International Journal of Computational Intelligence and Informatics, Vol. 5: No. 3, December 2015

A Novel Approach of Implementing Radix Sort by Avoiding Zero Bits Based on Divide and Conquer Technique

T Mathialakan
Department of Electrical and Computer Engineering
Michigan State University
East Lansing, MI, USA
mathialakan@gmail.com

S Mahesan
Department of Computer Science
University of Jaffna
Jaffna, Sri Lanka
mahesans@univ.jfn.ac.lk

***Abstract*-Sorting plays a major role in theoretical computations and makes data analysing easy. Even though existing algorithms show good performance in time and space consumption, new techniques are still being fabricated. An efficient non comparative, individual digit clustering, integer sorting algorithm, radix sort seems to be taking $O(n)$. However, in general, it takes $O(n\log(n))$ as the best performance comparison based algorithms do. In order to achieve a better performance algorithm than existing ones, we introduce an inspiring technique based on radix sort. This technique divides set of decimal integers in two groups with respect to bits in the same position in the binary representation. This steps is repeated within each subgroup and the process is continued until the list is sorted. Most significant non zero bit of maximum value is used as pivot bit for partition in each step and avoids to run through leading zero bits. This consumes $O(cn)$ time where c is significantly less than$\log n$.**

Keywords-divide and conquer technique, time complexity, radixes.

I. BACKGROUND

Sorting is a technique that arranges the elements in ascending or descending order. It is important to data canonicalization and algorithm optimization. A set of elements can be ordered numerically or lexicographically. Most of these algorithms compare elements on the entire list in a certain way in order to sort. These comparing and exchanging processes are executed iteratively or recursively based on the approaches adopted. The fundamental comparison-based sort, such as bubble sort or selection sort takes $O(n^2)$ time in average/worst case [1]. However, $O(n\log(n))$ is an achievable worst time by some comparison-based algorithms (e.g. merge, heap). The same time order of $O(n\log(n))$ is taken by quicksort and heapsort in average case [2], [3]. In best case, a non-comparative sorting algorithm radix sort can have an excellent performance with$O(n)$time, but it takes $O(n\log(n))$ time in average case [4]. We can realize usage of this sort in several applicable areas [5] and new techniques were developed frequently based on it [6]–[8], specially parallel versions which support high performance and GPU computing [9]–[14]. We are going to look at an overview of fundamental radix sort, and radix exchange sort, then explain our new approach followed by test cases and discussions.

A. Radix Sort

A radix sort deals with individual digits to sort integers.It examines digits which are in same position and sortsintegers based on those digits, in a certain way. In the case of decimal numbers, there are ten possible digits that need to be considered in comparison.It therefore requires another algorithm to organize the individual digits.Algorithm of radix sort can be simply written as follows:

$$\textbf{for } k := 0 \textbf{ to } d-1 \quad (1)$$

Sort the array of integers of d digits in a stable way, looking at k^{th}digit only.

The running time depends on the stable sort that is used to sort digits at each position. Suppose it takes$O(n)$, then the total time complexity will be$O(dn)$. Sorting begins from least significant or most significant digit based on approaches adopted:

Least Significant Digit radix sort (LSD): It starts from the least significant digit and moves the processing towards the most significant digit (that is from the right most bit to the left most bit). Thissort typically works as follows: *e.g.* consider the array of six 2-digit numbers `7, 12, 11, 23, 4, 20`. Sort by the digit at position zero, `20, 11, 12, 23, 4, 7` and then sort by the digit at position 1, to get `4, 7, 11, 12, 20, 23`. We need to consider as a bulk of numbers which have same digit at the processing position, and move the bulk

without collapsing inside during an arrangement at a stage. So, in this example the numbers 4 and 7 have zero at position 1 and move together without reordering themselves, similarly (11, 12) and (20, 23) keep their order within their bulk.

Most Significant Digit radix sort (MSD): It starts from the most significant digit and moves towards the least significant digit. MSD radix sorts use lexicographical order which is suitable for sorting strings such as words, or fixed-length integer representations. Consider the previous example, sort by digit at position 1, and then by digit at position 0: to get 7, 4, 12, 11, 23, 20and 4, 7, 11, 12, 20, 23.

In case of binary number sorting, there are no additional methods needed as they have only two possible digits. In order to use this bitwise comparison technique, integers would be represented in a binaryformat.

Binary level analyzing is a most suitable way for data represented in computers, because they use binaries in their lower level representations. Binary digits of data shall be considered to rearrange their original values in a particular order. Radix sort in binary level is a most applicable technique. Bitwise implementation of radix sort is called as radix exchange sort.

B. Radix Exchange Sort

Array of key elements x_is are partitioned into two groups based on k^{th}bit (*i.e.* bit at k^{th}position) of each step and this partitioning is processed recursively until all bits have been accessed.

$$\boldsymbol{for}\ k = d - 1\ \textit{to}\ 0\ \boldsymbol{do}$$

repeat
scan top-to-bottom to find a key having **1***at position* ***k****;*
scan bottom-to-top to find a key having **0** *at position* ***k****;*
exchange keys;
until *scan indices cross;*

It works like partitioning in quicksort, but it uses bit value in a significant position as a pivot element in each step. Elements which have 1 at the current bit are moved into one group and others will be into another group. E.*g*., let us sort a set of binary numbers $s = \{110, 010, 100, 111, 011, 101, 001\}$. Starting with the most significant bit at $k = d - 1(= 2)$ (assuming least significant digit index is zero), s is split into two sets s_0 and s_1 while keeping the relative order amongst those numbers having the same bit in position k : $s_0 = \{010, 011, 001\}$ and $s_1 = \{110, 100, 111, 101\}$, as shown in the tree of Figure 1. This process continues for bits at each bit position, until each set contains only one element.

Considering the bit at position 1,

$s_0$becomes s_{00}={001} and s_{01}={010,011}.
while $s_1$becomes s_{10}={100, 101} and s_{11}={110, 111}

Considering the bit at position 2,

s_{00}needs no further split as it is only one element,
s_{01} becomes s_{010}={010}, and s_{011}={011}
s_{11} becomes s_{110}={110}, and s_{111}={111}

Finally the sorted list obtained by collecting the leaf-nodes: left-to-right (for ascending order) or right-to-left (for descending order).

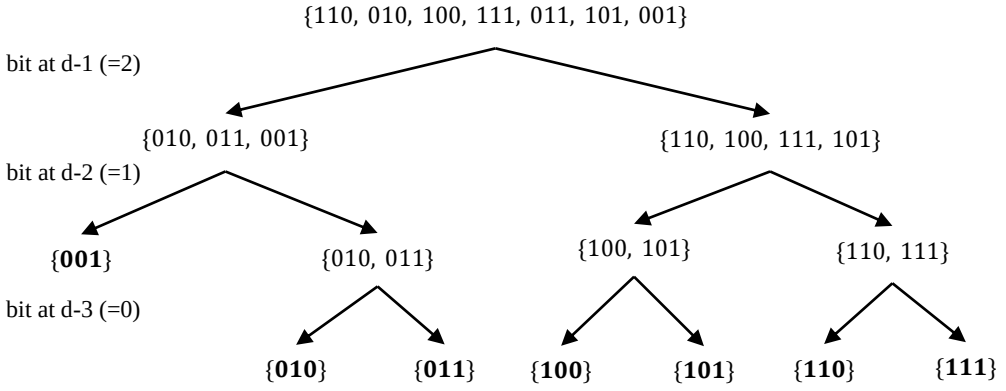


Figure 1. Tree Depicting Recursive Partitioning

The time complexity of this approach is $O(bn)$ where b is the number of digits, b may go up to 32 or 64 depending on the type of integer. Another important fact is range of given numbers. Comparisons shall be dodged substantially when the difference between maximum value and minimum of value of the array elements is large while the number of elements is small. Avoiding leading zeros would save the time. *E.g.* suppose s, (=`{000110, 000010, 000100, 000111, 000011, 000101, 000001}`) has the same numbers but in six digits with leading zeros as necessary. At any stage, the first three leading zeros can be avoided from comparison. The maximum value in the set of integers influences the amount of digit comparisons.

We have adopted a new approach on radix sort that avoids leading zero bits in comparison. In this approach, the number of digits,

$$b = \lceil \log(max_{0 \le i < n}(x_i)) \rceil \quad (2)$$

The time complexity would be $O(n)$if $b \ll n$.

II. OUR SCHEME

Take the most significant non-zero bit of the maximum into pivot and make partition by pivot bit until each partition has a single element. Each partition is further partitioned into two and so on until no more partitioning is needed. When a partition has no more than one element no partition is required. At each step partitioning is based on the next left most significant 1-bit among the set to be partitioned - that is, the bit of the largest element in the set. The leading zeros in binary representation of entire integers would be skipped without comparing.

How to make partition in each step? Figure 2 shows this clearly with an example. The number set `{154, 18, 22, 5, 1, 0}`is initially partitioned by the bit at position 7 for `154`, got modular values `{26, 18, 22, 5, 1, 0}`(these are given in parenthesis in Figure 2) and then by bit at position 4 is used to partition the set `{ 18, 22, 5, 1, 0}`because bits at positions 6 and 5 are zeros in each number in the current set, so here, first non-zero bit at bit position 4. Now we have modular set `{2, 6, 5, 1, 0}`and it is partitioned into two sets `{5, 1, 0}`and `{2, 6}`. Similarly bits at positions 4 and 3 are skipped in next step and both partitions `{5, 1, 0}`and `{18, 22}`are split by the bit at position 2. Finally leaf-nodes in the tree from left to right give numbers in ascending order.

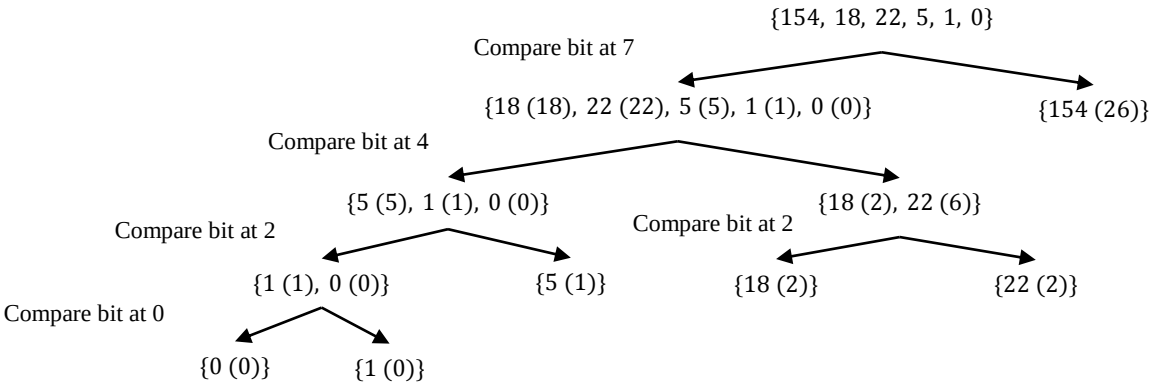


Figure 2.Bitwise partition based sorting

We have an array of integers, say$a[0:n-1]$, with fixed size $n \geq 0$that has to be sorted in ascending order. A sequence of pairs that represent the range of integers maintained through the looping. Integers within the range(f,l) are partitioned using the pivot bit into two sub-ranges, (f,i) and $(i+1,l)$ until ranges with one element, which will need no further partitions. The loop terminates with the comparison using the last bit. The partitioning is performed with the left most bit of the maximum value within each sub-range. This iteration starts with $s:=((0,n-1))$. The following program adopts the notations used by David Gries [15].

Program:

```
{Precondition: n≥2}
s:=((0, n-1));
{Invariant: s is a sequence of pairs (i, j) representing disjoint array sections
a[i:j] of a[0:n-1].It is ordered if and only if all the disjoint partitions given
in sequence are.}
{Bound function: (max(0,n-1))-size(s) }
do s≠()→(f,l),s:=s[0],s[1..];
   i,j,b:=f,l, lmb(max(f,l));
if j-i≤1 → skip;
```

```
   j-i>1 → {Invariant: bit(a[f:i-1],b)=0 ∧bit(a[j:l],b)=1}
{Bound function: -i }
do (i<j)∧(bit(a[i], b)=0)→i≔i+1;
(i<j)∧(bit(a[j],b)=1) → j≔j-1;
(i<j)∧~((bit(a[j],b)=1)∨(bit(a[i],b)=0)) → a[i],a[j],j≔a[j],a[i],j-1;
od
   s≔(f,i-1)|(i,l)|s;
fi
od
{Post condition: ∀i:0≤i<n-1:a[i]≤a[i+1]}
```

Predicate *max* is used to infer maximum value within a range in the array *a*. The *lmb* gives the bit position of the left most 1-bit of integer, and the *bit* yields a bit value at specific bit position of a given integer. According to the bound functions of loops, the complexity is of $O(cn)$, whenever $c < b$

This approach is implemented in two ways based on the technique using to keep track of bit position. First way reduces the time taking to perform arithmetic operations. Second way focuses on memory consumed by the program code.

A. Way 1

In this way, an additional array is used to keep modular value, the remainder (numbers denoted within brackets in Figure 2 yield by dividing non pivot bit,calculated in each step.

Algorithm

a:array of elements to be sorted.
ma: array to keep modular values, initialized with zero.
pd: dth place value for divisor.
DDSMax (f, l) *{*
i =f
j =l-1
size =l-f
max = $Max_{i \le k \le j}(a_k)$
if (max > 2^e *and max ≤* 2^{e+1} *) pd =* 2^e
if (size > 1) and (pd ≥ 1) {
while (i < j) {
while((i < j) and ($\frac{ma_i}{pd}$ *==0)) i = i+1*
while ((i < j) and ($\frac{ma_j}{pd}$ *==1)){*
$ma_j = ma_j \bmod pd$
j =j-1
} endWhile
if (i< j) {
$ma_i = ma_i \bmod pd$
swap (a_i *,*a_j*)*
swap (ma_i*,*ma_j*)*
j =j-1
} endIf
} endWhile
} endIf
if (ma_i*/ pd == 0) i = i+1*
else $ma_i = ma_i \bmod pd$
DDSMax (f,i)
DDSMax (i,l)
}

The needed memory space depends on the size of the array used to maintain runtime data. The occupied memory is doubled (2*n*) by using two arrays: *a*, *ma* with size of *n*. In order to minimize the memory consumption, we introduced the second way without using an additional array.

B. Way 2

Specific bit is taken in each step by locating the position.

Algorithm

```
a:array with elements that will be sorted.
d: most significant set(1) bit of max.
maxbit: the most significant non-zero bit of maximum on the previous stage, initialized with 31.

DDSMax ( f, l, maxbit ) {
        i =f
        j =l-1
        size= l-f
        max = MaxAtStep(f,l,maxbit)
        d = Maxbit(max)
        if (( size>1)  and (d≥0)) {
                while ( i< j) {
                        while ((i<j )  and ! IsBitSet(a_i,d)))      i=i+1
                        while ((i< j)  and IsBitSet(a_j,d)))       j =j-1
                        if ( i< j) {
                                swap( a_i,a_j)
                                j =j-1
                        }  endIf
                } endWhile
        } endIf
        if !IsBitSet(a_i, d)       i = i+1
        DDSMax (f,i,d)
        DDSMax (i,l,d)
}
```

Inthis way, additional calculation is needed to find maximum using the methodMaxAtStep in each stage. This method uses most significant bit of the maximum value on previous stage.

III. RESULTS AND ANALYSIS

We tested the algorithm implementing it in C# with randomly generated array of integer, in various sizes. The number of iterations are calculated for each test case analysed for performance evaluation. The number of elements (say *n*) and the number of bits in the maximum integer (say *b*) are the key facts in this time complexity analysis. Each case is tested with 20 sets of elements and the average (***t***) of execution times of these 20 sets is considered.

Computing time is calculated for each of the following two cases:

Case-1: *n* varies, *b* is fixed.
Case-2: *n* is fixed, *b* varies

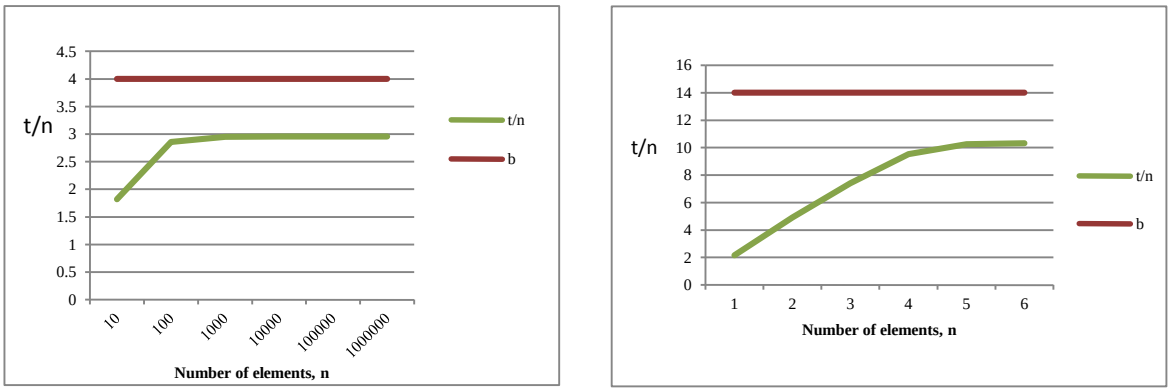
A. Case-1:Time Over n with Fixed b

In this case, time over n is calculated for two different instances of b – one, b is fixed as 4 and the other, fixed as 14, for varying n from 10 to 1,000,000 with the maximum of integer value being 14 (when $b = 4$) and being 9999 (when $b = 14$). Time complexity for conventional radix sort, $T_n = O(bn)$. Therefore T_n is proportional to bn and thus b is proportional to T_n/n

TABLE I. EXECUTION TIME FOR VARIOUS NUMBER OF ELEMENTS

n	**Average time (*t*) of 20 sets each of size *n***		***t/n***	
	b = 4	***b = 14***	***b = 4***	***b = 14***
10	18.2	21.7	1.82	2.17
100	285.65	492.35	2.8565	4.9235
1000	2952.15	7416.3	2.95215	7.4163
10000	29578.1	95268.35	2.95781	9.526835
100000	295874.25	1026028.9	2.9587425	10.260289
1000000	2958931.2	10326042.15	2.9589312	10.32604215

The graph t/n vs n, with $b = 4$ (a), and $b = 14$(b) is shown below:



(a) (b)

Figure 3. t/n versus n with b=4 (a) and b = 14 (b)

According to the above graph, (a), the value $t/n$is less than b. It increases linearly until $n = 100$ and stabilises reaching 3. When $b = 14$,t/n is also less than b, but it stays constant at 10 nearly by $n > 10000$. According to these results, we get a constant value for $t/n(< b)$ when the number of elements (n) goes beyond a particular value depending on b. Let us call this critical value M_b.

When$n > M_b$, t/n is limited to some constant (say c_1).

$$\frac{t}{n} = c_1 \qquad (< b) \tag{3}$$

This is possible when equivalent values are greater with unchangeable maximum value over n. That is a valuable reason for the constant, c_1.

When $n < M_b$,

$$\frac{t}{n} = f(n) \qquad (< b) \tag{4}$$

So, $f(n) = kn$

B. Case-2: Time Over b with Fixed n

Table 2 Time for Different Values of b with Fixed $n = 1000$

TABLE II. TIME VERSUS THE NUMBER OF BITS

Max	*b*	*T*	*t/n*
10	4	2794.2	2.7942
100	7	5150.55	5.15055
1000	10	6689.35	6.68935
10000	14	7435.35	7.43535
100000	17	7428.15	7.42815
1000000	20	7223.5	7.2235
10000000	24	7558.25	7.55825

A graph for *t/n* versus number of bits *b* is shown below:

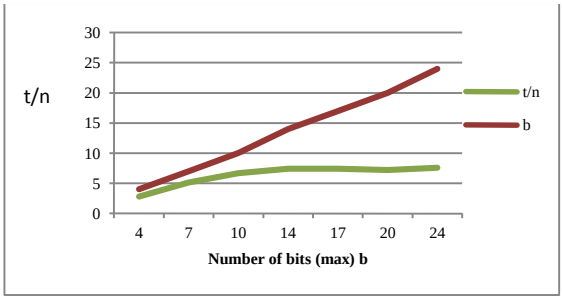


Figure 4. t/n versus b with fixed n=1000

The time maintains at nearly 7.5 over$b = 10$ that is less than *b.*

$n < M_b$,

$$\frac{t}{n} = c_2 \qquad (< b) \tag{5}$$

In general, from these analyses, t/n significantly and sharply increased until *n* is nearly equal to the maximum value. Then it stays at a constant. In this case, our approach achieves excellent performance.

C. Achivement Over Conventional Method

The time taken to sort the elements is reduced significantly using this novel approach. The time ratio between this novel approach and conventional method is analysed over the number of elements, *n.* Figure 5 shows this comparison with b= 4.

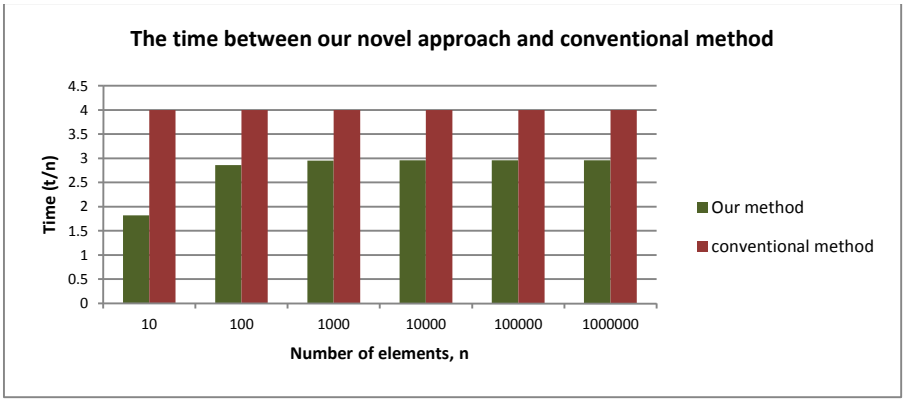


Figure 5. Comparison of t/n of this novel approach and conventional method

We achieved a prominent performance on the radix sort using this novel approach. This reduces the execution time nearly 25%.

IV. CONCLUSION

A better performance radix sort is implemented with a novel approach. This approach successfully overcame the drawbacks: greater time taken by sorting technique used inside of the radix sort and less flexibility. It also provides linear time complexity $O(cn)$ with $c < b$. This novel approach is ideal for elements of wide range.

REFERENCES

[1] J. D. U. Alfred V. Aho, John E. Hopcroft, The Design and Analysis of Computer Algorithms. Addison Wesley, 1974.

[2] V. Muniswamy, Design And Analysis Of Algorithms. I K International Publishing House, 2009.

[3] A. V. G. Baase, Sara, Computer algorithms : introduction to design and analysis, Pearson. 2009.

[4] M. Mcllroy, K. Bostic, and M. D. Mcllroy, "Engineering Radix Sort," Comput. Syst., vol. 6, no. 1, pp. 5–27, 1993.

[5] J. Kärkkäinen and T. Rantala, "Engineering radix sort for strings," Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 5280 LNCS, pp. 3–14, 2008.

[6] K. Hamada, D. Ikarashi, K. Chida, and K. Takahashi, "Oblivious Radix Sort : An Efficient Sorting Algorithm for Practical Secure Multi-party Computation," IACR Cryptol. ePrint Arch., 2014.

[7] T. Hayes, O. Palomar, O. Unsal, A. Cristal, and M. Valero, "VSR Sort: A Novel Vectorised Sorting Algorithm & Architecture Extensions for Future Microprocessors," IEEE 21st Int. Symp. High Perform. Comput. Archit., pp. 26–38, Feb. 2015.

[8] L. Z. G. Ding, "Algorithm for Computing Attribute Reduction Based on Radix Sort of Optimized Linked List Structure," Appl. Mech. Mater., vol. 721, pp. 547–552, 2014.

[9] D. Merrill and A. Grimshaw, "High Performance and Scalable Radix Sorting: a Case Study of Implementing Dynamic Parallelism for Gpu Computing," Parallel Process. Lett., vol. 21, no. 02, pp. 245–272, 2011.

[10] S. Bandyopadhyay and S. Sahni, "GRS - GPU radix sort for multifield records," 17th Int. Conf. High Perform. Comput., 2010.

[11] X. Liu and Y. Deng, "FastRadix : A Scalable Hardware Accelerator for Parallel Radix Sort," Front. Inf. Technol. (FIT), 12th Int. Conf., pp. 214–219, 2014.

[12] K. Zhang, "A Novel Parallel Approach of Radix Sort with Bucket Partition Preprocess," 2012.

[13] D. K. Shin-Jae Lee, Minsoo Jeon and A. Sohn, "Partitioned Parallel Radix Sort," J. Parallel Distrib. Comput., vol. 62, pp. 656–668, 2002.

[14] J. Wassenberg and P. Sanders, "Engineering a Multi-core Radix Sort," Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), pp. 160–169, 2011.

[15] D. Gries, The Science of Programming. USA: Springer New York, 1989.I. S. Jacobs and C. P. Bean, “Fine particles, thin films and exchange anisotropy,” in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

Thavappiragasam Mathialakan was born in Alaveddy, Jaffna, Sri Lanka. He received the BSc (Hons) in Computer Science from the University of Jaffna, Sri Lanka, in 2005, and earned Master in Computer Science from the University of South Dakota, SD, USA in 2013. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, Michigan State University, MI, USA. Previously, He worked as an assistant lecturer and then lecturer in the University of jaffna for more than 7 years. His research interests include computer modelling and simulations, parallel and high performance computing, alogirthm design,and expert systems.

Dr Sinnathamby Mahesan is a senior lecturer, Grade 1 at the University of Jaffna, Sri Lanka, from where he obtained the B.Sc. Honours Degree specialising in Statistics in 1982. He later obtained M.Sc in Computing and Ph.D. Computer Science from Cardiff University, Wales, UK in 1992. Belonging to the Department of Computer Science, Dr Mahesan has been lecturing in many topics including Advanced Algorithms, Artificial Intelligence, Natural Language Processing, Computer Graphics, Image Processing, Database Management Systems, Numerical Computing, Concepts of Programming Languages and many programming languages. His interests get extended to include Machine Learning, Big data analysis, and BioInformatics Computing too. Before joining to the Department of Computer Science, he was in the Department of Mathematics and Statistics lecturing in several topics in Mathematics and Statistics to all level of students from the first year to the fourth year.