# PARADIS: An Efficient Parallel Algorithm for In-place Radix Sort

Minsik Cho* Daniel Brand* Rajesh Bordawekar*
Ulrich Finkler* Vincent Kulandaisamy+ Ruchir Puri*
*IBM T. J. Watson Research Center, Yorktown Heights, NY, USA
+IBM Software Group, Hillsboro, OR, USA
{minsikcho, danbrand, bordaw, ufinkler, vinci, ruchir}@us.ibm.com

## ABSTRACT

In-place radix sort is a popular distribution-based sorting algorithm for short numeric or string keys due to its linear run-time and constant memory complexity. However, efficient parallelization of in-place radix sort is very challenging for two reasons. First, the initial phase of permuting elements into buckets suffers read-write dependency inherent in its *in-place* nature. Secondly, load balancing of the recursive application of the algorithm to the resulting buckets is difficult when the buckets are of very different sizes, which happens for skewed distributions of the input data. In this paper, we present a novel parallel in-place radix sort algorithm, PARADIS, which addresses both problems: **a)** "speculative permutation" solves the first problem by assigning multiple non-continuous array stripes to each processor. The resulting shared-nothing scheme achieves full parallelization. Since our speculative permutation is not complete, it is followed by a "repair" phase, which can again be done in parallel without any data sharing among the processors. **b)** "distribution-adaptive load balancing" solves the second problem. We dynamically allocate processors in the context of radix sort, so as to minimize the overall completion time. Our experimental results show that PARADIS offers excellent performance/scalability on a wide range of input data sets.

## 1. INTRODUCTION

Due to aggressive CMOS technology scaling, computing platforms have been evolving towards multi/many-core architectures, where a number of cores are connected to increasingly larger and faster hierarchical memory systems [34]. On the other hand, due to large amounts of information generated by mobile devices, wireless sensors, and others, the world's per-capita demand for information storage has been doubling nearly every 40 months since the 1980s [12].

Such trends in data volume and computing systems motivate large body of research on sorting, one of the most fundamental algorithmic kernels in data management. Various methods and approaches to speeding up sorting have been proposed including external/internal sorting, data-specific, or hardware-specific sorting [6,7,19,22,24,28]. Among them, in-memory sorting, where performance-critical workloads reside in DRAM rather than disk, has been of great interest. This is due to the poor latency and bandwidth of disk and the emergence of low-cost and high-density memory devices [9, 19, 23, 26].

Radix sort can be one of the best suited sorting kernels for many in-memory data analytics due to its simplicity and efficiency [1,3,5,16,25,29]. Especially *in-place* radix sort, which performs sorting without extra memory overhead, is highly desirable for in-memory operation [21] for two reasons: **a)** the large memory footprint of in-memory databases calls for memory-efficient sorting, **b)** in-place radix sort offers higher performance with significantly fewer cache misses and page faults than approaches requiring extra memory. Details on conventional radix sort are further discussed in Section 4.

Parallelizing in-place radix sort, however, is particularly challenging due to read-write dependency inherent in the *in-place* nature [25]. While many studies have proposed solutions, they either parallelize the non-critical preparation step only (histogram and partitioning) like Fig. 1 (a), or require an additional temporary/auxiliary array thus increasing the memory footprint like Fig. 1 (b). We are not aware of any prior research on fully parallel in-place radix sort.

In this work, we present PARADIS, a fully parallelized in-place radix sort engine with two novel ideas: speculative permutation and distribution-adaptive load balancing. Our theoretical analysis and experiment results show that PARADIS is highly scalable and efficient in comparison with several other parallel sorting libraries on realistic benchmarks, as well as on synthetic benchmarks with different sizes, data types, alignment and skewness [17, 28, 33]. The major contributions of this paper are:

- A speculative permutation followed by repair which are both efficiently parallelized. By iterating these two steps, PARADIS permutes all array elements into their buckets, fully in parallel and in-place.
- A distribution-adaptive load balancing technique for recursive invocations of the algorithm on the resulting buckets. For a skewed distribution, PARADIS minimizes the elapsed run-time by adaptively allocating more processors to larger buckets.

The rest of the paper is organized as follows. We review related works in Section 2 and present preliminaries


*Proceedings of the VLDB Endowment,* Vol. 8, No. 12

in Section 3. Section 4 presents our proposed PARADIS algorithm, with complexity analysis in the Appendix. Experimental results are in Section 5. Section 6 concludes this paper.

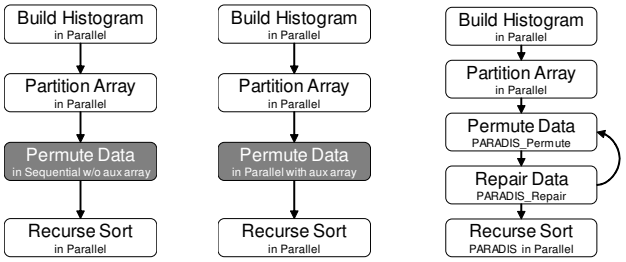
## 2. RELATED WORKS

Sorting algorithms have been a popular research area over the past few decades. Recent advancements in parallel computing platforms (e.g., multi-core CPU with SIMD, GPUs, IBM's Cell, etc) have drawn significant attention to parallel sorting techniques. Two strategies for parallel sort have been proposed for multi-core CPU: top-down and bottom-up. In top-down techniques [18, 20, 35], the input is first partitioned based on the key (e.g., radix-partition), and then each partition is independently sorted. In bottom-up techniques [3,17,31], the input is partitioned for load balancing, and all individually sorted partitions are merged into the final array. Further parallelization on CPU has been achieved with SIMD for comb sort [17] and bitonic sort [3].

Parallelization has been also researched for different computing platforms. Sorting algorithms based on bitonic sort [7], radix sort [22, 27, 28], or merge sort [27] have been proposed to utilize massive parallelism in GPUs. SIMD-based bitonic sort has been proposed in [6] to utilize co-processors.

Unlike comparison-based sorting (e.g., quicksort, mergesort), radix sort is a distribution-based algorithm which relies on a positional representation of each key (e.g., keys can be digits or characters). By reading a key as a sequence of numerical symbols from the most significant to the least significant (MSD) or in the other way (LSD), radix sort groups keys into buckets by the individual symbol sharing the same significant position, e.g., postman sort [16].

Many optimizations including parallelization have been done to speed up radix sort. Platform-based optimization for radix sort is discussed in [33], which takes advantage of virtual memory and makes use of write-combining in order to reduce the system's peak memory traffic. The early work on parallel radix sort is presented in [35], which shows how to build the histogram and perform data permutation in parallel. It uses an auxiliary array, making memory complexity $O(\mathcal{N})$, which is not desirable for in-memory data analytics. More advanced techniques for parallel radix sorting have been proposed in [18, 20, 25], but they are relatively inefficient due to their additional memory overhead




(a) radix-se (b) radix-ax (c) PARADIS

**Figure 1: Various parallel radix sort algorithms where parallel and in-place steps are in white. (a) Sequential in-place permutation, (b) Parallel permutation with auxiliary array (2x memory footprint) [10, 28, 35], (c) Parallel and in-place permutation in PARADIS**

**Table 1: Notations in this paper**

| | |
|---|---|
| $\mathcal{N}$ | set of array indices $\{0, 1, ..., \lvert\mathcal{N}\rvert - 1\}$ |
| $d[\mathcal{N}]$ | the array of size $\lvert\mathcal{N}\rvert$ to be sorted |
| $n, h, t$ | array index $\in \mathcal{N}$ |
| $\mathcal{P}$ | set of processor indices $\{0, 1, ..., \lvert\mathcal{P}\rvert - 1\}$ |
| $p, q$ | processor index $\in \mathcal{P}$ |
| $p_0, p_1, ...$ | shorthand for "processor 0", "processor 1", ... |
| $\mathcal{B}$ | set of bucket indices $\{0, 1, ..., \lvert\mathcal{B}\rvert - 1\}$ |
| $i, j, k$ | bucket index $\in \mathcal{B}$ |
| $\mathcal{L}$ | set of recursion levels $\{0, 1, ..., \lvert\mathcal{L}\rvert - 1\}$ |
| $l$ | recursion level $\in \mathcal{L}$ |
| $b(v)$ | index of the bucket where element $v$ should belong |
| $gh_i$ | head pointer of bucket $i$ |
| $gt_i$ | tail pointer of bucket $i$ |
| $ph_i^p$ | head pointer of the stripe for processor $p$ in bucket $i$ |
| $pt_i^p$ | tail pointer of the stripe for processor $p$ in bucket $i$ |
| $\mathcal{M}_i$ | $\{n \mid gh_i \leq n < gt_i\}$, i.e., the indices of bucket $i$ |
| $\mathcal{M}_i^p$ | $\{n \mid ph_i^p \leq n < pt_i^p\}$, i.e., the indices of stripe $p, i$ |
| $C_i$ | $\lvert\mathcal{M}_i\rvert = gt_i - gh_i$, i.e., size of bucket $i$ |
| $C_i^p$ | $\lvert\mathcal{M}_i^p\rvert = pt_i^p - ph_i^p$, i.e, size of stripe $p, i$ |
| $C_i(k)$ | $\lvert\{n \in \mathcal{M}_i \mid b(d[n]) = k\}\rvert$<br>i.e. the number elements in $\mathcal{M}_i$ belonging to $\mathcal{M}_k$ |
| $C_i^p(k)$ | $\lvert\{n \in \mathcal{M}_i^p \mid b(d[n]) = k\}\rvert$<br>i.e. the number elements in $\mathcal{M}_i^p$ belonging to $\mathcal{M}_k$ |

and/or all-to-all communication schemes [4]. Another work on parallel radix sort is in [28], which enhances [35] based on modern CPU architectural features such as TLB and cache configurations using user-level buffering. Parallel radix sort also is discussed in [10] with the overhead of an auxiliary output array. Fig. 1 (b) sketches these algorithms where an auxiliary array is required for parallel data permutation.

The load balancing problem in parallel radix sort is studied in [20,32]. Perfect load balancing idea is described in [32] at the cost of heavy communication between processors. [20] proposes an improved algorithm for load balancing where the radix key length (in bits) is increased in a trial-and-error way until good load balancing is obtained.

## 3. PRELIMINARIES

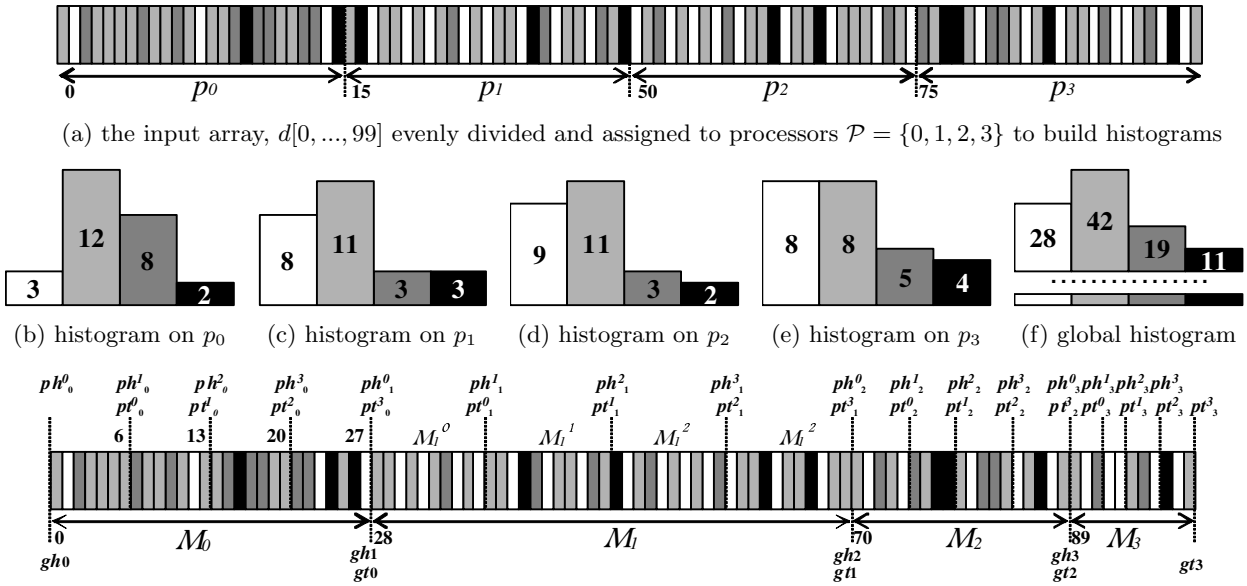
Table 1 lists our notations and concepts, which will be defined/referenced throughout the paper. We assume a given array of $\lvert\mathcal{N}\rvert$ elements to be sorted by the key of each element. An element consists of both key and payload, although PARADIS is also applicable to the case where keys and payloads are stored separately.

One can consider the example of sorting 8-byte integers. Then $\mathcal{L} = \{0, ..., 7\}$, and there are functions $b_0(), .., b_7()$. For an element $v$, $b_0(v)$ = most significant byte of $v$, $b_1(v)$ = second most significant byte of $v$, etc. $\mathcal{B} = \{0, ..., 255\}$ for all recursion levels $l$. At the first recursion level, $\mathcal{P}$ would consist of all available processors. On subsequent recursive calls, $\mathcal{P}$ would be only a subset of all available processors, namely those assigned to sort the sub-array $d[\mathcal{N}]$ (See Section 4.2.3).

In general, all the quantities in Table 1 are local to the invocation of the algorithm on each recursion level. Only the quantities $\mathcal{L}, \{..., b_l(), ...\}$ are global and prepared beforehand based on the type of data.

## 4. PARADIS

In this section, we propose our parallel in-place radix sort algorithm, PARADIS. We first discuss the challenges in parallelizing in-place radix sort, and then provide an overview of PARADIS in Section 4.1, highlighting our novel techniques.




(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 2: Parallel histogram construction and preparation for PARADIS**

Then we detail our parallelization techniques in Section 4.2 and 4.3 with comprehensive examples in Fig. 2 to Fig. 5.

## 4.1 Overview

In this section, we give an overview of PARADIS in Algorithm 2 with Fig. 1 (c). In this paper, we mainly focus on MSD radix sort [16, 21], but our ideas can be generally applied to LSD radix sort as well.

The nature of non-comparativeness enables $O(\mathcal{N})$ computational complexity. Memory complexity, on the other hand, can be $O(1)$ (in-place) or $O(\mathcal{N})$ (with auxiliary array). Sequential in-place MSD radix sort [21] permutes the elements in place, as sketched in Algorithm 1. In general, it consists of four steps:

**Step 1 (lines 4-7)** The unsorted input array is scanned to build a histogram of the radix key distribution.

**Step 2 (lines 8-11)** The input array is partitioned into $|\mathcal{B}|$ buckets by computing $gh_i$ and $gt_i$ (the beginning and end of partition for each radix key $i$).

**Step 3 (lines 12-20)** This is the core of the algorithm. Each element is checked on line 15 and permuted on line 16 if it is not in the right bucket.

**Step 4 (lines 21-25)** Once element permutation is completed, each bucket becomes a sub-problem, which can be solved independently and recursively.

The radix sort in Algorithm 1 depends on the following property (which is ensured by building a histogram)

$$C_i = \sum_j C_j(i) \quad (1)$$

which states that the amount $C_i$ reserved for bucket $i$ (on the left hand) must be exactly equal to the number of all the elements that should belong to bucket $i$, although those elements may be initially scattered through various buckets $j$ (on the right hand). Steps 1 and 2 are preprocessing phases whose purpose is to guarantee Eq. (1) during step 3.

**Algorithm 1** Radix Sort

```
1: procedure RadixSort(d[N],l)
2:     b = b_l                          ▷ Function giving bucket at level l
3:     B = the range of b()
4:     cnt[B] = 0                       ▷ Histogram of bucket sizes
5:     for n ∈ N do
6:         cnt[b(d[n])]++
7:     end for
8:     for i ∈ B do
9:         gh_i = Σ_{j<i} cnt[j]
10:        gt_i = Σ_{j≤i} cnt[j]
11:    end for
12:    for i ∈ B do
13:        while gh_i < gt_i do         ▷ Till bucket i is empty
14:            v = d[gh_i]
15:            while b(v)! = i do
16:                swap(v, d[gh_{b(v)}++])
17:            end while
18:            d[gh_i++] = v
19:        end while
20:    end for
21:    if l < L − 1 then               ▷ Recurse on each bucket
22:        for i ∈ B do
23:            RadixSort(d[M_i],l+1)
24:        end for
25:    end if
26: end procedure
```

PARADIS in Algorithm 2 is our parallelization of Algorithm 1. Steps 1 and 2 of Algorithm 1 are easy to parallelize based on the following partitioning as in [35]:

$\{..., \mathcal{A}_p, ...\} = \textbf{\textit{PartitionForHistogram}}$: *Partition $\mathcal{N}$ into disjoint subsets $\mathcal{A}_p \subset \mathcal{N}$, one for each processor $p \in \mathcal{P}$. The partitions should be as equal as possible, so each processor has $\frac{|\mathcal{N}|}{|\mathcal{P}|}$ elements.*

**Algorithm 2** PARADIS

```
1: procedure PARADIS(d[N],l,P)
2:     b = b_l                          ▷ Function giving bucket at level l
3:     B = the range of b()
4:     {..., A_p, ...} = PartitionForHistogram
5:     for p ∈ P in parallel do
6:         Build local histogram for d[A_p]
7:     end for
8:     Synchronization
9:     Build global histogram from the |P| local histograms
10:    Compute gh_i and gt_i, ∀i          ▷ As in Algorithm 1
11:    Synchronization
12:    {..., B_p, ...} = PartitionForRepair
13:    while Σ_i C_i > 0 do               ▷ Till all buckets are empty
14:        {..., M_i^p, ...} = PartitionForPermutation
15:        for p ∈ P in parallel do
16:            PARADIS_Permute(p)
17:        end for
18:        Synchronization
19:        for p ∈ P in parallel do
20:            for each i ∈ B_p do
21:                PARADIS_Repair(i)
22:            end for
23:        end for
24:        Synchronization
25:    end while
26:    if l < L − 1 then                  ▷ Recurse on each bucket
27:        {..., P_i, ...} = PartitionForRecursion
28:        for i ∈ B in parallel do       ▷ Sort each bucket
29:            PARADIS(d[M_i], l+1, P_i)
30:        end for
31:    end if
32: end procedure
```

Specifically, for step 1, each processor $p$ takes over a section $\mathcal{A}_p$ of the input array and builds its local histogram, see lines 4-8 of Algorithm 2. All the local histograms are then merged into a global histogram (line 9). Step 2 can be parallelized by using a parallel prefix sum technique. Step 4 can be naturally parallelized as each bucket can be sorted independently.

As an example, Fig. 2 shows how to begin sorting 100 elements with 4 processors, where there are four kinds of radix keys: white, gray, dark gray, and black (i.e., 2-bit radix sort). The entire input array is evenly partitioned and assigned to $p_{\{0,1,2,3\}}$ as in Fig. 2 (a). And then, each processor in parallel builds a histogram for its own partition. As a result, the processors generate the histograms in (b), (c), (d), and (e) which are merged into the global histogram in (f). Based on the global histogram, we can compute $gh_i$ and $gt_i, (0 \leq i < 4)$ as shown in (g). Such preparation steps for PARADIS are in lines 1-10 in Algorithm 2. Then the challenges in parallelization of Algorithm 1 come in two forms.

- Parallelizing step 3 is very challenging due to the read-after-write dependency on the $gh_i$.
- Unbalanced sub-problem sizes in step 4 can degrade the end-to-end performance.

Our proposed PARADIS algorithm addresses the above challenges with two novel techniques in Sections 4.2 and 4.3. To avoid the read-after-write dependencies we partition the given array among given processors in a share-nothing fashion. However, an arbitrary partitioning is unable to give each processor data that satisfies Eq. (1), which makes Algorithm 1 inapplicable. While partitionings satisfying Eq. (1) do exist, they are expensive to compute and do not guarantee balanced load for the processors. To address this, PARADIS *speculates* on a *good* partitioning (details in Section 4.2.1). Since this partitioning, in general, will not satisfy Eq. (1) (i.e., some buckets may be over-sized or under-sized), the output may not be completely permuted, which will be addressed by an additional repair stage. The two stages, permutation and repair, are iterated until a complete redistribution of all the array elements into their buckets is achieved. The speculative permutation is such that both stages can be executed in parallel, where all processors have an approximately equal load, achieving good scalability. In short, we have the extra cost of repairing elements due to speculative permutation, but the gain in scalability from two fully parallelized steps far outweighs such costs.

Once all elements are placed in their buckets, we have $|\mathcal{B}|$ independent sorting sub-problems. They can be highly different in size, causing poor load balancing. Thus, PARADIS performs load balancing through adaptive processor reallocation. Further details on speculative permutation and adaptive load balancing in PARADIS are discussed in Sections 4.2 and 4.3.

## 4.2 Speculative Permutation

In this section, we cover speculative permutation, a key technique to maximizing parallelism in element permutation. Essentially, it is an iterative algorithm which reduces the problem size significantly at each iteration. In detail, we have four steps in speculative permutation which will be explained in the following sub-sections.

### 4.2.1 Partitioning for Permutation

The first step is partitioning each bucket into stripes based on the following partitioning as in line 12 of Algorithm 2.

$\boldsymbol{\{..., \mathcal{M}_i^p, ...\} = PartitionForPermutation}$: *Partition each bucket $\mathcal{M}_i$ (of size $C_i$) into $|\mathcal{P}|$ disjoint* stripes *$\mathcal{M}_i^p$ (of size $C_i^p$) such that the stripes satisfy the following:*

$$C_i = \sum_p C_i^p \qquad \forall i \tag{2}$$

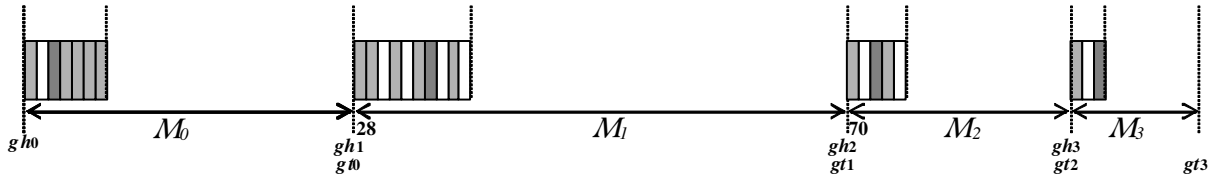
The goal of the procedure *PartitionForPermutation* is to let each processor own one stripe from each bucket. This allows each processor to permute elements among its stripes in parallel with other processors, but without any communication, see Section 4.2.2. As a heuristic to optimizing load balancing, *PartitionForPermutation* tries to solve

$$\text{min: } max\{\sum_i C_i^p \mid \forall p\} \tag{3}$$

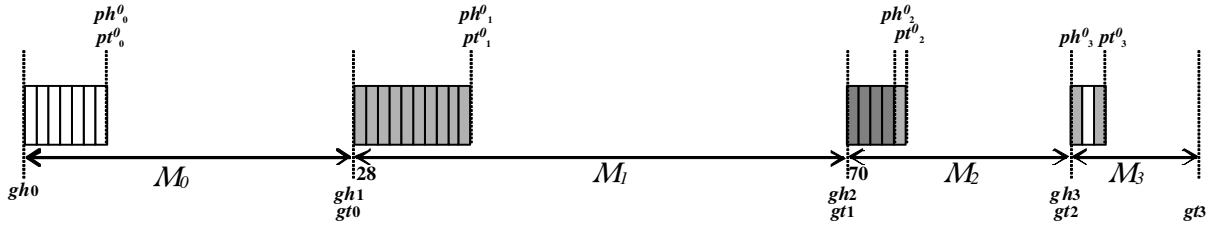
As a solution, PARADIS uses (see Table 1 for notations)

$$C_i^p = \frac{C_i}{|\mathcal{P}|} \quad \forall i, \forall p \tag{4}$$

to *speculatively* partition each bucket into equally-sized stripes. For simplicity we do not allow stripes to be arbitrary subsets of a bucket, but each stripe must be a single interval. As such, the stripes are delineated by indices $ph_i^p$ and $pt_i^p$,

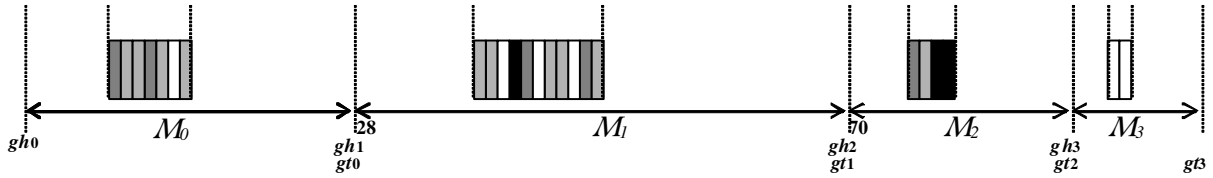



(a) Before: original data in stripes for each bucket for processor 0: $\{d[\mathcal{M}_i^0]|0 \leq i < 4\}$

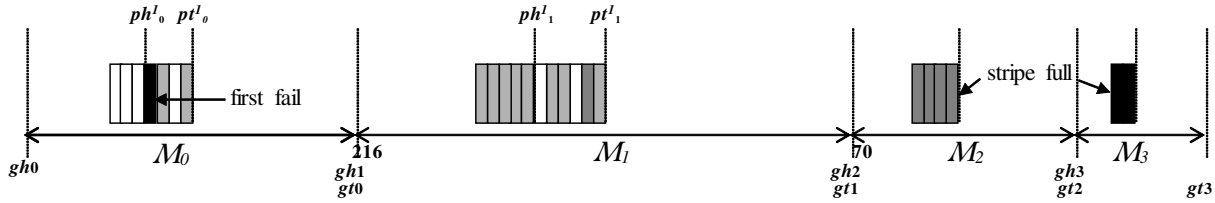



(b) After: almost permuted by PARADIS_Permute: $ph_i^0, \forall i$ marking the first wrong elements

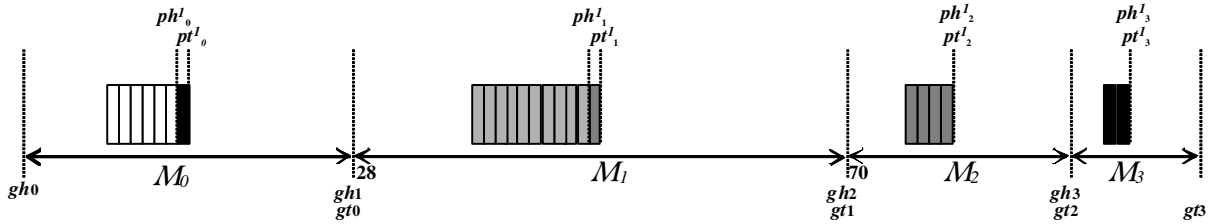
**Figure 3: Before-and-after by PARADIS_Permute by $p_0$**




(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 4: Before-and-after by PARADIS_Permute by $p_1$**

where $pt_i^p - ph_i^p = C_i^p$. For example, Fig. 2 (g) illustrates that $\mathcal{M}_1$ is evenly partitioned into stripes $\mathcal{M}_1^{\{0,1,2,3\}}$.

Eq. (3) expresses that the assignment of array elements to the processors should be balanced. There is another desirable optimization criterion, namely

$$\text{min:} \quad max\{C_i^p - \sum_j C_j^p(i) \mid \forall i, \forall p\} \tag{5}$$

which is a version of Eq. (1) restricted to those array elements assigned to processor $p$. Instead of requiring equality, it merely tries to minimize the difference between the left and right hand side of Eq. (1). Minimizing Eq. (5) will minimize the number of iterations of the loop on line 13 in Algorithm 2. However, that may conflict with balancing the workload in Eq. (3). We prefer the load balancing objective, as it directly impacts scalability. Therefore, PARADIS adopts Eq. (4), which is optimal for Eq. (3) (perfect work-

**Algorithm 3** PARADIS_Permute

```
1: procedure PARADIS_Permute(p)
2:     for i ∈ B do
3:         head = ph_i^p
4:         while head < pt_i^p do
5:             v = d[head]                          ▷ Keep moving v
6:             k = b(v)                             ▷ to its bucket k
7:             while k! = i and ph_k^p < pt_k^p do
8:                 swap(v, d[ph_k^p++])             ▷ v into its bucket k
9:                 k = b(v)                         ▷ New v and k
10:            end while
11:            if k == i then                       ▷ Found a correct element
12:                d[head++] = d[ph_i^p]
13:                d[ph_i^p++] = v
14:            else
15:                d[head++] = v
16:            end if
17:        end while
18:    end for
19: end procedure
```

load balancing). In addition, Eq. (4) also minimizes Eq. (5) in case of uniformly distributed radix keys (not to mention that Eq. (4) is easier to compute).

### 4.2.2 Parallel Data Permutation

Once all the buckets are partitioned into stripes based on Section 4.2.1, we can use Algorithm 3 in each processor $p$ to perform in-place permutation (invoked on lines 15-17 of Algorithm 2). Compared with step 3 of Algorithm 1, there are three fundamental modifications in Algorithm 3.

- Since the partitioning of each bucket is merely speculative, we check if the target stripe is full (line 7), in order not to overwrite existing elements.
- $ph_i^p$ increases (line 13) only if a correct element is found (line 11), which keeps all correctly placed elements before $ph_i^p$.
- At the end of Algorithm 3 all wrong elements in the bucket $i$ are kept between $ph_i^p$ and $pt_i^p$, which will be further repaired in Algorithm 4.

Fig. 3 and 4 show the before-and-after comparison of the stripes assigned to $p_0$ and $p_1$, respectively. In detail, let us focus on Fig. 4. At the beginning of processing its stripe in $\mathcal{M}_0$, we replace the first three elements with white ones. However, when processing the 4-th element (marked with **first fail**), we find out that the stripe for the black element is already full ($ph_3^1 = pt_3^1$) and does not have enough capacity to accept another one (marked with **stripe full**). Therefore lines 7 and 15 of Algorithm 3 will put the black element back to $\mathcal{M}_0$, which leads to the configuration in Fig.4 (b). Even with this failure, $p_1$ continues to process, and if a white element is found later, we simply move the black element at $ph_0^1$ to *head* location, and put the white element to $ph_0^1$ (lines 11-14). As a result, the black wrong element will continue moving toward $pt_0^1$ and will end up between $ph_0^1$ and $pt_0^1$ as in Fig.4 (c). In contrast, if processing a bucket encounters no failure, as in $\mathcal{M}_2$ and $\mathcal{M}_3$, then $ph_i^p = pt_i^p$ eventually.

Regarding the black elements in Fig. 3 and 4, we can observe that $p_0$ (Fig. 3) has allocated capacity for 3 black elements in $\mathcal{M}_3$, although $\mathcal{M}^0_{\{0,1,2,3\}}$ have no black elements. On the other hand, $p_1$ (Fig. 4) has allocated capacity for only 2 black elements, while there are 3 black elements in $\mathcal{M}^1_{\{0,1,2,3\}}$. Such over/under-allocation is owing to our speculative partitioning in Section 4.2.1, which needs to be repaired in Section 4.2.3. Note that it may be possible to improve speculation, if the information on the input array is known in advance.

### 4.2.3 Repairing Permutation

The output from Algorithm 3 may not be perfect; there are some elements left in the wrong buckets as in Fig. 3 and 4. Further, having such elements scattered in the bucket makes it hard to move $gh_i$ to the best location, as all elements in the right bucket must be before $gh_i$. Therefore, we follow with Algorithm 4, which will have the following outcome: **a)** in each bucket, all elements that belong there will be placed to the left, and all elements that do not belong there will be placed to the right. **b)** $gh_i$ will point at the first wrong element in bucket $i$, if there is any. In case Algorithm 3 succeeds in filling bucket $i$ with correct elements only, then $gh_i = gt_i$. Note that repairing a bucket does not involve visiting each element in order to identify a wrong element, as we scan only the remaining stripes $d[\mathcal{M}_i^p]$. (See Section 4.2.2). That arrangement will reduce the problem size for Algorithm 3 during subsequent iteration.

Algorithm 4 is parallelized by processing each bucket separately in a single processor based on the following:

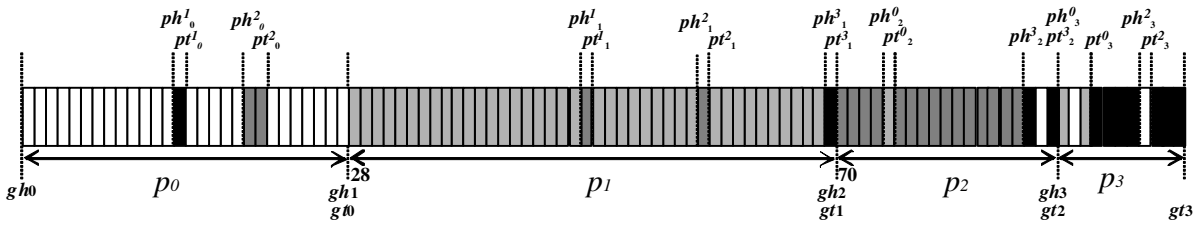
$\boldsymbol{\{..., \mathcal{B}_p, ...\} = PartitionForRepair}$: *Partition the existing set of buckets $\mathcal{B}$ into disjoint subsets $\mathcal{B}_p \subset \mathcal{B}$, one for each processor $p \in \mathcal{P}$.*

The objective is to balance the number of array elements contained in $\mathcal{B}_p$ by minimizing $max\{\sum_{i \in \mathcal{B}_p} C_i \mid \forall p\}$. Therefore, *PartitionForRepair* assigns buckets to processors, so that Algorithm 4 can be performed in parallel as in lines 19-23 of Algorithm 2 in a share-nothing fashion. We use a greedy linear algorithm to solve *PartitionForRepair*; simply by computing the average number of elements per processor in advance, we can keep assigning buckets to processors, until the number of elements for each processor is close to the average.

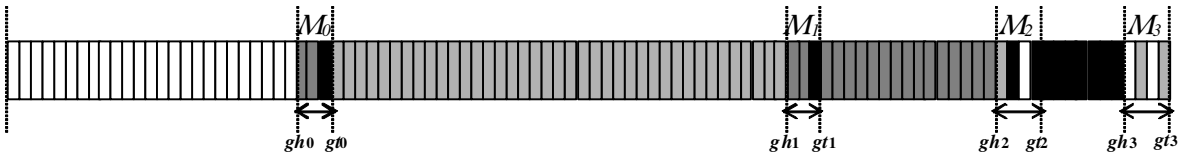
Fig. 5 (a) shows the state of the input array after Algorithm 3, where each $ph_i^p$ points at the first wrong element. Also, *PartitionForRepair* partitions $\mathcal{B}$ into $\{\mathcal{B}_0, \mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3\}$. After Algorithm 4 we will have a repaired input array as in Fig. 5 (b), where all wrong elements are moved to the end of each bucket and $gh_{\{0,1,2,3\}}$ are adjusted. The efficiency of this repairing step depends on finding the wrong elements quickly, and the arrangement of $ph_i^p$ in Algorithm 3 is designed for this purpose.

### 4.2.4 Iterative Permutation

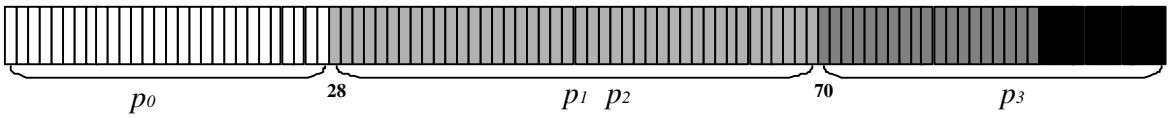
Once we complete an iteration (of the loop started on line 13 of Algorithm 2), we have a new permutation problem like Fig. 5 (b). This problem is usually an order-of-magnitude smaller than the initial problem, because we only need to permute elements in the reduced $\mathcal{M}_i$. In our example, the new problem size shrinks from 100 to 14. We then repeat Algorithms 3 and 4 with the *updated* $gh_i$, see line 13 of Algorithm 2, until all elements are placed in their correct bucket. Then we recurse the in-place radix sort on each bucket independently as in lines 27-34 of Algorithm 2, which will be discussed in Section 4.3.




(a) Almost permuted by PARADIS_Permute : $\mathcal{B}$ is partitioned $\mathcal{B}_0 = \{0\}, \mathcal{B}_1 = \{1\}, \mathcal{B}_2 = \{2\}, \mathcal{B}_3 = \{3\}$



(b) Wrong elements moved to the end of buckets by PARADIS_Repair : $gh_{\{0,1,2,3\}}$ adjusted to the first wrong elements




(c) Final permutation then load balancing by $PartitionForRecursion$ : $\mathcal{P}_0 = \{0\}, \mathcal{P}_1 = \{1, 2\}, \mathcal{P}_2 = \{3\}, \mathcal{P}_3 = \{3\}$

**Figure 5: Repairing and load balancing in PARADIS**

**Algorithm 4** PARADIS_Repair

```
1: procedure PARADIS_Repair(i)
2:     tail = gt_i                          ▷ Searches for w where b(w) = i
3:     for p ∈ P do
4:         head = ph_i^p                    ▷ Searches for v where b(v) ≠ i
5:         while head < pt_i^p and head < tail do
6:             v = d[head++]
7:             if b(v)! = i then            ▷ Element to fix
8:                 while head < tail do     ▷ Search from tail
9:                     w = d[--tail]
10:                    if b(w) == i then
11:                        d[head-1] = w
12:                        d[tail] = v      ▷ Swap v and w
13:                        break
14:                end if
15:            end while
16:        end if
17:    end while
18:    end for
19:    gh_i = tail                          ▷ gh_i to the first wrong element in i
20: end procedure
```

## 4.3 Distribution-adaptive Load Balancing

Parallelizing the above permutation step is one challenge, but achieving load balancing for recursion is the other challenge in parallel radix sort. If there is a bucket that has way more elements than other buckets, it is highly possible that sorting this large bucket will become the performance bottleneck. In PARADIS, we propose distribution-adaptive load balancing. Unlike existing approaches, where load balancing is achieved upfront at the cost of repeated counting and more radix bits [20], PARADIS dynamically reallocates processor resources only after it finds imbalance.

In generic parallelization, dynamic resource allocation is non-trivial, as the nature of workload may not be known and cannot be characterized effectively. However, since we are in a specific context of in-place MSD radix sort, we can efficiently perform resource allocation. The key observation is that the run-time complexity of radix sort is $O(\mathcal{N})$. Therefore, the resource allocation can be cast as a partitioning problem defined as follows:

$\{..., \mathcal{P}_i, ...\} = PartitionForRecursion$: *Assign each bucket $i$ to a non-empty subset $\mathcal{P}_i \subset \mathcal{P}$. To achieve a partitioning of the processors, for any two buckets $i$ and $j$, either $\mathcal{P}_i = \mathcal{P}_j$, or $\mathcal{P}_i \bigcap \mathcal{P}_j = \emptyset$.*

As a result, each partition of processors is assigned a separate subset of all the buckets. The key difference between $PartitionForRecursion$ and $PartitionForRepair$ is that $PartitionForRepair$ does not allow multiple processors to work on the same bucket, while $PartitionForRecursion$ does by recursively calling PARADIS. Fig. 5 (c) shows how $p_{\{0,1,2,3\}}$ are assigned to buckets for load balancing.

The objective of $PartitionForRecursion$ is to balance the workload assigned to the processors. We formulate the problem as follows:

$$\text{min: } max\{W(p) \mid \forall p\} \tag{6}$$

$$\text{where: } W(p) = \sum_{i \in \mathcal{B}_p} \frac{C_i \cdot log_{|\mathcal{B}|} C_i}{|\mathcal{P}_i|} \tag{7}$$

where $W(p)$ is an estimate of the workload assigned to processor $p$. Note that we denote by $\mathcal{B}_p$ the set of buckets assigned to processor $p$ as we did in $PartitionForRepair$.

The estimate in Eq. (7) is obtained as follows. The run-time complexity of an in-place MSD radix sort is known as $O(L|\mathcal{N}|)$, where $L$ is the number of recursion levels. In the worst case $L = |\mathcal{L}|$. While $\mathcal{L}$ is known statically from the size of the key, $L$ is a dynamic quantity – the recursion will stop as soon as we need to sort a sub-array of size 1. We estimate $L = log_{|\mathcal{B}|}|\mathcal{N}|$. Regarding the denominator $|\mathcal{P}_i|$,

since our parallelization of radix sort yields linear speedup, Eq. (7) can simply divide the complexity of sorting bucket $i$ by the number of processors assigned to bucket $i$.

In order to solve *PartitionForRecursion* in linear time, we first estimate its size $|\mathcal{P}_i|$ as follows, instead of finding each $\mathcal{P}_i$ directly:

$$|\mathcal{P}_i| = |\mathcal{P}| \frac{C_i \cdot log_{|\mathcal{B}|} C_i}{\sum_{j \in \mathcal{B}} C_j \cdot log_{|\mathcal{B}|} C_j} \quad (8)$$

where the numerator is the estimated time to sort bucket $i$ and the denominator is the estimated time to sort all the buckets. Then, we can find a solution fast by assigning processors to $\mathcal{P}_i$ based on *rounded* $|\mathcal{P}_i|$. For example, if $|\mathcal{P}_0| = 1.1, |\mathcal{P}_1| = 0.1$, and $|\mathcal{P}_2| = 2.6$, we find the following solution starting from $p_0$: $\mathcal{P}_0 = \{p_0\}, \mathcal{P}_1 = \{p_0\}$, and $\mathcal{P}_2 = \{p_1, p_2, p_3\}$.

Our proposed load balancing begins in line 27 of Algorithm 2 with *PartitionForRecursion*. Once a partitioning for recursion is obtained, we recursively call PARADIS to finish sorting all the buckets in parallel. Note that in case $|\mathcal{P}_i| = 1$ in line 29 of Algorithm 2, PARADIS seamlessly degenerates into the conventional sequential radix sort in Algorithm 1, by making speculation perfect (i.e., nothing to speculate and nothing to repair) and synchronization trivial.

Consider the example in Fig. 5 (c) which shows the array with each element placed in its correct bucket. If the recursive invocation of *PartitionForRecursion* used the same assignment of buckets to processors as in Fig. 5 (a), then sorting gray elements would become the bottleneck. However, *PartitionForRecursion* is allowed to assign multiple processors to a single bucket as shown in Fig. 5 (c); this is in contrast to *PartitionForRepair*, whose result is in Fig. 5 (a). This makes overall workload more balanced and enhances PARADIS performance.

# 5. EXPERIMENTAL RESULTS

We implemented PARADIS as a C++ template sorting function based on the pthread library. For cross-platform portability we avoided any hardware-specific features such as SIMD. We used GCC (ver. 4.4.4) to compile PARADIS. All experiments were performed on a RedHat Linux server (EL5 update 6) with Intel Xeon (E7- 8837) processor running at 2.67GHz (32 cores) and 512GB main memory. For comparison with GPU-based sorting, we used Nvidia K20x. We used radix key length of 1 byte. For arrays smaller than 64 elements, PARADIS calls std::sort. Our sorting experiments were for in-memory sorting (the entire input is assumed to be located in main memory), but PARADIS can be used as a sorting kernel in external sorting as well. We prepared a set of numeric benchmarks (8 byte key and 8 byte payload) with various sizes and skewness (random and zipfian $\theta$=0.25,0.5,0.75) [2,8,19]. We also prepared benchmarks for sorting strings (10 byte key and 90 byte payload) including random and skewed distributions using gensort [13]. Additionally, we extracted sorting benchmarks (ranging from 100 to 300 million records) from queries on large retail sales transactions. All numbers in this section are averages of 10 end-to-end elapsed times. We compared PARADIS with the following parallel sorting implementations.

**mptl (rel. 11-21-2006)** parallel introsort using pthread library from [15] (fails to sort numeric skewed inputs)

**omptl (rel. 04-22-2012)** parallel introsort using OpenMP ver 3.0 library from [15]

**mcstl (gcc ver. 4.4.4)** parallel hybridsort (multi-way mergesort and balanced quicksort) in libstdc++ [11, 30, 31]

**tbb (ver. 4.1 update 3)** parallel quicksort in Intel Thread Building Block Library [14]

**GPUsort** GPU-based radix sort [22]

**SIMDsort** SIMD-based parallel merge sort [3]

**Buffsort** Buffer-based radix sort [28]

**radix-ax** radix sort implementation using an auxiliary array for parallelization [18, 20, 35] as in Fig. 1 (b)

**radix-se** radix sort implementation parallelizing only recursion, which corresponds to Fig. 1 (a)

**radix-ip** PARADIS without our load balancing (hence, performs same as PARADIS on randomly distributed keys)

**mcstl**, **SIMDsort**, and **radix-ax** have $O(\mathcal{N})$ memory complexity (2x larger memory footprint than the others). With 96GB memory limitation, **mcstl** failed to complete sorting 64GB numeric input with 16 threads in an hour while other in-place algorithms completed in 120 seconds, which proves the criticality of in-place sorting for big data. As GPU has limited on-board memory, **GPUsort** first radix-partitioned the problem on CPU, then sorted each partition on GPU [22]. The runtime of **GPUsort** includes all communication overheads to capture the end-to-end performance.

Fig. 6 compares our load balancing with [20] which increases radix size to find a more balanced partitioning. We varied the radix bits (5-12 bits) for **radix-ip** and measured elapsed run-times and unbalanceness (the ratio between the max partition size and the min partition size) when sorting the numeric skewed (zipf 0.75) 16GB on 16 threads. As claimed in [20], increasing radix bits improves balance, which in turn minimizes the elapsed run-times. However, we found that increasing radix bits tends to increase cache-misses due to many head/tail pointers to keep track of, eventually saturating the overall performance improvement, not to mention the overhead in finding a good radix size. Meanwhile, **PARADIS** with 8-bit radix demonstrates over 2x smaller elapsed run-time in spite of unbalanced bucket size, which proves the effectiveness of our load balancing.

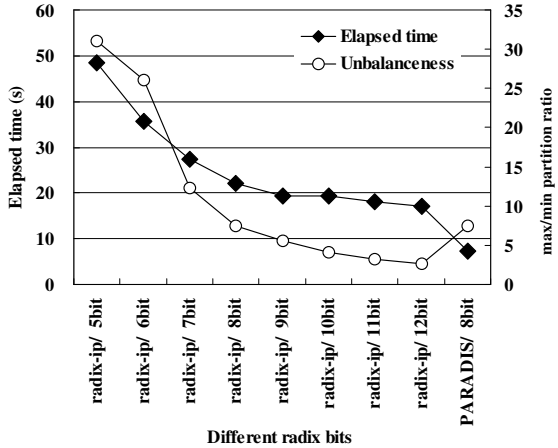
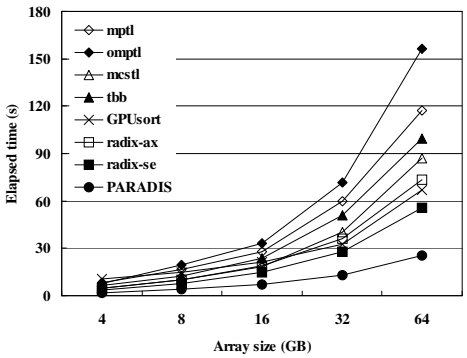
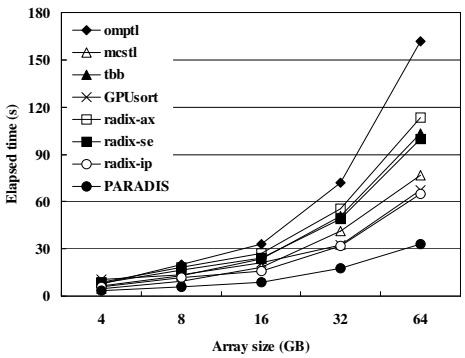


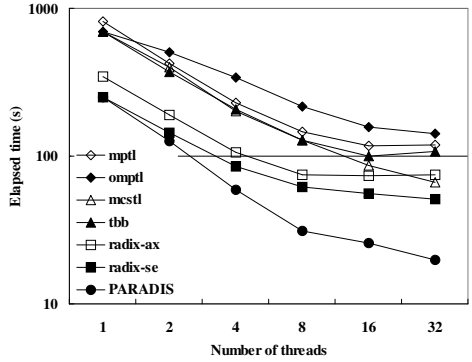

**Figure 6: Load balancing in PARADIS**



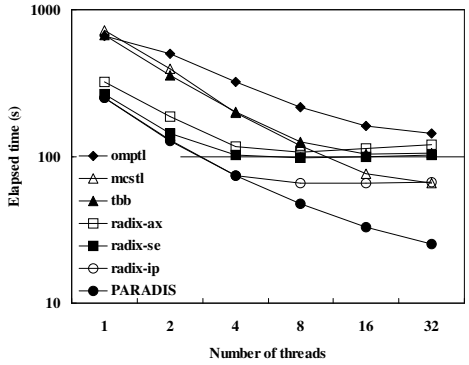

(a) Numeric random 16 threads



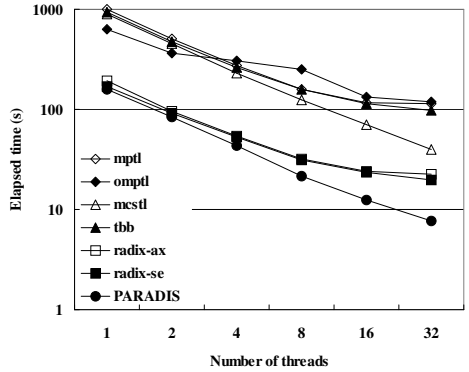

(b) Numeric skewed (zipf 0.75) 16 threads



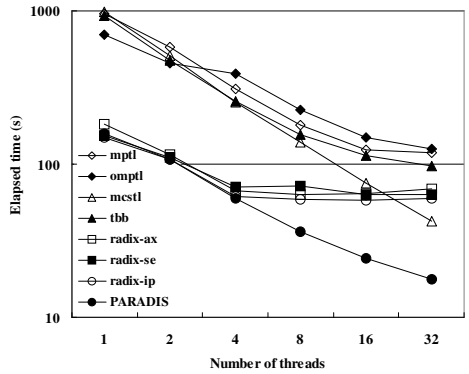

(c) Numeric random 64GB



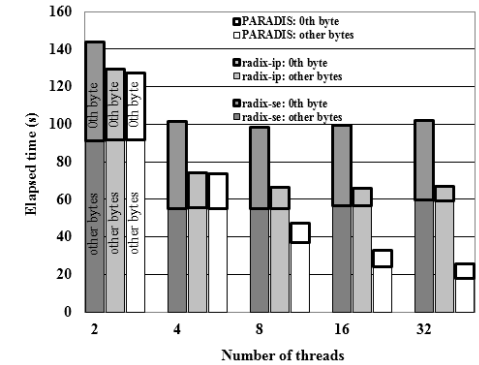

(d) Numeric skewed (zipf 0.75) 64GB



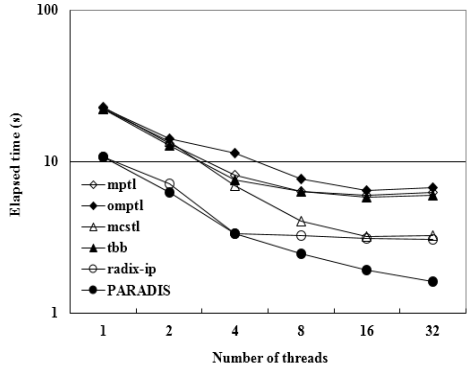

(e) String random 100GB




(f) String skewed (gensort -s) 100GB




(g) Breakdowns for numeric skewed 64GB




(h) Retail sales transaction (280M records)

**Figure 7: Performance of various sorting algorithms on numeric/string random/skewed inputs**

Due to lack of space and because of consistent trends on all benchmarks, we show only some representative results in Fig. 7 where (a)/(b) show the results of numeric benchmarks on 16 processors, (c)/(d) show the results from 64GB numeric benchmarks, (e)/(f) show the results from 100GB string benchmarks, (g) explains the high scalability in **PARADIS**, and (h) shows the results from a real-world case. We can observe similar trends from experiments, leading to the following observations.

- **PARADIS** shows the best performance as in Fig. 7 (a) and (b) due to $O(\mathcal{N})$ run-time and $O(1)$ memory complexity as well as effective load balancing.
- **radix-se** is consistently 10-40% faster than **radix-ax** even though **radix-se** does not parallelize the first level of recursion. Based on cache simulations using Valgrind, we find that while **radix-se** has nearly zero L2 cache write-miss (as it writes where it just read), **radix-ax** has over 0.5 write-miss rate.
- Ranging from 1 to 32 processors, existing in-place algorithms have a speed-up at most 6.5x for both numeric random and skewed cases. But, **PARADIS** has a speed-up 12.5x on random and 10.5x on skewed numeric data. For string benchmarks, **PARADIS** shows 20.5x speed-up on random and 8.3x on skewed data.
- The bottleneck is the first level of recursion as observed with **radix-se** in Fig. 7 (c) and (e). **PARADIS** enjoys good scaling of the 0th byte permutation (on 32 processors it is 5-6x faster than **radix-se**) as in Fig. 7 (g), thanks to our speculative permutation.
- A skewed case incurs performance degradation to **radix-se** and **radix-ip** due to poor load balancing as in Fig. 7 (g). With 2 or 4 threads, finding a balanced partitioning is easy enough for all three algorithms; they have similar runtime in sorting remaining bytes after the input is bucketized based on the 0th byte. However, with 8 or more threads, **radix-se** and **radix-ip** suffer when processing other bytes, as finding a balanced partitioning for many threads is difficult, while **PARADIS** continues to scale based on our processor-reallocation technique.
- **mcstl** shows good scalability due to its mergesort front-end at the cost of $O(\mathcal{N})$ memory requirement, but **PARADIS** is significantly faster and shows comparable scalability as in Fig. 6 (slightly better for randomly distributed inputs and slightly worse for skewed inputs). This is critical for performance when a system has limited memory capacity.
- Fig. 7 (h) shows that **PARADIS** outperforms other algorithms on a real-world data as well, where we sort the product codes in sales transactions. By comparing **PARADIS** with **radix-ip**, we can see the benchmark is highly skewed (i.e., some products are sold much more than others), yet **PARADIS** scales well thanks to the distribution-adaptive load balancing.

For comparison with **SIMDsort**, we scale the results in [3] based on their interpolation to large elements and system differences. Real-world applications require at minimum 16 byte elements (e.g, 8 byte key and 8 byte payload); yet most SIMD-based sort implementations, due to limited width of SIMD registers, handle 32bit keys only [3, 17]. According to our estimates, given 4GB of 16 byte elements on 4 cores, while **PARADIS** takes 4.6 sec, **SIMDsort** would take 9.9 sec due to the following slowdown: 1.75x due to key- payload tuple, 2x due to doubled key size, and 1.15x due to system differences. Also, note that **SIMDsort** requires 2x larger memory footprint.

We also compared **PARADIS** with **Buffsort** based on Fig. 7 from [28] where it shows **SIMDsort** based on [3] would be slightly (about 20%) faster than **Buffsort** for 16 byte elements. Accordingly, one can indirectly project that **PARADIS** would be 2.4x faster than **Buffsort** on the tested benchmarks.

# 6. CONCLUSION

In this paper, we presented PARADIS, a highly efficient fully parallelized in-place radix sort algorithm. Its speed and scalability are due to novel algorithmic improvements alone, which implies potential further speed-up when complemented with hardware-specific accelerations (e.g., SIMD). Two novel ideas, speculative permutation and distribution-adaptive load balancing, enable PARADIS to sort very efficiently large variety of benchmarks. With architectural trends towards increasing number of cores and larger memory systems, PARADIS is well suited for in-memory sorting kernels for many data management applications.

# 7. REFERENCES


[1] R. C. Agarwal. A Super Scalar Sort Algorithm for RISC Processors. In *Proc. SIGMOD*, pages 240–246, 1996.

[2] C. Balkesen, G. Alonso, J. Teubner, and M. T. Ozsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. In *Proc. VLDB*, pages 85–96, 2014.

[3] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture. In *Proc. VLDB*, pages 1313–1324, 2008.

[4] H. Dachsel, M. Hofmann, and G. Runger. Library Support for Parallel Sorting in Scientific Computations. *Euro-Par Parallel Processing*, 4641:695–704, 2007.

[5] A. C. Dusseau, D. E. Culler, K. E. Schauser, and R. P. Martin. Fast Parallel Sorting Under LogP: Experience with the CM-5. *IEEE Transactions on Parallel and Distributed Systems*, 7:791–805, 1996.

[6] B. Gedik, R. R. Bordawekar, and P. S. Yu. CellSort: High Performance Sorting on the Cell Processor. In *Proc. VLDB*, pages 1286–1297, 2007.

[7] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proc. SIGMOD*, pages 325–336, 2006.

[8] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *Proc. SIGMOD*, pages 243–252, 1994.

[9] Q. Guo, X. Guo, R. Patel, E. Ipek, and E. G. Friedman. AC-DIMM: associative computing with STT-MRAM. In *Proc. Int. Symp. on Computer Architecture*, pages 189–200, 2013.

[10] 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 989–994, 2012.

[11] http://algo2.iti.kit.edu/singler/mcstl/.

[12] http://en.wikipedia.org/wiki/Big_data/.

[13] http://sortbenchmark.org/.

[14] https://www.threadingbuildingblocks.org/.

[15] http://tech.unige.ch/omptl/.

[16] http://www.rrsd.com/.
[17] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 189–198, 2007.
[18] D. Jiménez-González, J. J. Navarro, and J.-L. Larrba-Pey. Fast parallel in-memory 64-bit sorting. In *Proc. Int. Conf. on Supercomputing*, pages 114–122, 2001.
[19] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, and J. Chhugani. CloudRAMSort: fast and efficient large-scale distributed RAM sort on shared-nothing cluster. In *Proc. SIGMOD*, pages 841–850, 2012.
[20] S.-J. Lee, M. Jeon, D. Kim, and A. Sohn. Partitioned parallel radix sort. *J. Parallel Distrib. Comput.*, 62(4):656–668, Apr. 2002.
[21] P. M. McIlroy, K. Bostic, and M. D. McIlroy. Engineering Radix Sort. *Computing Systems*, 6:5–27, 1993.
[22] D. Merrill and A. Grimshaw. High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(02):245–272, 2011.
[23] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, Jan. 2010.
[24] D. Pasetto and A. Akhriev. A comparative study of parallel sort algorithms. In *Proc. ACM Int. Conf. on Object Oriented Programming Systems Languages and Applications*, pages 203–204, 2011.
[25] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *Proc. SIGMOD*, pages 755–766, 2014.
[26] P. Ranganathan. From Microprocessors to Nanostores: Rethinking Data-Centric Systems. *Computer*, 44(1):39–48, 2011.
[27] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Proc. IEEE Int. Symp. on Parallel&Distributed Processing*, pages 1–10, 2009.
[28] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *Proc. SIGMOD*, pages 351–362, 2010.
[29] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast Sort on CPUs, GPUs and Intel MIC Architectures. Technical report, Intel Labs, 2010.
[30] J. Singler and B. Konsik. The GNU Libstdc++ Parallel Mode: Software Engineering Considerations. In *Proc. of Int. Workshop on Multicore Software Engineering*, pages 15–22, 2008.
[31] J. Singler, P. Sanders, and F. Putze. MCSTL: The Multi-core Standard Template Library. In *Proc. Int. Euro-Par Conf. on Parallel Processing*, pages 682–694, 2007.
[32] A. Sohn and Y. Kodama. Load balanced parallel radix sort. In *Proc. Int. Conf. on Supercomputing*, pages 305–312, 1998.
[33] J. Wassenberg and P. Sanders. Engineering a multi-core radix sort. In *Proc. Int. Conf. on Parallel Processing*, pages 160–169, 2011.
[34] www.itrs.net.
[35] M. Zagha and G. E. Blelloch. Radix Sort For Vector Multiprocessors. In *Proc. Int. Conf. on Supercomputing*, pages 712–721, 1991.

# APPENDIX

In this appendix, we will derive the complexity of PARADIS with the goal of establishing to what extend it can approach the theoretical optimum of $O(\frac{|\mathcal{N}|}{|\mathcal{P}|})$.

The procedure PARADIS in Algorithm 2 is invoked $\mathcal{L}$ times, where $\mathcal{L}$ is a constant dependent only on the size of the keys, therefore we can ignore it for asymptotic complexity. Lines 1 to 10 of Algorithm 2 can be clearly performed in $O(\frac{|\mathcal{N}|}{|\mathcal{P}|})$ steps, so we will analyze only the iteration of the loop on line 13.

For analysis purpose, we define the following:

- $r_i = \frac{C_i - C_i(i)}{|\mathcal{N}|}$: the ratio of wrong elements in a bucket $i$ over $|\mathcal{N}|$ after Algorithm 3.
- $r = \sum_i r_i = \frac{|\mathcal{N}| - \sum_i C_i(i)}{|\mathcal{N}|}$ : the ratio of all wrong elements over $|\mathcal{N}|$ after Algorithm 3.
- $E_i = \{p | C_i^p = 0\}$: the set of processors whose stripes are empty in bucket $i$ after Algorithm 3.
- $e_i = \frac{|E_i|}{|\mathcal{P}|}$: the fraction of stripes that are empty.
- $w = max\{\sum_{i \in \mathcal{B}_p} r_i | \forall p\}$ : the maximum fraction of elements to be repaired in PARADIS_Repair over all processors.

**Lemma 1:** $r_i \leq \frac{C_i}{|\mathcal{N}|}(1 - e_i), \forall i$

PROOF. *$e_i C_i \leq C_i(i)$, because $e_i C_i$ represents the number of elements permuted into bucket $i$ by processors in $E_i$. (The inequality may be strict because stripes $p \notin E_i$ may also contribute to $C_i(i)$).*

$$r_i = \frac{C_i - C_i(i)}{|\mathcal{N}|} \leq \frac{C_i}{|\mathcal{N}|}(1 - e_i) \tag{9}$$

□

**Lemma 2:** $r_i \leq e_i(1 - \frac{C_i}{|\mathcal{N}|}), \forall i$

PROOF. *Consider any $j \neq i$. In the bucket $i$, any stripe $p \notin E_i$ still has capacity to receive elements, and therefore any $p \notin E_i$ must have successfully permuted from bucket $j$ into bucket $i$ any element $d[n]$, where $n \in \mathcal{M}_j^p$ and $b(d[n]) = i$. Therefore in bucket $j$, any element still left belonging to bucket $i$ must be in a stripe $p \in E_i$. Thus*

$$C_j(i) = \sum_P C_j^p(i) = \sum_{E_i} C_j^p(i) \tag{10}$$

$$\leq \sum_{E_i} \frac{C_j}{|\mathcal{P}|} = \frac{|E_i|}{|\mathcal{P}|} C_j = e_i C_j \tag{11}$$

*Then, using Eq. (1) and $C_i + \sum_{j \neq i} C_j = |\mathcal{N}|$*

$$r_i = \frac{C_i - C_i(i)}{|\mathcal{N}|} = \frac{\sum_{j \neq i} C_j(i)}{|\mathcal{N}|} \tag{12}$$

$$\leq e_i \frac{\sum_{j \neq i} C_j}{|\mathcal{N}|} = e_i(1 - \frac{C_i}{|\mathcal{N}|}) \tag{13}$$

□

**Theorem 1:** $r_i \leq \frac{1}{4}$

PROOF. *Based on Lemmas 1 and 2,*

$$r_i \leq min(\frac{C_i}{|\mathcal{N}|}(1 - e_i), e_i(1 - \frac{C_i}{|\mathcal{N}|})) \tag{14}$$

$$= min(\frac{C_i}{|\mathcal{N}|}, e_i) - e_i \frac{C_i}{|\mathcal{N}|} \leq \frac{1}{4} \tag{15}$$

*Eq. (15) follows because "$min(x, y) - xy$" achieves maximum over the domain $0 \leq x, y \leq 1$ when $x = y = \frac{1}{2}$.* □

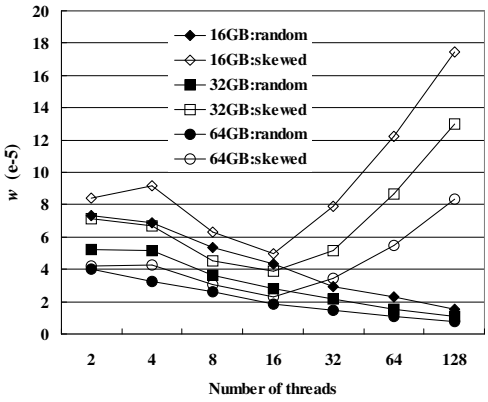



**Figure 8: $w$ values from numeric benchmarks**

**Corollary 1:** $r \leq 1 - \frac{1}{|\mathcal{B}|}$

PROOF.

$$r = \sum_i r_i \leq \sum_i \frac{C_i}{|\mathcal{N}|} - \sum_i (\frac{C_i}{|\mathcal{N}|})^2 \tag{16}$$

*which will be maximal with* $C_i = \frac{|\mathcal{N}|}{|B|}, \forall i$. *Thus*

$$r \leq 1 - \frac{1}{|\mathcal{B}|} \tag{17}$$

□

The complexity of each iteration in PARADIS depends on two parts: PARADIS_Permute and PARADIS_Repair. While the former has $O(\frac{|\mathcal{N}|}{|\mathcal{P}|})$ complexity as $\mathcal{N}$ is evenly divided to $|\mathcal{P}|$ processors by *PartitionForPermutation*, it is possible that PARADIS_Repair suffers from unbalanced work assigned to different processors. This can happen when one partition is left with many more incorrect elements than the others. The complexity of PARADIS_Repair depends on the maximum number of elements to be repaired by a single processor (i.e., $w$).

If we let $T(\mathcal{N})$ be the complexity of PARADIS, then

**Theorem 2:** $T(\mathcal{N}) \leq O(|\mathcal{N}|(\frac{1}{|\mathcal{P}|} + w))$

PROOF. *Without loss of generality, we let* $r$ *and* $w$ *represent their maxima over all iterations. Then*

$$T(\mathcal{N}) \leq (\frac{|\mathcal{N}|}{|\mathcal{P}|} + w|\mathcal{N}|) + r(\frac{|\mathcal{N}|}{|\mathcal{P}|} + w|\mathcal{N}|) + r^2(..) + ... \tag{18}$$

$$= \sum_{t=0}^{\infty} r^t (\frac{|\mathcal{N}|}{|\mathcal{P}|} + w|\mathcal{N}|) = (\frac{|\mathcal{N}|}{|\mathcal{P}|} + w|\mathcal{N}|)\frac{1}{1-r} \tag{19}$$

*By Corollary 1,* $\frac{1}{1-r} \leq |\mathcal{B}|$ *which is constant. Hence*

$$T(\mathcal{N}) \leq O(|\mathcal{N}|(\frac{1}{|\mathcal{P}|} + w)) \tag{20}$$

□

The above proof does not rely on any bound on the number of iterations $t$. Nevertheless, since repairing stops when $|\mathcal{N}| \cdot r^t < 1$ (i.e., less than a single element left for repair after $t$ iterations), we can state the number of iterations is bounded as follows:

$$t < -\log_r |\mathcal{N}| \tag{21}$$

Corollary 1 then provides a theoretical upper bound on $r$. As $w$ and $r$ are the worst repair load for one processor and the total repair load over all processors respectively, we can use the quantity $w$ and the relation $r \leq w|\mathcal{P}|$ for practical estimates of the number of iterations.

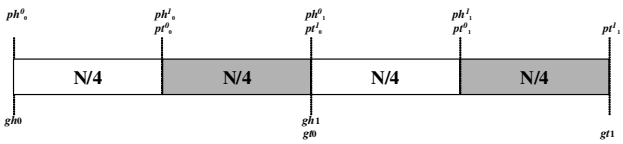
Since $w$ does not scale with $|\mathcal{P}|$, understanding the impact of $w$ on various problems is critical. Fig. 8 shows the $w$ values on the benchmarks in Section 5. As $w$ represents the maximum percentage of elements requiring repair over all processors, $w$ serves as an indicator of how good our speculation is (when poor, $w$ increases requiring higher repairing efforts). For skewed benchmarks the $w$ values get larger after 16 processors, because the largest bucket cannot be repaired by multiple processors and the buckets are more fractured. Nonetheless, we can see that $w$ values are very small regardless of size/skewness, and get smaller with larger $\mathcal{N}$ (as only a fraction of $\mathcal{N}$ needs repair). This is what makes PARADIS highly scalable for big data and leads to Corollary 2.

**Corollary 2:** $T(\mathcal{N})$ *converges to* $O(\frac{|\mathcal{N}|}{|\mathcal{P}|})$, *as* $w$ *goes to 0.*

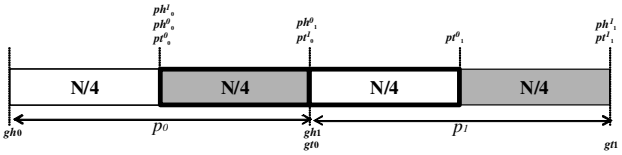
PROOF.

$$\lim_{w \to 0} O(|\mathcal{N}|(\frac{1}{|\mathcal{P}|} + w)) = O(\frac{|\mathcal{N}|}{|\mathcal{P}|}) \tag{22}$$

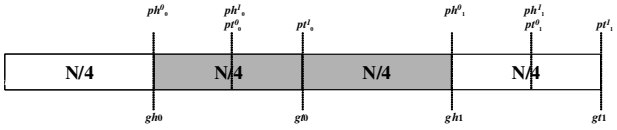
□




(a) the worst case for PARADIS in the 1st iteration




(b) the worst case for repairing with $r_{\{0,1\}} = w = \frac{1}{4}$




(c) the ideal case for PARADIS in the 2nd iteration

**Figure 9: A pathological case for PARADIS**

Fig. 9 (a) is the pathological case for PARADIS, where $\mathcal{M}_0$ and $\mathcal{M}_1$ are for white and gray elements, respectively. As you see, PARADIS_Permute cannot permute any element, which creates the worst case for PARADIS_Repair as in (b), with $w = \frac{1}{4}$. Fig. 9 (c) shows that PARADIS_Repair efficiently shrinks down the problem for the second iteration, in spite of the first iteration being the worst case for PARADIS. As a result, the problem becomes smaller and *ideal* for PARADIS_Permute, with $w = 0$ (i.e., no more iterations).