Program Structures and Algorithms (INFO6205)

Fall 2021 - Final Project

Literature Survey on Radix Sort

Akhilanand Sirra - 002197798, Pramithi Jagdish - 002192816, Venkata Srinivas Kompally - 002137855

I. Abstract:

Radix sorting methods have excellent asymptotic performance on string data, for which comparison is not a unit-time operation. Attractive for use in significant byte-addressable memories, these methods have nevertheless long been eclipsed by more easily programmed algorithms. In this paper, we describe our implementation of MSD radix sort and compare it with three variations of the algorithm. We also analyze the performance of Huskysort, a new algorithm proposed by Prof Hillyard that combines dual-pivot quicksort and Timsort and can run significantly faster than either algorithm alone for the types of objects which are expensive to compare.

II. Introduction:

As the amount of data continues to grow, efficient computing algorithms are crucial for processing it. Despite appearing to be a straightforward and familiar topic, sorting has long been one of the most studied research areas in computing. For sorting strings, you can't beat radix sort. The idea is simple, deal the strings into piles by their first letters. One pile gets all the empty strings. The next gets all the strings that begin with A-; another gets B- strings, and so on. Split these piles recursively on second and further letters until the strings end. When there are no more piles to split, pick up all the piles in order. The strings are sorted.

Radix sorting 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. In the following section we will look at the brief background about Radix sort, followed by a deeper dive into various types within them, followed by the Algorithm, implementation, and analysis of the MSD

Radix sort used in our project and a final conclusion.

III. Background:

Radix sort dates back as far as 1887 to the work of Herman Hollerith on tabulating machines. Radix sorting algorithms came into common use as a way to sort punched cards as early as 1923. The first memory-efficient computer algorithm was developed in 1954 at MIT by Harold H. Seward. Computerized radix sorts had previously been dismissed as impractical because of the perceived need for variable allocation of buckets of unknown size. Seward's innovation was to use a linear scan to determine the required bucket sizes and offsets beforehand, allowing for a single static allocation of auxiliary memory. The linear scan is closely related to Seward's other algorithm — counting sort.

IV. MSD Radix Exchange sort:

For a binary alphabet, radix sorting specializes to the simple method of radix exchange. It will Split the strings into three piles: the empty strings, those that begin

with 0, and those that begin with 1. For classical radix exchange assume further that the strings are all the same length. Then there is no pile for empty strings and splitting can be done as in quicksort, with a bit test instead of quicksort's comparison to decide which pile a string belongs in.

Program 1.1 sorts the part of array A that runs from A[lo] to A [hi - 1]. All the strings in this range have the same å-bit prefix, say x-. The function split moves strings with prefix -r0- to the beginning of the array, from A[0] through A[mid - 1], and strings with prefix x1- to the end, from A[mid] through A [hi – 1].

Program 1.1.

```
RadixExchange(A, lo, hi, b) =
        if hi - lo ≤ 1
                then return
        if b ≥ length(A[lo])
                then return
        mid := Split(A, lo, hi, b)
        RadixExchange(A, lo, mid, b+1)
        RadixExchange(A, mid, hi, b+1)
```

After enjoying a brief popularity, radix exchange was upstaged by quicksort. Not only was radix exchange usually slower than quicksort, it was not good for programming in Fortran or Algol, which hid the bits that it depends on. Quicksort became known as simple and nearly unbeatable; radix sorting disappeared from textbooks.

Nevertheless, radix exchange cannot be bettered in respect to amount of data looked at. Quicksort doesn't even come close. Quicksort on average performs O (log n) comparisons per string and inspects O (log n) bits per comparison. By this measure the expected running time for quicksort is Ω (n $\log^2 n$), for radix exchange only Ω (n log n). Worse, quicksort can "go quadratic," and take time Ω(n^2logn) on unfortunate inputs. This is not just an abstract possibility. Production quicksort routines have gone quadratic on perfectly reasonable inputs.

In other ways, quicksort and radix exchange are quite alike. They both sort in place, using little extra space. Both need a recursion stack, which we expect to grow to size O (log n). In either method, if the strings are long or have different lengths, it is well to address strings through uniform descriptors and to sort by rearranging small descriptors instead of big strings.

V. List based radix sort:

For most modern machines, the 8-bit byte is a natural radix, which should overcome the bit-picking slowness of radix exchange. A byte radix makes for 256- or 257 -way splitting, depending on how the ends of strings are determined. This raises the problem of managing space for so many piles of unknown size at each level of recursion. An array of linked lists is an obvious data structure. Dealing to the piles is easy; just index into the array. Picking up the sorted piles and getting them hooked together into a single list is a bit tricky, but takes little code.

Program 2.1. Simple list-based sort.

```
typedef struct list {
        struct list     *next;
        unsigned char   *data;
} list;
list *rsort(list *a, int b, list *sequel)
{
#define ended(a, b)     b>0 && a->data[b-1]==0
#define append(s, a)    tmp=a; while(tmp->next) tmp=tmp->next; tmp->next=s
#define deal(a, p)      tmp = a->next, a->next = p, p = a, a = tmp
        list            *pile[256], *tmp;
        int             i;
        if(a == 0)
                return sequel;
        if(ended(a, b)) {                       /* pile finished */
                append(sequel, a);
                return a;
        }
        for(i = 256; --i >= 0; )                /* split */
                pile[i] = 0;
        while(a)
                deal(a, pile[a->data[b]]);
        for(i = 256; --i >= 0; )                /* recur on each pile */
                sequel = rsort(pile[i], b+1, sequel);
        return sequel;
}
```

The input variables are:

a - linked list of null-terminated strings.

b - the offset of the byte to split on; the strings agree in all earlier bytes.

sequel - a sorted linked list of strings that compare greater than the strings in list a.

Three in-line functions are coded as macros:

ended (a, b) - tells whether byte position b is just beyond the null byte at the end of string a.

append (s, a) - appends list s to the last element of non-empty list ø.

deal (a, p) - removes the first string from list a and deals it to pile p.

Program 2.1 works slowly. Empty piles are the root of the trouble. Except possibly at the first level or two of recursion, most piles will be empty. The cost of clearing and recursively "sorting" as many as 255 empty piles for each byte of data is overwhelming. Some easy improvements will speed things up but still would not sort stably. Because piles are built by pushing records on the front of lists, the order of equal-keyed records is reversed at each deal. To stabilize the sort, we can reverse each backward pile as we append to it. Alternatively, we can maintain the piles in forward order by keeping track of the head and tail of each pile. Program A does the latter. Sorting times differ negligibly among forward/reverse and stable/unstable versions.

VI. Two-Array Radix Sort

Imagine the strings as an array as for radix exchange. In basic radix exchange, the two piles reside in known positions at the bottom and top of the array. For larger radixes, we can calculate the positions of the piles in an extra pass by counting how many strings belong in each pile. Knowing the sizes of the piles, we don't need linked lists.

Program 3.1 gets the strings home by moving them as a block to the auxiliary array ta, and then moving each element back to its proper place. The upper ends of the places are precomputed in array pile. Elements are moved stably; equal elements retain the order they had in the input. The stack is managed explicitly; the stack has a third field to hold the length of each subarray. Program 3.1 is amenable to most of the improvements listed in section 2; they appear in Program B. In addition, the piles are independent and need not be handled in order. Nor is it necessary to record the places of empty piles. These observations are embodied in the "find places" step of Program B.

Program 3.1. Simple two-array sort.

```
typedef unsigned char *string;
void rsort(string *a, int n)
{
#define push(a, n, b)   sp->sa = a, sp->sn = n, (sp++)->sb = b
#define pop(a, n, b)    a = (--sp)->sa, n = sp->sn, b = sp->sb
#define stackempty()    (sp <= stack)
#define splittable(c)   c > 0 && count[c] > 1
        struct { string *sa; int sn, sb; } stack[SIZE], *sp = stack;
        string          *pile[256], *ak, *ta;
        static int      count[256];
        int             i, b;
        ta = malloc(n*sizeof(string));

        push(a, n, 0);
        while (!stackempty()) {
                pop(a, n, b);
                for(i = n; --i >= 0; )                  /* tally */
                        count[a[i][b]]++;
                for(ak = a, i = 0; i < 256; i++) {      /* find places */
                        if(splittable(i))
                                push(ak, count[i], b+1);
                        pile[i] = ak += count[i];
                        count[i] = 0;
                }
                for(i = n; --i >= 0; )                  /* move to temp */
                        ta[i] = a[i];
                for(i = n; --i >= 0; )                  /* move home */
                        *--pile[ta[i][b]] = ta[i];
        }
        free(ta);
}
```

As we mentioned earlier, radix sorting works best for large arrays. As the piles become smaller, we may benefit from other low-overhead sorting methods such as insertion sort or Shell sort. Arrays are more natural for library interfaces than lists, but a two-array sort dilutes this advantage with O(n) working space and dynamic allocation of storage. Our next variant solves this problem.

VII. American flag Radix Sort

Instead of copying data to an auxiliary array and back, we can permute the data in place. American flag sort differs from the two-array sort mainly in its final phase. The "permute home" phase in Program 4.1 accomplishes the effects of the "move to temp" and "move home" of Program 3.1. This phase fills piles from the top, making room by cyclically displacing elements from pile to pile.

Program 4.1. In-place permutation to substitute in Program 3.1.

```
#define swap(p, q, r) r = p, p = q, q = r
string          r, t;
int             k, c;

for(k = 0; k < n; ) {
        r = a[k];                       /* Figure 4.2 */
        for(;;) {
                c = r[b];               /* Figure 4.3 */
                if(--pile[c] <= a+k)
                        break;
                swap(*pile[c], r, t);
        }
        a[k] = r;                       /* Figure 4.4 */
        k += count[c];
        count[c] = 0;
}
```

Let a[k] be the first element of the first pile not yet known to be completely in place. Displace this element out of line to r (Figure 4.2). Let c be the number of the pile the displaced element belongs to. Find in the c-pile the next unfilled location, just below pile [c] (Figure 4.3). This location is the home of the displaced element. Swap the displaced element home after updating pile[c] to account for it. Repeat the operation of Figure 4.3 on the newly displaced element, following a cycle of the permutation until finally the home of the displaced element is where the cycle started, at a[k]. Move the displaced element to a[k]. Its pile, the current c- pile, is now filled. Skip to the beginning of the next pile by incrementing ft. (Values in the count array must be retained from the "find places" phase.) Clear the count of the just-completed pile, and begin another permutation cycle. It is easy to check that the code works right when a * k : pile[c] initially, that is, when the pile is already in place. When all piles but one are in place, the last pile must necessarily be in place, too.

VIII. The MSL Algorithm

The MSL (Map shuffle loop) algorithm processes bits left to right recursively, it 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 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)

Radix computing in MSL is similar to adaptive MSD (AMSD). AMSD radix setting is simple to understand and is modified to work on bits instead of digits. The following modifications are applied to the implementation of MSL:
(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

IX. Conclusion:

Although radix sorts have unbeatable asymptotic performance, they present problems for practical implementation:
(1) managing scattered piles of unpredictable size and
(2) handling complex keys.
We have shown that the piles can be handled comfortably. Although it costs memory roughly proportional to the volume of keys, this strategy is simple and effective for sorting records.

List-based radix sort is faster than pure array-based radix sorts. The speed disparity is overcome by hybrid routines that divert from radix sorting to simple comparison-based sorting for small arrays. The natural array-argument interface makes them attractive for library purposes. Both list-based and two-array sorts entail O(n) space overhead. That overhead shrinks to O(log n) in American flag sort, which, like quicksort, trades off stability for space

efficiency. We recommend American flag sort as an all-round algorithm for sorting strings.

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.

X. References:

1. https://link.springer.com/chapter/10.1007/978-3-540-89097-3_3
2. https://arxiv.org/pdf/2012.00866.pdf
3. https://link.springer.com/content/pdf/10.1007%2F978-3-540-24687-9_83.pdf
4. https://en.wikipedia.org/wiki/Radix_sort