

ARL, a faster in-place, cache friendly sorting algorithm

Arne Maus, arnem@ifi.uio.no

Department of Informatics

University of Oslo

Abstract

*This paper introduces a new, faster sorting algorithm (ARL – Adaptive Left Radix) that does in-place, non-stable sorting. Left Radix, often called MSD (Most Significant Digit) radix, is not new in itself, but the adaptive feature and the in-place sorting ability are new features. ARL does sorting with only internal moves in the array, and uses a dynamically defined radix for each pass. ARL is a recursive algorithm that sorts by first sorting on the most significant ‘digits’ of the numbers – i.e. going from left to right. Its space requirements are $O(N + \log M)$ and time performance is $O(N * \log M)$ – where M is the maximum value sorted and N is the number of integers to sort. The performance of ARL is compared with both with the built in Quicksort algorithm in Java, `Arrays.sort()`, and with ordinary Radix sorting (sorting from right-to-left). ARL is almost twice as fast as Quicksort if $N > 100$. This applies to the normal case, a uniformly drawn distribution of the numbers $0:N-1$. ARL is also significantly faster than Quicksort for 8 out of 9 other investigated distributions of the numbers to be sorted. ARL is more cache friendly than Quicksort and also maybe ordinary Radix by generating fewer cache misses. Even though Radix is faster in most cases, ARL becomes faster than Radix for large n ($n > 5 * 10^6$). Since the space requirement for Radix is twice that of ARL (and Quicksort), this paper proposes ARL as a better general sorting algorithm than Quicksort.*

1. Introduction

Sorting is maybe the single most important algorithm performed by computers, and certainly one of the most investigated topics in algorithmic design. Numerous sorting algorithms have been devised, and the more commonly used are described and analyzed in any standard textbook in algorithms and data structures [Dahl and Belsnes, Weiss] or in standard reference works [Knuth, van Leeuwen]. Maybe the most up to date coverage is presented in [Sedgewick03]. New sorting algorithms are still being developed, like Flashsort [Neubert] and ‘The fastest sorting algorithm’ [Nilsson]. The most influential sorting algorithm introduced since the 60’ies is no doubt the distribution based ‘Bucket’ sort which can be traced back to [Dobosiewicz].

Sorting algorithms can be divided into comparison and distribution based algorithms. Comparison based methods sort by comparing two elements in the array that we want to sort (for simplicity assumed to be an integer array ‘a’ of length N). It is easily proved [Weiss] that the time complexity of a comparison based algorithm is at best $O(N \log N)$. Well known comparison based algorithms are Heapsort and Quicksort [Hoare]. Distribution based algorithms on the other hand, sort by using directly the values of the elements to be sorted. Under the assumption that the numbers are (almost) uniformly distributed, these algorithms can sort in $O(N * \log M)$ time where M is the maximum value in the array. Well

known distribution based algorithms are Radix sort in its various implementations and Bucket sort.

Quicksort is still regarded as the in practice fastest known sorting algorithm [Weiss]. Its good performance is mainly attributed to its very tight and highly optimized inner loop [Weiss]. The execution time of a sorting algorithm is in general affected by:

1. CPU speed.
2. Length of the data set – N .
3. Distribution of data – standard is Uniformly drawn (0: $N-1$).
4. The algorithm itself, the number of instructions performed per sorted number.
5. Cache friendliness of the algorithm.

In this paper we will investigate the last four of these factors when comparing ARL with Quicksort and Radix.

Even though the investigation is based on sorting non-negative integers, it is well known that any sorting method well suited for integers is well suited for sorting floating point numbers, text and more general data structures [Knuth].

2. Radix Sorting Algorithms

Basically, there are two Radix sorting algorithms, Left Radix, also called MSD (Most Significant Digit) and Right Radix or LSD (Least Significant Digit). Right Radix is the by far most popular variant of radix sorting algorithms.

We will in the following denote ordinary Right Radix just Radix, where one starts with the least significant digit, sort on that digit in the first phase by copying all data into another array. Radix then sorts on the next digit, right-to-left. Each time, Radix copies the data between two arrays of length N , and runs through the whole data set in each pass.

In MSD or Left Radix, one starts with the leftmost, most significant digit, and sort all data based on the value of that digit. Afterwards, LR sorts recursively each subsection with equal value of the previous digit to the left, until the data is sorted on the last, rightmost digit.

A digit is a number of consecutive bits, and can in both radix algorithms be allowed to be set individually for each pass. It is typically from 12 to 4 bits, and is not confined to an 8-bit byte. ALR owes its good performance to its ability to vary the size of the sorting digit when sorting sparse subsections of the array (sparse being defined as few numbers to sort compared to the number of bits left to sort on).

The first computer version of the MSD-radix algorithm improved upon in this paper is formulated in [Hildebrandt]. In the description of the MSD-radix algorithm, the original 1-bit algorithm of Hildebrandt and Isbitz is described in [Cormen, Leeuwen, Sedgewick88, Knuth] where internal move in the array is possible. When sorting on a larger sized digit than 1 bit in each phase [Aho, Knuth], an extra array of length N or a list-based implementation, which amounts to an extra space requirement for N pointers, is used. In all descriptions of the MSD-radix algorithm, a radix of fixed size is used. The speed of 1 bit

MSD-radix is reported to be slightly faster than Quick-sort if the keys to be sorted are comprised of truly random bits, but Quick-sort can adapt better to less random situations [Knuth, Sedgewick88].

3. The ARL algorithm

The algorithm presented here, can be described as a recursive Left-radix algorithm with a dynamically determined number of bits in its digits for each sorting of a segment of the array. It sorts by internal moves using permutation cycles within the given array and is optimized with Insert-sort for sorting shorter sub-sequences (with length less than some constant – say 20). It is coded in pseudo-Java and is presented in Program 1.

Comments to the pseudo code:

0. If the maximum value in 'a' is known, the highest bit can easily be calculated. (the bits are numbered from left to right: 31,30,...,1,0 – and bit 31 is the most significant). Otherwise we can find maxBit faster by simply do exclusive 'or' on all the elements in 'a' into a variable and then find the max bit set in that variable.
1. The adaption (to determine 'numbits') is based on tree principles:
 - a. The size (=number of bits) of the next digit must be smaller than some max value (here:11) to make two supporting arrays each of length 2^{numbit} fit nicely into the first level cache.
 - b. The size must also be larger than some min value (here:3 bits) to ensure a decent progress.
 - c. Between these values, we try to make the execution times of the two following loops 2) & 3) equal, the first of length 2^{numbit} and the second of length 'num'.
2. We need two supporting arrays of length 2^{numbit} to:
 - a. keep track of the number of elements in 'a' for each possible value of the digit and
 - b. where to place each of these sets when sorted.
(two arrays are needed when we have equal valued elements on current digit).
3. The permutation cycle. Pick up first unsorted element, remove it and calculate where it should be placed when sorted on current digit. Swap it with the element in that position, calculate that elements sorted position... and so on until we place an element where that cycle started. Go to next unsorted element and repeat a new cycle until finished.
4. We test if there are more digits to the right to sort on.
5. If so, recursively call arl(..) for each set of equal values for the digit we just sorted on, If that segment in 'a' is shorter than some constant (here 20), use insertionSort() instead.

It must be noted that ALR is *not* a stable sorting algorithm meaning that equal valued input might be interchanged on output. If input to ALR is {2,1,1} the output will be {1,1,2}, but the two '1s' on input will be exchanged on output. The reason for that is that we use permutation cycles to exchange elements instead of copying back and forth between arrays.

```

ARLsort(int [] a) {
    // 0.) Find max bit in 'a'
    < maxBit = highest bit set to '1' in array a>;
    arl (0, a.length -1, a, maxBit);
}

arl ( int a[], int start , int end, int leftbitno )
{
    num = end-start+1;

    // 1) . Adaptive part - determine 'numbit' = size of next digit
    numbit = Math.min(leftbitno+1,maxbitno);          // maxbitno = 10,11 or 12
    while( (1<<numbit) > num && numbit > 4) numbit --;

    // 2). count in how many numbers there are in 'a[start..end] ' of each value in this digit.
    // Also make indexes to where each of these sets of equal values should end up
    // in 'a' when sorted

    // 3.) Use permutation cycles to move all elements in place on current digit
    while ( totalmoved < num ) {
        // pick up first, not sorted element
        < t1 = temp = first unsorted element>

        // next permutation cycle
        do { find out where 'temp should end up (based on the digit value of temp)
            swap (temp, element at that position)} until (temp ends up at t1's place);
        }
    }

    // 4.) Test if last digit
    if ( leftbitno - numbit > 0)      {
        p = start;
        for ( segSize = size of next segment of equal values for current 'digit' (now sorted)) {
            // 5.) Recursivly call for each set of equal values with length> 1
            if (segSize < 20)
                insertionSort( that segment: a[p..p+segSize]) ;
            else
                alr (a, p, p+segSize-1, leftbitno-numbit);          // sort on next digit
            p = p + segSize;
        } }
    }/* end alr */
}

```

Program 1. Pseudo code for the ALR-algorithm

4. The effect of caching

I have previously reported [Maus] the effect of caching on sorting for Radix type of sorting algorithms. Since the speed difference between the CPU and main memory has increased by a large factor since that paper, I will briefly describe the effect, and bring some new results.

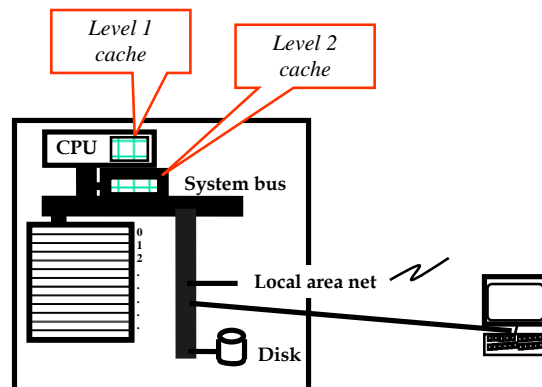


Figure 1. Level 1 and level 2 cache memories.

Typically, a modern CPU operates on 2 GHz. Hence its execution time for simple instructions is 0,5 nsec, while an access (read or write) in main memory typically takes 100-200 nsec. There is then a factor of 200 or more between accessing a register in the CPU and accessing a variable in main memory! Because of this, two or three of cache memories, level 1 (size: 8-32 kB), level 2 (size: 256kB – 1 Mb) and maybe also level 3 (as on the new 64-bit Intel Itanium processor) are placed between the CPU and main memory to smooth out this huge speed difference.

When the CPU tries to access a memory location (for data or instructions), it first looks in its level 1 cache. If it is not found there it generates a 'cache miss' to the next level cache to see if it can be found there,..., and so on until it at least is found in main memory. When a cache miss occurs, typically a cache line of 32 byte is transferred from where it is found and all the way up to the level 1 cache before that memory location is used by the CPU. This scheme is effective because programs has a tendency for sequentially accessing elements in memory and programs have loops. If access on the other hand is truly random and the data set a program uses is (far) larger than the sizes of the cache memories, this scheme does not work well.

To test this speed difference, I performed the same number of instructions, as shown in Program fragment 2. That code is inspired by similar statements found in sorting algorithms .

In figure 2, the following parameters are varied:

- n- the length of b,
- k - the number of nested references to b in a,

- the contents of the array b .
If $b[i] = i$, the sequential case, then this code will create almost no cache misses. But if we fill b with randomly drawn numbers in the range $0..n-1$, then almost every nested reference will generate a cache miss when the size of b is large compared with the size of a cache memory.

In figure 2, the results are shown normalized: The random case divided by the sequential case for each length of b . We then get one curve for each k , the number of nested b 's inside a , and the number we get on the y-axis gives the factor by which random access is slower than the sequential case. This test is described in more detail in [Maus].

```
for(int i= 0; i < n; i++)
    a[b[b[...b[i]...]]] = i ;
```

Program 2. The cache miss test.

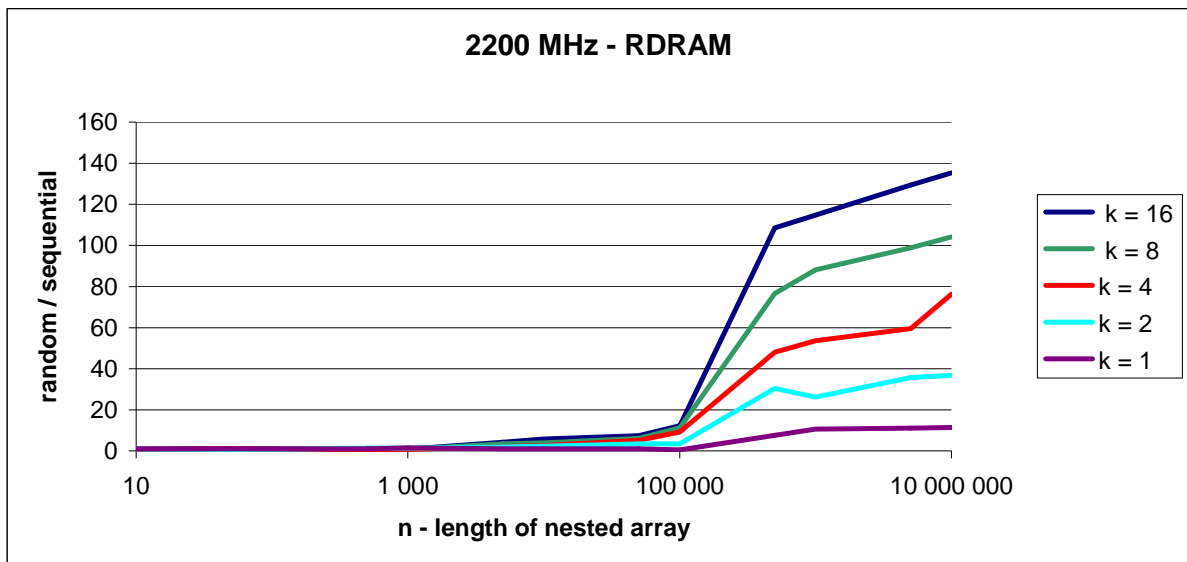


Figure 2. The ratio between the time for a sequential and a random content for the array b in Program 2 when we vary the nesting level k and the length of b .

In [Maus] a factor of up to 23 was reported for a 460 MHz Pentium III PC. In figure 2 we find a factor of more than 130 for a 2200 MHz Pentium IV with RDRAM. This factor will increase as CPUs get faster. CPU speed will soon reach 10 GHz, but memory basically can not be made to run much faster because of the heat memory chips generate. If we want air cooling and dense memory chips, then memory seems to be stuck with today's speed. The price we then pay for a cache miss to main memory might then by the end of this decade increase to a factor of 500 or more than accessing the level 1 cache.

From the curves in figure 2, we see that when b is short, less than 1000 integers or 4 kB, we get no cache miss (sequential and random access times are equal). From $n=2000$ to $100,000$,

we see that we get an increasing number of level 1 cache misses to the level 2 cache. For $n > 100\,000$, or approx. 0,5 MB, we see that we get an increasing number of cache misses down to main memory, and after $n = 1\,000\,000$, we see that the probability that any access is not a cache miss to main memory, is small and the curve levels out. The reason that we get different curves for different k – the depth of nested references to b , is that the total execution time is a mix of some very cache friendly accesses, the program code itself, which is a very tight loop, and the data-accesses. Increasing k , increases the amount of cache unfriendliness in the mix of accesses made by the CPU.

5. Comparing Quicksort, Radix and ALR in the normal test case

By far the most popular distribution used for comparing sorting algorithms, is a random uniform distribution $0..N-1$, where N is the length of the integer array. In Figure 3 we use that distribution and give relative performance figures for Radix, ALR and Quicksort, where the execution times for ALR and Radix for each value of N is divided by the execution time for Quicksort to get relative performance figures.

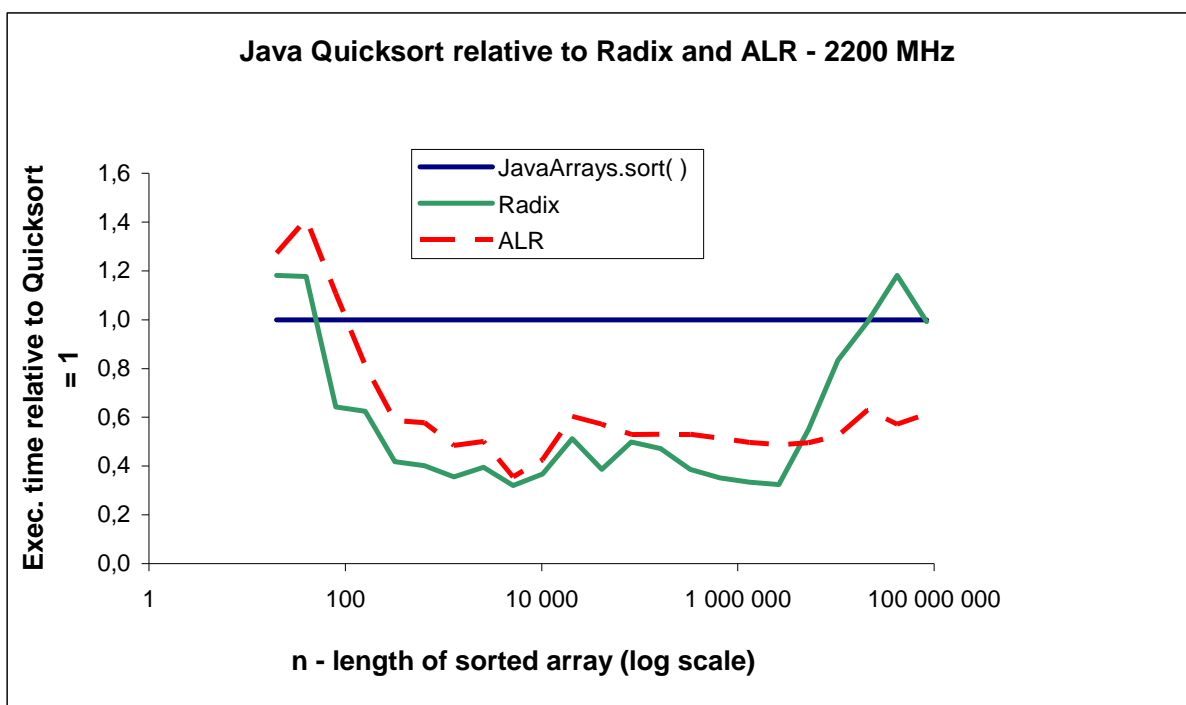


Figure 3. The relative performance of Radix and ALR versus Quicksort (= Java Arrays.sort(int[] a))- note a logarithmic x-axis

We can conclude that for the standard test case, Radix and ALR are much faster than Quicksort – on the average 50% or more for $N > 200$. For all but the extreme large values of N we see that Radix is superior, closely followed by ALR when compared to Quicksort. For very large values of $N > 10^6$, the performance of Radix degenerates. We will comment on that in section 7.

6. Comparing Radix, ALR and Quicksort for ten different distributions

Since not all numbers we sort are taken from a uniform distribution $0..N-1$, it would be interesting to test other distributions, of which there are as many as our imagination can create. In Figure 4 and 5 the following 10 distributions are used:

1. Uniform distribution $0:N/4$
2. Uniform distribution $0:N-1$
3. A random permutation of the numbers $1..N$
4. A sequence of sorted numbers $1,2,\dots,N$
5. A sequence of almost sorted numbers, every 7'th element permuted
6. A sequence of inverse sorted number $N, N-1,\dots,2,1$
7. Uniform distribution $0:3N-1$
8. Uniform distribution $0:10N-1$
9. Uniform distribution $0:2^{30}-1$
10. The Fibonacci sequences starting with $\{1,1\},\{2,2\},\{3,3\},\dots$
One sequence is used until it reaches maximum allowable value for a positive integer or we reach N numbers. (gives short sorted subsequences with a few large numbers)

We show the results in two figures: Figure 4 Radix versus Quicksort, and Figure 5: ALR versus Quicksort. In both cases all execution times are normalized with Quicksort = 1, so we get relative performance figures.

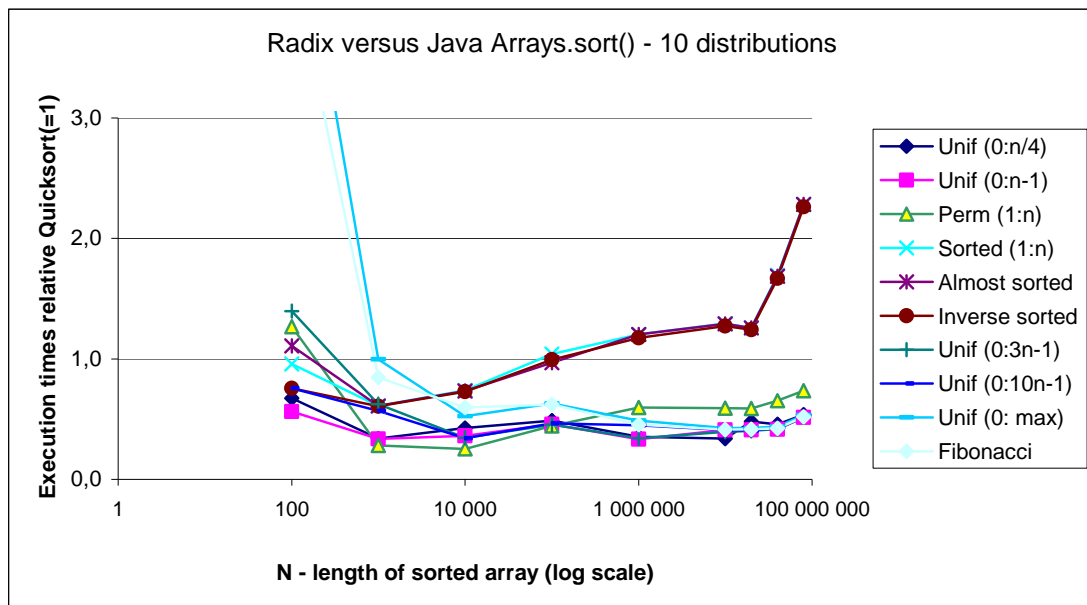


Figure 4. Radix versus Quicksort, 10 different distributions.

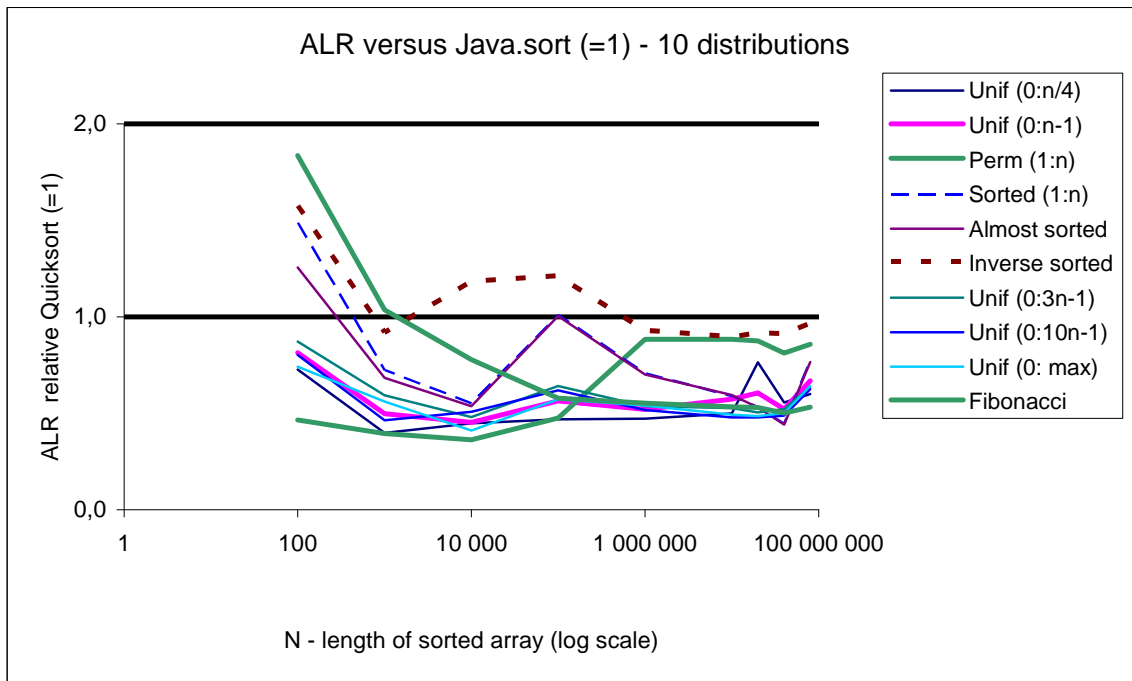


Figure 5. ALR versus Quicksort, 10 different distributions.

We see that both Radix and ALR perform better than Quicksort, but Radix ‘looses’ clearly in three of the ten distributions (sorted, inverse sorted and almost sorted) against Quicksort. ALR is better in nine out of ten distributions, and only in the inverse sorted case, is ALR not as good for some values of N, but better for really large values.

Since Radix uses twice as much memory as ARL, I would advise that one should prefer ARL as a general purpose sorting algorithm if one can accept that ARL is not a stable sorting algorithm.

7. Cache friendly sorting

A first question to ask is whether the cache tests in section 4, which were designed to force the memory system to generate cache misses, carry over to ordinary applications like a sorting algorithm. The answer is yes. A careful counting of all the loops in the radix sorting algorithm when $n = 1\,000$, shows that it performs 6 000 loop iterations in 0.0518 millisecc., but when $n = 1\,000\,000$, it performs 6 002 000 loop iterations in 90.2250 millisecc. This is 1742 times as long time with only 1000 times as many loop iterations. The explanation for this must be the effect of caching when the array to be sorted grows from 4kB to 4MB. Cache friendly comparison based sorting algorithms have been reported in [LaMarca & Ladner].

One might also wonder how it is possible to write a cache friendly sorting algorithm. After all, sorting a set of randomly distributed numbers, consists of either reading them sequentially

and writing them back into memory in a ‘randomly’, now (more) sorted order – or the other way around – reading ‘randomly’ and writing back sequentially. The answer to that is that a sorting algorithm must always do at least one such cache unfriendly pass of data, but it doesn’t have to do it repeatedly. Of the three sorting algorithms investigated in this paper:

- The ordinary Radix algorithm does this scanning for each pass of data – basically two sequential reads and one ‘random’ write of the whole array for each say 11 bit radix we sort on (plus one back-copying into the original array if we do an odd number of such passes). Radix also uses two arrays of length N and hence meets this cache unfriendly situation before any in-place sorting algorithm.
- ARL does first a (very) cache unfriendly pass over data by dividing the dataset into 2048 separate datasets. Afterwards, each such data set can be sorted individually and fits nicely into either the level 1 (or at least the level 2) cache. ARL hence only does one such cache unfriendly pass.
- Quicksort on the other hand, is basically cache friendly in the sense that it reads sequentially (forwards and backwards) data swapping unsorted data, each time splitting data into two separate data sets that can be sorted individually. The trouble with Quicksort, is that it does so many of these passes. When N is large, it takes quite a number of passes before the data sets get small enough to fit into the caches.

From the above analysis, it can be assumed that ARL should be more cache friendly, and hence relatively faster than Radix for large data sets. In figure 3 we find such a possible gain for ARL over Radix when $n > 5 * 10^6$. It must also be noted, however, that there are many ways to tune these algorithms (maximum radix size, adaptive radix size and when to use insertSort for shorter subsequences with Quicksort and ARL). The cache advantage of ARL over Radix occurs for some of these cases but not for all. For all normal values of such tuning parameters and with the standard test case (uniform distribution 0:N-1), however, both Radix and ARL are superior to Quicksort by a factor of approx. 2.

8. Conclusions

The purpose of this paper has been twofold. First to present a new algorithm ARL, that introduces two new features to the left radix sorting algorithm. These additions make ARL an in-place sorting algorithm and adapts it well to almost any distribution of data. It has been demonstrated that it is clearly superior to Quicksort, and in most cases by a factor of 2. A disadvantage with ARL is that it’s not a stable sorting algorithm – meaning that equal elements on input might be interchanged on output. For most applications, that is not a problem. It is thus proposed to use ARL instead of Quicksort as a preferred general purpose sorting algorithm.

The second theme of this paper has been the effect of caching on our programs. An access that creates a cache miss to main memory now takes more than 100 times as long as to data already in the closest cache. This effect will increase in the next decade to maybe a factor of 500 or more with the coming of 10 GHz, 64bit processors. Both Radix and ARL outperform Quicksort by doing far fewer (but more complicated) accesses to data, thus

limiting the number of cache misses. That might be a general message, fewer accesses to data and programs that makes the most out of data it has fetched to the CPU might be preferable to many fast passes over the same data- usually taking a binary decision each time.

References

- [Dahl and Belsnes] - Ole-Johan Dahl and Dag Belsnes : „*Algoritmer og datastrukturer*“ Studentlitteratur, Lund, 1973
- [Dobosiewicz] - Wlodzimierz Dobosiewicz: “Sorting by Distributive Partition”, *Information Processing Letters*, vol. 7 no. 1, Jan 1978.
- [Hoare] - C.A.R Hoare : “ Quicksort”, *Computer Journal* vol 5(1962), 10-15
- [Knuth] Donald E. Knuth: „*The art of computer programming - vol.3 Sorting and Searching*“, Addison-Wesley, Reading Mass. 1973
- [LaMarca & Ladner] - Anthony LaMarcha & Richard E. Ladner: “The influence of Caches on the Performance of Sorting”, *Journal of Algorithms* Vol. 31, 1999, 66-104.
- [van Leeuwen] - Jan van Leeuwen (ed.), *Handbook of Theoretical Computer Science - Vol A, Algorithms and Complexity*, Elsevier, Amsterdam, 1992
- [Neubert] – Karl-Dietrich Neubert, *Flashsort*, in Dr. Dobbs Journal, Feb. 1998
- [Nilsson] - Stefan Nilsson, *The fastest sorting algorithm*, in *Dr. Dobbs Journal*, pp. 38-45, Vol. 311, April 2000
- [Sedgewick88] Robert Sedgewick, *Algorithms*, Second. ed., Addison-Wesley, 1988
- [Sedgewick03] – Robert Sedgewick, *Algorithms in Java*, Third ed. Parts 1-4, Addison Wesley, 2003
- [Weiss] - Mark Allen Weiss: *Datastructures & Algorithm analysis in Java*, Addison Wesley, Reading Mass., 1999

[Aho 74] {Aho} {Aho V.A, Hopcroft J.E., Ullman, J.D., The Design and Analysis of Computer Algorithms, second ed. (Reading, MA:Addison-Wesley, 1974)}

\bibitem

[Baase 88] {Baase}

{ Baase S., Computer Algorithms, second ed. . (Reading, MA: Addison-Wesley, 1988)}

\bibitem [Cormen et al. 90] {Cormen}

{ Cormen T.H., Leiserson C.E., Rivest R.L., Introduction to Algorithms.\ (Cambridge, Ma: The MIT Press, 1990)}

\bibitem [Dahl and Belsnes 73] {Dahl} { Dahl, O.J. and Belsnes D., Algoritmer og datastrukturer.(in Norwegian) (Studentlittertur, Lund 1973)}

\bibitem [Gonnet and Baeza-Yates 91] {Gonnet} {Gonnet G.H., Baeza-Yates R., Handbook of Algorithms and Data Structures. second ed. (Workingham, England: Addison-Wesley, 1991) }

\bibitem [Hildebrandt and Isbitz 59] {Hildebrandt} { Hildebrandt P. and Isbitz H., Radix exchange - an internal sorting method for digital computers. J.ACM 6 (1959), p. 156-163 }

\bibitem [Hoare 61] {Hoare}{ Hoare C.A.R., Algorithms 63 and 64. Com. ACM 4 (1961), p.321 }

\bibitem [Knuth 75] {Knuth}{ Knuth D. E., The art of Computer Programming, vol 3, second printing. (Reading, MA : Addison-Wesley, 1975)}

\bibitem [Leeuwen 90] {Leeuwen}{ Leeuwen J. van (ed.), Handbook of Theoretical Computer Science - vol A (Amsterdam, The Netherlands, 1990)}

\bibitem [Manber 89] {Manber}{Manber U., Introduction to Algorithms - A Creative Approach. (Reading, MA: Addison-Wesley, 1989)}

\bibitem [Reingold et al. 77] {Reingold}{Reingold E.M., Nievergelt J., Deo N., Combinatorial Algorithms, Theory and Practice. (Engelwood Cliffs, New Jersey, : Prentice Hall, 1977)}

\bibitem [Sedgewick 88] {Sedgewick}{Sedgewick R., Algorithms (second. ed.) (Reading, MA: Addison-Wesley, 1988)}

\bibitem [Stubbs and Webre 85] {Stubbs}{Stubbs D.F., Webre N. W., Data Structures with Abstract Data Types and Pascal. (Monterey, Cal.: Brooks/Cole, 1985)}

\bibitem [Weiss 92] {Weiss}{Weiss, M. A., Data Structures and Algorithm Analysis. (Benjamin\sla Cummings, Redwood, CA 1992)}

\bibitem [Wood 93] {Wood}{Wood D., Data Structures, Algorithms and Performance. (Reading, MA. Addison-Wesley, 1993}