MSD Radix sort for natural language simplified Chinese

Varun Rajamudi(002108570)
Diksha Bhatia(002175782)
Shreekara SS (001545668)
Department of Information Systems, Northeastern University
360 Huntington Ave, Boston, MA 02115

ABSTRACT
On string data, where comparison is not a unit-time process, radix sorting techniques exhibit outstanding asymptotic performance. These approaches, while appealing for application in huge byte-addressable memories, have since been surpassed by more simply programmed algorithms. We show and analyze some innovative string sorting optimization and implementation strategies. We focus on Forward radix sort, a recently disclosed radix sorting algorithm with provably good worst-case behavior. Radix sorting is significantly faster (sometimes more than twice as fast) than comparison-based sorting algorithms, according to our findings. Even with tiny input sequences, this is true. We also show that a radix sort with good worst-case running time may be implemented without affecting average-case efficiency.

1. Introduction:

Radix sorting is a straightforward and very effective sorting algorithm that has gotten much too little attention. A widespread fallacy is that a radix sorting algorithm must either check all of the input characters or use an excessive amount of time or space. However, as various academics have demonstrated, efficient systems are attainable with careful implementation. In fact, for average data, radix sorting can be constructed to run much quicker than strictly comparison-based sorting. We show in this essay that radix sorting can be implemented in a way that ensures good worst-case behavior. An algorithm is used to accomplish this. We're sorting simplified Chinese characters with several MSD radix sort, LSD radix, Quick sort dual pivot, and Pure husky sort algorithms, scaling from 250k to 4M chinese words, to see which algorithms work best for natural language.
[1]Simplified Chinese characters are standardized Chinese characters used in Mainland China, Malaysia and Singapore, as prescribed by the Table of General Standard Chinese Characters. Along with traditional Chinese characters, they are one of the two standard character sets of the contemporary Chinese written language. The government of the People's Republic of China in mainland China has promoted them for use in printing since the 1950s and 1960s to encourage literacy. In the digital age, Pinyin has become exceedingly useful, as it is the most popular and common way to type out Chinese characters on a typical keyboard. Touch-screen devices allow you to draw out the character, which is often unreliable (depending on how bad your handwriting is) and more time consuming Pinyin is the Romanization of Chinese characters based on their pronunciation. In Mandarin Chinese, the phrase “Pin Yin” literally translates into “spell sound.” In other words, spelling out Chinese phrases with letters from the English alphabet.

Now , We are using the knowledge of Pinyin to sort the Chinese characters.

1.1. LSD Radix Sort:
Stability: Yes
Time Complexity: O(wn) for all the cases.
Space Complexity: O(n+r) for auxiliary

1.2 Dual-Pivot QuickSort:
Time Complexity:

- Best Case: Ω(n log n)
- Worst Case: Θ(n log n)
- Average Case: O(n^2)

Space complexity: The Dual Pivot Quicksort algorithm makes recursive calls which makes its space complexity O(n).

1.3 Pure HuskySort:
Time Complexity: O(n log n)
Space complexity: O(n)

2. Implementing MSD Radix sort for Chinese words
One of the greatest sorting methods for strings or characters is MSD radix sort. We use a Pinyin translation of Chinese characters whose Unicodes are unknown since we are categorizing Chinese characters with unknown Unicodes. So we start by converting Chinese words to Pinyin words, sorting those Pinyin words using MSD radix sort, and then converting back to Chinese words using HashMap.
We need to adapt the methods for partitioning an array that we've seen in quicksort implementations to implement MSD radix sort. When there are only two or three divisions, these methods, which are based on pointers that start from the two ends of the array and meet in the middle, work well, but they do not immediately generalize. Fortunately, the key-indexed counting method for sorting files with minimal key values is ideal for our needs. We utilize a table of counts and an auxiliary array, counting the number of occurrences of each leading digit value on the first run through the array. These numbers indicate where the partitions will be placed. The counts are then used to relocate items to the proper location in the auxiliary array on a second run through the array.
Quicksort's recursive structure is generalized by it. To avoid excessive recursion depth, should we do the largest of the subfiles last? The depth of recursion is restricted by the length of the keys, thus probably not. Should we use a simple method like insertion sort to sort small subfiles? Definitely, because there are so many of them.
To partition the array, we utilize an auxiliary array with the same size as the array to be sorted. Alternatively, we might utilize key-indexed counting in-place. We must pay special attention to space because recursive calls may take a lot of it for local variables.
Because we are usually manipulating references to such data, extra capacity for the auxiliary array is not a serious worry in many practical implementations of radix sorting that include large keys and records. As a result, the extra space is only for reordering references and is minor in comparison to the space required for the keys and records (although still not insignificant). If space is limited and performance is critical (as it is when using radix sorts), recursive argument switchery, similar to merge sort, can be used to avoid the time required for the array copy.

3. Performance

3.1 Randomly Shuffled Array Inputs:
MSD string sort accesses enough characters to distinguish among the keys,

and the running time is sublinear in the number of characters in the data.

3.2 Non Random Sorted Array Inputs:
The string sort will take sublinear time and examine more characters than it would take in a randomly stuffed array. It would take linear time, in case of multiple equal keys.

3.3 Worst case Scenario:
All keys are equal. The same problem arises when large numbers of keys have long common prefixes, a situation often found in applications.

4. Disadvantages of MSD string sort

- Accesses memory randomly, (cache inefficient)
- Inner loop has a lot of instructions
- Extra space for count[]
- Extra space for aux[]

5. Improving MSD radix sort

5.1 In-Place MSD Radix Sort

[2][PARADIS: A PARALLEL IN-PLACE RADIX SORT ALGORITHM, ROLLAND HE] introduced improvements in extra memory usage, in order to reduce the amount of extra memory needed from O(n) to O(1), we can no longer rely on a separate buckets data structure. Instead, we need to swap elements in place. In order to successfully permute the elements into their correct positions, we modify our MSD radix sort algorithm. Instead of explicitly bucketizing each element, we build a histogram to count the number of elements that belong to each bucket – this will require a single pass through the data.

5.2 Improving cache usage

For MSD RadixSort we used the two array implementation. A code example is here:

```
int[] count = new int[radix+2];

for (int i = lo; i < hi; i++)
    count[charAt(a[i], d) + 2]++;

for (int r = 0; r < radix + 1; r++)
    count[r + 1] += count[r];

for (int i = lo; i < hi; i++)
    aux[count[charAt(a[i], d) + 1]++] = a[i];

if (hi - lo >= 0) System.arraycopy(aux, 0, a, lo, hi - lo);

for (int r = 0; r < radix; r++){
    sort(a, lo + count[r], lo + count[r+1] - 1, d+1);
}
```

Another way of improving the cache is as below:
[7][Cache efficient radix sort for string sorting Waihong Ng*, Katsuhiko Kakehi] introduced The improvement is achieved by uniquely associating a small block of main memory called key buffer to each key and temporarily storing a portion of each key into its corresponding key buffer. Key buffers are equal-sized and can hold maximum *z* characters each and extra spaces are required for storing the key buffers. The value of *z* can be tuned for targeting a particular class of inputs for achieving high performance in sorting that class of inputs.

6. CONCLUSION

In conclusion, we evaluated the performance of multiple sorting algorithms on a dataset of Chinese names. The results reveal that, for high quantities of input, RadixSort algorithms outperform comparison based algorithms, with the exception of Quicksort Dual pivot. To avoid cache misses, we can modify the MSD RadixSort by adding a new array.

6.References

[1]https://en.wikipedia.org/wiki/Mainland_China
[2]PARADIS: A PARALLEL IN-PLACE RADIX SORT ALGORITHM ROLLAND HE rhe@stanford.edu https://stanford.edu/~rezab/classes/cme323/S16/projects_reports/he.pdf
[3]10.3 MSD Radix Sort | Algorithms in Java, Parts 1-4 (3rd Edition) (Pts.1-4)
[4]Implementing Radixsort Arne Andersson Department of Computer Science, Lund University Box 118, S-221 00 Lund, Sweden e-mail: arne@dna.lth.se Stefan Nilsson Department of Computer Science, Helsinki University of Technology P. O. Box 1100, FIN-02015 HUT, Finland e-mail: Stefan.Nilsson@hut.fi
[5]https://static.usenix.org/publications/compsystems/1993/win_mcilroy.pdf
[6] Husky Sort Reference: https://arxiv.org/pdf/2012.00866v1.pdf
[7]Cache efficient radix sort for string sorting
Waihong Ng*, Katsuhiko Kakehi
https://waseda.pure.elsevier.com/en/publications/cache-efficient-radix-sort-for-string-sorting