The Spreadsort High-Performance General-Case Sorting Algorithm

Steven J. Ross
P. O. Box 513
Clinton, NY 13323

Abstract

A high-performance nearly in-place general-case sorting algorithm named SpreadSort is demonstrated. It is approximately 4X as fast as Quicksort in normal cases, and up to 18X as fast with distributions of limited variation (much like Bucketsort). The technique is mixed distributional and comparison-based, merging many of the advantages of both techniques. Spreadsort can operate recursively, but is O(n) for continuous integrable functions, and has better than O(nlog(n)) worst-case performance when used with distributions where the keys have finite length, so recursion past the second iteration is rare. This algorithm can be modified to be in-place with a modest speed loss.

Keywords: Sorting, Quicksort, In-place, Algorithm, Bucketsort, Radixsort

1. Introduction

"Sorting represents one of the most basic operations in computer science"[1] as it is the ordering of a set in ascending or descending order, an operation with broad applicability. Along with significant theoretical interest, it has enormous practical application, using approximately 20 percent of computing resources today[2]. It is time-critical for applications varying from Databases[3] to Data Compression and web searches[4]. Prof. Knuth argues that $\theta(n\log(n))$ performance is the best to be expected of general-case sorting, depending on comparison-based methods in his proof[5].

Knuth was wrong about sorting. His proof is correct, based on his assumptions, but his assumptions and therefore conclusions are wrong. A sorting algorithm does not need to be purely comparison-based to perform well in all cases. Additionally, a comparison is not always a constant-time operation. The Spreadsort sorting algorithm described here will work on any problem with a total ordering (including ones with duplicates), which Knuth has stated as a requirement to consider the problem sorting[5]. It will also perform faster, with better than or equal average case and worst case computational order as opposed to comparison-based sorting for all distributions. In this paper, the weaknesses in Knuth's assumptions are identified and discussed. Then, the Spreadsort algorithm is described. Finally, an analysis is made of the relative performance of Spreadsort and comparison-based sorting, including such techniques as Quicksort and Mergesort.

1.1 Sorting Assumptions 1: Comparison as a Constant Time Operation

One implicit assumption that is common in the field of sorting is that a comparison is a constant-time operation. This assumption is made in [5], in the discussion of Shellsort, algorithm D, when $O(n\log(n)^2)$ comparisons are considered equivalent to $O(n\log(n)^2)$ running time. Knuth avoids stating this assumption explicitly[5]. It is an assumption that is used when $O(n\log(n)^2)$ comparisons are considered equivalent to $O(n\log(n)^2)$ operations during a sort, where n is the number of items being sorted.

A comparison does not take constant time in the worst case. A comparison is an attempt to determine which of two variables is greater, or if they are equal. It will not stop until one of these results is determined. There is only a finite length of the keys that can be compared at one time, without shifting

memory to load in another section. On a modern microprocessor, this usually corresponds to the bus width. Thus a 32-bit processor can only compare 4 bytes in one operation; it needs at least one more operation to compare 8 bytes if the first four bytes are equal and the end hasn't been reached. This comes down to a worst-case performance of $O\left(\frac{b}{w}\right)$ where b is the length in bits of the key and w is the width in bits of the maximum section the processor can handle at a time. In the case of long strings that differ at a random point in their length, this worst-case performance will be achieved. This means that a comparison-based algorithm that takes O(n log(n)) comparisons takes $O\left(\frac{nb\log(n)}{w}\right)$ time.

1.2 Sorting Assumptions 2: General-case sorting must be Comparison-based

"Now if we set N = n!, we get a lower bound for the average number of comparisons in any sorting scheme. Asymptotically speaking, this lower bound is $\lg n! + O(1) = n\lg n - \frac{n}{\ln 2} + O(\log(n))$." [5] pg. 193.

The above quotation assumes that a purely comparison-based algorithm obtains the minimum number of comparisons. It may seem necessary to assume that an algorithm be purely comparison based to evaluate its relative performance, but if it uses a small constant number of O(n) operations, it does not add to the computational order of the comparisons. Spreadsort can obtain significantly superior average-case performance by splitting up a problem for easier comparison sorting.

"Studies of the inherent complexity of sorting have usually been directed towards minimizing the number of times we make comparisons between keys while sorting n items." [5] pg. 181.

Partially due to the above logic, it is common to assume that algorithms having any distributional basis are not useful in the general case[5]. While this is clearly true of Bucketsort, where the number of memory locations goes up exponentially with the length of the key in bytes, that does not make the assumption true in general.

For example, Radixsort, takes O(nb) time, but can be used on any distribution. If log(n) is comparable in size to the bit width w, then Radixsort will have comparable worst-case performance to a comparison-based technique, and it can be even faster for very large n. Radixsort is a high-performance general-case distributional sorting algorithm that is usually slower than the best know comparison-based algorithm Quicksort[6], but not badly so[7]. It is thus a general-case distributional sorting algorithm that refutes the common assumption. Spreadsort, described below, is a mixed distributional and comparison-based algorithm that has high average-case performance and excellent worst-case performance. The code used in testing exhibits is fully generalized with a user-defined value() method, much like Quicksort's compare() method. The value() method must take in an object and return its corresponding integer value. For large byte widths, a position must also be passed in, and there must be a method to return the length in bytes of an object.

2 SpreadSort

Most modern comparison-based algorithms are based on some variant of the divide and conquer technique, where the list is recursively split in half until each of the pieces is small enough to be quickly sorted, and then the list is reformed fully sorted. Mergesort implements this process in reverse, but the same logarithmic progression is apparent. One question rarely asked about these splittings is: why divide by two, and not by 3, 4, or some much larger number? What is the optimal number of pieces to split into at each step? It is easiest to implement splitting by factors of two, but that doesn't necessarily correlate with the best performance. Spreadsort uses the theory that the optimum number of bins is a fraction of *n*. This

optimum number is determined by minimizing the sum of the average bin overhead time and the average bin subsorting time, both of these times being functions of the number of bins. This bin count has been empirically found to be in the range of $n/4$ to $n/8$ on most systems.

The Spreadsort algorithm is a different divide and conquer algorithm, dividing by a fraction of n instead of by 2. It is a recursive algorithm, as with other divide and conquer algorithms. The recursive part begins by calculating the maximum and minimum values of the distribution (a quick O(n) task), then evenly splitting up the distribution of possible key values (m) in between these values into (n/c) bins, where c is a small positive integer. Note that a similar technique can be used on keys of indeterminate length, if the key is assumed to be followed by an infinite succession of minimum values. Each item's key value is then divided by a previously calculated factor (m/(n/c)) to decide which bin to put it in. n items are thus mapped to (n/c) bins in an O(n) operation. By doing this, the distribution size for each bin is cut by (n/c), and the average number of items per bin is c. Then a series of tests is applied: If the number of bins is greater than or equal to the range of keys, then the data is already sorted (see Bucketsort), and no bin-by-bin tests are necessary. If the bins aren't already fully sorted, then the comparison below is checked:

$$\frac{(\log(n_2))^2}{2} < \log(m_2)$$

If this comparison is true, then the worst case for the recursive application of Spreadsort (assuming constant-time comparisons for the moment) is worse than the normal case for Quicksort, so Quicksort is selected for the bin. Otherwise, recursive application of Spreadsort can continue cutting up the problem both in terms of key size and concentration. This recursive application has a worst case performance that can be calculated using the assumption of the branching tree structure, with a division into two equal-sized pieces per recursive operation:

x is the number of Spreadsort recursive operations necessary to sort a list in worst case.
x is the smallest integer such that

$$\frac{m}{\left(\frac{n}{c2^0}\cdot\frac{n}{c2^1}\cdot\ldots\cdot\frac{n}{c2^x}\right)} \le 1$$

Taking the case where the sides are equal and multiplying the series, x can be solved for.

$$m = \frac{n^x}{c^x 2^{(x(x-1))}}$$

$$\log_2(m) = x\log_2(n) - x(x-1)\log_2(2) - x\log_2(c)$$

$$-x^2 + x(\log_2(n) - 1 - \log_2(c)) - \log_2(m) = 0$$

$$-x^2 + x\left(\log_2\left(\frac{n}{c}\right) - 1\right) - \log_2(m) = 0$$

$$x = \frac{-\left(\log_2\left(\frac{n}{c}\right) - 1\right) \pm \sqrt{\left(\log_2\left(\frac{n}{c}\right) - 1\right)^2 - 4\log_2(m)}}{-2}$$

$$x = \frac{\left(\log_2\left(\frac{n}{c}\right) - 1\right) - \sqrt{\left(\log_2\left(\frac{n}{c}\right) - 1\right)^2 - 4\log_2(m)}}{2}$$

This requires:

$$\left(\log_2\left(\frac{n}{c}\right) - 1\right)^2 \ge 4\log_2(m)$$

It is notable that if this condition is true, x is always less than $\log_2(n)$, as by inspection:

$$\frac{\left(\log_2\left(\frac{n}{c}\right) - 1\right)}{2} < \log_2(n)$$

As long as $\left(\log_2\left(\frac{n}{c}\right) - 1\right)^2 \ge 4\log_2(m)$ is true, then the data is fastest sorted by Spreadsort. When it is not true, Quicksort is used. Another algorithm, such as Mergesort can be used instead of Quicksort if strict O(nlog(n)) comparisons performance is considered necessary. The decision stop because the

data is already sorted, continue with Spreadsort, or stop and use Quicksort is made with every bin created. This gives Spreadsort the same absolute worst-case performance as the comparison-based algorithm it is used with. Distributions where $\left(\log_2\left(\frac{n}{c}\right)-1\right)^2 > 4\log_2(m)$, which are relatively common, have a better worst-case performance than O(nlog(n)) comparisons:

$$x = \frac{\left(\log_2\left(\frac{n}{c}\right)-1\right) - \sqrt{\left(\log_2\left(\frac{n}{c}\right)-1\right)^2 - 4\log_2(m)}}{2}$$

$$x = \left(\log_2\left(\frac{n}{c}\right)-1\right)\left(\frac{1-\sqrt{1-\frac{4\log_2(m)}{\left(\log_2\left(\frac{n}{c}\right)-1\right)^2}}}{2}\right)$$

$$x \approx \frac{\log_2(m)}{\left(\log_2\left(\frac{n}{c}\right)-1\right)}$$

Using the approximation that the square root of 1 minus a small value is 1 minus half that value. It is notable that if the small value equals 1, the below result is still correct. The number of operations is n plus n times the number of recursive calls x:

$$O(n + nx)$$

$$O(n + n\log_n(m))$$

The major advantage of Spreadsort is that each separation takes only O(*n*) time and splits the problem into θ(*n*) pieces. If the distribution is random, then the bins can be sorted separately with a net O(*n*) time, assuming a generally small constant number of items , c, per bin. If the distribution is Gaussian, then it will actually operate much like a simple random distribution for this case, as the tails of a Gaussian taper off rapidly, limiting the total size of the distribution being cut up. For the more spiky distributions, the large spikes can be recursively cut down in key size to the point that they are fully distributionally sorted. Additionally, unless the spike has a discontinuous shape, the first iteration will turn each spike into its own bin, which is then sorted normally. With distributions of signification size (1MB+), no recursive calls after the second are usually necessary on high peaks.

Worst case performance occurs for Spreadsort when the key size is large and the distribution branches with each application of the sort into just a few branches (two, worst case), and each of these branches are at the edges of the previous distribution bin. This will limit each recursive distribution size cut, forcing many recursive calls. With a large enough key size and relatively small *n*, this type of problem will force Spreadsort to fall back on an O(*n*log(*n*)) comparisons technique.

3 Other Algorithms:

A description of the main types of sorting algorithms currently in use can be found in reference 5. Short descriptions of the primary algorithms pertinent to Spreadsort are provided in Appendix A. These include Quicksort, Mergesort, Bucketsort, and Radixsort. On an Altivec processor, only Bucketsort was capable of outperforming Spreadsort under any circumstances, and that being when the range of possible keys is smaller than the number of items being sorted. Mergesort is capable of operations on serial media, but there is a high-performance version of Spreadsort that also works with serial media and is much faster than Mergesort.

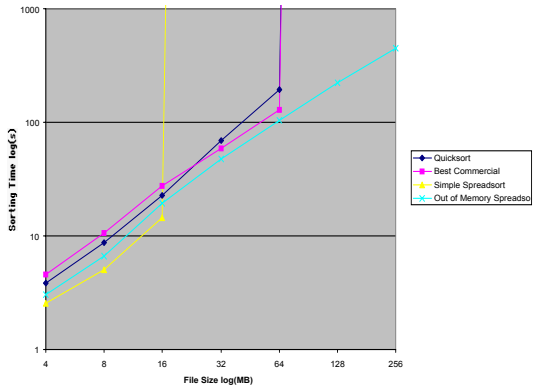
4 Performance Comparison:

Spreadsort reduces most sorting problems to one or two distribution-based steps, followed by comparison-based sorting of small subbins, taking about 4 comparisons per item. In contrast a comparison-based algorithm will take 20 to 30 operations per item for a list of a few million items, but only the first few operations will be in main memory and the rest will move onto the cache. The net

effect of this difference in number of operations is to give a factor of four to six times speed advantage to the Spreadsort algorithm over Quicksort, for large key ranges. This advantage is maximized for distributions that consist of only evenly spread (random) numbers and large, thin clumps. It is smaller for distributions with more mixed groupings of medium-sized clumps and empty sections. In cases where the range of key values is comparable to or less than the number of items being sorted, the Spreadsort speed ranges between seven and eighteen times as fast as Quicksort.

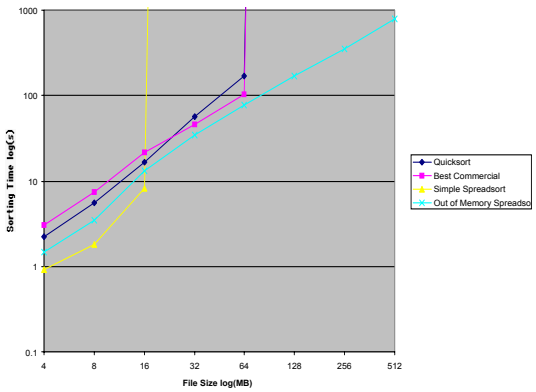
The latest version of Spreadsort uses about 20% more memory than Quicksort, to hold bin information. By increasing the bin size from 8-16 (a good average size) to 32-64, and more complex rearranging of elements, the speed degrades about 10%, but the memory usage becomes less than 1.05n, only a minor increase over Quicksort. If truly in-place sorting is desired, using bin counts that are between the 3/4 and 9/10 power of the number of items works well. The test shown in Figure 1 for semirandom data sets of variable size illustrates this speed

Figure 1: SpreadSort Speed Comparison



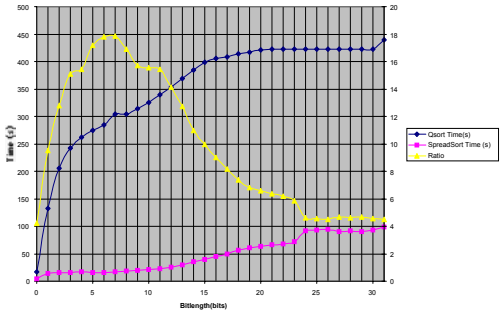
performance advantage with tests that include read, sort, and write time on a Pentium II 266MHz running LINUX with 64MB of RAM. The spikes leading upwards are where the computer ran out of memory, noticeably early with the simple Spreadsort due to its memory requirements. Many of the algorithms slow down significantly on virtual memory right before running out of memory. It should be noted that the file I/O time for SpreadSort and Quicksort in these examples is an equal .4s/MB. In the interest of using exact data, this compensation was not included in Figure 1, but makes the speed improvement for Spreadsort more visible in Figure 2. It should be noted that the file I/O time for the commercial application being tested against is unknown, but "sorting" time is nearly identical for already sorted data, and that the out of memory algorithm is dependent on hard drive speed.

Figure 2: Algorithm Comparison Compensated for File I/O



A limited ANSI C version of SpreadSort is up to 5.2X as fast as Quicksort for sorting on an Altivec, but uses a little more than twice the memory necessary to hold the data. Figure 3 shows the results for the fully-optimized nearly in-place (up to 1.2n memory) SpreadSort operating on an Altivec, as the key range varies. The transition from a bitlength of 24 to 23 is due to the onset of bucketsorting. The transition at bitlength 31 is probably an anomaly due to the start of the test. It is notable that on the Altivec Spreadsort never drops below 4X as fast as Quicksort.

Figure 3: Relative Algorithm Performance for 250MB on a 400MHz Altivec



Spreadsort defeats the θ(nlog(n))

comparison-based limit by combining two aspects of the problem, the distribution and the bin size. By solving both problems simultaneously, it keeps cutting down until one or the other is ready for a quick O(n) solution. With most distributions, this should be just one or two iterations, which has been proven for a similar technique[8]. The simple generalization of Quicksort is obtained for Spreadsort by using a value function that returns a value for a key. This ends up bringing Spreadsort to the point where it can solve the vast majority of sorting problems within 2 iterations plus the time it takes to comparison-sort an 8 item bin.

5 Conclusion

Spreadsort is a practical general-case sorting algorithm with $\theta(n)$ average-case performance and good worst-case performance, being $O(n\log(n))$ in comparisons and $O(n\log_n(m))$ in time. It can be used in any situation where a definite ordering can be applied to all possible values, even values of infinite key length. Because the core Spreadsort technique divides the problem both distributionally and numerically (smaller bucket sizes), it makes the problem simpler to solve for both subsidiary comparison-based and distributionally-based algorithms. Each splitting operation takes O(n) time, but will cut the remaining key length by a fraction of n, while cutting the bucket size. With all distributions, each operation will divide the distribution into multiple pieces, commonly a large number. If the distribution is not cut into many pieces, then it is well set up for recursive application and eventual final distributional sorting. If the distribution is cut into many pieces, then a comparison-based sort can easily sort the small subsidiary bins.

This improved divide-and-conquer technique provides a significant real performance enhancement over conventional $\theta(n\log(n))$ techniques, such as Quicksort. This performance enhancement is gained by using a normally constant number of time-consuming operations instead of a $\log(n)$ number of quick operations. In practical applications processor caches and memory consumption influence speed, but Spreadsort shows a clear improvement. This improvement has been verified by experiment, and shows a general-case distributional algorithm that has superior performance to Quicksort.

References

[1] J. D. Bright, G. F. Sullivan, and G. M. Masson, “A Formally Verified Sorting Certifier,” IEEE Transactions on Computers, Vol 46, No. 12, December 1997.

[2] M. H. Nodine and J. S. Vitter, “Large-Scale Sorting in Parallel Memories,” 3rd ACM Symp. On Parallel Algorithms and Architectures, pp. 29-39, 1991.

[3] V. Markl and R. Bayer, “A Cost Function for Uniformly Partitioned UB-Trees”, Database Engineering and Applications Symposium, 2000, pp 410-416.

[4] K. Sadakane, “A Fast Algorithm for Making Suffix Arrays and for Burrows-Wheeler Transformation,” Data Compression Conference 1998. pp 129-138.

[5] Donald E. Knuth, The Art of Computer Programming -- Sorting and Searching, vol. 3, 1997.

[6] C.A.R. Hoare, “Quicksort,” Computer J., vol. 6, no. 1, pp. 10-15, 1962.

[7] A. Andersson and S. Nilsson. A New Efficient Radix Sort. In 35th Symp. On Foundations of Computer Science, pp. 714-721, 1994.

[8] Markku Tamminen, “Two Levels are as Good as Any” J. Algorithms 6, pp. 138-144, 1985