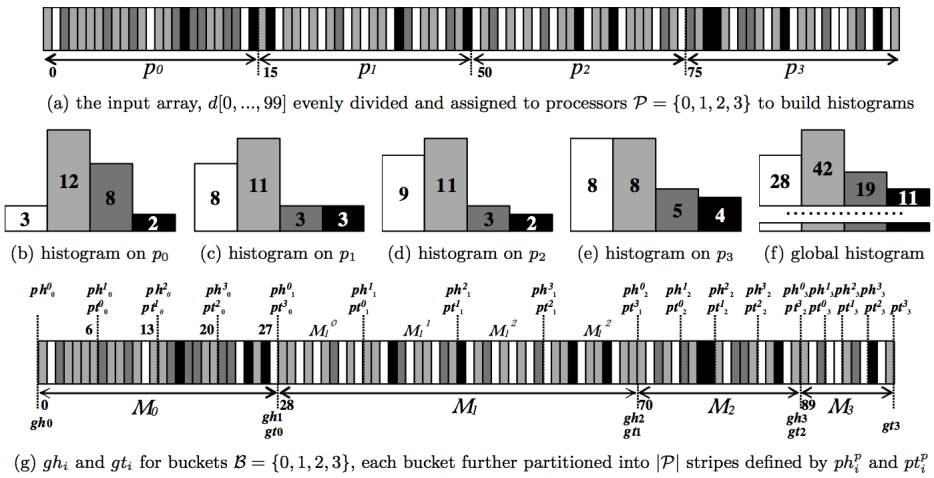
goes through k passes of the data and performs constant work for each element, the total runtime is $O(kn)$. Moreover, the memory complexity is $O(n)$, as during any given pass, an auxiliary data structure needs to be used to bucketize all the elements.

2.2. **MSD Radix Sort.** LSD radix sort unfortunately lends itself to a sequential nature. In order to introduce some parallelization, MSD radix sort can be used. Instead of starting from the least significant digit, we instead start sorting from the most significant digit. The algorithm is as follows: After the first pass through the

Algorithm 2 MSD radix sort algorithm

```
function msdRadixSort(data, l, k)
    buckets = list of 10 buckets
    if number of elements in data = 1 then
        return data
    end if
    for elem in data do
        d = l'th most significant digit of elem
        Place elem in buckets[d]
    end for
    if l ≤ k then
        for bucket in buckets do
            bucket = msdRadixSort(bucket, l + 1, k)
        end for
    end if
    Replace data with the elements from buckets (in the same order)
    return data
end function
```

data, we have 10 buckets just as previously, and can make recursive calls to sort each bucket. Furthermore, since the buckets do not depend on each other, these



(a) the input array, $d[0, ..., 99]$ evenly divided and assigned to processors $\mathcal{P} = \{0, 1, 2, 3\}$ to build histograms

(b) histogram on p_0 (c) histogram on p_1 (d) histogram on p_2 (e) histogram on p_3 (f) global histogram

(g) gh_i and gt_i for buckets $\mathcal{B} = \{0, 1, 2, 3\}$, each bucket further partitioned into $|\mathcal{P}|$ stripes defined by ph_i^p and pt_i^p

FIGURE 1. Parallel histogram building and preparation for in-place MSD radix sort, using 4 processors with 4 buckets. Here, gh and gt represent the head and tail pointers, respectively. Image used from [1].

calls are independent, and can be parallelized. Therefore, the work and depth can be represented as follows:

$$W(n) = 10W\left(\frac{n}{10}\right) + O(n) \tag{2.1}$$

$$D(n) = D\left(\frac{n}{10}\right) + O(n) \tag{2.2}$$

which can easily be solved via the Master Theorem to give $W(n) = O(n \log n)$ and $D(n) = O(n)$.

A further optimization can be made by parallelizing the bucketizing of elements. Instead of scanning through the elements sequentially, we can instead assign each processor an equal chunk of the data to place in the buckets, since placing each element in a bucket can be done independently. While this does nothing to lower the work, it does improve the depth:

$$D(n) = D\left(\frac{n}{10}\right) + O(1) \tag{2.3}$$

which can be solved to give $D(n) = O(\log n)$.

2.3. **In-Place MSD Radix Sort.** In order to reduce the amount of extra memory needed from $O(n)$ to $O(1)$, we can no longer rely on a separate buckets data structure. Instead, we need to swap elements in place. In order to successfully permute the elements into their correct positions, we modify our MSD radix sort algorithm.

Instead of explicitly bucketizing each element, we build a histogram to count the number of elements that belong to each bucket – this will require a single pass through the data. We can once again take advantage of parallelization by splitting this counting between processors. Thus, assuming there are p processors, each process will have to only count $O\left(\frac{n}{p}\right)$ elements; at the end, we simply add the corresponding counts together to get the number of elements in the data that

belong to each bucket (which can also be done in parallel via parallel summation). Suppose we define this building of the histogram as the function:

buildHistogram($data, l, k, p$) – Returns an array with 10 elements of counts corresponding to the number of elements in $data$ with digit d at the l'th MSD position.

We omit the pseudocode for the histogram building, as it is trivial and described above. With the histogram built, we can proceed to the rest of the algorithm. We want to use head and tail pointers to represent the start and end of each bucket; then, we iterate through the data, continually swapping elements into their correct bucket. The head and tail pointers will keep track of which elements are already swapped in the correct bucket, and will be incremented/decremented accordingly. Finally, after all elements are swapped to the correct buckets, we can continue sorting on the next digit, just as before. Figure 1 shows the histogramming process. The pseudocode of the algorithm is below:

Algorithm 3 MSD in-place radix sort algorithm

```
function inPlaceMSDRadixSort(data, l, k, p)
    if number of elements in data = 1 then
        return data
    end if
    histogram = buildHistogram(data, l, k, p)
    heads = [0]
    tails = [histogram[0]]
    for i from 1 to 9 do
        heads[i] = heads[i − 1] + histogram[i − 1]
        tails[i] = tails[i − 1] + histogram[i]
    end for
    for i from 0 to 9 do
        while heads[i] < tails[i] do
            elem = data[heads[i]]
            while elem is not in the correct bucket do
                b = correct bucket of elem
                Swap values of elem and data[heads[b]]
                heads[b] + +
            end while
            data[heads[i]] = elem
            heads[i] + +
        end while
    end for
    if l < k − 1 then
        for i from 1 to 9 do
            d = data from bucket[i]
            inPlaceMSDRadixSort(d, l + 1, k, p)
        end for
    end if
    return data
end function
```

It can easily be verified that the work is still $O(n \log n)$, as the recurrence for work stays the same. Moreover, it is also clear that only a constant amount of

memory is required for this algorithm. However, depth is now back to $O(n)$, as the permutation process is sequential, and therefore requires total $O(n)$ work and depth.

3. PARADIS

PARADIS improves upon the existing method parallel in-place radix sort algorithm with 2 new ideas: speculative permutation and distribution-adaptive load balancing.

3.1. **Speculative Permutation.** The permutation idea presented in Algorithm 3 is inherently sequential, due to the fact that any one bucket can swap elements with any other bucket. This results in read-write dependency, which prevents effective parallelization. Speculative permutation solves this problem by giving each processor exclusive ownership of multiple contiguous data segments, thus avoiding read/write conflicts, as can be seen in Figures 1 and 2.

However, there is no longer guarantee that all elements will be permuted correctly at the end of the permutation phase, as each processor will only be able to swap with a limited number of elements with each bucket. Thus, a second repair phase is introduced after permutation to fix any elements not currently in the correct bucket. In particular, the repair phase starts from the tails of each bucket, and finds other elements to swap into place such that both elements are now placed correctly.

The pseudocode is presented below. Note that the heads or tails are now processor specific, as each processor will be assigned several contiguous stripes of data.

Algorithm 4 PARADIS algorithm

```
function PARADIS(data, l, k, p)
    if number of elements in data = 1 then
        return data
    end if
    histogram = BUILDHISTOGRAM(data, l, k, p)
    heads = [0]
    tails = [histogram[0]]
    for i from 1 to 9 do
        heads[i] = heads[i − 1] + histogram[i − 1]
        tails[i] = tails[i − 1] + histogram[i]
    end for
    PARADIS_PERMUTE(d, l, k, p, heads, tails)
    for bucket in buckets do
        PARADIS_REPAIR(d, l, k, p, bucket_start, bucket_end)
    end for
    if l < k − 1 then
        for i from 1 to 9 do
            d = data from bucket[i]
            PARADIS(d, l + 1, k, p)
        end for
    end if
    return data
end function
```

Algorithm 5 PARADIS-permute

```
function PARADIS_PERMUTE(data, l, k, p, heads, tails)
    for each processor do
        for i from 0 to 9 do
            head = heads[i]
            while  dohead < tails[i]
                v = data[head]
                b = correct bucket of v
                while b! = i and heads[b] < tails[b] do
                    Swap values of v and d[heads[b]]
                    heads[b] + +
                    b = correct bucket of v
                end while
                if k == i then
                    data[head] = data[heads[i]]
                    data[heads[i]] = v
                    heads[i]+ = 1
                else
                    data[head] = v
                end if
                head+ = 1
            end while
        end for
    end for
end function
```

Algorithm 6 PARADIS-repair

```
function PARADIS_REPAIR(data, l, k, p, heads, tails, i)
    tail = tails[i]
    for e doach processor
        head = heads[i]
        while head < tails[i] and head < tail do
            v = data[head]
            head + +
            bv = correct bucket of v
            if b! = i then
                while head < tail do
                    tail − −
                    w = data[tail]
                    bw = correct bucket of w
                    if bw == i then
                        data[head − 1] = w
                        data[tail] = v
                        break
                    end if
                end while
            end if
        end while
    end for
end function
```

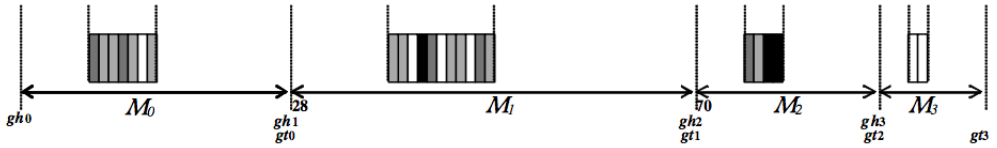
3.2. **Distribution-Adaptive Load Balancing.** Distribution-adaptive load balancing is an approach to spread the work of recursive calls to PARADIS evenly. As the number of elements in each bucket can vary greatly depending on the data, we can modify the assignment of buckets to different processors in an intelligent way so as to distribute work evenly among the processors.

While it doesn't improve the average-case runtime of the algorithm, it does provide marginal improvement on the worst case runtime. For this paper, I have not implemented or tested the effects of load balancing, though more information about the load balancing method can be found in [1].

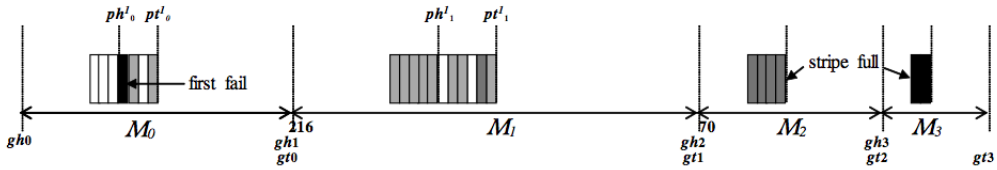
4. Preliminary Results and Analysis

All the following experiments were performed on a MacBook Pro with a 2.7 GHz i5 dual-core processor and 8 GB ram, and the implementation is in Python.

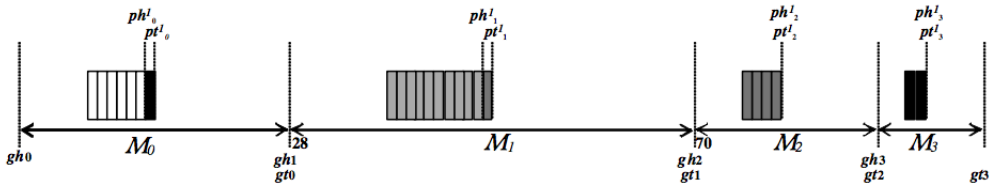
4.1. **Experiment Results.** First, we can see from Figure 3 that MSD radix sort in general does better than naive radix sort, which is expected due to the divide and conquer nature of MSD radix sort. Next, we can see that in-place radix sort does worse than both the out-of-place algorithms; this can perhaps be attributed to the permutations and constant swaps, resulting in a longer runtime, albeit with lower memory usage. Finally, PARADIS performs a bit better than the standard in-place radix sort implementation, though still worse than the sequential out-of-place algorithms.



(a) Before: original data in stripes for each bucket for processor 1: $\{d[\mathcal{M}_i^1]|0 \leq i < 4\}$



(b) First failure due to insufficient capacity for the black key: $C_3^1 = 2 < \sum_j C_j^1(3) = 3$



(c) After: almost permuted by PARADIS_Permute: $ph_i^1, \forall i$ marking the first wrong elements

FIGURE 2. Before and after PARADIS-permute. We can see that after permutation, there are still some elements not in the correct buckets, which is why the repair phase is necessary.

There could be many reasons why the experiments show PARADIS as only marginally better. First, I used Python, which unfortunately does not lend itself very well to parallel computations. Next, running the algorithms on a single laptop machine with small amounts of data often lead to relatively large overhead costs, which dampens the benefits from parallelization. Finally, I haven't yet implemented load balancing, which could further optimize the performance.

More complete results of PARADIS can be found in [1]; indeed, the speedups in the experiments there are quite drastic for PARADIS, often reaching 2 to 3 times faster than competing radix sort algorithms; our experiments unfortunately have not been able to replicate the results to the same extent.

4.2. **Analysis.** The runtime complexity is shown in [1] to have an upper bound of $O(n(\frac{1}{p} + w))$, where w is the largest fraction of elements repaired by a single processor. It is also shown that w is bounded above by $\frac{1}{4}$. Given random data, w is generally very small, and therefore, the runtime complexity of the PARADIS algorithm achieves close to the optimal $O(\frac{n}{p})$.

Moreover, the only additional work done by PARADIS as compared to the regular in-place radix sort algorithm is from paradis-repair, which requires $O(wn) = O(n)$ work. Therefore, the total amount of work done at each step of the recursion is $O(n)$, so total work is still $O(n \log n)$. Similarly, depth is also $O(\log n)$. Though the theoretical work and depth of PARADIS remain the same as compared to the

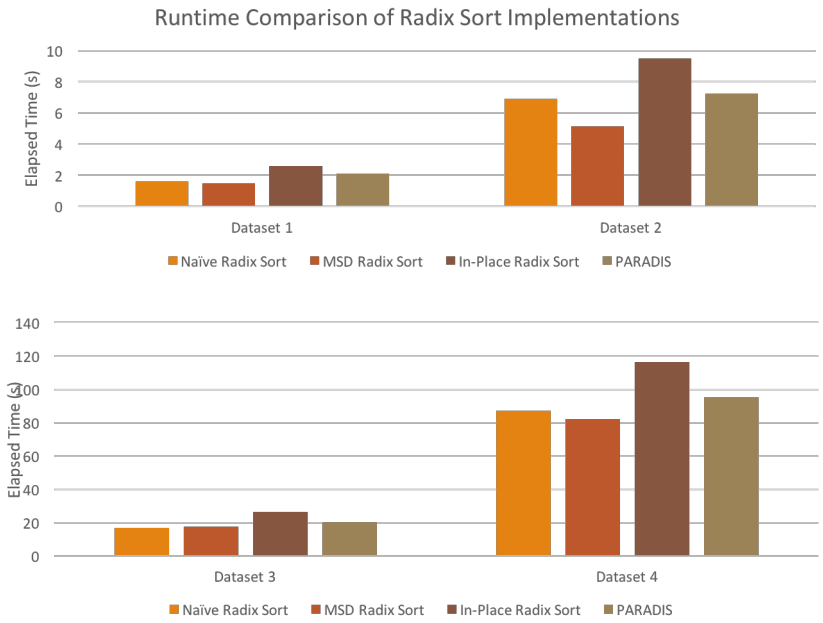


FIGURE 3. Comparison of runtimes between the different radix sort methods. Here, datasets 1 and 2 consist of 1 million randomly generated numbers up to 4 digits and 16 digits, respectively, selected uniformly. Datasets 3 and 4 consist of 10 million randomly generated numbers up to 4 digits and 16 digits, respectively, selected uniformly.

regular in-place algorithm, parallelizing the permutation process is helpful in lowering the constant factor, and provides many practical benefits. Favorable results are shown in [1], even though the preliminary results achieved here did not reveal the same trends.

A final important aspect to consider is that no communication between processors is required for PARADIS, which is greatly helpful in terms of runtime efficiency.

5. Conclusion

In this paper, we have explored radix sort along with its in-place variant, as well as many parallelization ideas that can be applied to radix sort to increase its runtime efficiency. Moreover, the PARADIS algorithm, which introduces the novel concept of speculative permutation, was explored.

Though the preliminary results were not able to match the trends shown in the original paper, the limited technological capabilities of a single laptop computer sorting on small amounts of data in Python likely limits any potential gains from parallelizing radix sort. Nonetheless, the results found in [1] are certainly promising, and demonstrate the efficiency of PARADIS. Furthermore, there is no doubt that parallelization can indeed improve performance of radix sort, allowing it to remain competitive with other sorting methods.

References

[1] Minsik Cho, Daniel Brand, Rajesh Bordawekar, Ulrich Finkler, Vincent Kulandaisamy, Ruchir Puri PARADIS: An Efficient Parallel Algorithm for In-Place Radix Sort http://www.vldb.org/pvldb/vol8/p1518-cho.pdf.

[2] J. Harkins, T. El-Ghazawi, E. El-Araby, and M. Huang. A Novel Parallel Approach of Radix Sort with Bucket Partition Preprocess. In Proc. IEEE Conf. on Embedded Software and Systems, pages 989994, 2012.

[3] S.-J. Lee, M. Jeon, D. Kim, and A. Sohn. Partitioned parallel radix sort. J. Parallel Distrib. Comput., 62(4):656668, Apr. 2002.

[4] D. Jimenez-Gonzalez, J. J. Navarro, and J.-L. Larrba-Pey. Fast parallel in-memory 64-bit sorting. In Proc. Int. Conf. on Supercomputing, pages 114122, 2001.

[5] Radix Sort
`https://en.wikipedia.org/wiki/Radix_sort`

[6] Comparison sort
`https://en.wikipedia.org/wiki/Comparison_sort`