IPS, sorting by transforming an array into its own sorting permutation with almost no space overhead[1]

Arne Maus, arnem@ifi.uio.no
Department of Informatics
University of Oslo

Abstract

This paper presents two new algorithms for inline transforming an integer array 'a' into its own sorting permutation - that is: after performing either of these algorithms, a[i] is the index in the unsorted input array 'a' of its i'th largest element (i=0,1..n-1). The difference between the two IPS (Inline Permutation Substitution) algorithms is that the first and fastest generates an unstable permutation while the second generates the unique, stable, permutation array. The extra space needed in both algorithms is O(log n) – no extra array of length n is needed! The motivation for using these algorithms is given along with their pseudo code. To evaluate their efficiency, they are tested relative to sorting the same array with Quicksort on 4 different machines and for 14 different distributions of the numbers in the input array, with n=10, 50, 250.. 97M. This evaluation shows that both IPS algorithms are generally faster than Quicksort for values of n less than 10^7, but degenerates both in speed and demand for space for larger values. These are results with 32 bit integers (with 64 bits integers this limit would be proportionally higher, at least 10^{14}). The two IPS algorithms do a recursive, most significant digit radix sort (left to right) on the input array while substituting the values of the sorted elements with their indexes.

Introduction

Sorting an array *a* is defined in [6] as finding a permutation *p* such that if your read *a* in that order, i.e. a[p[i]] *i=0,1,…,n-1*, it will be in sorted order. This paper introduces two algorithms for transforming the input array *a* into its own sorting permutation.

Numerous sorting algorithms have been devised, and the more commonly used are described and analyzed in any textbook in algorithms and data structures [1, 13, 14] or in standard reference works [6, 8]. Maybe the most up to date coverage is presented in [13, 4]. Sorting algorithms are still being developed, like "The fastest sorting algorithm" [11], ALR [10], Stable ALR [15] and Permsort [9]. The most influential sorting algorithm introduced since the 60'ies is no doubt the distribution based 'Bucket' sort which can be traced back to [3].

Sorting algorithms can be divided into comparison and distribution based algorithms. Comparison based methods sort by repeatedly 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 [14] that the time complexity of a comparison based algorithm is O(*n logn*). Well known comparison based algorithms are Insertsort, Heapsort and Quicksort [14, 5, 2]. Distribution based algorithms, on the other hand, sort by using directly the values of the elements to be sorted. Well known distribution based algorithms are Radix sort, Bucket sort and Counting sort.

In theory these distribution based algorithms sort in O(*n*) time, but because of the caching schemes on any modern CPU they are in practice outperformed by other implementations of the same algorithms having an O(*n log m*) execution time, where *m* is the maximum valued element in the array[16]. Caches, two or three smaller and faster memories between the processor and main memory, have been introduced to overcome the speed difference between the CPU and main memory, which is at least in the order

[1] *This paper was presented at the NIK-2008 conference. For more information, see //www.nik.no/*

of 1:100 [12]. The more efficient distribution algorithms hence sort in succession on smaller sections (digits) of the values of each element. This is done to keep all frequently accessed data in the caches [10, 16]. Hence the elements in the array has to be broken into a number of smaller sorting digits proportional to the number of bits in *m* and the execution time becomes O(*n log m*). If the numbers to be sorted are uniformly distributed U(*n*) (uniformly drawn numbers in the range 0:n-1)., then *m* in practice equals *n,* and both classes of sorting algorithms then have an execution time of O(*n log n*).

The main reason that Radix type sorting in practice is 2 to 4 times faster than Quicksort, is that Radix based sorting accesses and moves each element far fewer times than Quicksort. The difference between Radix sort and Bucket sort is that the Radix algorithms sort on the value of successive parts of the elements (digits) while in Bucket sort and Counting sort, the algorithm sorts on the whole value of the elements by using only one large digit to sort the array in one pass. Counting sort can be implemented as inline sorting by overwriting the original array with the sorted sequence[22], but needs an extra array of length *m* which is the maximum value in the array in order to count how many elements there are of each value. The two new algorithms presented in this paper are based on rather large modifications to the unstable version of the ALR a radix type sorting algorithm introduced in [10].

Quicksort is still in some textbooks regarded as the fastest known sorting algorithm in practice, and its good performance is mainly attributed to its very tight and highly optimized inner loop [14].

As mentioned, the effect of caching in modern CPUs has a large impact on the performance on various sorting algorithm. In [16] it is demonstrated that if one uses a too large digit in radix based sorting, one might experience a slowdown factor of up to 7 sorting large arrays. In [17] the effects of the virtual memory system and especially the effects of the caching of the TLB (Translation Lookaside Buffer) are analyzed. The effects of caching are obviously of great importance for the absolute performance of algorithms, but somewhat outside the scope of this paper. Here the purpose is to introduce two new algorithms for generating a sorting permutation by employing and modifying unstable ALR and evaluate their efficiency by comparing them to sorting the same data with Quicksort.

The sorting permutation

When sorting, it is usually not only a single array *a* we want to sort, but also a number of accompanying arrays *b, c,.. , d* holding satellite data (say you want to sort peoples names and addresses) according to the keys in array *a* (say income). We could, of course, if we knew these other arrays, do exactly the same exchanges to them as we do when sorting the keys in a. But this would be slow and we would have to write specific sorting algorithms in each case. Instead, one could from *a* generate a sorting permutation p such that b[p[i]],…d[p[i]] are the i'th (on *a*) sorted element in these arrays. This is the motivation for finding the sorting permutation.

The second part of this papers title "with almost no space overhead" covers that the expected extra space needed is O(log m) in all but some extreme cases, which will be commented on later. Most other Radix type algorithms need an extra amount of O(n) extra space. The IPS algorithms can then in this sense be described as 'better algorithms'.

This author has previously presented one such algorithm for finding the sorting permutation, pSort [9]. This was a very fast algorithm for the standard case, where the numbers in a followed a U(n) distribution. But since it was a single digit Radix type algorithm, it did not scale well for large values of n, because of the effects of caching. It also had a requirement for extra space that was O(n+m) and hence was slow and unusable or sparse distributions or distributions with large maximal values.

Stable and unstable sorting

Stable sorting is defined as equally valued elements in input keep their relative order when sorted. An unstable algorithm does not give any such guarantee. The stable feature is never a disadvantage and is required if you want to sort a data set on more than one key.

Say you want to sort e-mails by sender, and for each sender by date, you simply with a stable sorting algorithm first sort by date and then by sender. Since equal valued elements on senders name will keep their relative order, which was sorted on date, we get the desired result. To achieve the same with an unstable algorithm, one has to first sort on sender, and then, for each sub segment with the same sender, sort on date, which clearly more complicated. Such cascaded sorting can in practice have more than two levels; e.g. sorting people by first family name, then state, town, street address and finally, first name.

Outlining the IPS (Inline Permutation Substitution) algorithms

The Stable IPS and Unstable IPS algorithms can, as mentioned, best be described as modifications and additions to the unstable ALR (Adaptive Left Radix) sorting algorithm. ALR is a left-to right recursive radix type sorting algorithm with inline exchange of elements to be sorted. However, since the ALR algorithm now has been somewhat optimized since its introduction, and only uses a single array of length $2^{numBits}$, where numBits is the number of bits in the current sorting digit, it is also presented in this paper. ALR is adaptive in the sense that it sets the size of each sorting digit, between a predefined max and min value, according to number of elements to sort. Basically, it tries within these limits to have a sorting digit of length = numBits such that $2^{numBits}$ = 'the length of the subsection of the array we now are sorting'.

The same calculations of the length of each sorting digit are also used by the IPS algorithms. In another paper [15] a stable version of ALR is described. That algorithm is not used here, but the central idea of Stable ALR inspired the design of both IPS algorithms (which is that during a recursive decent sorting, left to right, we gather all elements in *a* that has the same bit pattern for some consecutive bits to be handled by one recursive instance. These bit fields can then (temporary) be used for storing other type of information, because their single common value can be stored in the method instance. We can then later restore the original values if needed.)

The unstable ALR sorting algorithm

In Program 1 we first present the improved pseudo code for the unstable ALR algorithm. This algorithm uses an extra array, point, to first count how many elements there are of each possible value of the current sorting digit. These counters are then added together such that point[i] is the index in a where to place the (next) element with value i in the sorting digit. Also, ALR negates already moved elements during sorting. This limits ALR to sort 31 bit numbers on a 32 bit machine.

```
1. First find the maximum value in a, and hence the leftmost bit set in any number in a.

2. For the segment of the array a we sort on, decide a digit size: numBits between
predefined max and min values for the next sorting digit, and count how many there are of
each value for this digit in an integer array point. After counting; in a second loop, add
these numbers  into pointers  (new  p[i] = start+ sum(old p[k],k=0,.,i-1) Then  point [i] is
the first address to where elements with the i'th values should be stored.

3. We have now found the $2^{numBits}$ pointers to boxes to place sorted elements,  one for
each value of current digit. We then have a simple variable p1, with initial value start of
segment.
Move elements in permutation cycles until all have been moved, repeat as long as $p1 \le$
end-of-segment:
        a) Select the next element e1= a[p1]. If it is negative (then it has been moved),
        store
        -e1-1back in same position; else you have found the element  with the lowest
        index that hasn't been moved (start of a new cycle). The value of the current
        sorting digit in e1, we call v1:
                b) Find out where e1 it should be moved. Set p2= point[v1], and increase
                point[v1]by 1.
                c) Place -e1-1 in p2 and repeat b-c, now with the element e2 (with
                digit value v2) from p2 (later e3 with digit value v3 from p3,...)
                until we end up in some pk = p1 (where the cycle started), and
                we place ek there in position p1. Increase p1  by 1.

4. We have now sorted all elements in a on the current sorting digit. For each box of
length > 1of the current sorting digit, call ALR recursively (to sort on the next digit) – i.e.
perform 1,2 and 3 until there are no more digits to sort on.
  (an obvious speed optimization is here to use Insertion sort if the length of the box is <
               some constant – say 20 - and terminate this recursive branch).
```

Program 1. *The pseudo code for the Unstable ALR sorting algorithm. The step 3 with the in place permutation cycle, makes this an unstable algorithm.*

The Unstable IPS algorithm

The unstable IPS algorithm is a modification to the unstable ALR and is described in pseudo code in Program 2. Note that the modifications are when sorting with the first (leftmost) sorting digit and finally, a shift operation on all the elements.

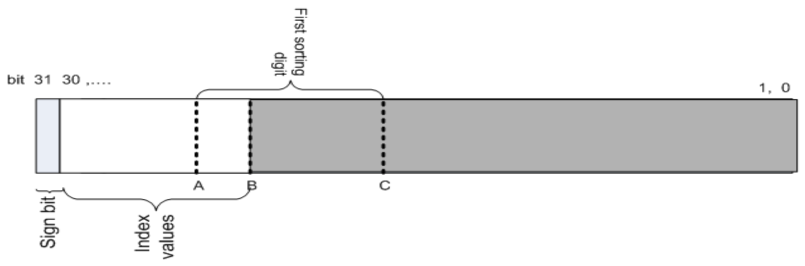

Figure 1. *How the index in input array a is placed in the bit field 30-B, possibly overlapping (replacing) some parts of the first sorting digit. Since we can choose the size of the first sorting digit, this field will always have enough bits to hold the index values.*

a) Determine **B** (figure 1) as **B** = 30-⌈log(n)⌉+1 , which is the lowest bit in the bit field [30:B] when substituting index values into an element.

b) For the first sorting digit: As in unstable ALR, do steps 1 and 2. Then do step 3 with the following modification: When we have read an element **ei** and the corresponding value **vi**, substitute in **ei** from bit **B** and upwards as explained in Figure 1, its index **pi**, possibly overwriting (some or all of) the value field for the first sorting digit.

c) Then do ordinary unstable ALR sort recursively (steps 1-4) for the remaining sorting digits (with Insertsort optimization on the values of the sorting digit).

d) Finally shift down **B** places all elements in **a** such that the index numbers (**p1,p2,..,**) from the first sorting digit appear in bit 0 and upwards.

Program2. *The pseudo code for the Unstable IPS sorting algorithm presented as modification to unstable* ALR.

The Stable IPS algorithm

The Stable IPS algorithm will here be described in pseudo code as modifications to the Unstable IPS algorithm. The modifications are firstly that one can not do Insertsort optimization when doing the recursive sort on the elements, and secondly, that when the recursion is completed on the values of the elements, the index numbers are shifted down for each leaf in the recursion tree and then every such subsection of the array is sorted on these index numbers (recursively with the same method).

a) Do an unstable IPS sort, but drop phase c) and the Insertsort optimization of the recursion tree. Instead, in each recursion tree leaf, when we have finished sorting with the last sorting digit, shift down the index part **B** places to bit 0 (overwriting totally the original numbers) in the elements.

b) Then in each leaf at the bottom of the recursion tree representing a section of length > 1 in **a**; sort on that subsection of indexes (such a leaf represents the indexes of some equal valued elements).

c) Phase b) is done by a recursive call to the IPS stable algorithm (setting the parameters such that this will not be a first sorting digit) for each such subsection. The Stable IPS will then do an unstable ALR sort on the indexes of each such subsection, but since all indexes are unique, this will also be a stable sort.

Phase b-c) should also be optimized with Insertsort when sorting a short subsection.

Program3. *The pseudo code for the Stable IPS sorting algorithm presented as modification of the Unstable IPS algorithm.*

The sorting permutation and IPS

The second feature of these two algorithms is that they do this transformation from an array of keys to the permutation array with no extra space needed, except what is needed for the recursion stack. Each method body on that stack, in addition to some simple variables, contains a single integer array of length $k=2^{numBit}$. In all but extreme situations k is less or equal to 2^{10}. The extreme situation is when n is large (say $n>10^7$) *and* where m (= max a[i]) is at least equal to n. Then k and the first sorting digit have to be large (figure 1) to contain the index numbers.

Why optimize on space one could ask. Modern computers usually have enough. There are two reasons. First is that algorithms that use less space are generally faster because of the cached memory system. The second reason is that some systems like the JVM (Java Virtual Machine) has a fixed sized heap, and if a sort of 10M numbers suddenly asks for 10 or 20 M extra integers (80Mb), then your program might run out of heap space and abort.

One might also argue that transforming and hence destroying the contents of an array is extreme. Almost always do you need to keep the content of the original array and in addition get a permutation array. Here there are two answers: First, you could always take a copy of the original array before calling IPS (System.arraycopy in Java is very fast) and then call IPS with one of them - which would still be fast. Alternatively, you could make a special version of your favorite sorting algorithm where you first allocated an array p of length n, initialized it with 0,1,2,…,n-1 and then did all the same exchanges on p as you do when sorting a. IPS would however be faster than this last alternative, and in addition, in the first alternative you would have the advantage of both having a sorting permutation and having the array a in the same order as the accompanying arrays: b,c,..d.

Two alternatives to make the sorting permutation, are either to sort by rearranging the elements in the accompanying arrays b,c,.,d together with the elements in a in an ordinary sort, which takes a heavy time penalty, or alternatively, store all the information belonging to each logical unit in objects and then sort these objects based on the key inside each object.

The generation of the sorting permutation is not, however, a new idea. Even though I am not able to find any published paper on it, nor finding a published algorithm, at least two public software libraries have these functions [HPF, Starlink/PDA]. The implementations of these library functions are not (easily) available.

From evolutionary studies in molecular biology, however, there is an active research into sorting permutations. Their problem is to find by the minimum number of reversals one has to perform on a given permutation to get back to the unity permutation (1,2,..,n). In short, they pose the question of how many changes do we have to perform to genes in one organism to get to the same sequence of these genes in another organism [18,19, 20]. The problem in this paper is not totally the same. I make no claim that the number of exchanges made in the IPS algorithms is minimal. These papers on the other hand, construct algorithms that assume a main memory where every location has the same access time, thus avoiding the problem and opportunity of caches in any modern computer.

Some comments on implementing the IPS algorithms

The implementations of the two IPS algorithms were done in Java and are available for free download at the author's website [21]. Both algorithms has, like Quicksort, an interface routine, that either calls a PermInsertsort routine if $n < 30$ (optimization) or else finds the maximum value in the input array and hence the maximum bit set in any element in the array. Then it calls the recursive IPS sorting method. The Unstable IPS is some 160 lines of code (LOC), and uses MaskedInsertsort, a version of Insertsort that sorts only on the bits specified by a mask, for optimization. The Stable IP is somewhat longer, 190 LOC, since it has an extra sorting phase at the end. Stable IPS also uses an ordinary Insertsort to optimize this last sorting on indexes.

Testing the Unstable and Stable IPS algorithms

To test the running time of these two algorithms, the three factors that are found to influence the execution time of integer sorting algorithm the most: n, the length of the array, the distribution of the numbers to be sorted and the type of CPU (with different caching and memory channel schemes), are tested.

This is presented as two sets of tests, first the standard distribution U(n) , uniformly distributed numbers in the range 0:n-1, are tested on four different CPUs for n=10, 50, 250,..,97M (19M in the case of a Dell D400 laptop with a Intel CoreDuoT2500 1.2 GHz CPU, due to limited memory). For every n shorter than the maximum, a number of shorter arrays were sorted, the sum of which was the maximum number. The figures are then the average these shorter arrays. The second test was using 14 different distributions of the numbers to be sorted on one of the CPUs, the AMD 254 Opteron.

Testing Unstable and Stable IPS on 4 different CPUs

We have here tested the U(*n*) distribution for *n*= 10, 50, 250,…,97M as described above for four different CPUs:

a) AMD Athlon 64X2 4800 (2.49GHZ) dual core each with a 64KB L1cache and a 1 Mb shared L2 cache
b) Intel Xeon E5320(1,86GHz), quad core, each with a 64KB L1 cache and two 4MB L2 caches, each shared between two of the cores.
c) Intel CoreDuo T2500 (1,2 GHz) dual core, each with 64KB L1 cache and 1MB L2 cache (separate)
d) AMD Opteron 254 (2,8 GHz) single core, 128 KB L1 cache and 1MB L2 cache

For the quad and dual core CPUs, only one core is used. Since the tests were performed on stand alone machines, almost all of the caches can be assumed occupied with sorting.

We see in Figure 2 that the IPS algorithms perform almost identically on three out of the four CPUs relative to sorting the same data with Quicksort on the same CPU. On the fourth CPU, the AMD Athlon, the IPS algorithms perform somewhat poorer compared with Quicksort. However, it is safe to say that Unstable IPS is clearly faster than Stable IPS by some 30-40%, and that Stable IPS again is by and large twice as fast as Quicksort on any CPU for 50 < n < 4M for the U(n) distribution. For larger values of n, up to 100M, both algorithms are up to 50% slower on three of the CPUs and 120% slower on the AMD Athlon. The reason for this degradation for large values of n is simple: To make room for the index values, the first sorting digit has to be made so large that the point array does not fit into the L2 cache, and we get too many cache misses to main memory, each of which takes at least a factor 100 times longer [12] than accessing an already cached element.

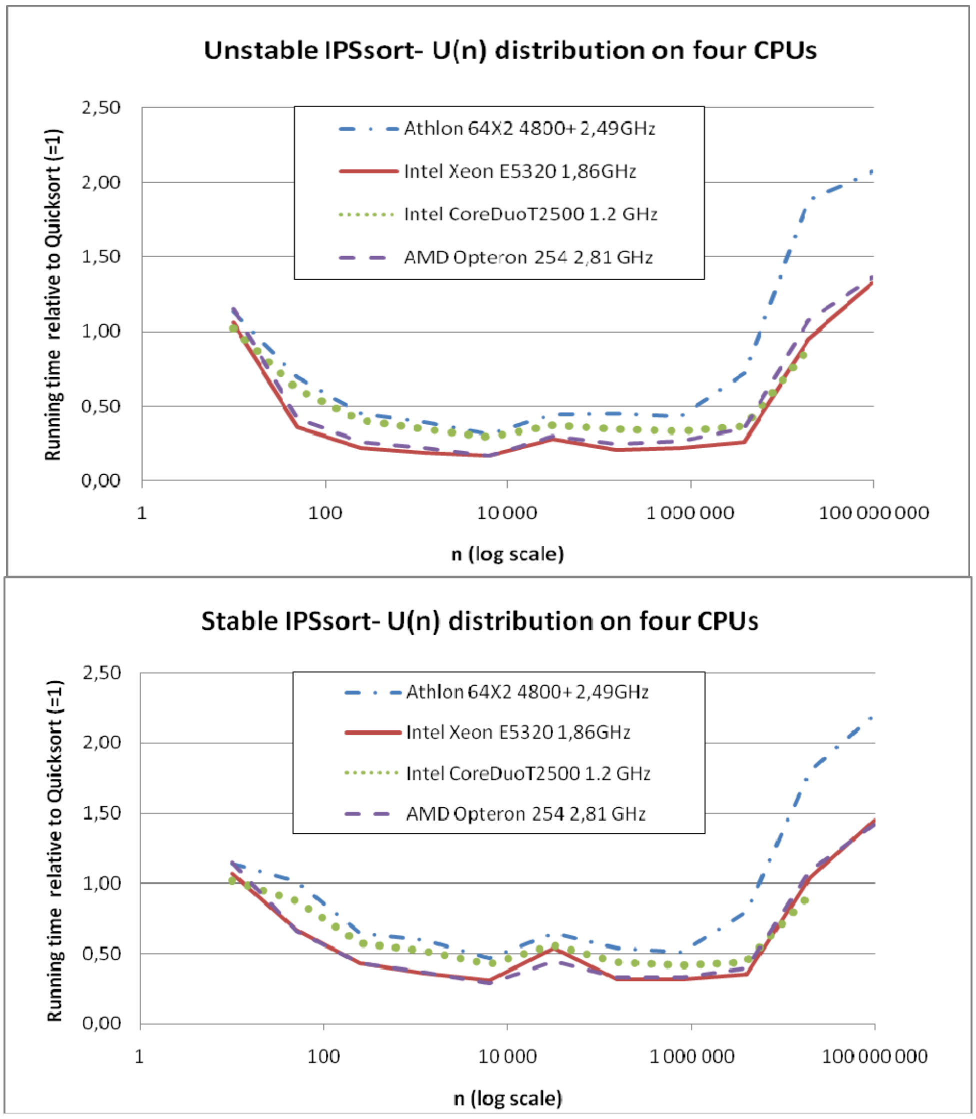

Figure 2 a) and b). *The Unstable and Stable IPS algorithms tested with the U(n) distribution - uniformly distributed numbers in the range 0:n-1 for 4 different CPUs. Running times are plotted relative to use Quicksort on the same CPU and data, with n =10,50,250,…,97M (19M for the Intel Core Duo). Values lower than 1 is faster than Quicksort.*

Testing Unstable and Stable IPS for 14 different distributions

The fastest CPU, the AMD 254 Opteron is used to test 14 different distributions of the numbers to be sorted. There are two dense distributions with an expected fixed number of duplicates per value (U(*n*/10) and U(*n*/3)), a number of distributions where the there are basically *n* numbers in the range *0:n*-1 to sort (the U(*n*), the permutation of *0..n-1*, the sorted and reverse sorted and the almost sorted distribution). Then there are some thin distributions (U(3*n*), U(10*n*), U(0:2^{30}), and the Fibonacci numbers starting with first 1,1 then 2,2,.., which is an exponential distribution).

Finally, there are distributions with a limited number of possible values (i%3, i%29 and i%171, randomly placed, i=0,1,..,n-1). These last tree distributions are quite common in practice when you sort a large collections of elements, say people, on a key with a fixed limited range, often determined by some classification. E.g. sorting the population by county, height, age or (as an extreme case) by gender (male/female). In the most extreme case tested, the i%3 distribution, we have more than 32M equal valued elements when *n* = 97M, and it will later be commented on why this distribution stands out when comparing Quicksort and IPS.

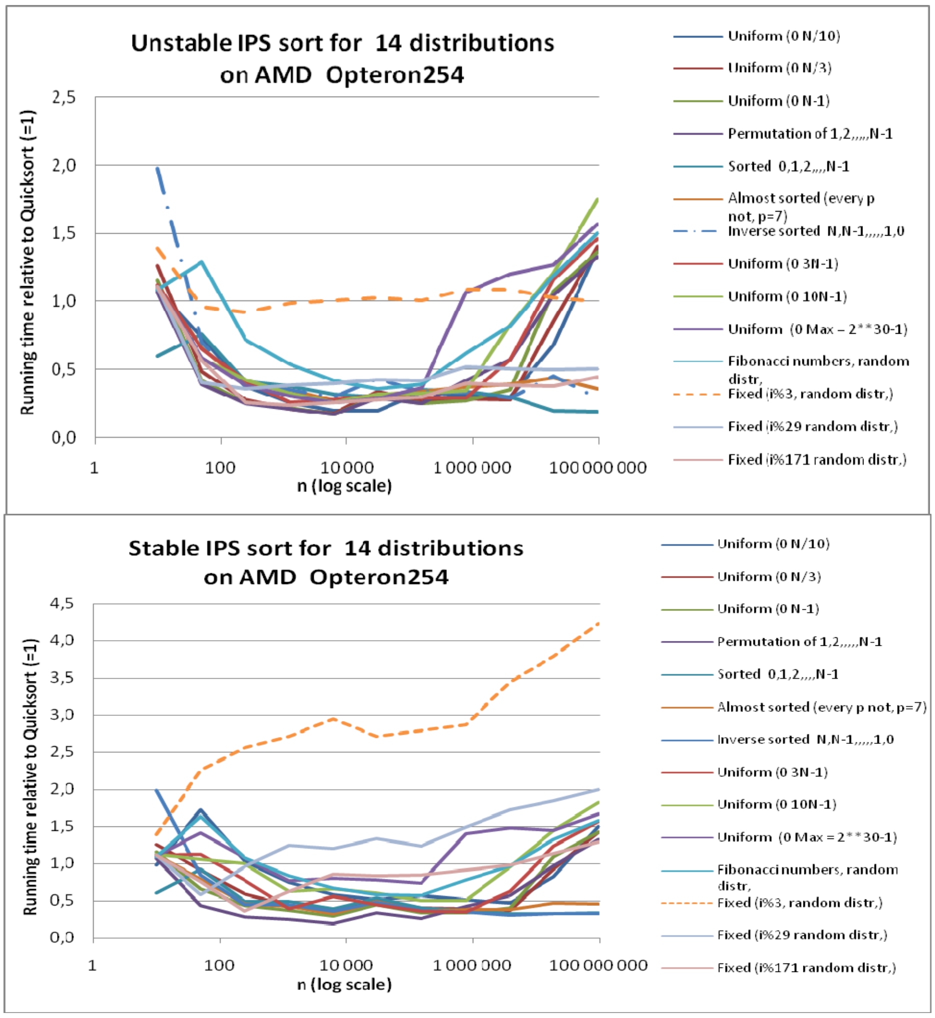

Figure 3 a) and b). *The Unstable and Stable IPS algorithms tested 14 different distributions as described in the text on a AMD 254 Opteron with n= 10,50,250,…,97M. Running times are relative to sorting same data with Quicksort (lower than 1 is better).*

We see that for almost all, but one of the distributions, we observe as with the previous test, that Unstable IPS is some 30%-40% faster than Stable IPS which again is some 30% -40% faster than Quicksort for 50 < n < 4M. For values of n approaching 100M, both algorithms need both more time and space. The reason for needing more time is the effects of caching. In order to make room for large index values, the IPS algorithms are forced to use a large first sorting digit. The point array that count how many there are of each value, does not fit into the largest cache, and we get many cache misses. The size of this array is then such that the claim in the title of this paper then does not quite hold. To be short, the IPS algorithms deteriorate for values greater than 10M.

Note that the 14 graphs in figures 3 a and b does not show absolute processing speed, but the running times relative to Quicksort sorting the same data. It seems that IPS is slow for the i%3 distribution where there are only tree different values to sort. To see that this is actually not the case, we first have to see the absolute running time of how Quicksort handles the standard test case, the U(n) distribution, and the i%3 distribution on two CPUs in Figure 4.

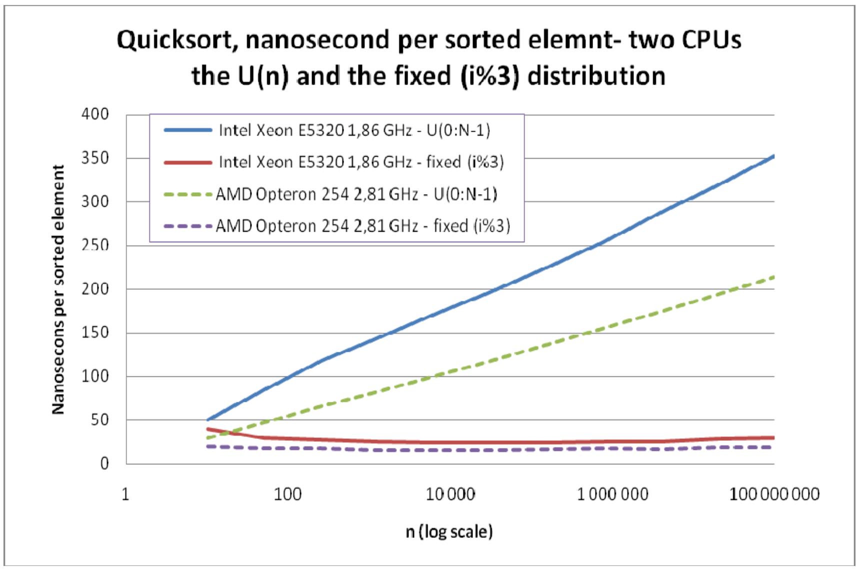

Figure 4. *The absolute execution time of the Quicksort algorithm for two different distribution on the AMD 254 Opteron and Intel Xeon E5320 CPU with n= 10,50,250,…,97M. Running time is plotted as nanoseconds per sorted element.*

We see first that the AMD 254 Opteron is a faster CPU than the Intel Xeon E5320, but more interestingly, Quicksort is much faster and sorts any array in O(n) time for the i%3 distribution, but have an O(n logn) behavior with the U(n) distribution.

The Unstable IPS algorithm shows similar performance curves for the i%3 distribution, but does not improve so much as Quicksort and ends up 'only' just as fast as Quicksort.

The fact that the Stable IPS algorithm is so much slower than the unstable Quicksort is easily explained. First, Stable IPS does an Unstable IPS of these three numbers, using the same time as Quicksort, but then afterwards has to sort three sets of indexes, each n/3 in size. And since they contain unique numbers in the range 0:n-1, this is equivalent to sort three times a U(n) distribution of length n/3, each of which takes O(n log n) time (figure 3), and Stable IPS hence has an overall execution time of O(n log n) . This will of course not be as fast as the O(n) curve for Quicksort for this distribution, when n gets large.

Note also that Stable and Unstable IPS are faster than Quicksort for the i%171 distribution and only Stable IPS gets somewhat slower for the i%29 distribution for the same reasons as mention above for the i%3 distribution. But now each segment of indexes for equal valued elements is much smaller.

Using IPS sorting for other types of data

Sorting a mix of positive and negative numbers can be handled by the IPS algorithms. A trivial solution is finding *s*, the smallest number in *a*, and if that is negative, add –*s* to all elements, then sort only positive numbers, and afterwards add *s*. Note that if *s* is a large positive number, the same subtraction by *s* on all elements to make room for the substitutions would also be beneficial in cases where also the maximum value *m* and *n* also are large,.

Sorting floating point numbers as integers give a correct result because of the IEEE definition of floating point numbers is designed with this in mind This is possible in languages as C but not in type safe languages like Java. Also, a substitution algorithm like IPS would work well with strings because substitutions in strings are easy. Knuth [6] also claims that any integer sorting algorithm is generally well suited for sorting texts and floating point numbers.

Conclusions

Two algorithms for inline generating the sorting permutation have been introduced. Both rely on the observation that, when we have read the first sorting digit, that digit is no longer used for any other purpose and that the space it occupied, can be used for holding other information. The Stable IPS also uses the observation that an unstable sorting like Unstable ALR can be used to achieve a stable sorting because all index numbers are unique.

We have also observed that IPS is generally faster than Quicksort, and for values less than 10^7 only uses extra space of O(log n). Using little extra space is advantageously in two respects, there is less chance that we hit some fixed memory limit and get aborted. Secondly, algorithms with a smaller working memory demand are generally faster because of the effects of caching[16].

Acknowledgements

I will like to thank Stein Krogdahl and Birger Møller-Pedersen for commenting earlier versions of this paper.

Bibliography

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

[2] Jon L. Bentley and M. Douglas McIlroy: *Engineering a Sort Function*, Software-Practice and Experience, Vol. 23(11) P. 1249-1265 (November 1993)

[3] Wlodzimierz Dobosiewicz: *Sorting by Distributive Partition,* Information Processing Letters, vol. 7 no. 1, Jan 1978.

[4] http://en.wikipedia.org/wiki/Sorting_algorithm

[5] C.A.R Hoare : *Quicksort*, *Computer Journal* vol. 5(1962), 10-15

[6] Donald Knuth :*The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Second Edition. Addison-Wesley, 1998.

[7] Anthony LaMarcha & Richard E. Ladner: *The influence of Caches on the Performance of Sorting,* Journal of Algorithms Vol. 31, 1999, 66-104.

[8] Jan van Leeuwen (ed.), *Handbook of Theoretical Computer Science - Vol A, Algorithms and Complexity,* Elsivier, Amsterdam, 1992

[9] Arne Maus, *Sorting by generating the sorting permutation, and the effect of caching on sorting*, NIK'2000, Norwegian Informatics Conf. Bodø, Norway, 2000 (ISBN 82-7314-308-2)

[10] Arne Maus. *ARL, a faster in-place, cache friendly sorting algorithm*. in NIK'2002, Norwegian Informatics Conf, Kongsberg, Norway, 2002 (ISBN 82-91116-45-8)

[11] Stefan Nilsson, *The fastest sorting algorithm*, in Dr. Dobbs Journal, pp. 38-45, Vol. 311, April 2000

[12]John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 4th edition, 2007. ISBN 978-0-12-370490-0.

[13] Robert Sedgewick, *Algorithms in Java*, Third ed. Parts 1-4, Addison Wesley, 2003

[14] Mark Allen Weiss: *Data structures & Algorithm analysis in Java 2/E,* Addison Wesley, Reading Mass.2007

[15] Arne Maus, *Making a fast unstable sorting algorithm stable,* in NIK'2006, Norwegian Informatics Conf, Molde, Norway, 2006 (ISBN 978-82-519-2186-2)

[16] Arne Maus and Stein Gjessing: *A Model for the Effect of Caching on Algorithmic Efficiency in Radix based Sorting,* The Second International Conference on Software Engineering Advances, ICSEA 25.Aug. 2007, France

[17] Naila Rahman and Rajeev Rahman, *Adapting Radix Sort to the Memory Hierarchy*, Journal of Experimental Algorithmics (JEA) ,Vol. 6 , 7 (2001) , ACM Press, New York, NY, USA

[18] H. Kaplan, R. Shamir and R.E. Tarjan, "Faster and Simpler Algorithm for Sorting Signed Permutations by Reversals", Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (1997), ACM Press.

[19] Caprara, A., Lancia, G., and Ng, S. 2001. Sorting Permutations by Reversals Through Branch-and-Price. INFORMS J. on Computing 13, 3 (Jun. 2001), 224-244.

[20] Feng, J. and Zhu, D. 2007. Faster algorithms for sorting by transpositions and sorting by block interchanges. ACM Trans. Algorithms 3, 3 (Aug. 2007), 25.

[Starlink/PDA] see software | programming | mathematical at : http://star-www.rl.ac.uk/

[21] Arne Maus‘ sorting homepage: http://heim.ifi.uio.no/~arnem/sorting

[22] Henrik Blunck and Jan Vahrenhold. In-Place Algorithms for Computing (Layers of) Maxima. In: Lars Arge and Rusin Freivalds, editors: Proceedings of the 10th Scandinavian Workshop on Algorithm Theory (SWAT 2006), volume 4059 of Lecture Notes in Computer Science, pages 363-374. Springer, Berlin, 2006.