RadixInsert, a much faster stable algorithm for sorting floating-point numbers

Arne Maus (em)
Dept. of Informatics, University of Oslo
arnem@ifi.uio.no

Abstract

The problem addressed in this paper is that we want to sort an array *a*[] of *n* floating point numbers conforming to the IEEE 754 standard, both in the 64bit double precision and the 32bit single precision formats on a multi core computer with *p* real cores and shared memory (an ordinary PC). This we do by introducing a new stable, sorting algorithm, RadixInsert, both in a sequential version and with two parallel implementations. RadixInsert is tested on two different machines, a 2 core laptop and a 4 core desktop, outperforming the not stable Quicksort based algorithms from the Java library – both the sequential Arrays.sort() and a merge-based parallel version Arrays.parallelsort() for $500<n<250$mill by a factor from 3 to 10.

The RadixInsert algorithm resembles in many ways the Shell sort algorithm [1]. First, the array is pre-sorted to some degree – and in the case of Shell, Insertion sort is first used with long jumps and later shorter jumps along the array to ensure that small numbers end up near the start of the array and the larger ones towards the end. Finally, we perform a full insertion sort on the whole array to ensure correct sorting. RadixInsert first uses the ordinary right-to-left LSD Radix for sorting some left part of the floating-point numbers, then considered as integers. Finally, as with Shell sort, we perform a full Insertion sort on the whole array. This resembles in some ways a proposal by Sedgewick [10] for integer sorting and will be commented on later. The IEE754 standard was deliberately made such that positive floating-point numbers can be sorted as integers (both in the 32 and 64 bit format). The special case of a mix of positive and negative numbers is also handled in RadixInsert. One other main reason why Radix-sort is so well suited for this task is that the IEEE 754 standard normalizes numbers to the left side of the representation in a 64bit double or a 32bit float. The Radix algorithm will then in the same sorting on the leftmost bits in *n* floating-point numbers, sort both large and small numbers simultaneously. Finally, Radix is cache-friendly as it reads all its arrays left-to right with a small number of cache misses as a result, but writes them back in a different location in *b*[] in order to do the sorting. And thirdly, Radix-sort is a fast O(*n*) algorithm – faster than quicksort O(*nlogn*) or Shell sort $O(n^{1.5})$. RadixInsert is in practice O(*n*), but as with Quicksort it might be possible to construct numbers where RadixInsert degenerates to an $O(n^2)$ algorithm. However, this worst case for RadixInsert was not found when sorting seven quite different distributions reported in this paper. Finally, the extra memory used by RadixInsert both in its sequential and parallel versions, is *n* + some minor arrays whereas the sequential Quicksort in the Java library needs basically no extra memory. However, the merge based Arrays.parallelsort() in the Java library needs the same amount of *n* extra memory as RadixInsert.

Keywords: Radix sort, Insertion sort, IEEE 754, parallel algorithms, multicore, Java, Shell sort.

1. Introduction

The chip manufacturers have since 2004 not delivered what we really want, which simply is ever faster processors. The heat generated with an increase of the clock frequency will make the chips malfunction and eventually melt above 4 GHz with today's technology. Instead, they now sell us multi core processors with 2-32 processor cores. Some special products with 50 to72 cores are also available [2, 21], and the race for many processing cores on a chip is also found in the Intel Xeon Phi processor with its fast, unconventional memory access and 62 to72 cores [2]. However, many cores on a chip does not guarantee much faster execution – more often than not the bottleneck for such processing chips is memory and the memory channels. Many of these processors, but not all, are hyperthreaded, where some of the circuitry is duplicated such that each core can switch between two threads within a few instruction cycles if the active thread is waiting for some event like access to main memory.

To the operating system one such hyperthreaded core then acts as a separate core. The conclusion to all this parallelism is that if we faster programs, we must make either new faster algorithms and parallel versions of current or such new algorithms. This paper presents a new sorting algorithm for floating point numbers with two different parallel implementations.

The rest of this paper is structured as follows. First, the sequential RadixInsert algorithm is explained with some comments on how it is implemented in Java. Then RadixInsert is compared with other algorithms in the literature – where the most relevant turned out to be Shell sort from 1959[1] and the ideas in [10]. The two parallel implementations of RadixInsert, one uses an improved merge algorithm [11], and the second uses a full parallel Radix-sort that is explained in more detail. Then graphs for sorting different distributions of numbers on two different machines are presented comparing RadixInsert with the Quicksort based Arrays.sort and Arrays.parallelsort in the Java library. Observations on the differences between the performance on 32bit and 64bit numbers are discussed. Finally, this paper concludes.

2. The sequential RadixInsert algorithm

The problem addressed is that we want to sort a array *a*[] of length *n* (with 64 bit or 32 bit floating point numbers in the IEEE 754 [2,3,13] standard) on a shared memory machine with *p* cores. This we do by first sorting on *m* bits the left part of all the numbers by using the ordinary right-to-left Radix sort (LSD) reading the IEEE 754 numbers as 64 bit (or 32 bit) integers and sort them as integers. The IEEE 754 standard defines many floating point representations; in this paper we focus on the 64bit and 32 bit formats. An IEEE754 number consists of three parts – the leftmost bit is a signbit, then a modified exponent part of length 11 bits for the 64 double representation and 8 bit for the 32 bit format. The thirdrightmost part is the mantissa – the significant bits (with this exponent and sign bit).
We assume that the LSD Radix-sort is well known[4,10]. If we have sorted the left part on *m* bits from bit 63-*m* to bit 63, the sign bit (bits are numbered 63..0), we have then created 2^m sub regions or buckets where all elements in one bucket are larger than all elements in any bucket to the left, and smaller than all elements in any bucket to the right – this because they all have different values in the *m* leftmost binary digits and are sorted on these *m* bits. How we treat negative numbers is described later.

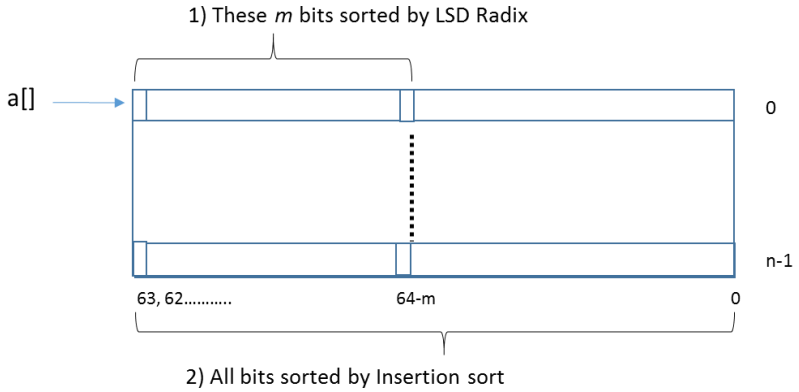


Figure 1. *The array a[] of 64 bit IEEE754 numbers, first partially sorted by LSD Radix and finally by Insertion sort.*

Within each such bucket the elements are not sorted, they are in the order they appear on input. That is another way of saying that this partial LSD sorting of *a*[] is stable.

It is also the case that these small buckets are not of the same size - that is data dependant. Finally, in RadixInsert the whole array *a*[] is sorted using the ordinary, stable Insertsort. This ensures that we have done a stable full sorting of all elements in *a*[] – regardless of their initial distribution. Because LSD sorting makes such a good job of localizing almost equal sized numbers, this last Insertsort phase can be made very quick, but in a theoretical worst case can be $O(n^2)$.

Some more details of the rationale behind this new algorithm. It is based on three observations:

1) The IEEE 754 is such that positive numbers can be sorted as integers.
2) The IEEE 754 standard left-justifies all numbers such that large and small numbers alike have their most significant bits starting at bit 63 (31). When sorting on the same left part of all numbers, they are all sorted to the same degree. If the left part of *a[]* is sorted on *m* bits, then *a*[] is dvided int 2^m buckets, and:
   all elements in bucket $_i$ > elements in bucket $_{i-1}$ (but each bucket is not in any way sorted internally – they are in the same order as on input).
3) When sorting a mix of positive and negative floating-point numbers with LSD Radix, this way, because negative numbers have their sign bit set, negative numbers will be sorted last in reverse order posing as the largest numbers and with the smallest negative number rightmost in *a[n-1]*. Negative numbers we solve by testing the last element after the LSD sorting but before Insertionsort. If that element is negative, by binary search we find the first, leftmost negative number. Then we use the extra array *b*[] of the same length as *a*[], copy first the negative part of *a*[] to the front of *b*[] while swaping it. Last in *b*[] we copy the positive section. As the last but one step we copy *b*[] to *a*[].To keep the stable property also for negative numbers, we then have to walk through the negative section once more because equal elements was first stably sorted to the last section in *a*[]. When we swaped it to the front of *b*[], and later back to *a*[], we reverse that order. As a last step we walk through the negative section element by element and when finding a subsection with equal elements, we swap them a second time, and hence ensure stable sorting.

3. Comparison with other algorithms

The use of Insertion sort for finalizing sorting of a partial sorted array using LSD Radix was nowhere to be found for sorting of floating point numbers, but for integer sorting, Sedgwick proposes first sorting half of the bits with LSD Radix and then finalizing the sorting by applying insertion sort on the whole array. A webpage on sorting IEEE 754 numbers by only using Radix sorting on all 32bits on the short format was found [9]. Almost all papers on sorting concern the sorting of integers, and might miss the interesting observation 2) above which might make floating point numbers more easy to sort. Not many specific algorithms specifically suited for IEEE 754 are presented. Almost all remarks on the net [9] describe using Radix sorting of floating-point numbers with first converting IEEE754 number to some other format, byte or decimal, before sorting. They seem unaware of observation 1) above. These authors seem to rely on the correct observation in Donald Knuth [5] that algorithms for integer sorting are also well suited for sorting floating point numbers in the sense that they concentrate on integer sorting algorithms. A mixed sorting of LSD Radix finished with a sweeping insertion sort on the last bits is not found for floating point numbers. One good reason might be that a pure Radix sorting of all significant bits on the (right justified) integers is usually faster. However, when

investigating distribution based integer sorting algorithms such as Quicksort, the use of Insertion sort for finalizing the sort, are abundant.
RadixInsert for IEEE754 numbers resembles in many ways Shell sort[1] from 1959. In Shell sort Insertion sort is first used along long jumps for shifting small elements to the left and larger to the right of *a*[], later progressively shorter jumps are used until Shell sort finish by sorting all elements using Insertion sort. Shell sort can be described as an Insert-Insert sort algorithm.

4. The run time efficiency of sequential RadixInsert.

a) The general efficiency on two computers,
We first give performance figures for the two computer used for development of this algorithm; a 2core (4 hyperthreaded) laptop, with an Inteli7-4600U @ 2.1 GHz-2.2.7GHz CPU, and a Desktop with 4 core(8 hyperthreaded) Intel i7- 6700 CPU @ 3.40 GHz.

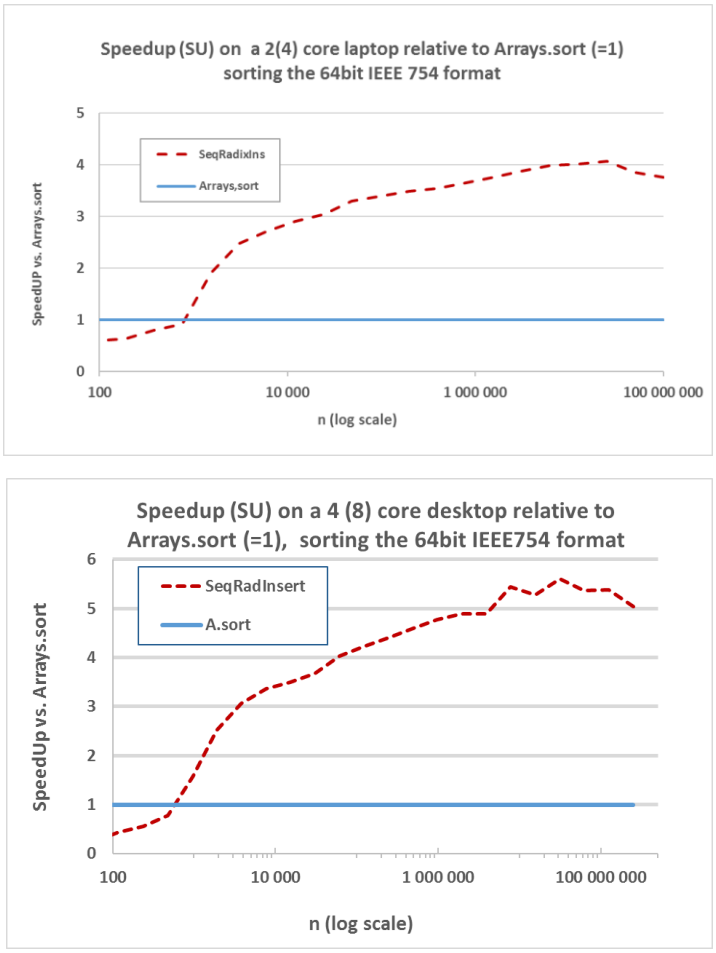


Figure 2. *The Speedup of a Desktop (n= 100.. 250 mill.) and a Laptop (n=100..100mill.) for sorting n numbers with the sequential RadixInsert algorithm compared with the sequential QuickSort based Arrays.sort() algorithm (=1) in the Java library.*

We see that sequental RadixInsert is at best 4 to 5 times faster than Arrays.sort when sorting 64bit IEEE754 floating point numbers. This speedup is basicly only sensetive to *m* , the number of bits we sort on in the Radix phase as explained in the next section.

b) The number of bits used by LSD Radix.
The most inportant issue for the speed of this algorithm is how we determine *m*, the number of bits we sort on in Radix phase . The following formula is used when sorting *n* numbers in the 64 bit IEEE754 format:

$$m = log_2 n + 13$$

The reason for the $log_2 n$ term is simple, to make the length of the average bucket equal to 1. The reason for the addition of 13 bits is motivated by the sign bit plus the 12 bit exponent part of IEEE754 for 64 bit numbers. For most distributions of numbers to be sorted, this part varies little, but must be sorted on. See fig.3 below.

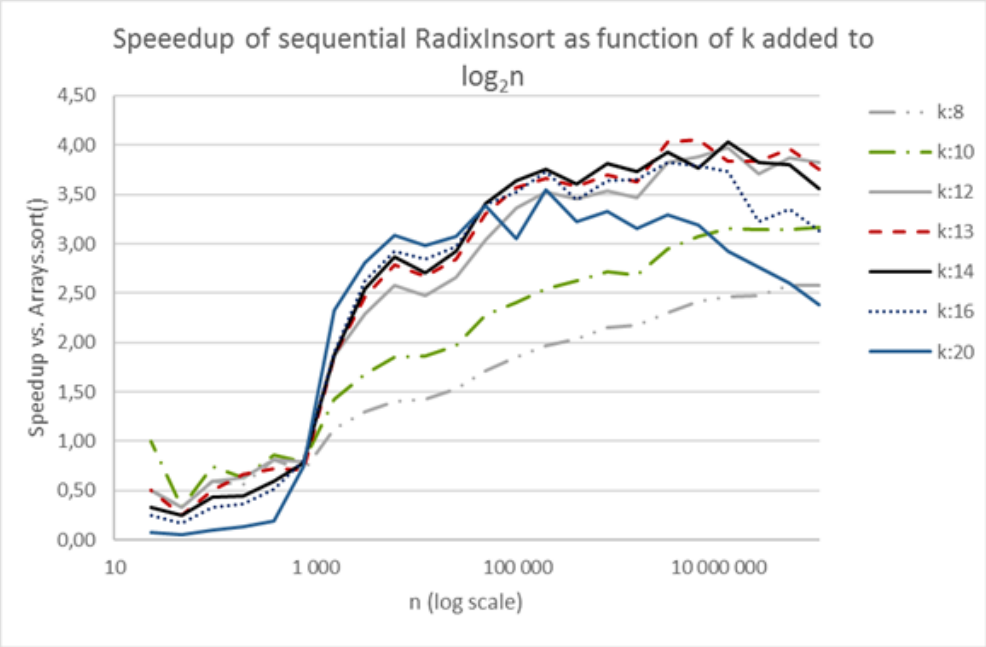
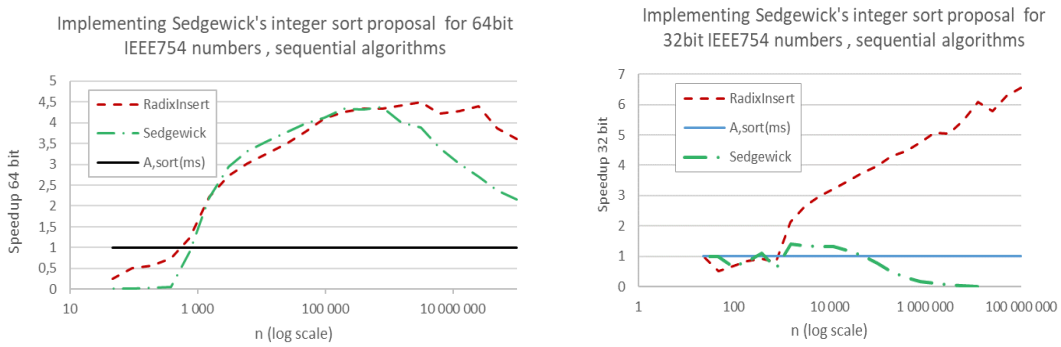


Figure 3. *The Speedup of the sequential RadixInsert algorithm as a function of the number of additional bits to log2(n) we sort on versus the sequential Java library Arrays.sort algorithm on a 4(4) core Desktop for the 64bit IEEE 754 format, n = 10.. 100mill.*

Finaly, it must be mentioned that the extra amount of memory used by RadixInsert is an array *b*[] of size *n* + some minor arrays, whereas the sequential Quicksort in the Java library needs basically no extra memory. However the merge based Arrays.parallelsort() in the same library needs the same amount of extra memory as RadixInsert does in both its sequential and parallel versions.

c) Comparison with Sedgewick's proposal.
Sedgewick has proposed[10], for integer sorting that we should sort on half the bits with Radix and then use Insertsort. If we implement this idea for IEEE754 sorting in 64 bit and 32 bit floating point rutines stating that *m*= 64/2 and *m*= 32/2, we get Fig 4a and Fig 4b below with RadixInsert compared with Sedwick's proposal: 4a for 64bit and 4b for 32bit.



Figures 4a and 4b. *We see that Sedgewick's integer proposal (fig a) of 64/2 bit radix sort is good for 64 bit IEEE754 numbers but inferior to the formula in this paper. However, it is a total disaster (fig b) with 32/2bit radix sort for 32 bit numbers.*

We conclude that Sedwicks propososal does not work well, especially for the 32bit version.

d) The Java optimizer.

Java code is optimized during runtime. The tests calling the sort algorithm in Arrays.sort() and the three RadixInsert classes, are iterated many times, for most graphs 5 times for the largest value of *n* tested and progressively more times for smaller values of *n*. This we do because Java does runtime optimization of programs as they are used. First time the byte-code from the class-file is executed, it is compiled into machine code. With more runs of the same code this machine code is optimized two or three times. The final result is a speedup of more than 100 000 for some operations like the new operation on a class or method calls, while a user written Insert sort method will be optimized with a speedup of 20-30 [13].

The figures presented are always the median of these many runs. It must also be made clear that the two methods from the Arrays class undergo the same optimizations as they are executed and called the same number of times.

e) The distribution of numbers sorted.

The performance of the three RadixInsert algorithms reported in this paper, the sequential and the two parallel versions of that, varies little with the distribution of numbers they sort. We tested seven different distributions; some of which were constructed to make RadixInsert slow, but the results vary little. The seven distributions used when initialising n elements *a*[i], were: " nextDouble()", (which generates a random 64bit double between 0 and 1), "nextDouble()*nextInt(n) (nextInt(n) generates a random integer between 0 and *n* – here converted to a double)", " nextDouble – 0,3"(for generating a mix of positive and negative numbers) ,"nextDouble()+nextInt(n)", "1.0+nextInt(n)", " (n-i) *nextInt(n)-i/10.0", and " (n-i) * r.nextDouble()".

The highest speedup is found with distribution: nextDouble()*nextInt(n) with a best speedup of 4.04, while the worst performing is: nextInt(n) – i/10.0 with speedup of 3.83. Hence most figures use the nextDouble() distribution.

The balance between the 1) Radix and 2) Insert-sort phases in sequential RadixInsert

1) When counting the values of the initial count-arrays (one for each digit), they are all declared and counted at the top method. Since most of this sorting use a 4 digit LSD radix sort, this saves 4 – 1 = 3 reads and a conversion double to long for each element, but the elements also have to be read for each digit it sorts on. Thus, a total of. (1+4)*n = 5n reads

and 4n writes is done in the Radix phase.
Insertsort consist of a double loop. In the outer loop it tests if a[i] ≤ a[i+1], and if that fails, it enters the inner loop to leftshift element a[i+1] in place. Each such shift adds one read and one write. We have then counted this number of such shifts as a function of the %-age of n, the numbers sorted. This %-age is almost constant for all n (1000 <n< 250 mill), but varied significantly with the k, the number of elements added to $\log_2 n$. This is shown in figure 4 for n=10 mill. If k =5, it results in 26n reads and 25n writes in Insert-sort, but if k=14, the figures are 1,025n reads (including 1 read for the outer loop) and 0,025n writes and hence the time used by insert phase is neglect able. We conclude that the time taken by Insert sort decreases rapidly with the number of bits sorted on by the Radix phase, but for values < 22 (last value tested) it 'never' goes to exact 0, so the Insertsort phase is always needed.

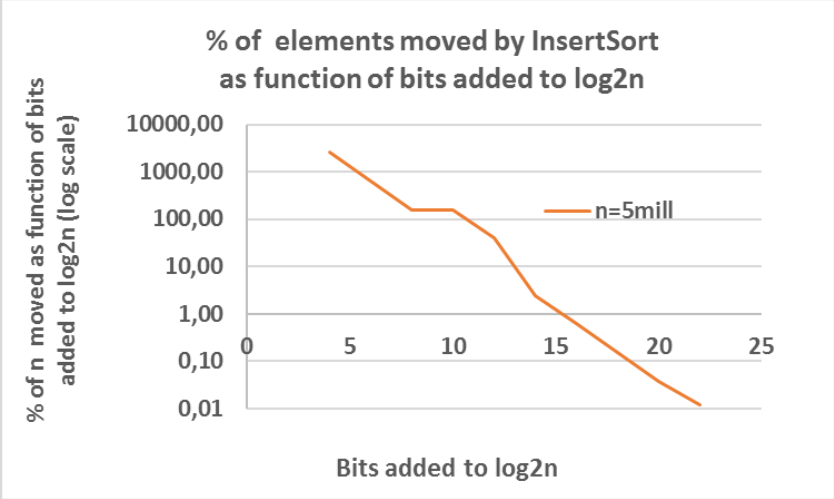


Figure 5. *The %-age of n elements moved by Insertsort as function of the number of bits added to $\log_2 n$ in RadixInsert. This figure shows the work done for 64bit format by InsertSort as a function of k, the number of bits added to log2(n). We see that when k = 5, InsertSort shift 10 times as any elements than n, the amount of elements we sort. But with k = 15, we only move 1% of these n elements – obviously a negligible amount of work.*

5. Two parallel implementations of RadixInsert

The two parallel algorithms described here parallelize both the Radix part of the RadixInsert and the Insertsort part of the algorithm.

a) Merge Para

In [14], with *k* cores, it first divides data into *k* parts and then, in a top down fashion, starts two threads at each level until it has started k threads at the last level. Each thread then sorts its part with sequential RadixInsert which include Insertsort. On backtrack each node merge two segments from both ends, small elements from the left and largest elements from the right end. This is an all parallel merge algorithm, meaning that the top level it's two-parallel, at the next level its four-parallel, ... In the paper it is demonstrated that this merging is faster than ordinary merging when *k* >2 and *n* > 1000 000. Ordinary merging, which only merge from the left part of its segments, has a sequential merge stage at the top.

b) Full Para RadixInsert

This is a full parallel algorithm meaning that it starts *p* threads, one for each core, and apart from two synchronizations between the threads for each digit sorted on, all threads can work at full speed until a full LSD Radix sort is done. Since it has not yet been published, a short description is also given here. Right to left Radix sorting of *a*[] on *m* bits first determine a number of digits to sort on, RadixInsert basically use 4 digits (for n <1000 it uses 2 digits), dividing the *m* bits into 4 more or less equal parts (each 6 to 10 bits long). LSD Radix, starting with the least significant (rightmost) digit, will then, for each digit move data back and forth between arrays *a*[] and *b*[] based on the values on that digit after all elements with smaller values . Doing this four times, then final sorted result ends in *a*[]. The following describes sorting on one such digit. The full sorting is just an iteration of doing this four times sorting on the next set of bits to the left.

Stages in sorting on one digit with $2^{digbits}$ different values 0: $2^{digbits}$-1, with the threads numbered: 0, … (*p*-1). With *p* threads we divide the array *a*[] into *p* equal parts. $thread_0$ owns the leftmost part of *a*[], $tread_1$ the next part,…Each $thread_i$ then owns $a[from_i \,..to_i]$ and does all its sorting on that part to $b[from_i \,..to_i]$.

1. Each thread has an integer array *count*[0..2^{dbits}-1] and counts all different values of the digit in its part of *a*[] .
2. In the shared data area there is declared a two-dimensional array *allCount*[0:*p*-1][]. Each thread sets its *count*[] array into *allCount*. In Java that is a single statement: *allCount*[i] = *count*;
3. All threads synchronize on the same ReentrantCyclicBarrier.
4. Each $thread_i$ creates a second array *count2*[0..$2^{digbits}$-1] and initializes its content following this rule: count2[s] = $\sum_{t=0}^{k-1}[\sum_{j=0}^{s-1} allCount2[\,t][j] + \sum_{r=0}^{i-1} allCount2\,[s][r]_{\_}$, or verbally: *count2[s]* is initialized to the sum of all elements in *allCount [][]* with smaller value than *s* + the sum of all values in allCount [][] with the same value *s* and a smaller thread-index than *i*. An example is that if $thread_i$ finds a 3 in a[] it must be placed after all 0, 1 and 2s from all threads including itself and also after all 3's found by threads with smaller index than *i*. This last part of the rule also secures that RadixInsert is stable.
5. Now all threads can in parallel go through its part of *a*[] and sort its different values *w* to *b*[] and after each such placement in place *b*[count2[*w*]] and increase count2[*w*] by one. Then all threads will write to the correct placement and hence into different elements in *b*[].
6. All threads synchronize on the same ReentrantCyclicBarrier.

This gives a full parallel stable sort in the Radix phase.

6. Making the Insert phase full parallel

The above section describes how the radix phase can be made full parallel. As decribed earlier, we have now partitioned *a*[] into buckets where all elements in one bucket are larger than all elements in all buckets to the right and smaller than all elemets in buckets to the right, but internally no bucket is sorted – they are in the input order. On the average they are of

length 1, but that is data dependant. They might be of any length ≥ 0. Here is how we make the insertsort full parallel:

1. After the Radix phase, all threads sort their part of *a*[] with Insertsort in parallell.
2. All threads synchronize on the same ReentrantCyclicBarrier.
3. All threads but the last has to fix a possible problem with the next thread in case a bucket with two or more elements is split between this thread and the next thread (see fig. 6). Even though each part on this division line is sorted, it might not be sorted across this division.

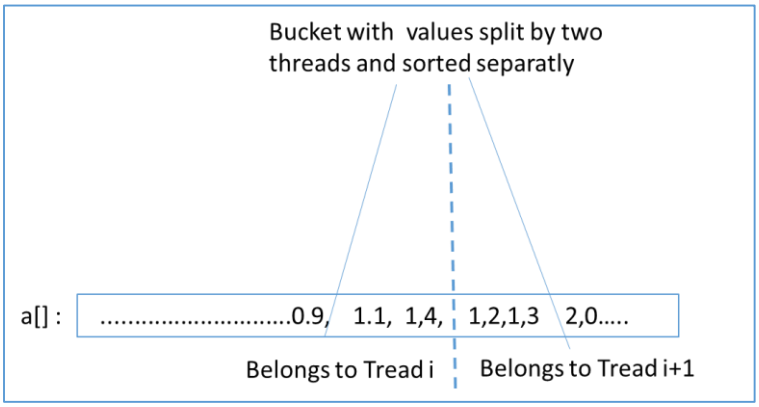


Figure 6. *The problem that might arise if a bucket with more than one element is divided by two threads. This problem is solved by starting insertion sort, beginning with the leftmost element of thread i+1 and shifting smaller elements leftwards into the area for thread$_i$ until this split bucket is fully sorted.*

This problem occurs empirically less than one in 100 million numbers sorted by RadixInsert, but has none the less to be solved. This sorting will be very short stopping with the next buckets left and right.

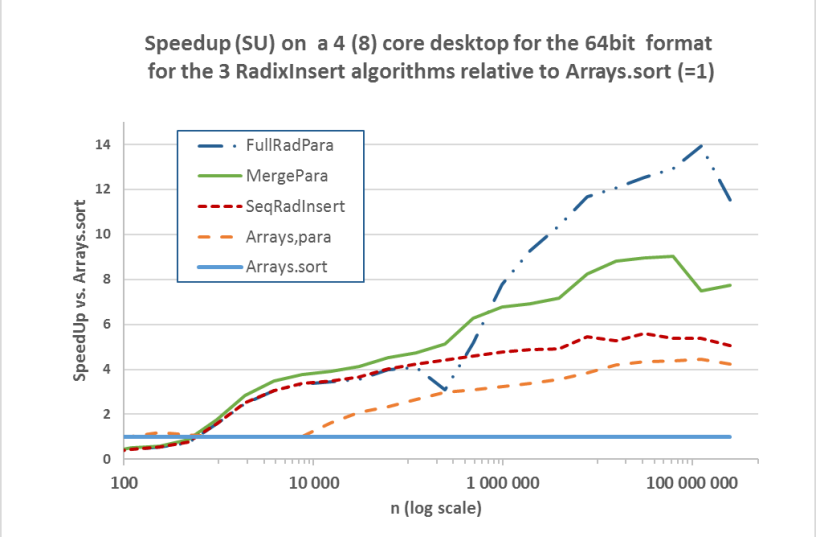


Figure 7. *The Speedup of the sequential, the two parallel RadixInsert algorithms and the Arrays.para algorithm on a 4(8) desktop, the 64bit format*

7. Sorting the 32bit IEEE754 format with RadixInsert

In figure 7 we gave measurements when sorting 64bit floating point numbers. What if we sort 32 bit IEEE754 numbers? The short answer is that the speedup is then even better. The only change except for the obvious recompiling double arrays to float, is that the formula now for selecting m, the number of bits for the LSD Radix to sort is now:

$$m = Math.\min(log_2\, n + 9,\ \ 32)$$

The reason for $\log_2 n$ is the same as for the 64bit double. The addition of 9 here is because the exponent part + sign bit is only 9bit.

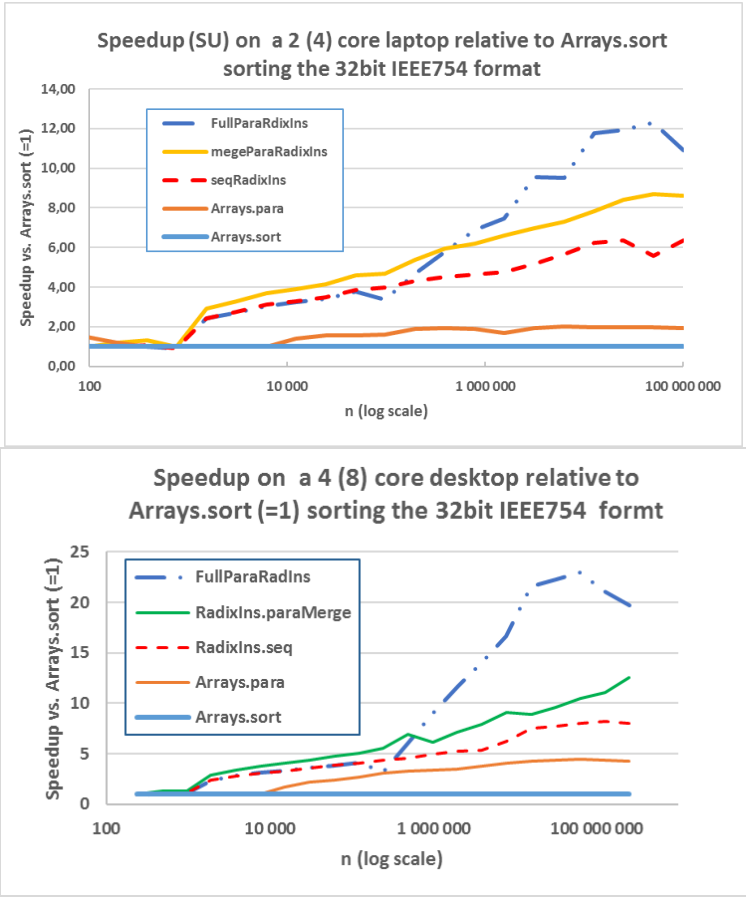


Figure 8. *The speedup of the three RadixInsert algorithms sorting 32bit IEEE 754 numbers on a 2(4) core laptop and a 4(8) core desktop with the nextFloat() distribution. In general RadixInsert sorting the 32 bit format is almost twice as fast as sorting the 64 bit format.*

We also note that the 4(8) core Desktop is approximately twice as fast as the laptop because it has twice as many cores. In addition to the full parallel RadixInsert is up to more than 20 times as fast than Arrays.sort and almost 5 times as fast as Arrays.parallelsort on the Desktop for the 32bit format.

8. Notes on the implementations

The Java program that implements sequential RadixInsert and its two parallel implementations and the tests producing the graphs are implemented as four classes:

a. TestParaRadixInsDouble – does all statistics and collecting run times for all tested algorithms including Arrays.sort() and Arrays.parallelsort(),
b. SeqRadixIns – containing all tuning parameters and the sequential version of RadixInsert. Its user interface is static method: sort (double a[])
c. RadixMergePara – containing the merge parallel version of RadixInsert. This class calls the sorting method in SeqRadixIns. Its user interface is method: *sort* (double a[])
d. RadixFullPara – containing the full parallel version of RadixInsert. This class calls the sorting method in SeqRadixIns. Its user interface is method: sort (double a[])

In a sorting libray, only class SeqRadixIns and class RadixFullPara are needed, This code will be available on the authors home page [15] before the conference. The three last classes also is available in 32bit *float* versions.

- To ensure that all RadixInsert algorithms are correct, the result from the call to Arrays.sort() is kept for each run, and all arrays sorted by the three RadixInsert method are afterwards compared element by element with the Arrays.sort() sorted array.

- For reading a double as a long, the Java library method *Double. doubleToRawLongBits(double value)* is used. If one implementst this algorithm in C, a union beween a *long long* and a *double* can be used to the same effect.

- In some algorithms it is a marked effect to overbook the number of cores/threads we tell the program to use compared with the actual cores present. The effect of this is small here, but a slight 4% increase of speedup of the full parallel radix algorithm with a doubling the number of cores reported by the operating system can be used. In effect then the 2(4) core laptop then runs with 8 threads.

- Like most sorting methods that employs threads, it only starts parallelism when n > some limit (here: numCores * 15 000). If not, the parallel method only uses the sequential algorithm because the time it takes to start *p* threads is larger than sorting such a 'short' array. If $n < 50$, it only uses insertsort.

9. Discussion

We have presented RadixInsert, a new sorting algorithm for sorting 32bit and 64bit floating point numbers following the IEEE 754 standard. Sequential RadixInsert is up to some 4 times faster for the 64bit format and more than 6 times faster for the 32bit format than the standard sequential Java sort method. The best parallel RadixInsert is also at least some 3 times faster than the standard Arrays.parallelsort. What is new in these algorithms is that we have dynamic number of bits we sort on and first and foremost that we sort floating point numbers.

Although there are some disputes between Intel and NIVIDA on arithmetic on this standard [2,3], and that the Java library have two ways of reading such a floating point number, the other with normalized NaN values (Not a Number), it is our claim that as long as all such IEEE754 numbers comes from the same computing device with the same encoding RadixInsert is a valid and much faster sorting method. Two additional facts that strengthen this claim is that we do not do any arithmetic or change any bit – we only read bits in their representation. Also, our sort is always checked as being equal to what quicksort achieves element by element by value. The IEEE 754 standard is supported by Intel, AMD and the Arm (which dominates the mobile phone market) and probably all other CPU and GPU producers. As opposed to Quicksort, RadixInsert is a stable sorting algorithm, which makes serial sorting on more than one data field possible.

The reason that 32bit RadixInsert sorting is almost twice as fast as 64bit sorting, while the quicksort based routines in the java library have little speedup 32bit versus 64 bit, we explain by the execution times reflects more the number of bytes read & written to and from memory. The 32bit RadixInsert reads less than half the number of bytes than 64bit sorting and thus fit better into the caching system. Quicksort on the other hand has far more reads and writes (with n= 1mill, it is in the order of 20) and thus stressing the cache system.

10.Conclusion

We have presented RadixInsert, a new sorting algorithm for sorting 32bit and 64bit floating point numbers following the IEEE 754 standard. Sequential RadixInsert is up to some 4 times faster for the 64bit format and more than 6 times faster for the 32bit format than the standard sequential Java sort method. The full parallel RadixInsert is also 3 to 5 times faster than the standard Arrays.parallelsort.

11. References.

[1] *Shell, D. L. (1959). "A High-Speed Sorting Procedure" . Communications of the ACM. **2** (7): 30–32.*

[2] *ANSI/IEEE 754-1985. American National Standard - IEEE Standard for Binary Floating-Point Arithmetic. American National Standards Institute, Inc., New York, 1985.*

[3] *IEEE 754-2008. IEEE 754–2008 Standard for Floating-Point Arithmetic. August 2008.*

[4] *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Section 5.2.5: Sorting by Distribution, pp. 168–179.

[5] https://docs.nvidia.com/cuda/floating-point/index.html inspected 19.05.2019

[6] https://software.intel.com/en-us/forums/watercooler-catchall/topic/681450 inspected 18.05.2019

[7] http://stereopsis.com/radix.html, inspected 19.05.2019

[8] Knuth, Donald E. (1997). "Shell's method". The Art of Computer Programming. Volume 3: Sorting and Searching (2nd ed.). Reading, Massachusetts: Addison-Wesley. pp. 83–95. ISBN 978-0-201-89685-5.

[9] V. J. Duvanenko, "Faster LSD Radix Sort", https://duvanenko.tech.blog/2019/02/27/lsd-radix-sort-performance-improvements/ February 2019 inspected 19.05.2019

[10] R. Sedgewick, "Algorithms in C++", third edition, 1998, p. 424-427

[11]M. J. Atallah (ed.): «Algorithms and theory of Computation Handbook», ch. 3, ISBN 0-8493-2649-4, CRC Press 1999.

[12] K.Hegna, A.Maus: «Javaprogrammering – kort og godt», s. 236, Universitetsforlaget 2017.

[13] https://en.wikipedia.org/wiki/IEEE_754 (inspected 03.spt. 2019)

[14] A. Maus: "A faster, all parallel Merge sort algorithm for multicore processors" NIK'2018, Norwegian Informatics Conf. Svalbard, 2018. Tapir, www.nik. no

[15] Arne Maus homepage: http://arnem.at.ifi.uio.no/sorting/