INFO 6205 Final Project Literature Review

Qinyun Lin , Xinlei Bian , and Yiqing Huang

College of Engineering
Northeastern University
Boston, MA
{*lin.qiny, bian.xin, huang.yiqin*}*@northeastern.edu*

Abstract—**In this literature review, we mainly focus on how to parallelize MSD radix sort. The second paper is about in-place radix sort whose idea is applied in the other three papers. The other papers make MSD radix sort parallel in different approaches.**

1. PARADIS: An Efficient Parallel Algorithm for In-place Radix Sort [1]

1.1. 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 difficult 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.

1.2. Algorithms

The author in this paper proposed a parallel in-place radix sort algorithm, PARADIS.

Table 1 lists the related notations and concepts. Let's assume that a given array of $|N|$ elements to be sorted by the key of each element. An element consists of both key and payload, although PARADIS is also applicable to be the case where keys and payloads are sorted separately.

TABLE 1. RELATED NOTATIONS

N	set of array indices $0, 1, ..., \|N\| - 1$
$d[N]$	the array of size $\|N\|$ to be sorted
n, h, t	array index $\in N$
P	set of processor indices $0, 1, ..., \|P\| - 1$
p, q	processor index $\in P$
$p_0, p_1, ...$	shorthand for "processor 0", "processor 1", ...
B	set of bucket indices $0, 1, \|B\| - 1$
i, j, k	bucket index $\in B$
L	set of recursion levels $0, 1, ..., \|L\| - 1$
l	recursion level $\in L$
$b(v)$	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 process p in bucket i
M_i	$n\|gh_i \leq n \leq gt_i$, i.e., the indices of bucket i
M_i^p	$n\|ph_i^p \leq n \leq pt_i^p$, i.e., the indices of stripe p, i
C_i	$\|M_i\| = gt_i - gh_i$, i.e., size of bucket i
C_i^p	$\|M_i^p\| = pt_i^p - ph_i^p$, i.e., size of stripe p, i
$C_i(k)$	$\|n \in M_i\|b(d[n]) = k\|$ i.e. the number elements in M_i belonging to M_k
$C_i^p(k)$	$\|n \in M_i^p\|b(d[n]) = k\|$ i.e. the number elements in M_i^p belonging to M_k

1.2.1. Overview. *Algorithm 1* shows the original MSD Radix Sort. There are 4 steps:

Step 1 The unsorted input array is scanned to build a histogram of the radix key distribution

Step 2 The input array is partitioned into $|B|$ buckets by computing gh_i and gt_i

Step 3 Each element is checked and permuted if it is not in the right bucket

Step 4 Once element permutation is completed, each bucket becomes a sub-problem, which can be solved independently and recursively

Algorithm 2 shows the PARADIS algorithm which is designed in this paper. For *step 1* and *step 2* of the radix sort, it is easy to turn into a parallel version. However, parallelizing *step 3* is very challenging due to the read-after-write

Algorithm 1 Radix Sort

```
1: procedure RADIXSORT($d[n], l$)
2:     $b = b_l$
3:     $B =$the range of $b()$
4:     $cnt[b] = 0$
5:     for $n \in N$ do
6:         $cnt[b(d[n])] + +$
7:     for $i \in B$ do
8:         $gh_i = \sum_{j<i} cnt[j]$
9:         $gt_i = \sum_{j\leq i} cnt[j]$
10:    for $i \in B$ do
11:        while $gh_i < gt_i$ do
12:            $v = d[gh_i]$
13:            while $b(v)! = i$ do
14:                $swap(v, d[gh_{b(v)} + +])$
15:            $d[gh_i + +] = v$
16:    if $l < L - 1$ then
17:        for $i \in B$ do
18:            RadixSort($d[M_i]$, $l + 1$)
```

Algorithm 2 PARADIS

```
1: procedure PARADIS($d[N], l, P$)
2:     $b = b_l$
3:     $B =$the range of $b()$
4:     $..., A_p, ... = PartitionForHistogram$
5:     for $p \in P$ in parallel do
6:         Build local histogram for $d[A_p]$
7:     Synchronization
8:     Build global histogram from the $|P|$ local histograms
9:     Compute $gh_i$ and $gt_i$, $\forall i$
10:    Synchronization
11:    $..., B_p, ... = PartitionForRepair$
12:    while $\sum_i C_i > 0$ do
13:        $..., M_i^p, ... = PartitionForPermutation$
14:        for $p \in P$ in parallel do
15:            PARADIS_Permute(p)
16:        Synchronization
17:        for $p \in P$ in parallel do
18:            for each $i \in B_p$ do
19:                PARADIS_Repair(i)
20:        Synchronization
21:    if $l < L - 1$ then
22:        $..., P_i, ... = PartitionForRecursion$
23:        for $i \in B$ in parallel do
24:            PARADIS($d[M_i], l + 1, P_i$)
```

dependency on the gh_i, and the unbalanced sub-problem sizes in *step 4* can degrade the end-to-end performance.

This paper proposed PARADIS algorithm addresses the above challenge with two novel techniques: *Speculative Permutation* and *Distribution-adaptive Load Balancing* to solve the two drawbacks mentioned above.

1.2.2. Speculative Permutation. Speculative permutation is a key technique to maximize parallelism in element permutation. Essentially, it is an iterative algorithm which reduces the problem size significantly at each iteration.

Algorithm 3 PARADIS_Permute

```
1: procedure PARADIS_PERMUTE($p$)
2:     for $i \in B$ do
3:         $head = ph_i^p$
4:         while $head < pt_i^p$ do
5:             $v = d[head]$
6:             $k = b(v)$
7:             while $k! = i$ and $ph_k^p < pt_k^p$ do
8:                 $swap(v, d[ph_k^p + +])$
9:                 $k = b(v)$
10:            if $k == i$ then
11:                $d[head + +] = d[ph_i^p]$
12:                $d[ph_i^p + +]$
13:            else
14:                $d[head + +] = v$
```

Once all the buckets are partitioned into strips, use *Algorithm 3* in each processor p to perform in-place permutation. There are three fundamental modification comparing to the *step 3* of **Algorithm 1**.

- Since the partitioning of each bucket is merely speculative, in order not to overwrite existing elements, check if the target stripe is full
- ph_i^p increases only if a correct element is found, which keeps all correctly placed elements before ph_i^p
- At the end of *Algorithm 3*, all wrong elements int the bucket i are kept between ph_i^p and pt_i^p, and *Algorithm 4* repairs it

Algorithm 4 PARADIS_Repair

```
1: procedure PARADIS_REPAIR($i$)
2:     $tail = gt_i$
3:     for $p \in P$ do
4:         $head = ph_i^p$
5:         while $head < pt_i^p$ and $head < tail$ do
6:             $v = d[head + +]$
7:             if $b(v)! = i$ then
8:                 while $head < tail$ do
9:                     $w = d[- - tail]$
10:                    if $b(w) == i$ then
11:                        $d[head - 1] = w$
12:                        $d[tail] = v$
13:                        break
14:    $gh_i = tail$
```

1.2.3. Distribution-adaptive Load Balancing. Using distribution-adaptive load balancing method, PARDIS dynamically reallocates processor resources only after it finds imbalance.

1.3. Conclusion

In this paper, the authors presented PARDIS, 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 speedup when complemented with hardware-specific accelerations (e.g., SIMD). Two novel ideas, speculative permutation and distribution adaptive load balancing, enable PARDIS to sort very efficiently large variety of benchmarks. With architectural trends towards increasing number of cores and larger memory systems, PARDIS is well suited for in-memory sorting kernels for many data management applications.

2. Formulation and analysis of in-place MSD radix sort algorithms [2]

2.1. Abstract

We present a unified treatment of a number of related inplace MSD radix sort algorithms with varying radices, collectively referred to here as ”Matesort” algorithms. These algorithms use the idea of in-place partitioning which is a considerable improvement over the traditional linked list implementation of radix sort that uses $O(n)$ space. The binary Matesort algorithm is a recast of the classical radix exchange sort, emphasizing the role of in-place partitioning and efficient implementation of bit processing operations.

This algorithm is $O(k)$ space and has $O(kn)$ worst-case order of running time, where k is the number of bits needed to encode an element value and n is the number of elements to be sorted. The binary Matesort algorithm is evolved into a number of other algorithms including ”continuous Matesort” for handling floating point numbers, and a number of ”general radix Matesort” algorithms. We present formulation and analysis for three different approaches (sequential, divide-and-conquer and permutation-loop) for partitioning by the general radix Matesort algorithm. The divide-and conquer approach leads to an elegantly coded algorithm with better performance than the permutation-loop-based American Flag Sort algorithm.

2.2. Algorithm Details

The main idea of this ”Matesort” is quite like Quicksort. The difference is in the way to partition in each step, and an extra ”bit location” input parameter. In addition to text data, there is a useful trick for exploiting data redundancy.

2.2.1. Encoding And Partition. The elements in the array to be sorted will be encoded using k bits. After that, the array can be partitioned based on the rules that all elements with $b_{k-1} = 0$ appear before elements with $b_{k-1} = 1$. Then what remains to be done is to sort each group. In particular to Bit Partition, it is said that given an integer array $A[1..n]$ and a bit position i, return k such that bit b_i in any of $(A[1], A[2], ..., A[k]) \leq 0 < b_i$ in any of $(A[k+1], A[k+2], ..., A[n])$.

2.2.2. Bit Operation. To isolate the bit at a bit location, one way is to use the shifting and a mask. The expression is: $(A[i] >> bitloc)$ & 0×1. However, a more efficient method is to use the expression $(A[i]$ & $Mask)$, where $Mask = 2^{bitloc}$.

2.2.3. Exploit Data Redundancy. If we know beforehand that the text is limited to English uppercase letters, then, rather than scanning the whole range of values (0–255) represented by a byte, the range should be limited to 65–90 (corresponding to the ASCII encoding of letters ’A’–’Z’). For us, we can also make use of this idea to reduce the Chinese Pinyin index.

2.2.4. Conclusion. The main idea for this algorithm is similar to Huskysort - making use of encode of elements to put them as close as possible to where they should be. While Huskysort uses the value of encode to do the comparable sort, which is different from here.

As for Matesort, it will scan the encode bit by bit like classical radix sort. Experiments have shown that Matasort is not only in-place, but also better than Quicksort when data redundancy increases.

3. Sequential And Parallel Hybrid Approach for Non-Recursive Most Significant Digit Radix Sort [3]

3.1. Abstract

Sorting and parallelism are widely studied subjects and their combination is a challenging topic of importance in many scientific fields. Efficient sorting is important because of reduced time and energy consumption; implementation of an efficient parallel sorting algorithm is the main focus of this article. In this study, traditional (recursive) sequential and alternative (non-recursive) sequential versions of the most significant-digit radix sort algorithms are implemented, and their performance is compared. Furthermore, two-hybrid sequential and parallel versions combining the most significant-digit radix sort and Quicksort are implemented and evaluated. OpenMP API is utilized to implement the parallel algorithms.The resulting parallel versions outperform the well-known efficient Quicksort algorithm.

The goal of this paper is to come up with an alternative sequential most significant digit radix sort algorithm that performs better than the traditional recursive version. The second goal is to implement the sequential and parallel versions of a hybrid combination of the most significant digit radix sort and quicksort algorithm and study their performance. Radix sort uses buckets for the sorting process. Since this paper uses an integer to implement sorting algorithms, a total of ten buckets each representing all possible values of a digit is needed. Seven algorithms that include the traditional recursive and non-recursive MSD radix sort, sequential Quicksort, 2 versions of sequential hybrid MSD radix sort, and 2 versions of parallel hybrid MSD radix sort are implemented for comparative study.

3.2. Algorithms

3.3. The Non-recursive MSD Radix Sort

The non-recursive MSD radix sort using the first bucket, second bucket, and middle bucket. The first bucket and the second bucket size depend on the number of input elements. The size of the middle bucket is 10 for the integer element. The algorithm starts with allocating elements into the $first\_bucket$ based on the most significant digit. Then each index in the first bucket is a vector that stores one or more elements whose most significant digit equals the index number. Then each vector in the first bucket needs to be sorted based on the second most significant digit by allocating them in the middle bucket. Then, the result in the middle bucket is put in the second bucket at the corresponding index as a vector. Then, the middle bucket is used to sort based on the third most significant digit from the second bucket, and the result is stored in the first bucket. The process repeats until the size of all vectors is 1 or all digits have been sorted. The process is shown in *Figure 1*.

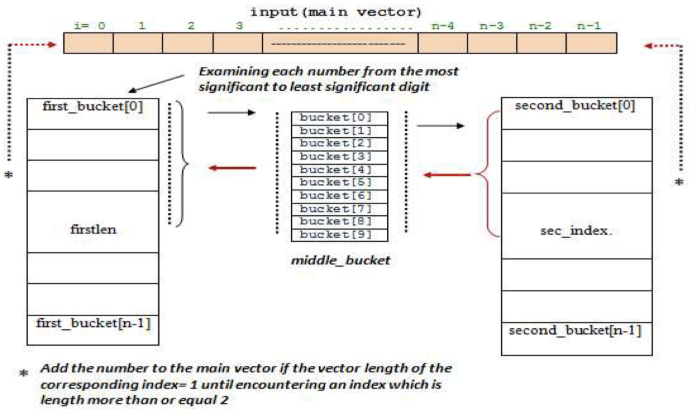

Figure 1. Non-Recursive MSD Radix Sort Algorithm Sorting Process Diagram

3.3.1. Sequential and parallel Hybrid MSD Version 1. In the hybrid versions, the value of ”step” is determined before the sorting process begins. ”steps” indicates the number of times MSD Radix sort is applied to the input before Quicksort. This version is based on the non-recursive MSD radix sort described above.

First, input is processed with MSD radix sort ”step” rimes. The result is a list of buckets containing vectors. Then quicksort is applied to sort the elements in each vector and sorted vectors are written in order as a result.

Since after application of the MSD Radix Sort, the individual vectors generated by the $first\_bucket$ and the $second\_bucket$ respectively can be independently and in parallel using the Quicksort. It is easy to parallelize Hybrid MSD Version 1 by sorting independent vectors in different threads.

3.4. Sequential and parallel Hybrid MSD Version 2

In Version 2, it uses an additional data structure, $middle\_bucket1$, to split the input based on the most significant digit in the first iteration of MSD radix. Then applying Version 1 to each of the vectors in $middle\_bucket1$.

The parallel section of this algorithm is the application of Quicksort to independent buckets. Each vector is implemented quicksort in different threads.

3.5. Conclusion

This survey shows the non-recursive MSD radix sort version performs remarkably better than the traditional recursive MSD radix sort. Moreover, both parallel hybrid version performs faster than their sequential counterparts and they yield significant performance gain over Quicksort.

4. Engineering a Multi-core Radix Sort [4]

4.1. Abstract

We present a fast radix sorting algorithm that builds upon a micro architecture-aware variant of counting sort. Taking advantage of virtual memory and making use of write-combining yields a per-pass throughput corresponding to at least 89% of the system’s peak memory bandwidth. Our implementation outperforms Intel’s recently published radix sort by a factor of 1.64. It also compares favorably to the reported performance of an algorithm for Fermi GPUs when data-transfer overhead is included. These results indicate that scalar, bandwidth-sensitive sorting algorithms remain competitive on current architectures. Various other memory-intensive applications can benefit from the techniques described herein.

4.2. Algorithms

4.2.1. Software Write-Combining. There are several cases that may have a serious impact on performance with numerous memory accesses. First case is writing to multiple streams, second is a large number of write-only accesses, these two cases can lead to negative influence to cache performance.The third problem is single memory accesses can involve significant bus overhead.

Therefore, the author recommended the use of Software Write-Combining whenever a core’s active write destinations outnumber its write-combine buffers to avoid these problems.

4.2.2. Virtual-Memory Counting Sort.

The author described an improved variant for Counting Sort, which makes use of virtual memory and write-combining. First optimization goal is to avoid the initial counting pass, and then is to efficiently write values to storage using the write-combining technique.

4.2.3. Radix Sort.

The author introduced a new variant based on the Virtual-Memory Counting Sort described above. In particular to MSD, they also explain that the overhead of managing numerous (nearly empty) buckets makes MSD radix sort less suited for relatively small N. While for LSD, by contrast, the bucket setup cost will be amortized over the number of elements, and avoids the possibility of load imbalance for parallelization as the price of data copy increases.

To solve this problem, they make use of ”reverse sorting”, which means use one or more MSD at first to partition data into bucket, and use LSD sort in each bucket. This method turns out to be more advantageous for Non-Uniform Memory Access (NUMA) systems because each processor is responsible for writing a contiguous range of outputs, thus ensuring the OS allocates those pages from the processor’s NUMA node.

4.2.4. Conclusion.

The author introduced improvements to counting sort and a novel variant of radix sort for integer key/value pairs. Bandwidth measurements indicate their algorithm’s throughput is within 11% of the theoretical optimum for the given hardware, which indicates that bandwidth-sensitive sorting algorithms still have their place on current architectures.

Since their Radix sort is limited to relatively small integer keys, they may try to solve it by sorting extremely large buckets from the MSD phase using multiple processors in the future.

References

[1] M. Cho, D. Brand, R. Bordawekar, U. Finkler, V. Kulandaisamy, and R. Puri, “Paradis: an efficient parallel algorithm for in-place radix sort,” *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1518–1529, 2015.

[2] N. Al-Darwish, “Formulation and analysis of in-place msd radix sort algorithms,” *Journal of information science*, vol. 31, no. 6, pp. 467–481, 2005.

[3] A. A. Aydin and G. Alaghband, “Sequential and parallel hybrid approach for nonrecursive most significant digit radix sort,” in *10th International Conference on Applied Computing*, 2013, pp. 51–58.

[4] J. Wassenberg and P. Sanders, “Engineering a multi-core radix sort,” in *European Conference on Parallel Processing*. Springer, 2011, pp. 160–169.