Fast In-place Integer Radix Sorting

Fouad El-Aker

Computer Science Department, University of Petra,
P.O. Box 940650, Amman 11194, Jordan
elaker_fouad@maktoob.com
elaker_fouad@yahoo.ca

Abstract. This paper presents two very efficient sorting algorithms. MSL is an O(*N***B*) in-place radix sorting algorithm, where *N* is the input size and *B* is the keys length in bits. This paper presents an implementation of MSL that is sub-linear in practice, for uniform data, on Pentium 4 machines. We also present an O(*N***logN*) hybrid quicksort that has a non-quadratic worst case.

1 Introduction

Right to left LSD and left to right MSD are O(*N***B*) radix sorting algorithms. *N* is the input size and *B* is the length of keys in bits. LSD and MSD use an extra space of size *N*. ALR [4] and MSL [1] process bits left to right, however unlike MSD, ALR and MSL are in-place and cache friendly. MSD, ALR and MSL execute recursively, for every partition. LSD body code is executed only *B/D* times, where *B* is the length of keys in bits and *D* is the used digit size. This makes the design of algorithms faster than LSD quite difficult. LSD is faster than MSD in [6]. We present a sub-linear run time MSL implementation suitable for sorting 31 bits and 63 bits integers in Java in this paper. MSL implementation in Section 2 uses small digit sizes increasing data cache friendliness. MSL loops were implemented reducing the number of instructions, and therefore increasing instruction cache friendliness. In addition, section 3 presents a non-quadratic implementation of quicksort, called switch sort. Hybridizing switch sort and MSL does not improve over MSL. Section 4 presents the test results. Section 5 gives the conclusions and future work.

2 MSL and Smaller Digit Sizes

MSL and ALR use a permutation loop in order to avoid reserving an extra array of size *N*, which is performed in MSD. The main steps of MSL are presented in [1]. MSL permutation loop, shuffles keys into their target groups. In the circular list of keys, K = <K_1, K_2, .. , K_L>, assume the Target Address (K_J) = Array Location (K_{J+1}), where J is not equal to L, and Target Address (K_L) = Array Location (K_1). Digit extraction and group end address lookup are used in computing a key's target address. MSL permutation loop moves keys in a circular list K to their target addresses. K_1 is named the initial key in K, and is computed prior to the permutation loop.

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3516, pp. 788 – 791, 2005.

Many permutation loops are required in order to shuffle all keys in a group to their target addresses. ALR searches for K_1 sequentially, and preserves the property that all keys to the left of the current K_1 key are in their correct target addresses. MSL searches groups' information sequentially for the left most group, G_{Left}, which has at least one key possibly not in its target address. MSL uses the top key in G_{Left}, as K_1.

In [5], sections 4 and 5, digit size 6 was determined as appropriate for radix sorting algorithms. This is because of data cache friendliness. The importance of cache friendliness in radix sorting algorithms is emphasized in [5] and [4]. MSL also cuts to insertion sort for group sizes 20 or less, same as [4].

3 Switch Sort

Hybridized quicksort [3] implementation in this paper selects one from many pivot computations and is described in this section. Assume that we are interleaving the execution of a constant number, *K,* of divide and conquer algorithms whose worst cases are $f_1, f_2 , .. f_K$. The list of algorithms is denoted $AL = (A_1, A_2, .. A_K)$. When A_J in *AL* performance is degenerate, we interrupt A_J and switch execution to the next algorithm in the circular list *AL*. *AL* worst case is equal to $K * f_W$ provided that the following conditions hold. (1) $f_W = Min\ (f_1, f_2 , .. f_K)$. (2) We can determine that the current call to A_J is futile in constant time. (3) We can switch execution to the next algorithm in the circular list *AL* without loosing the processing done so far. If each A_J in *AL* executes a futile call, execution returns to A_W after circular calls to other algorithms in *AL*. A quicksort example is shown and described next.

```
SS(int A[], int A_Pivot, int l, int r, int min, int
max){
if ( r - l + 1 <= 10 ) insertionSort (Array, l, r );
} else {  Step 1: switch (Apply_Pivot) {
  case 0 : Pivot = max/2 + min/2 ; break;
  case 1: Pivot = Median_3 (A) ; break; }
  Step 2: Pos = partition (A, Pivot, l, r) ;
  Step 3.1:  Compute R%;
  Step 3.2: if (R < 0.05) A_Pivot = A_Pivot ^1;  // xor
  Step 4: Quicksort(A, A_Pivot, l, Pos, Pivot,
    max);
    Quicksort(A, A_Pivot, Pos+1, r,  min, Pivot);
 }
}
```

We measure balanced partitioning in quicksort to determine that the current call is futile. The partitioning ratio is defined as the size of the smaller partition divided by the size of the input group in quicksort. Let P% be the minimum acceptable partitioning ration, over all the algorithms in *AL*, equals 5% in Step 3.2. R% is the partitioning ration for the current quicksort call. When R% < P%, Step 3.2, partitioning is named degenerate or a failure. *AL* code above has only quicksort implementations, and a switch statement is used to decide which pivot computation to use, see Step 1 above. We call the algorithm switch sort (SS). Step 3.2 selects an alternative pivot computation for recursive calls. Max-Min average pivot computation in the first line in Step 1 is an adaptive implementation of taking the middle value of the input range in radix exchange [7]. Median of three quicksort passes down the actual lower partition max and the actual upper partition min. Radix exchange always divides the input range by half on recursive calls, independent of data. *AL* worst case is *O(2 * NlgN)*, where the worst case of radix exchange is *O(2 * NlgN)*.

4 Experimental Results

In Table 1, MSL run time is non-significantly sub-linear in experiments. The test data is uniform. The machine used for the displayed results is 3 GHz Pentium 4, 1GB RAM, 1MB level 2 cache, and 16 KB level 1 cache, with 400 MHz RAM speed. MSL sub-linear run time was confirmed on other Pentium 4 machines. In Table 1, add the sizes at columns headings to the sizes at each row to get the array size at a cell. Row 30M+ (30 millions+) and column +5M refer to the cell for the array size 35 millions.

In Table 1, MSL running time for array size 35 millions is 4000 milliseconds, and for array size 70 millions is 7875 milliseconds, 31 bits integers. In Table 1, the running time for array size 25 millions, is 4032 milliseconds, and for array size 50 millions, is 7735 milliseconds, for 63 bits integers.

Cutting to insertion sort is an important factor in MSL. On the other hand, we could not improve the running time of MSL by hybridizing MSL with switch sort. MSL and switch sort are compared against other algorithms in Table 2.

Table 1. MSL running times in milliseconds. Sizes are multiple of $M=10^6$

31Bits	+1M	+2M	+4M	+5M	+6M	+8M	+10M
N=0+	93	188	437	657	890	1234	1469
N=10M+	1563	1671	1875	1984	2094	2266	2500
N=20M+	2578	2678	2891	2984	3078	3281	3484
N=30M+	3594	3703	3906	4000	4125	4344	4547
N=40M+	4671	4766	5000	5079	5218	5390	5625
N=50M+	5782	5875	6062	6172	6313	6532	6734
N=60M+	6890	6953	7172	7313	7453	7656	7875
63Bits	+1M	+2M	+4M	+5M	+6M	+8M	+10M
N=0+	141	250	594	843	1125	1531	1859
N=10M+	2016	2157	2485	2594	2765	3016	3281
N=20M+	3469	3578	3859	4032	4141	4438	4735
N=30M+	4860	5016	5328	5469	5625	5891	6203
N=40M+	6375	6500	6797	6953	7125	7406	7735

LSD, digit size 8 (LSD8) is faster than LSD with digit size 16, LSD16, and other digit sizes, on the test machine. LSD processes the total keys bits. MSL processes only the distinguishing prefixes, but is recursive (section 1). In Table 2, MSL has half the run time of LSD8 for 63 bits data. In addition, MSL is better than LSD8 for larger 31 bits arrays. See size 16 and 32 millions as well as MSL sub-linear run time in Table 1. Switch sort (SS), is faster than LSD16, 63 bits longs data. Switch sort is also faster than the two algorithms, which Switch sort alternates, quicksort and Max-Min Average (MMA). Java built in tuned quicksort (JS), which is a tuned implementation of [2], is used in Table 2, instead of our own slower median of three quicksort.

Table 2. MSL running times in milliseconds. Sizes are multiple of $M=10^6$

31Bits	1/2M	1M	2M	4M	8M	16M	32M
MSL	47	93	188	437	1234	2094	3703
LSD8	47	109	234	454	938	1875	3859
LSD16	94	234	500	1031	2047	4250	8656
JS	109	234	516	1062	2219	4640	9672
SS	109	250	500	1031	2141	4500	9546
MMA	109	234	500	1031	2172	4516	9438
63Bits	1/2M	1M	2M	4M	8M	16M	32M
MSL	62	141	250	594	1531	2765	5016
LSD8	172	344	672	1328	2719	5563	10953
LSD16	250	516	1015	2031	4563	8609	18891
JS	156	329	672	1422	2969	6203	12922
SS	140	313	640	1344	2781	5860	12203
MMA	141	312	641	1360	2829	5906	12359

5 Conclusion and Future Work

MSL is a sub-linear in-place radix-sorting algorithm, for uniform data. Switch sort is a non-quadratic implementation of quicksort. Future work includes low run time algorithms and models for sorting as well as for other problems.

References

1. Al-Badarneh Amer, El-Aker Fouad: Efficient In-Place Radix Sorting, Informatica, 15 (3), 2004, pp. 295-302.
2. J. L. Bentley, and M. D. McIlroy: Engineering a Sort Function, Software-Practice and Experience, 23 (1), 1993, pp. 1249-1265.
3. F. El-Aker, and A. Al-Badarneh: MSL: An Efficient Adaptive In-place Radix Sorting Algorithm, ICCS, Part II, 2004, pp. 606-609.
4. Maus, A.: ARL: A Faster In-place, Cache Friendly Sorting Algorithm, Norsk Informatikkonferranse, NIK'2002, 2002, pp. 85-95.
5. N. Rahman and R. Raman: Adapting radix sort to the memory hierarchy, Proc. 2nd Workshop on Algorithm Engineering and Experiments, ALENEX, 2000.
6. Sedgewick, R.: Algorithms in Java, Parts 1-4, 3rd Ed., Addison-Wesley, 2003.