MSL: An Efficient Adaptive In-Place Radix Sort Algorithm

Fouad El-Aker[1] and Amer Al-Badarneh[2]

[1] Computer Science Department, New York Institute of Technology,
P.O. Box 940650, Amman 11194, Jordan
Fouad_elaker@yahoo.ca
[2] Computer Information Systems Department, Jordan University of Science and Technology
P.O. Box 3030, Irbid 22111, Jordan
Amerb@just.edu.jo

Abstract. This paper presents an in-place pseudo linear average case radix sorting algorithm. The proposed algorithm, MSL (Map Shuffle Loop) is a modification of the ARL algorithm. The MSL permutation loop is faster than the ARL counterpart since it searches for the root of the next permutation cycle group by group. The permutation cycle loop maps a key to its target group and shuffles the input array. The performance of MSL is compared with Java quicksort, as well as MSD and LSD radix sorting algorithms.

1 Introduction

Radix sorting [1] algorithms fall into two major categories, depending on whether they process bits left to right, or right to left. After the k^{th} step, MSD (Most Significant Digit) sorts the input keys according to their left most k digits. Similarly, LSD (Least Significant Digit) processes digits in the opposite direction, but it is non-recursive, unlike MSD. MSL processes bits left to right recursively. MSL is described in Section 2. In Section 3, we present the algorithm analysis and experimental results. Finally, Section 4 gives the conclusion and future work.

2 The MSL Algorithm

The main steps of MSL sorting algorithm are presented first. Each step executes a loop whose time complexity is indicated. N is the size of input data, while K is the number of groups.

```
Step 1: Compute groups' sizes.                    O(N)
Step 2: Compute groups' start and end addresses.  O(K)
Step 3: Permutation cycles (Map shuffle) loop.    O(N)
Step 4: Process groups recursively.               O(K)
```

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3037, pp. 606–609, 2004.

MSL is a modification of the ARL algorithm [2]. Both algorithms use permutation cycles, which is the primary reason for the in-place property of the algorithms. MSL and ARL differ in how they search for and select the root of the next permutation cycle, Step 3, and in how they compute the radix size.

Radix computing in MSL is similar to adaptive MSD (AMSD) [3]. AMSD radix setting is simple to understand, and is modified to work on bits instead of digits. The following modifications are applied to the implementations of both MSL and AMSD: (1) Recompute the radix size continually on recursive calls instead of only once, initially. (2) Use multiple of 4 radix size values and disallow the use of small radix size values, 1 to 3, to improve the running time. The radix sizes used in MSL are 4, 8, and 12, etc. The initial radix size value 15 is used in Section 3 in sorting large arrays of 31 bits integers, to avoid using radix size values 1 to 3.

In addition, the implementations of MSL and AMSD use one iteration of quicksort partitioning before calling insertion sort. This initial call inserts the smallest key of each of the two partitions in its correct place. This speeds up insertion sort, which is called frequently. Insertion sort in this case has a redundant if-condition that is removed from its inner loop.

The digit size is computed in MSL as follows: (1) For array sizes 1..25, no radix is used, and the quicksort partition method is called instead. (2) For array sizes 26..5000, 5001..80000, and larger than 80001, a digit size 4, 8, and 12 is used, respectively. In comparison, ARL computes the digit size in terms of the largest key value in the array, and applies a maximum threshold on digit sizes.

MSL, like ARL, requires 2 supporting arrays, each of size 2^d, where d is the current digit size. For MSL extra space requirements, see Section 3. ARL uses the supporting arrays to store groups' start addresses and sizes. MSL replaces the groups' sizes by the groups' end addresses. In Step 2 above, K extra assignments are executed to replace sizes by end addresses.

MSL inserts a key in its target group, at the key target group's end address. The key target group's end address is decremented immediately beforehand. MSL performs a total of N decrement operations for this purpose. In addition to these N decrement operations, ARL needs extra N addition operations to compute keys' target addresses. A key's target group's address is computed by summing its target group's start address and size.

ARL and MSL permutation cycles, Step 3, insert keys into their target groups. Each permutation cycle starts and ends at the same location, which is the root key location. The root key is the initial key used in the cycle, and is recomputed when a new permutation cycle is started.

In ARL, the exchange cycle root key is the first key, counting from the left end of the array that is not in its target group. ARL uses an independent loop of total N increment operations to find all roots for permutation cycles. In MSL, a total of K increment operations are needed instead.

MSL Step 3 is organized into 3 substeps. In substep 1, MSL finds the next root key, by searching through groups, which makes it faster in principle than ARL. The *originGroup* variable advances through groups until a group is found that is not fully processed (when the group's start and end addresses do not match). At this point, *originGroup* points to the left most group, which has a key that may not be in its target group (See example below). MSL permutation code stops when the last non-empty group is reached. The condition used is: if (*originGroup* == *maxGroup*).

In substep 2, root key information is updated. The root key (*root_key*) of the next exchange cycle is computed, in addition to its group (*root_key_group*) and its destination group (*dest_group*). Target addresses are computed directly from target groups using the group's end address.

Substep 3, is the actual permutation cycle. As exchanges are performed, *root_key_group* remains unchanged. For each body execution of the permutation cycle, *root_key* is exchanged with the key that is in its destination address, where the destination address is computed from *dest_group*. Next, *dest_group* is updated to the destination group of the current *root_key*. The current exchange cycle stops when a key is found whose target group is equal to *root_key_group*. The condition used is: if (*dest_group* == *root_key_group*).

The following is a trace of the 3 substeps above. The example assumes we are sorting a group of 8 bit integers on the left most 4 bits, where the radix size value is 4. There are a total of three groups, whose identities, start and end addresses are provided. The changes made by a substep are bolded. Only the modified group information after a substep execution is displayed.

Initial array data: ⟨3, 20, 1, 21, 41, 22, 40, 42, 2⟩;
Initial groups data: *minGroup* = 0; *maxGroup* = 2; (*Gid* = 0, *Start* = 0, *End* = 3);
(*Gid* = 1, *Start* = 3, *End* = 6); (*Gid* = 2, *Start* = 6, *End* = 9);
After substeps 1 & 2: *originGroup* = 0; *root_key_group* = 0; *root_key_address* = 2;
After exchanges in substep 3:
First exchange: ⟨3, 20, **1**, 21, 41, 22, 40, 42, 2⟩; (*Gid* = 0, *Start* = 0, *End* = **2**);
// Key value 1 is already in its target group
After substeps 1 & 2: *originGroup* = 0; *root_key_group* = 0; *root_key_address* = **1**;
After exchanges in substep 3:
First exchange: ⟨3, , 1, 21, 41, **20**, 40, 42, 2⟩; (*Gid* = 1, *Start* = 3, *End* = **5**);
Next exchange: ⟨3, , 1, 21, **22**, 20, 40, 42, 2⟩; (*Gid* = 1, *Start* = 3, *End* = **4**);
Next exchange: ⟨3, , 1, 21, 22, 20, 40, 42, **41**⟩; (*Gid* = 2, *Start* = 6, *End* = **8**);
Next exchange: ⟨3, **2**, 1, 21, 22, 20, 40, 42, 41⟩; (*Gid* = 0, *Start* = 0, *End* = **1**);
// Key value 2 is inserted in root key address, closing the exchange cycle
After substeps 1 & 2: *originGroup* = 0; *root_key_group* = 0; *root_key_address* = **0**;
After exchanges in substep 3:
First exchange: ⟨**3**, 2, 1, 21, 22, 20, 40, 42, 41⟩; (*Gid* = 0, *Start* = 0, *End* = **0**);
// Key value 3 is already in its target group
After substeps 1 & 2: *origin group* = **1**; *root_key_group* = **1**; *root_key_address* = **3**;
After exchanges in substep 3:
First exchange: ⟨3, 2, 1, **21**, 22, 20, 40, 42, 41⟩; (*Gid* = 1, *Start* = 3, *End* = **3**);
// Key value 21 is already in its target group
After substeps 1 & 2: *originGroup* = 2; Exit, since *originGroup* == *maxGroup*;

3 Experimental Results

In the following, assume that the average applied digit is A, and that we are sorting B bit integers. For the worst case, the algorithm is called recursively to the maximum possible number of levels, $L = B/A$, where partitioning is always severely unbalanced. The worst case is $O(N*L)$. The average case is $O(N \log_C N)$, when partitioning is always balanced, and the algorithm is called recursively to level $\log_C N$, where $C = 2^A$.

In Table 1, all the tests take the average of five runs. The machine used in the tests is Compaq, 500 MHZ Pentium III, with 128 MB RAM. The distributions used are random with varying densities. We divide the full integer range into *x* subranges and use the first subrange to randomly generate the data for the random distribution named R/*x*. The smallest integer value used in the distribution R/*x* is zero, and the largest integer value used is Java_Max_Integer/*x*.

MSL is compared with AMSD, LSD, and Java tuned quicksort. For all 4 algorithms, the run time improves as the range is decreased. MSL run time is more than twice as fast as Java quicksort, for all runs. MSL run time is better than AMSD. MSL and LSD running time are relatively superior compared to Java quicksort and AMSD. The time taken to allocate extra space is excluded. The extra space allocated for LSD is *N*. The extra space allocated for MSL and AMSD is 0.14**N*, 0.07**N*, 0.35**N*, and 0.0175**N*, for the array sizes 0.5, 1, 2, and 4 millions, respectively. For other random distributions, the algorithms behave similarly.

Table 1. A comparison shows MSL, AMSD, LSD, & Java tuned quicksort (QSort) running times in milliseconds. Sizes are multiple of 10^6

Sizes * 10^6		**0.5**	**1**	**2**	**4**		**0.5**	**1**	**2**	**4**
LSD	R/1	372	770	1550	3130	R/10^3	306	604	1204	2426
AMSD		474	868	1614	3186		362	682	1408	2988
QSort		614	1352	2868	6064		636	1330	2778	5734
MSL		340	726	1430	2910		272	570	1222	2332
LSD	R/10	330	660	1338	2702	R/10^6	198	406	808	1670
AMSD		428	680	1516	3186		322	582	1228	2558
QSort		670	1360	2866	6046		372	750	1528	3196
MSL		284	626	1408	2746		208	328	738	1500

4 Conclusion and Future Work

In-place radix sorting includes ARL and MSL. The run time of MSL is competitive with other radix sorting algorithms, such as LSD. Future work on in-place radix sorting algorithms includes engineered in-place radix sorting. In addition, testing in-place radix sorting algorithms on 64 and 128 bits integers is important.

References

1. Sedgewick, R.: Algorithms in Java, Parts 1-4, 3rd Edition, Addison-Wesley, (2003).
2. Maus, A.: "ARL: A Faster In-place, Cache Friendly Sorting Algorithm", Norsk Informat-ikkonferranse, NIK'2002, (2002) 85-95.
3. Anderson, A., Nilsson, S.: Implementing Radixsort, ACM Journal of Experimental Algorithms. Vol. 3(7). (1998) 156-165.