IMPROVED RANDOMIZED SIGNATURE SORT: IMPLEMENTATION AND PERFORMANCE ANALYSIS

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Computer Science and Engineering

Submitted By
Tamana Pathak
(800932025)

Under the supervision of:
Dr. Deepak Garg
Assistant Professor

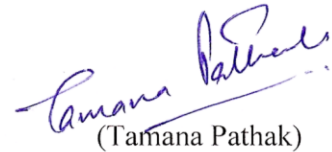
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2011

Certificate

I hereby certify that the work which is being presented in the thesis entitled, "*Improved Randomized Signature Sort: Implementation and Performance Analysis* ", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Computer Science and Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Dr. Deepak Garg* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

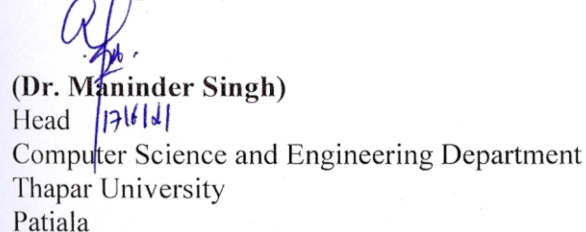


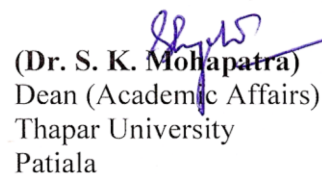
(Tamana Pathak)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Dr. Deepak Garg)
Computer Science and Engineering Department
Thapar University
Patiala

Countersigned by





(Dr. Maninder Singh)
Head
Computer Science and Engineering Department
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

Acknowledgement

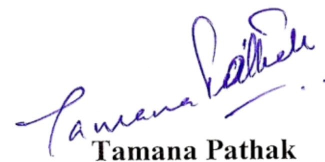
First of all, I submit my thesis at the feet of **Almighty**. I thank Almighty for his blessings and guiding me to the right path. I wouldn't have been able to reach so far without his grace.

It is a great privilege to express my gratitude and admiration toward my esteemed venerable supervisor **Dr. Deepak Garg,** Assistant Professor, Department of Computer Science and Engineering, Thapar University, Patiala. He has been an esteemed guide and moving spirit behind this uphill task. Without his able guidance and painstaking efforts and encouragement, the work wouldn't have been what it is. I am truly grateful to him for extending his total co-operation and understanding when things were not too smooth

I wish to express my heartiest thanks to **Dr. Maninder Singh**, Head, Department of Computer Science and engineering, Thapar University, Patiala for providing me the opportunity and all necessary facilities to accomplish this endeavor successfully.

I am also thankful to my true friends for their help and support in the hour of need.

I find it difficult to pin down my deepest sense of gratiyude toward my parents and sister Purnima, who soulfully provided me their constant support and the right impetus to undertake the challenge.



Tamana Pathak

(800932025)

ii

Abstract

Sorting is one of the most fundamental computational problems, and it is known that n keys can be sorted in $O(n\,log\,n)$ time by any of a number of well-known sorting algorithms. Research done in the area of integer sorting has considerably improved the lower bound and achieved with comparison sorting i.e. $O(n\,log\,n)$ to $O\left(n\sqrt{\log\log n}\right)$ [1] for a deterministic algorithms or to $O(n)$ for a radix sort algorithm in space that depends only on the number of input integers. Andersson et al. [2] presented signature sort in the expected linear time and space which gives very bad performance than traditional quick sort. It is well known that n integers in the range [1, n^c] can be sorted in $O(n)$ time using radix sorting. Integers in any range [1,U] can be sorted in $O\left(n\sqrt{\log\log n}\right)$ time [1]. However, these algorithms use $O(n)$ words of extra memory.

In this thesis, the intent is to present a simple and stable variant of signature sort for integer sorting, which works in $O(n)$ time and uses only $O(1)$ words of extra memory. Also to improve the performance of the signature sort by implementing differently and comparing its performance against traditional sorting algorithms and to see the effect of register size on the algorithm.

Table of Contents

Certificate ………………………………………………………………………….. (i)

Acknowledgement ……………………………………………………………. (ii)

Abstract ………………………………………………………………………. (iii)

Table of Contents ……………………………………………………………. (iv)

List of Figures ……………………………………………………………….. (vi)

List of Tables ………………………………………………………………... (viii)

Chapter-1: Introduction ……………………………………………………........ 1

1.1. Signatures……... ……………………………………………………….......... 2
1.2. Randomized Algorithms……… ……………………………………….......... 3
1.3. Types of randomized algorithms……………………………………….......... 4
    1.3.1. Las Vegas Algorithm………………………………………………… 4
    1.3.2. Monte Carlo Algorithm……………………………………………… 5
1.4. Parallel Algorithm …….. ……………………………………………........ 5
1.5. Integer sorting using signature ……………………………………............ 6
1.6. Structure of Thesis ……………………………………………………........... 7

Chapter-2: Literature Review ……………………………………………........... 9

2.1. Range Reduction 11
    2.1.1. Range reduction scheme 12
2.2. Packed Sorting 12
2.3. Sorting with packed merge 14
2.4. Sorting recursively 15
2.5. Trie construction 15
2.6. Result from existing signature sort 17

Chapter-3: Problem Statement 19

Chapter-4: Improved Randomized Signature sort using hashing and bitwise operators 20

4.1. Machine Model 21
4.2. Signatures 22
4.2.1. Hashing 23
4.3. Packing 23
4.4. Comparison sorting 25
4.5. Unpacking 30

Chapter-5: Testing and Results 32

5.1. Testing 32
5.1.1. Running Time comparison 32
5.1.1.1. Improved randomized signature sort 33
5.1.1.2. Improved randomized signature sort v/existing randomized signature sort 34
5.1.1.3. Improved randomized signature sort v/s randomized quick sort 35
5.1.1.4. Overall Comparison 37
5.1.2. Memory Requirement 38
5.2. Results 41

Chapter-6: Conclusion and Future Scope 45

6.1. Conclusion 45
6.2. Future Scope 47

References 48

List of Publications 51

List of Figures

Figure-1.1: Formation of signatures 2

Figure-1.2: Randomized algorithm 3

Figure-2.1: Packing b-bit integers into w-bit word 13

Figure-2.2: Runtime Comparison between Randomized Quick and Existing Randomized Signature Sort 17

Figure-4.1 Packing of Signatures in a word 24

Figure-4.2: Packing of signatures with extra bits 25

Figure-4.3: Comparison within word 27

Figure-4.4: Word comparison 29

Figure-4.5: Comparison between words 30

Figure-5.1: Runtime plot of improved randomized signature sort 34

Figure-5.2: Runtime Comparison: Improved randomized signature sort v/s existing randomized signature sort 35

Figure-5.3: Runtime Comparison: Improved randomized signature sort v/s quick sort 37

Figure-5.4: Runtime comparison 38

Figure-5.5: Memory requirement of improved randomized signature sort 40

Figure-5.6: Comparison of Memory Requirement 41

Figure 5.7: Random input and signature created for improved randomized signature sort 42

Figure 5.8: Words formed and sorted output of improved randomized signature sorting 44

List of Tables

Table-2.1: Comparison of sorting algorithms 9

Table-5.1: CPU runtime details 33

Table-5.2: Memory comparison details 39

Chapter-1

Introduction

Searching and Sorting, are one of the most important and frequent operations that are faced in many computer applications. That's why researchers have attempted umpteen times in past and still working on it, to develop more optimized and efficient algorithms. The efforts for making algorithms optimized, belongs to the area of time complexity as well as space complexity, by reducing these complexities, efficiency and optimization can be done. Combination of both searching and sorting has become the most used algorithm in computing, and it has been said that time spend by the computer is almost half of the CPU time performing them. So this indicates that if somehow the amount of time of sorting algorithms is reduced then the amount of time spend by computers also reduces as the process of sorting becomes faster. This is the main reason that research is being done on optimizing and making efficient sorting algorithms.

Therefore, the fundamental field of computing is the development of fast, efficient and inexpensive algorithms for sorting and ordering lists and arrays. Thus whole computation can become faster just by making sorting algorithms optimized.

As sorting is one of the most basic and frequent computational problems, and it is known that in order to sort n keys, time taken by the sorting algorithm is $O(n\ log\ n)$ time by any of common comparison based sort algorithms. These comparison based sort algorithms operates in particular settings where they obtain information about the relative order of keys solely through pair-wise comparisons. Integer sorting being sorting of integer keys apart from general keys has always been an important task in applications related to the digital computer. In integer sorting problem, size of elements are generally considered of w-bits.

In 1988, first introduction to the signature sort was given by Andersson, Hagerup, Nilsson and Raman [1]. It sorts n w-bit integers in $O(n)$ time when $w = \Omega(log^{2+\varepsilon} n\ )$

for some $\varepsilon > 0$. Arne Andersson [1] presented signature sort and that was the combination of two techniques i.e. Packed Sorting and Range reduction. Later on, signature sort was transformed into randomized signature sort by using the concept of randomized algorithms with it. Still many researchers are trying to optimize the expected time and memory requirements of signature sort by using different and new concepts in different phases of signature sort [1-5].

1.1 Signatures [1, 5]

In order to decrease the size of integers to be operated upon, signatures are created which have the lesser bit size than the original inputs. To sort n keys where each key is of b bits, q number of logical partitioning is done of each key where each partitioning is of k-bit field i.e. $k = b/q$. A unique signature of $O(log\, n\,)$ bits can be obtained from each field's value by applying a universal hash function. A hash function [1] h: $\{0\ldots\ldots 2^k - 1\} \rightarrow \{0,\ldots\ldots 2^l - 1\}$ where $l = O(log\, n)$, that operates on the set of all fields occurring in the input keys. The hash function $h(n_i)$ is called for each field. If a key P consists of fields $i_1, i_2\ldots\ldots i_q$ the concatenated signature of P is obtained by concatenating the signatures $h(i_1),\ldots. h(i_q)$. The signatures of every field in a key can be computed together in constant time.

In Figure 1.1, Field consists of the inputs keys i_1, i_2 and i_3. $s1, s2$ and $s3$ are the signatures created from the inputs i_1, i_2 and i_3, when hash functions $h(i_1), h(i_2)$ and $h(i_3)$ are applying on inputs.

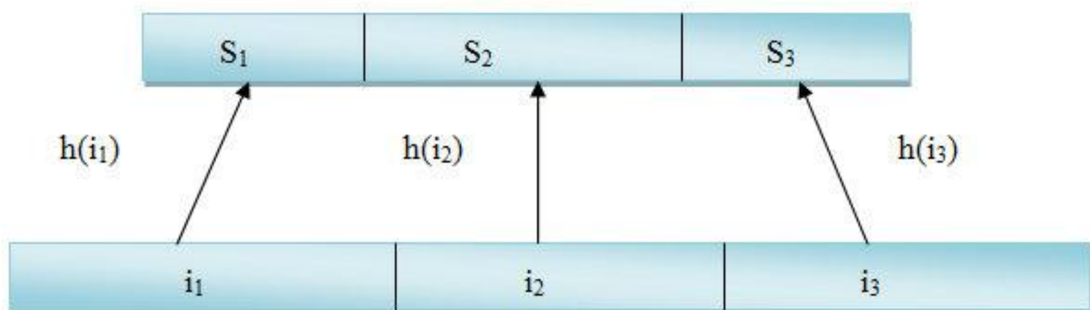


Figure 1.1: Formation of signatures [5]

1.2 Randomized Algorithms

The output of algorithm depends upon the design of the algorithm. Algorithm may be deterministic algorithm or randomized algorithm. The behavior of deterministic algorithm states that the output of the algorithm will be correct for the input and behave same for the same input entered. Also it performs quickly. But sometimes there is an advantage to randomized algorithms, which are non-deterministic, takes a source of random numbers and make random choices during execution. Also, the behavior of randomized algorithms can vary on fixed input too as the random numbers will affect the performance and the result. Randomization may provide a faster or simpler solution than the best known deterministic algorithm.

A randomized algorithm uses a degree of randomness as a part of its logic and these random values guide the behavior so as to obtain good performance in the average-case. In a randomized algorithm the choices are main addition to inputs, algorithm takes a source of random numbers and makes random choices during execution. Behavior can vary even on a fixed input.

Randomized (probabilistic) algorithm can be non-deterministic. They can make random choices but gives correct decisions. The same algorithm may behave differently when it is applied twice to the same instance of a problem. They may be not very precise sometimes. Usually if more time is given then better precision can be obtained.

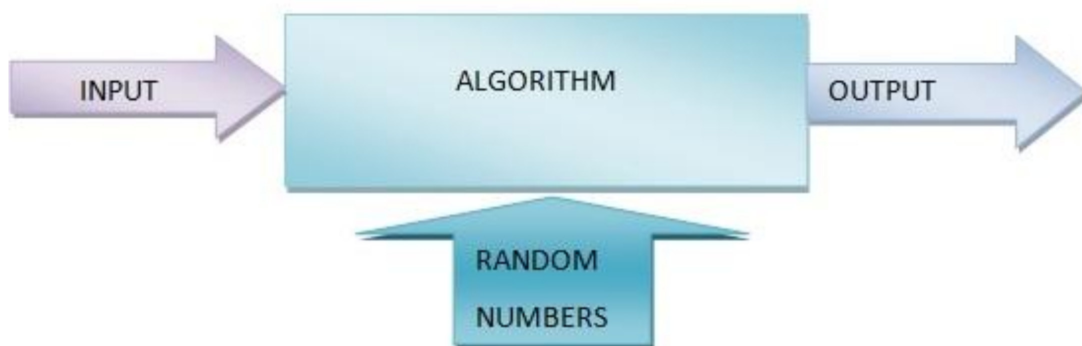


Figure 1.2: Randomized algorithm

A randomized algorithm is a deterministic having a capability of making random choices during the compilation that do not dependent on the input values. By applying randomization, some worst-case situations can be hidden so that the probability of their occurrence becomes small. Therefore the expected runtime is better than worst-case runtime. Two main advantages of randomized algorithm are as: first is performance; they run faster than the best known- deterministic algorithms for many problems; and secondly, they are simpler to describe and implement than deterministic algorithms of comparable performance.

1.3 Types of Randomized algorithms

Randomized Algorithms is mainly categorized into two types, depending on how the algorithm delivers the output with correctness and accuracy.

1.3.1 Las Vegas Algorithm

A Las Vegas algorithm is a randomized algorithm that always gives correct results; that is, it always produces the correct result or it informs about the failure. In other words, a Las Vegas algorithm does not gamble with the verity of the result; it only gambles with the resources used for the computation. A simple example is randomized quick sort, where the pivot is chosen randomly, but the result is always sorted. The usual definition of a Las Vegas algorithm includes the restriction that the expected run time always be finite, when the expectation is carried out over the space of random information, or entropy, used in the algorithm.

Las Vegas algorithms can be used in situations where the number of possible solutions is relatively limited, and where verifying the correctness of a candidate solution is relatively easy while actually calculating the solution is complex. The name refers to the city of Las Vegas, Nevada, which is well-known within the United States as an icon of gambling.

Las Vegas algorithms can be contrasted with Monte Carlo algorithms, in which the resources used are bounded but the answer is not guaranteed to be correct 100% of the time. By an application of Markov's inequality, a Las Vegas algorithm can be converted into a Monte Carlo algorithm via early termination.

1.3.2 Monte Carlo Algorithm

A Monte Carlo algorithm is a randomized algorithm whose running time is deterministic, but whose output may be incorrect with a certain (typically small) probability. The related class of Las Vegas algorithms is also randomized, but in a different way: they take an amount of time that varies randomly, but always produce the correct answer.

A Monte Carlo algorithm can be converted into a Las Vegas algorithm whenever there is a procedure to verify that the output produced by the algorithm is indeed correct. If so, then the resulting Las Vegas algorithm is merely to repeatedly run the Monte Carlo algorithm until one of the runs produces an output that can be verified to be correct.

Whereas the answer returned by a deterministic algorithm is always expected to be correct, this is not the case for Monte Carlo algorithms. For decision problems, these algorithms are generally classified as either false-biased or true-biased. A false-biased Monte Carlo algorithm is always correct when it returns false; a true-biased behaves likewise, mutatis mutandis. While this describes algorithms with one-sided errors, others might have no bias; these are said to have two-sided errors. The answer they provide (either true or false) will be incorrect, or correct, with some bounded probability.

1.4 Parallel Algorithm

A parallel algorithm or concurrent algorithm, as opposed to a traditional sequential (or serial) algorithm, is an algorithm which can be executed a piece at a time on many different processing devices, and then put back together again at the end to get the correct result [6-15].

Some algorithms are easy to divide up into pieces like this. For example, splitting up the job of checking all of the numbers from one to a hundred thousand to see which are primes could be done by assigning a subset of the numbers to each available processor, and then putting the list of positive results back together [6-15].

The performance of a parallel algorithm can be specified by bounds on its principal resources namely, processors and time. Let P denote the processor bound, and T denotes the time bound of a parallel algorithm for a given problem, the product PT is, clearly, bounded from below by the minimum sequential time, Ts, required to solve this problem. It is said that a parallel algorithm is optimal if PT = $O(Ts)$. Optimal parallel sorting for both general and integer keys remained an open problem for a long time. Many optimal algorithms, both deterministic and randomized, for sorting general keys take $O(\log n)$ time. As in the sequential case, many parallel applications of interest need only sort integer keys. Until recently, no optimal parallel algorithm existed for sorting n integer keys with a run time of $O(\log n)$ or less [6-15].

By general keys, it is meant that a sequence of *n* elements drawn from a linearly ordered set s whose elements have no known structure. The only operation that can be used to gain information about the sequence is the comparison of two elements. If each of the *n* elements in a sequence is an integer in the range [1, *nc*] (for any constant *c*), these keys are called as integer keys. General sort is the problem of sorting a sequence of general keys, and integer sort is the problem of sorting a sequence of integer keys.

1.5 Integer sorting using Signatures

Arne Andersson [1] presented signature sort and that was the combination of two techniques i.e. Packed Sorting and Range reduction. Packed sorting was introduced by Paul and Simon [16] and developed further in [6] and [9]saves on integer sorting by packing several integers into a single word and operates at the same time on all of them at unit cost. But this is the case when several integers to be sorted fit in one word.

Range reduction, on the other hand, which underlies both radix sorting and the algorithm of Kirkpatrick and Reisch [17], reduces the problem of sorting integers in a certain range to that of sorting integers in a smaller range. The combination of these two techniques is straightforward: First Range reduction is applied as first task before packed sorting so as to replace the original full-size integers by small integers of which several fit in one word, and these later are sorted by applying packed sorting.

Later, an existing range reduction was combined with an existing packed sorting to obtain deterministic sequential results. Then newly introduced range reduction based on the use of signatures, short unique identifiers for long bit strings; the resulting algorithm, called as signature sort.

In order to sort n keys of b bits each, first conceptual partitioning is done of each key into q k-bit fields where $k = b/q$. Each value occurring in one or more fields can be represented by a unique signature of $O(log\ n\ )$ bits, obtained by applying a universal hash function to the value. The signatures of all fields in a key can be computed together in constant time and their concatenation is an integer of $O(q\ log\ n)$ bits. After sorting the concatenated signatures of input keys in linear time, path-compressed trie is constructed that takes also $O(n)$ time. The trie is a tree of $2n$ edges. Each leaf corresponds to an input key, and edge is associated with a distinguishing signature. All that remains is to sort the siblings below each node in the tree by the original $(b/q) -$ bit field's values corresponding to their distinguishing signatures, since after the operation a sorted output of n input keys can be read-off the tree in a left-to-right scan. Signature Sort by Andersson [1] performs in the expected linear time and space but gives very bad performance regarding CPU running time and memory requirement.

1.6. Structure of The Thesis

The rest of thesis is organized in the following order:

Chapter-2: This chapter will provide the overview of all recent done in area of integer sorting, randomized algorithms, parallel algorithms and integer sorting using signatures.

Chapter-3: This chapter gives the problem statement and methodology used to solve the problem.

Chapter-4: This chapter provides the solution to the problem discussed in chapter-3. This chapter also gives the algorithm for improved randomized signature sort.

Chapter-5: This chapter explains the implementation, testing and result of algorithm given in chapter-4.

Chapter-6: This chapter gives the conclusion of the thesis with the future scope of topic.

Chapter-2

Literature Review

Integer sorting has always been an important task in connection with the digital computer. The table-1 depicts various existing comparison based algorithms and their complexities with relative performance [18].

Table-2.1: Comparison of sorting algorithms

Name	Best	Average	Worst	Memory	Relative Performance
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Average
Shell Sort	$O(n)$	$Depends$	$O(n(logn)^2)$	$O(n)$	Average
Binary tree sort	$O(n)$	$O(nlogn)$	$O(nlogn)$	$O(n)$	Good
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Average
Heap sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(1)$	Good
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Average
Merge sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	Depends	Good
Quick sort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(\log n)$	Good
Randomized Quick sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(\log n)$	Good
Signature Sort	$O(n)$	$O(n)$	$O(n)$	Linear	Poor
Randomized signature sort	$O(n)$	$O(n)$	$O(n)$	Linear	Poor

The sorting problem is to sort n elements, according to a given ordering, has a tight $O(n \log n)$ bound in the comparison model. This lower bound is achievable without randomization.

Sorting integers is a recurring problem in computer science. The question is how fast sorting can be done to sort n integers from $\{0,1, \ldots., u-1\}$ sorting on word RAM with w-bits words, where $b = log\ u \leq w$.

Few important used sorting algorithms and running times [18]:

Comparison sort - $O(n\ log\ n)$

Counting sort - $O(n+u)$, $O(n+u)$ space

Radix sort - $O(n \frac{\log u}{\log n})$

Van Emde Boas's sort - $O(n\ log\ log\ u), O(u)$ space

Algorithms sorting n keys in $O(n\ log\ n)$ time are considered of the comparison-based sorting category [19, 20]. These algorithms operate in the comparison-based settings i.e. they obtain information about the relative order of keys exclusively through pair-wise comparisons. However, this model may not always be the most natural one for the study of sorting problems, since real machines allow many other operations besides comparison.

The field of better than $O(n\ log\ n)$ integer sorting opened up around 1990 when Subsequently, Andersson [1] improved the bound to $O(n\ log\ log\ n)$ time and linear space using a combination of Kirkpatrick and Reisch's range reduction scheme [17] and packed sorting [16]. Nilsson successfully implemented with algorithm and demonstrated it performing well in practice [23]. However, there is some debate to its cache performance, which could point to a future area of study. Very recently the Andersson [1] bound was improved to $O\left(n\sqrt{\log \log n}\right)$ by Han and Thorup [4]. Each paper has a slightly different take on the computational model, reasonable word size, and operations available. Along with the $O(n\ log\ log\ n)$ sorting algorithm [2, 24-26],

Andersson [1] also presents an $O(n)$ expected running time algorithm called signature sort.

The first signature sort was developed in 1988 by Andersson, Hagerup, Nilsson and Raman[1]. It sorts n w-bit integers in $O(n)$ time when $w = \Omega(log^{2+\varepsilon} n)$ for some $\varepsilon > 0$. But the algorithm for the sort was very complicated and had many drawbacks. The sorting algorithm described by Andersson [22] in 1996 achieved signature sort's running time through two main tools, range reduction and packed sorting.

Many researchers have worked on parallel integer sorting [6-11, 27] and lot of work has done to achieve better tight bound for random access machines [16, 28-30].

2.1 Range Reduction

Range Reduction turns a problem of sorting n integers of b bits into a problem of n integers of $< b$ bits. Repeated use of range reduction will eventually reduce the problem to a sorting problem that can be solved with packed sorting. The range reduction technique for signature sort computes a hash, or signature, for each of the input integers being sorted and then recursively sorts these hash values.

The first tool, range reduction, reduces the problem of sorting n integers with b bits to sorting n integers with $b/2$ bits in $O(n)$ time. Using randomization, it can be done in $O(n)$ space, $O(\sqrt{u})$ space is needed otherwise.

Applying range reduction $2\, log\, log\, n$ times will reduce the number of bits from $b = log\, u$ to $b/log^2 n$. If word-size $w \geq b\, log^2$, then fit $log2^2 n$ elements in each word.

In following subsection range reduction scheme by Kirkpatrick and Reisch is discussed in detail in order to provide insight knowledge of reduction technique. This technique has been modified by many researchers but original version is still preferred.

2.1.1 Range Reduction Scheme [17]

Range reduction reduces sorting n numbers of $b \leq w$ bits to sorting n numbers of $b/2$ bits at a cost of $O(n)$ time. There are four steps:

1. Cluster each number *x* according to high(x).
2. Form coordinate pairs and sort.
3. Cluster sorted pairs by second coordinate.
4. Concatenation min(B[high]) and high cluster for each high in order from * cluster.

Range reduction uses $O(\sqrt{u})$ space for B and cluster arrays. It is possible to cluster together like values using hashing in $O(n)$ time and space.

Range reduction reduces the sorting problem by sorting coordinate pairs by the first coordinate. It is necessary to move the second coordinate, the auxiliary data, with the first as sorted.

2.2 Packed Sorting

Packed sorting capitalizes on the ability of computers to operate on words in constant time. By packing several integers into a single word, parallelism can be used to sort faster. This idea was first introduced by W. Paul and J. Simon [16], and then further developed by numerous others, including a good description by Albers and Hagerup [6].

Packed Sorting; the second tool used in the sorting algorithm, allows for $O(n)$ sorting if the word-size $w \geq b \, log \, n \, log \, log \, n$.

For packed sorting, append element's auxiliary data to each element. This requires increasing the number of bits needed to store each entry from b to $2b$ and adds one more range reduction step to the overall algorithm.

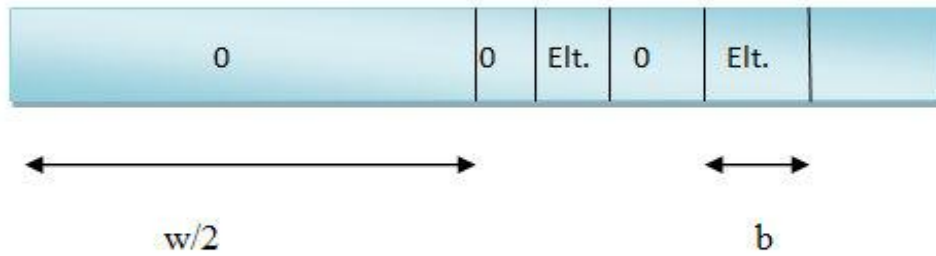


Figure 2.1: Packing b-bit integers into w-bit word.

Packed sorting, due to Albers and Hagerup [6], can sort n integers of b bits in $O(n)$ time, given a word size of $w \geq 2(b+1) lg\, n\, lg\, lg\, n$. Therefore pack $log\, n\, log\, log\, n$ elements into one word in memory. Then leave one zero bit between each integer, and *w/*2 zero bits in the high half of the word; an adapted version of merge sort to sort the elements. There are four main operations that allow doing this [6]:

1. Merge a pair of sorted words with $k \leq log\, n\, log\, log\, n$ elements into one sorted word with 2*k* elements.

2. Merge sort $k \leq log\, n\, log\, log\, n$ elements, yielding a packed word with elements in order. Using (1) for the merge operation, this takes time $T(k) = 2T(\ k2) + O(\lg k)$. Using the master theorem or drawing the recursion tree shows the leaves dominate the running time, so $T(k) = O(k)$.

3. Merge two sorted lists of r words, each word containing $k = logn\, log\, log\, n$ sorted elements, into one sorted list of 2r sorted words. By moving the first word of each list and merging them. The first half of the resulting word can be output, since its k elements are necessarily the smallest of all those remaining. Then masking is done on the second half of the word, which contains the larger *k* elements. This word is placed at the beginning of the list which formerly contained the maximum element in the word, maintaining the sorted manner of the lists. This takes $O(log\, k)$ time to output a word, so the merge operation takes total time $O(r\, log\, k)$.

4. Merge sort with (3) as the merge operation and (2) as the base case, yielding a recurrence of

$$T(n) = 2T(n2) + O(nk \log k), \text{Where } k = logn \log \log n.$$

There are $\log nk = O(\log n)$ internal levels in the recursion tree, each taking total time $O(n\, k \log k) = O(\, n \log n)$. So internal levels contribute a cost of $O(n)$. The $n\, k$ leaves each take $O(k)$ time, so the total cost of the leaves is also $O(n)$.

2.3 Sorting with Packed Merge

The above merge operation is used to pack words and then sort the packed words. A word is packed using a merge sort which uses the above merge as the merge operation. The word-packing is the base case of another merge sort that uses the same merge operation to merge two sorted lists of r packed words in $r \log k$ time. This saves a factor of $\Theta\,(k/ \log k)$ over a normal merge sort. Given that $k = \Theta\,(\log n \log \log n)$, this savings equates to [5]:

$$nlogn.\frac{\log(\log n \log \log n)}{\log n \log \log n}$$

$$nlogn.\frac{\log \log n + \log \log \log n}{\log n \log \log n}$$

$$nlogn.\frac{1}{\log n}$$

Or $O(n)$ time [5].

In 2003, Ben Vandiver and Alex Rolfe [5] also implemented the expected linear time and space requirement signature sort algorithm of Andersson [1] to compare its performance against traditional sorting algorithms. They used packed sorting idea give by W. Paul and J. Simon [16] and further developed by Albers and Hagerup [6]. They tried using the concept of maintaining ranks and range reduction. The other phases included were sorting recursively and trie construction.

2.4 Sorting Recursively [5, 6, 16]

The next step after sorting using packed merge is to sort the signatures recursively. If the signatures are small enough, they can be sorted directly using the packed sorting. Otherwise, recursively call the signature sorting algorithm.

2.5 Trie Construction [5, 16]

The sorted signatures do not give us the sorted input values directly. Instead, the sorted signatures will allow us to compute a path compressed trie of the signatures in linear time. Constructing this trie requires that compute the longest common prefix for all adjacent pairs of signatures in the sorted list and then use a particular construction algorithm that first builds a binary Cartesian Tree to get the path compressed trie. There are n−1 internal nodes and n leaves, one per signature, in the resulting trie. Once the algorithm has built a binary trie, the tree is flattened; if the fields of the signature were each l bits, then at most l levels of the tree will be flattened into one node to give a maximum out-degree of l. Thus, each node/edge in the trie represents on of the 2l possible values of the signature field at the position corresponding to the depth in the tree. If the hash function generated no collisions, then sort the nodes of the tree to produce the sorted output. Each leaf, of which there are n, is labeled with the input value corresponding to the signature with which the leaf was first labeled. Sort the internal nodes (working upwards towards the root), labeling each with the minimum and maximum values of its children. To find the minimum and maximum value for internal nodes (for leaves, the minimum and maximum are just the input value with which it was labeled), sort the children by their minimum value. Since each node has constant out degree (assuming l is fixed), this sorting can be done in constant time at a node. Since there are at most n nodes (the binary trie had n−1 internal nodes; converting to the more general form of the trie may have removed internal nodes, but didn't add any), the total time to sort the trie is $O(n)$.

Sorting the children of a node produces a process similar to radix sort. In the trie, each level corresponds to one field of l bits in the signature. Recall that each field in the signature is the hash of one field in the original input value. Thus, sorting at the ith level in the trie is equivalent to sorting based on the ith field of the original input

value. If the hash worked well (without collisions), then input integers with the same value in the ith field will all be grouped in the same node. Thus, at the top level, for example, sorting the children is equivalent to sorting the groups of input values by their first (most significant) field, where all nodes in a group/child share the same first field. At the second level, the sort organizes the input values with the same first field by their second field.

Once the trie has been fully sorted, the sorted input values can be read by traversing the leaves from left to right. Again, this can be done in $O(n)$ time even if traverse all $O(n)$ of the internal nodes in the process. If the hash function generated collisions, then the grouping in the trie will not be correct and all input elements group together at the ith level will not share the same value in their ith field. Since it is not possible to discover these collisions in the allotted time, the algorithm runs to completion, ignoring the possibility of collisions, and then checks its output at the end. This check ensures that the output sequence is non-decreasing and can be performed with a linear-time scan of the proposed output. If the output is not properly sorted, the algorithm merely runs again.

But whole implementation didn't conclude into expected output as again the output was poor compared to quick sort. The reason for the output is considered as the number of operations in the signature sort range reduction is much larger:

The number of linear time operations in the signature sort range reduction is much larger:

1. Computing each signature takes a multiplication, several shifts, and several logical operations.
2. Building the binary trie requires copying at least $2n$ integers.
3. Building the binary trie requires $O(n)$ comparisons
4. Converting the binary trie to a general form requires at least n comparisons.
5. Building and converting the trie requires following at least $2n$ pointers.
6. Traversing the tree to generate the output requires following at least n pointers and copying n integers.

7. The sorted signatures require $O(n)$ auxiliary (not input or output) memory, reducing the gains our algorithm might see from the CPU's cache.
8. The trie requires $O(n)$ auxiliary memory (around 20 bytes per input integer), further reducing the cache's effectiveness.
9. Many of the trie construction procedures are recursive, so there are at least two function calls for every input element.

In experiments, to measure the performance of the randomized signature sort with the quick sort, packed sorting and signature sorting algorithms were ran on a variety of input sequences. The same algorithms were also run for binary inputs but of no use. The difference in the performance was clearly visible.

2.6 Result from Existing Signature Sort

In figure 2.2 shows the comparison between quick sort and existing randomized signature sort.

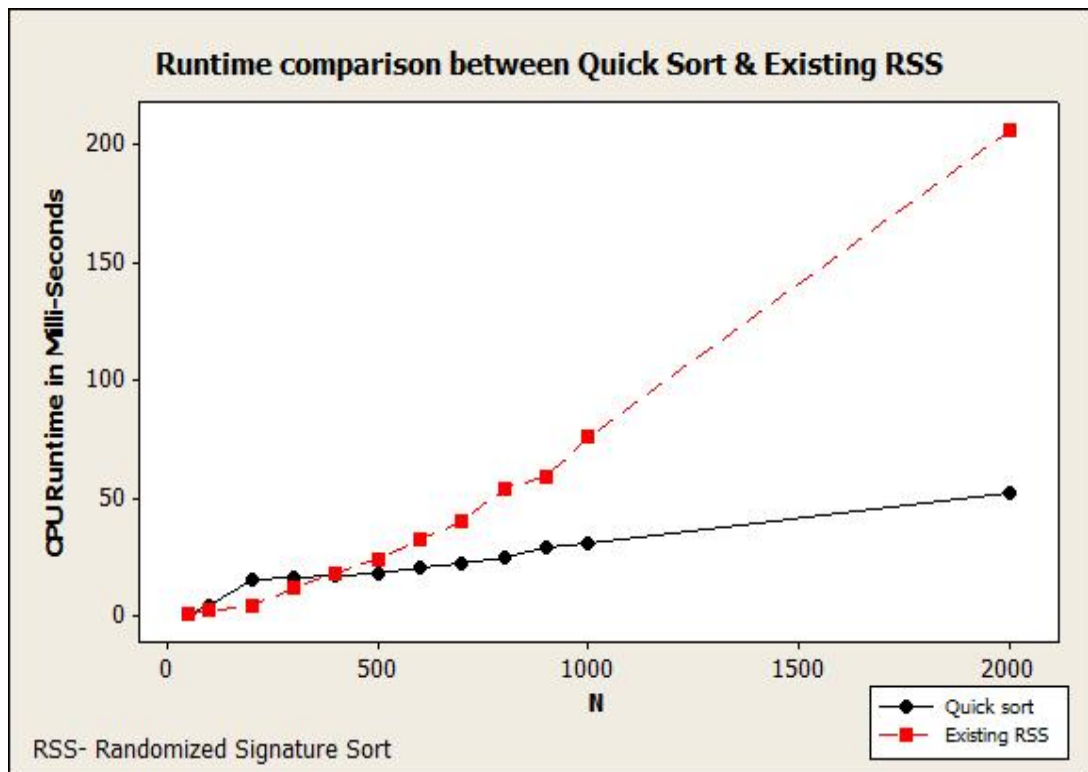


Figure 2.2: Runtime comparison between randomized quick sort and existing randomized signature sort

The graph depicts that may be for smaller inputs the performance of both the algorithms is almost similar but as the input sequence length increases the difference between the performance of the two also increases. Thus signature sort showed very poor performance because the time consumption for sorting n integers increases as the size of n increases.

Chapter -3

Problem Statement

Computational complexity means how efficiently problems can be solved on computers. In order to achieve efficient signature sort algorithm, randomization technique seems better option than deterministic algorithm.

On implementing randomizing signature sort algorithm, it gives relatively poor performance than the traditional algorithm like randomize quick sort. It happens due to extra operations associated with randomized signature sort. This operation includes packing, unpacking and creation of signatures. So there is a need to enhance the performance of randomize signature sort either by reducing extra operation or by modifying the operations in such a way that the running time of the operations improves. All major operations of randomize signature sort i.e. Signature creation, packing, word comparison and unpacking; can be improved.

In order to make randomize signature sort more efficient, one has to work on its main two phases; namely Packed sorting and range reduction [1, 5, 6, 16]. Though many researchers have worked in optimizing these two phases but as optimizing a randomized integer sorting algorithm is always a challenging task therefore, there is always a scope to improve it for better performance.

Better packing mechanism is major concern because if packing of signature is not done properly than chances are there that few bits may get lost. So the mechanism should be efficient to work in accordance to the size of signature bits so as to be fitted in the limited word-size.

Also, better comparison method is another scope of improvement. By using better implementation ways like use of appropriate hashing function, bitwise operators, and faster computation can be achieved. Thus intent is to simple and stable randomized signature sort running efficiently and lesser complex than its previous versions.

Chapter-4

Improved Randomized Signature Sort Using Hashing and Bitwise Operators

In this thesis, the performance of randomized signature sort has been improved with the use of hashing and bitwise operators. It gives significant improvement in the performance as compared to existing randomized signature sort. Improved randomized signature sort also reduces the extra operation required by the signature sort. Instead of dividing integer into fields, simply hash each integer into signature which is only $O(log\ n)$ bit size long. It reduces the requirement of dividing integers into field as integer has only one signature of reduced bit size. Then multiple signatures are packed into one word. In this way, single word is used for multiple integers, instead of using one word for one field of integer. Thus, the operation required after this in only comparison.

The signature sorting algorithm uses the signatures which are computed by applying hash function on the integer. The main purpose of using this idea is to implement it on input values to reduce the size of the integers being sorted. The range reduction depends on the hashing algorithm not generating collisions; if collisions occur, the output will not actually be sorted, making this a Monte Carlo algorithm. Thus, the hash function is very important and it must not generate collision at all for better results. The hash function used hashes integer into $O(log\ n)$ bit size signature.

The benefit of using bitwise operators in the implementation is to impose implicit parallelism. Also, bitwise operations are slightly faster than addition and subtraction operations and usually significantly faster than multiplication and division operations.

The improved randomized signature sort will work in following steps:

1. Reading input.
2. Creating signatures from input values of $O(log\ n)$ bit length.
3. Packing of signatures in words.

4. Comparing words using bitwise operators and sorting.
5. Unpacking signatures from words and getting sorted sequence.

4.1 Machine Model

The machine model is a normal computer which holds an instruction set corresponding to what is programmed in common standard programming language such as C/C++ or JAVA. A processor determined word-size W, confines how large integers can be processed in constant time. It is assumed here that each input integer fits in a single word and for generic code, the type of a full word integer e.g. long int, should be a macro parameter in C or template parameter in C++ or a primitive type in JAVA. The unit-cost time measure is adopted where each operation takes constant time [4]. Interestingly, the traditional theoretical RAM model of Cook and Reckhow [29] allows infinite words.

According to Yijie Han and Mikkel Thorup [4], the outcome of infinite words with operations like shifts or multiplications, exponentially big parallel processor can be simulated, solving all problems in NP in polynomial time. So they suggested banning such operations from unit-cost theory RAM, making it even more contrived from a practical view-point. But it is possible to achieve such algorithms that can be implemented in the real world considering the real-world limitation of a fixed word-size in mind. This model is named the word RAM (Hagerup [9]). The word RAM has a fairly long tradition within integer sorting, being advocated and used by Kirkpatrick and Reisch [17].

In word RAM, addition and subtractions can be performed on integers and this is considered as an advantage over the comparison based model by word RAM. Hence, this word RAM can be used to code multiple comparisons of short integers combined in single words. The idea of multiple comparisons was first introduced by Paul and Simon [16]. The word RAM model is different from the comparison based model as well as from the pointer machine in which integers can be used, and segments of integers, as addresses. This idea links to radix sort where an integer is viewed as a vector of characters, and these characters are used as addresses. Another idea of word RAM is to hash the integers into smaller ranges, naming these hash integers as

signatures in this paper. Here radix sort goes back at least to 1929 [18] and hashing goes back at least to 1956 [18], both being developed for efficient problem solving in the real world. Also research has been done on the further use of RAM for advanced tabulation of complicated functions over small domains.

As a simple example, considering computer architecture of 64 bits i.e. 64 bits are processed in single instruction cycle. Actual comparison operation is performed on maximum of 32-bit integers; if having a range of integers then these integers can be mapped to lower number of bits. Thus, several integers are packed into a single word. Thus 64 bit machine can compare 16 signatures i.e. hash value of input integers with their size of 8 bits in single instruction cycle.

Summing up, the concept facilitated has been discussed by the word RAM is well established in the practice of writing fast code. Hence, on disallowing these concepts, the time complexity of running imperative programs on real world computers, are not being discussed. Thus this RAM model plays a vital role in reducing the overhead and enhancing the performance of the algorithm.

4.2 Signatures

In order to decrease the size of integers to be operated upon, signatures are created which have the lesser bit size than the original inputs. The signatures are computed for each input with a universal hash function. Such signatures must have the size of $O(logn)$ where n is the number of input integers. The hash function used to create these signatures, must assure collision free hashing.

The signatures created with this method must follow the property:

$$\text{If} \quad A_i \leq A_j \ \forall \ i,j \ \in \ \{0, 1 \ldots n-1\} \text{ then}$$

$$S_i \leq S_j \ \forall \ i,j \ \in \ \{0, 1 \ldots n-1\}.$$

Where A_i's are the input integers, S_i's the corresponding signatures and n is the number of input integers.

4.2.1 Hashing

The hash function is applied on integers to reduce their size by creating signatures of $O(log\ n)$ bit size. The hash function must provide collision free hashing to ensure accurate and better result. The hash function must take $O(n)$ expected time. This will improve the overall performance of algorithm. The following hash function from [5] is being implemented that is used for hashing of integers into signatures. Division must be avoided in hash function for better performance. The hash function is,

$$h_a(x) = (ax\ mod\ 2^k)/2^{k-l}$$

Where, k is the number of bits in the input integer,

l is the number of bits in the signature which will be $O(logn)$.

a is randomly chosen between zero and 2^k.

Since the division in the above function is division by a power of two, it can be implemented as a left shift. This function will take $O(n)$ time. The above said hash function assures $O(logn)$ bit size of signatures and also collision free result [5].

4.3 Packing

In Andersson's concept [1] integers are divided into fields and each of these fields is packed into words. This is a bit of overhead as each field of integers is required to be packed and compared. The concept of word formation has been improved using hashing. As discussed above hash function will hash whole integer into a signature with reduced bit size of $O(log\ n)$. After that the packing of multiple signatures into one word will be done.

It is an important phase as multiple integers i.e. signatures (hashes) of integers will be packed in a word. Also ensure that this phase runs error free while implementing.

If w is word size of the machine, sb is the number of bits in the signature, wl is the word length, m is the number of words, l is the number of signatures in a word and n is the number of input integers then:

$$l = wl/sb$$

$$m = ceil(n/l)$$

There will $O(log\ n\ log \log n)$ number of words be created overall. The word formation phase will take $O(n)$ time. The following pseudo code is for word formation:

Word and sign are the array of words and array of signatures respectively.

Step 1: Repeat for i=1 to m by 1.

Step 2: Repeat for j=1 to L by 1.

Step 3: word[i]= (word[i] << sb) | sign[i*L+j].

Step 4: Return.

Diagrammatically above algorithm can be represented as:

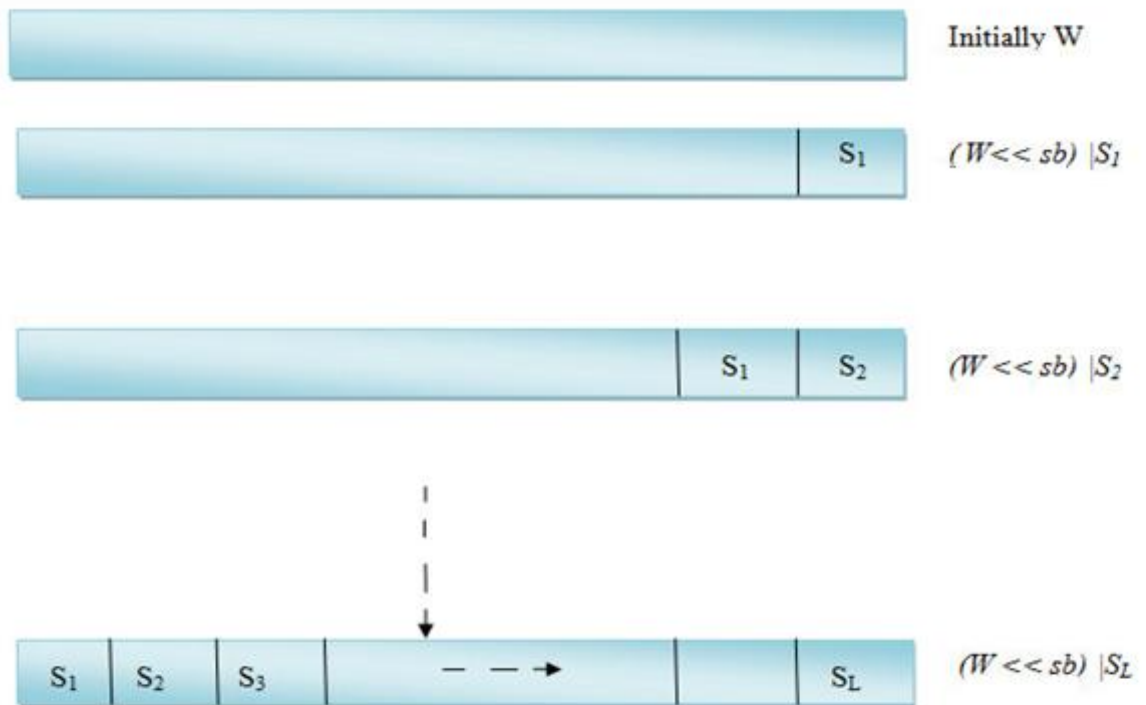


Figure 4.1: Packing of signatures in a word.

In above figure, W represents the actual word, sb represents the signature bit size and L is the number of signatures that can be packed in a single word. Above case discussed happens when the $w = sb * l$ where w is word length. If the perfect packing is not possible then some bits remains useless in a word.

Figure 4.2: Packing of signatures with extra bits.

4.4 Comparison Sorting

Comparison sorting is another important phase which is applied on words in order to get the sorted result. Here sorting means exchanging the positions of the signatures from one word to another word and also exchanging positions within word itself. This is the phase where actual comparison will occur, which will result in sorting. As now, there are multiple integers in one word thus, the word itself also be sorted after sorting has occurred in between words. Thus, the whole procedure of sorting can be divided into two sub-phases.

The first sub-phase is sorting between words. This can be done by comparing two words and checking corresponding bits of signatures to merge them. The idea is to use bitwise operations to get the result. As bitwise operators are comparatively fast thus use of these operators will speed up the processing the algorithm. Only two bitwise operators XOR and AND are required to perform this task. The task includes applying XOR and AND in such a way that the result will show which signatures need to be swapped. First, apply XOR on words and then apply XOR and AND operators on result with the word in which keep smallest of this signatures.

The above discussed operations will give non-zero value on corresponding bits where signatures are needed to be swapped. Thus, swap those signatures again using XOR operator. Hence, only a constant time is required to perform the above said operation. As there will be a maximum of $O(log\ n\ log\ log\ n)$ words which are to be compared, this will lead to $O((logn\ loglogn)^2)$ number of maximum comparisons. These comparisons can be further reduced by using any traditional deterministic algorithm

like quick sort. By applying the traditional deterministic algorithm the number of comparison will be reduced to $O(logn\ loglogn\ log(logn\ log\ logn))$.

Knowing that, $log\ log\ n < (logn)^{2}$

Taking log on both sides, $log(log\ log\ n) < 2\ loglog\ n$

Multiplying both sides with $log\ n\ log\ log\ n$,

$$log\ n\ log\ log\ n\ log(log\ n\ log\ log\ n) < 2\ log\ n\ (log\ log\ n)^2$$

Now as, $(log\ log\ n)^2 \leq log\ n$

Thus, $log\ n\ log\ log\ n\ log(log\ n\ log\ log\ n) \leq 2(log\ n)^2$

Now, $(log\ n)^2 \leq n$

Hence, $log\ n\ log\ log\ n\ log(log\ n\ log\ log\ n) \leq 2n$

This proves that total number of comparisons will be $O(n)$. As there is constant number of signatures in every word thus there is need to perform this procedure only a constant number of times which is independent of number of integers. The overall expected time for this sub-phase is only $O(n)$.

Now after first sub-phase the words will be in sorted order with each other, but there are multiple signatures in every word thus there is need to take care of that. Here there is need to perform sorting within word itself. This can be done in similar fashion as sorting is done between words. The one more thing is needed to take care of here is masking of signatures which are not participating in sorting operation. The task consists of comparing signatures within word. Here also bitwise operators will be used as above. Hence, there is constant number of signatures in a word. This will take constant number of operations. There is at most $O(log\ n\ log\ log\ n)$ number of words, that implies $O(log\ n\ log\ log\ n)$ number of operations are required which is less than $O(n)$.

Let's suppose there are m signatures in a word. So on comparing each signature with other, it will take $O(m^2)$ expected time. This expected time is further reduced to

enhance the performance by simply using divide and conquer technique on word. This is shown in the figure-4.3:

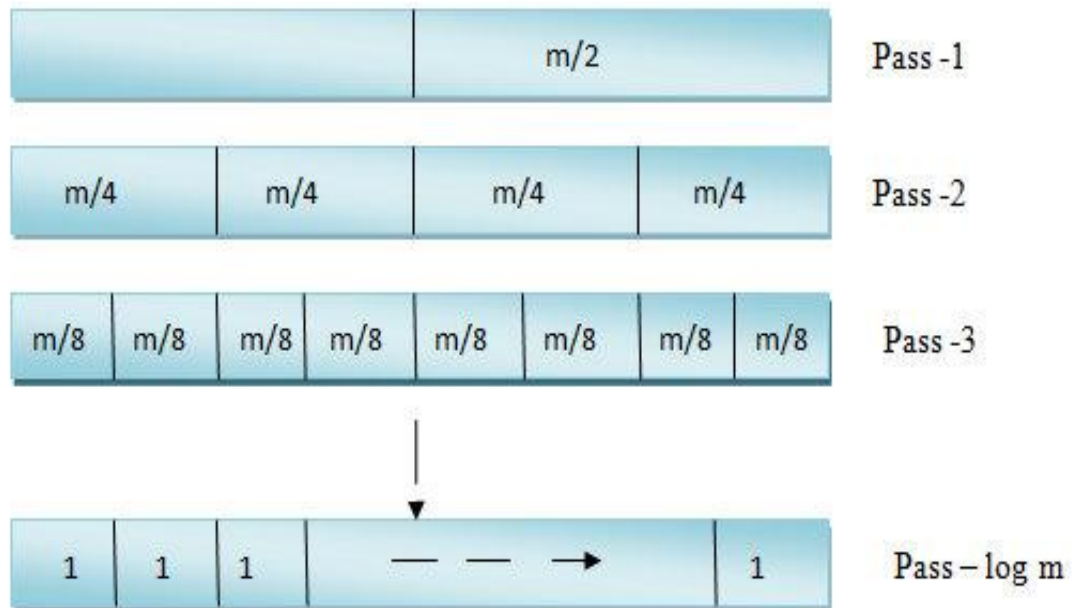


Figure 4.3: Comparison within word

The algorithm for comparison within word is given as follows:

Step 1: Set i=1.

Step 2: Divide word into 2^i parts.

Step 3: Compare each adjacent parts i.e. 1st with 2nd, 3rd with 4th and so on.

Step 4: Exchange the positions of signatures according to the result of step 3 as exchanged while performing comparison between words.

Step 5: Stop.

This algorithm first divide a word into 2 halves and compare them bitwise operations. And according to the result of bitwise operations, the positions of signatures are exchanged, and in second pass each half considered as individual word and same task is performed on those. In this way, parallelism of uniprocessor system is being exploited as done while performing comparison between words. The number of

passes will be log m and total number of comparison will be c log m where m is the number of signatures in a word. Now there is need to call the algorithm for all words formed. Thus, finally words will be sorted itself. This shows that there will be at most $O(log\ m)$ passes and in each pass only constant number of comparisons is required. Thus total number of comparisons will be $c\ log\ m$.

Considering the space requirement it is clear that the only extra space requirement is $O(1)$ which is used to perform bitwise operations on words.

The above discussion shows that the whole sorting operation can be completed in $O(n)$ expected time and uses $O(1)$ extra memory. In this phase, actual sorting will be implemented. Here sorting means exchanging the positions of the signatures from one word to another word and also exchanging positions within word itself. The whole sorting procedure is divided into two sub-phases.

The first sub-phase is sorting between words. The sorting between words includes applying XOR and AND in such a way that the result will show which signatures need to be swapped. First, apply XOR on words and then apply XOR and AND operators on result with the word in which smallest of this signatures will be kept. This operation will give non-zero value on corresponding bits where signatures are needed to be swapped. Later swap those signatures again using XOR operator. The overall expected time for this sub-phase is only $O(n)$. This is pseudo code for the first sub-phase: The sort function will take two words as input and sort them. Let two words be x and y.

Sort(x, y)

Step 1: Set temp=x XOR y

Step 2: Set temp = temp XOR x

Step 3: Set temp= temp AND y

Step 4: For all non zero signatures in temp, Swap the corresponding signatures in x and y.

Step 5: Return.

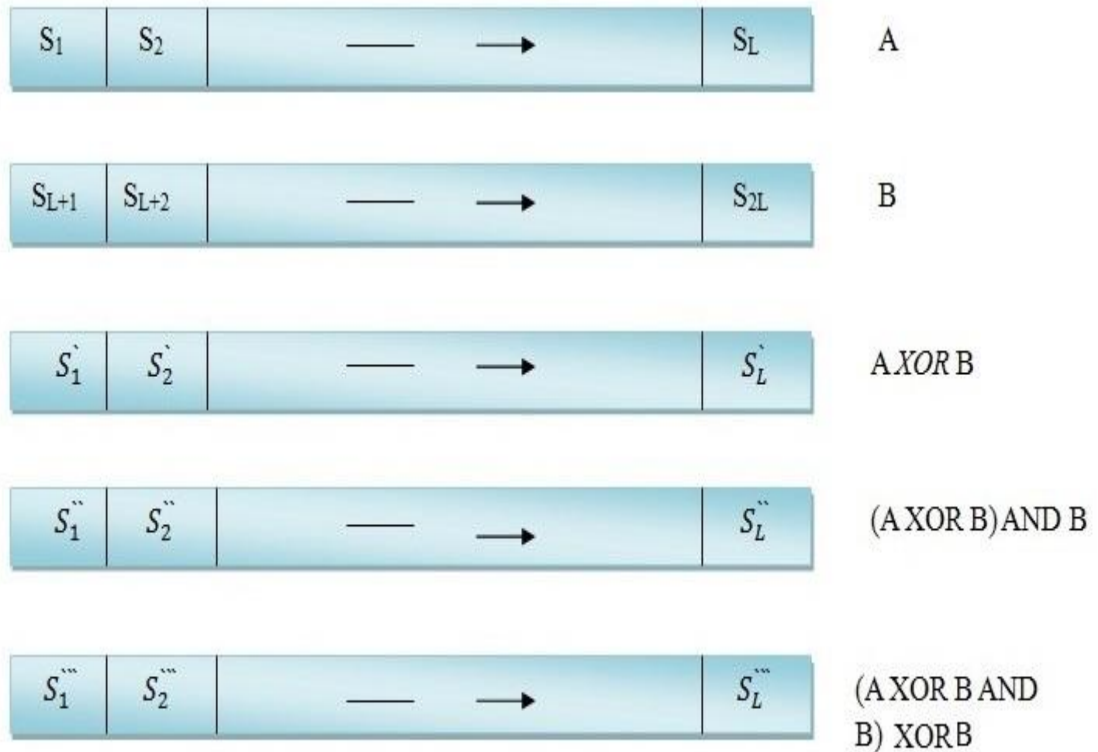


Figure 4.4: Word comparison

In above figure 4.4, there are two words taken, let it be A and B, both contains signatures formed by hashing integers. Now in order to sort first perform XOR operation between the two words. Then again XOR operation is performed with the previous resultant.

And, finally the new result obtained will be ANDed with second word i.e. B. If in the final result obtained, all bit in $S_i^{```}$ is zero then the S_i and S_{L+i} are need to be swapped in order to keep the smallest signatures in a single word.

In figure 4.5, while comparing the words in iteration, only one signature of word A is compared to the single signature of another word B. It cannot assure that all signatures of a word are smaller than all signatures of another word. So shifting is needed in either of a word.

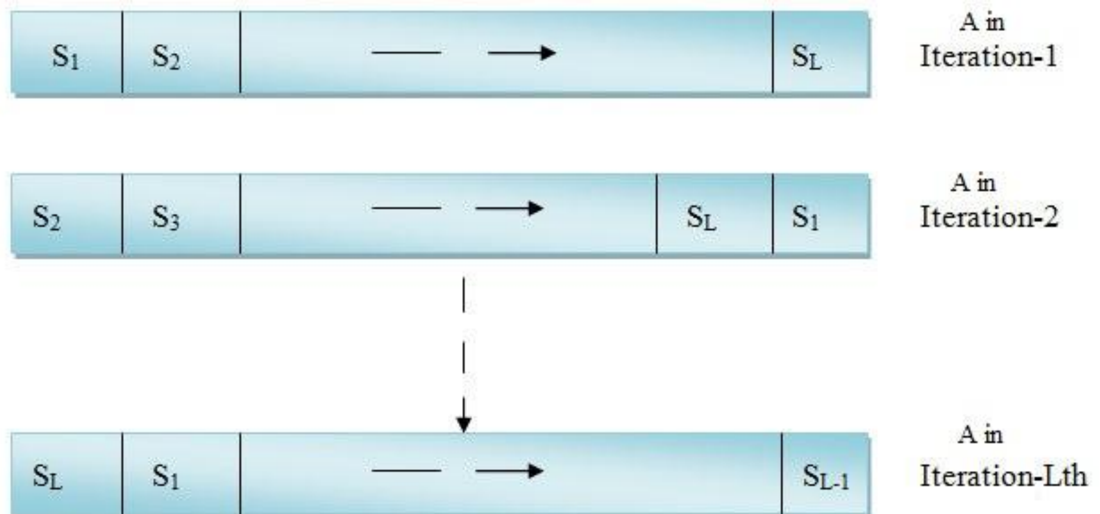


Figure 4.5: Comparison between words.

Figure 4.5 shows that iteration 1 to L each time one signature of a word from leading side is shifted to tail side. This will insure that all signature of a word will be smaller than the signatures of another word.

After first sub-phase the words will be in sorted order with each other, but there are multiple signatures in every word, hence there is need to perform sorting within word itself. Then comparison is performed by recursively dividing a word into 2 halves until reaches to single signature in a half. The expected time for this sub-phase will be $O(log\ m)$.

The algorithm for comparison within word is given as follows:

Step 1: Repeat for i=1 to logL.

Step 2: Divide word into 2^i parts.

Step 3: Call Sort() for each adjacent parts i.e 1st and 2nd, 3rd and 4th, so on.

Step 4: Return.

4.5 Unpacking

The unpacking implies when the sorting has been done there is a need to get back the original input integers from the sorted signatures. In order to perform this task there

are two ways: one possible way is to keep track using index and another is to create reverse hash function. During the implementation of improved randomized signature sort, the promoted idea is the usage of indexing. If indexing is used then indexes are swapped as the signatures are swapped therefore it's easier to sort the signatures and get back the original input.

The unpacking mechanism can also be designed to get sorted signatures from words. This can be done in similar fashion as done in packing. For this a mask will be required. AND operation of the mask with the words will be taken. This gives a single signatures, after that shifting of mask is done according to the size of signatures and this process is repeated until all signatures from words is unpacked.

Chapter -5

Testing and Results

5.1 Testing

In order to compare the performance of improved randomized signature sort with the existing signature sort as well as quick sort, the algorithms is run several time with different number of input size and the result obtained are then compared. The comparison is done on two main basis: the CPU running time and, the memory requirement. 32-bit integers are used as input and 64-bit long data type provided by Java to store words. The input integers are generated using random function and stored in a file. The implementation of algorithms is done in Java6.0 on Eclipse-IDE using OOP approach. The platform used is Intel 64-bit with Core i3 processor having a frequency of 2.40 GHz with Windows 7(64-bit) Enterprise Edition running on it. The System had a RAM of 3GB. While measuring the performance i.e. in order to collect the details, all other extra processes were terminated so that actual measure can be taken.

5.1.1 Runtime Comparison

The algorithm ran several times on a sequence of randomly generated number to find the average running time. The runtime of the algorithm is measured using currentTimeMillis() which is provided by the System Class. The function claims to provide CPU time measuring in milliseconds. The running time includes reading input from a file and writing output to another output file. The runtime also includes time consumed in the process of hashing of integers into signatures, packing of signatures into words and extracting signatures from words.

The table-1 shows the data collected of running time for improved randomized signature sort, existing randomized signature sort and randomized quick sort algorithm. The CPU time shown in table for all algorithms is in Milli-Seconds.

Table-5.1: CPU runtime details

N	Existing randomized signature sort	Quick sort	Improved randomized signature sort
50	1	1	1
100	2	4	5
200	4	15	11
300	12	16	16
400	18	17	22
500	24	18	31
600	32	20	32
700	40	22	32
800	54	25	44
900	59	29	46
1000	76	31	47
2000	206	52	94
4000	623	83	234

5.1.1.1 Improved Randomized Signature Sort

The figure 5.1 shows that the graph when plotted between CPU time, measured in Milliseconds and number of input size sequence length, the graph grows close to linear. As the input size increases the CPU time also increases proportionally. For small input sequence lengths, the running times are relatively linear. As improved randomized signature sort using hashing and bitwise operators, the tasks including packing, comparison and unpacking tasks takes relatively takes lesser time than the existing one.

The increase in running time of algorithm is directly proportional to number of input integers. The slope clearly shows that whenever number of input integers is increased then the running time is increased proportionally. This implies that the performance of the exponential tree sorting is very good.

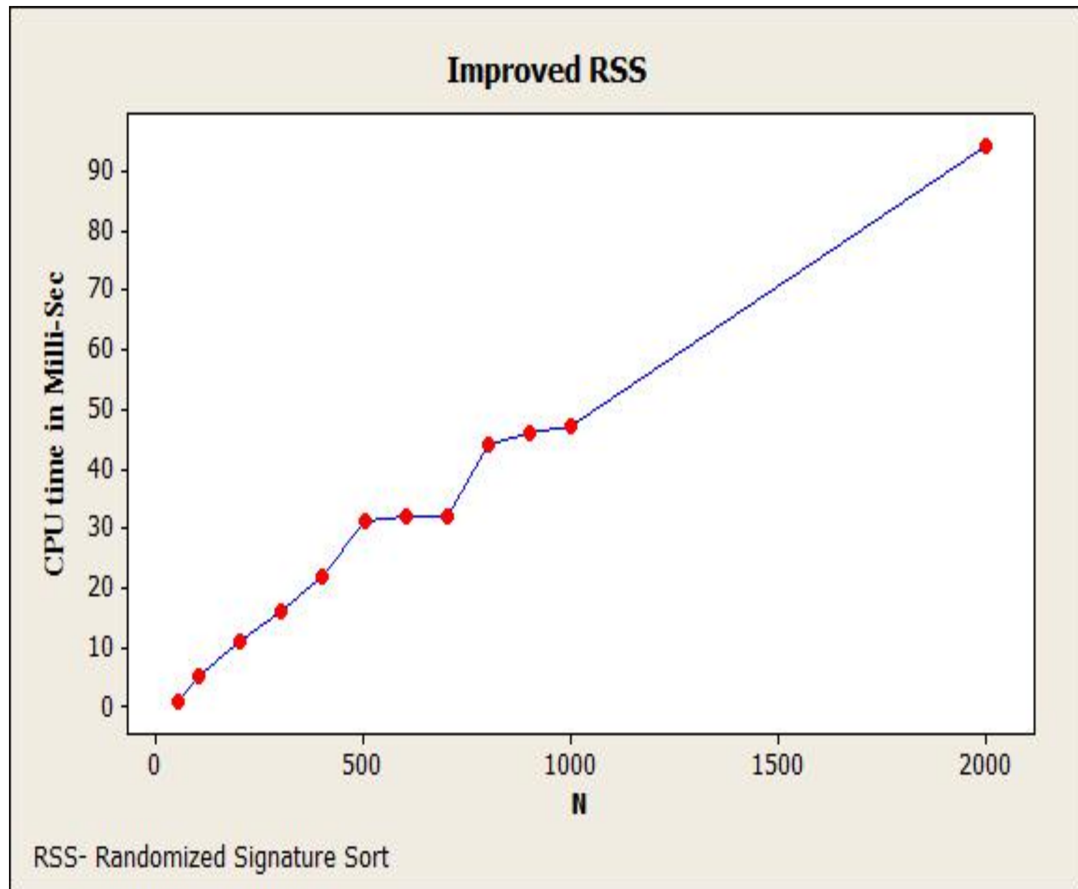


Figure 5.1: Runtime plot of improved randomized signature sort

5.1.1.2 Improved Randomized Signature Sort V/S Existing Randomized Signature Sort

The figure 5.2 displays the runtime performance comparison of improved randomized signature sort algorithm and existing randomized signature sort algorithm. The line for existing randomized signature sort algorithm grows apart from the improved one, indicating that for values ranging from 50 to 1000 the runtime of CPU for both the algorithms is almost close to each other but for values of n when becomes greater than 1000, the difference between the lines grows greatly. This implies that for larger input sequence length the CPU runtime consumption for existing randomized signature sort is more than improved randomized signature sort.

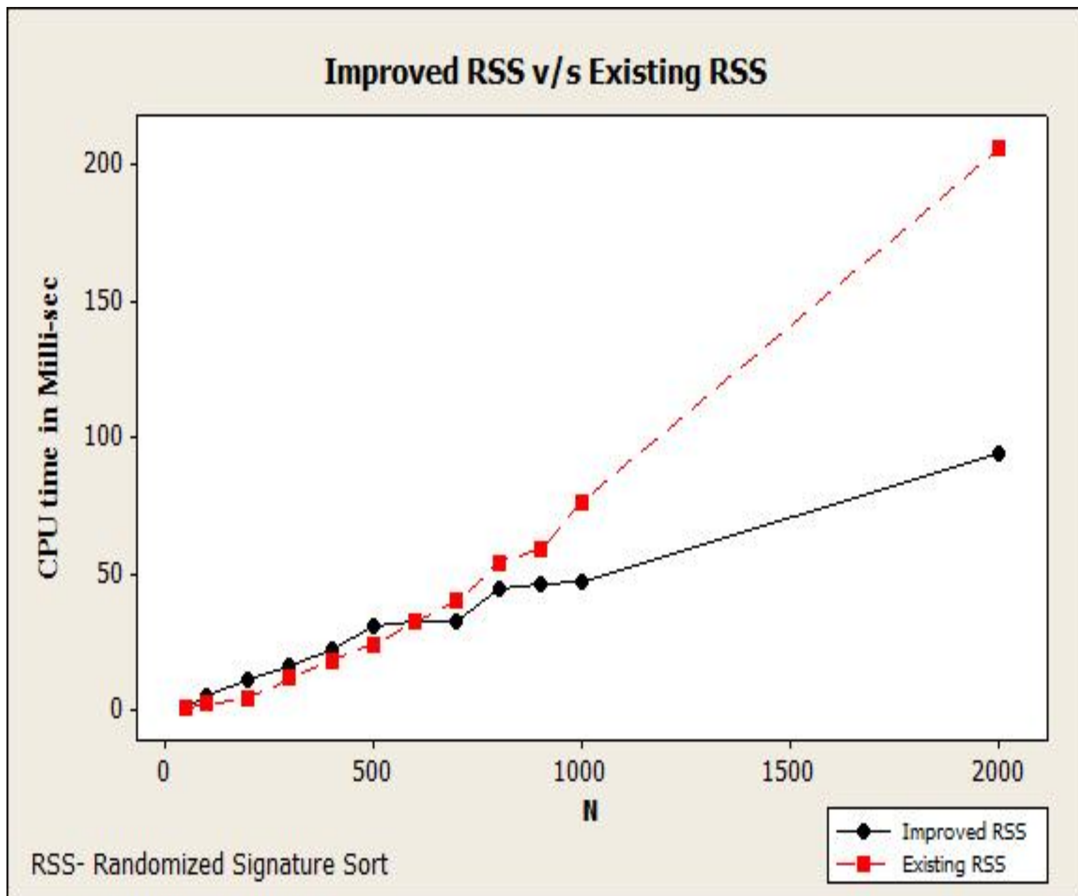


Figure 5.2: Runtime comparison: Improved randomized signature sort v/s existing randomized signature sort

The existing randomized signature sort divides integers into fields. Therefore, there is need to consider each field for comparison operation. That implies each integer will be considered for comparison as many time as the number of fields, which is extra overhead. In improved randomized signature sort, first task is of hashing which include hashing large number of bits to smaller number of bits. This means there is need to consider each integer for comparison for lesser number of times than existing randomized signature sort.

Hence, the performance (in respect of CPU runtime) of improved randomized signature sort algorithm is better than existing randomized signature sort algorithm.

5.1.1.3 Improved Randomized Signature Sort v/s Randomized Quick Sort

Randomized quick sort has been considered for comparison of performance with improved randomized signature sort because of the reason that quick sort is also a

comparison based sorting technique. Quick sort makes $O(n\,log\,n)$ comparisons to sort the n integers. Quick sort is considered faster in practice than the other algorithms. It is widely used and considered best among all. Therefore, it's better to compare the improved randomized signature sort with quick sort to measure actual performance.

The figure 5.3 depicts the runtime comparison between improved randomized signature sort and the traditional quick sort algorithm. In the starting when the input sequence length is small the performance for both the algorithms is almost close to each other. The line for improved randomized signature sort algorithm grows apart from the quick sort, indicating that for values ranging from 50 to 1000 the runtime of CPU for both the algorithms is almost close to each other but for values of n when becomes greater than 1000, the difference between the lines grows greatly. This implies that for larger input sequence length the CPU runtime consumption for improved randomized signature sort is more than quick sort. Hence, the performance (in respect of CPU runtime) of improved randomized signature sort algorithm is close to quick sort but not better than quick sort algorithm.

In order to compare the running time performance of improved randomized signature sort with existing signature sort and quick sort, both algorithms were run for the same input and then the graph was plotted to see the difference.

The input integers for both improved randomized signature sort and randomized quick sort are generated by using random function and stored in a file. Later that file is used as an input for all sorting algorithms. The output of each sorting technique is also stored in a file to avoid the loss of output sequence which happens when printing the sequence on console. Other benefit of doing so is that the output of each algorithm later can be compared to ensure the desired result.

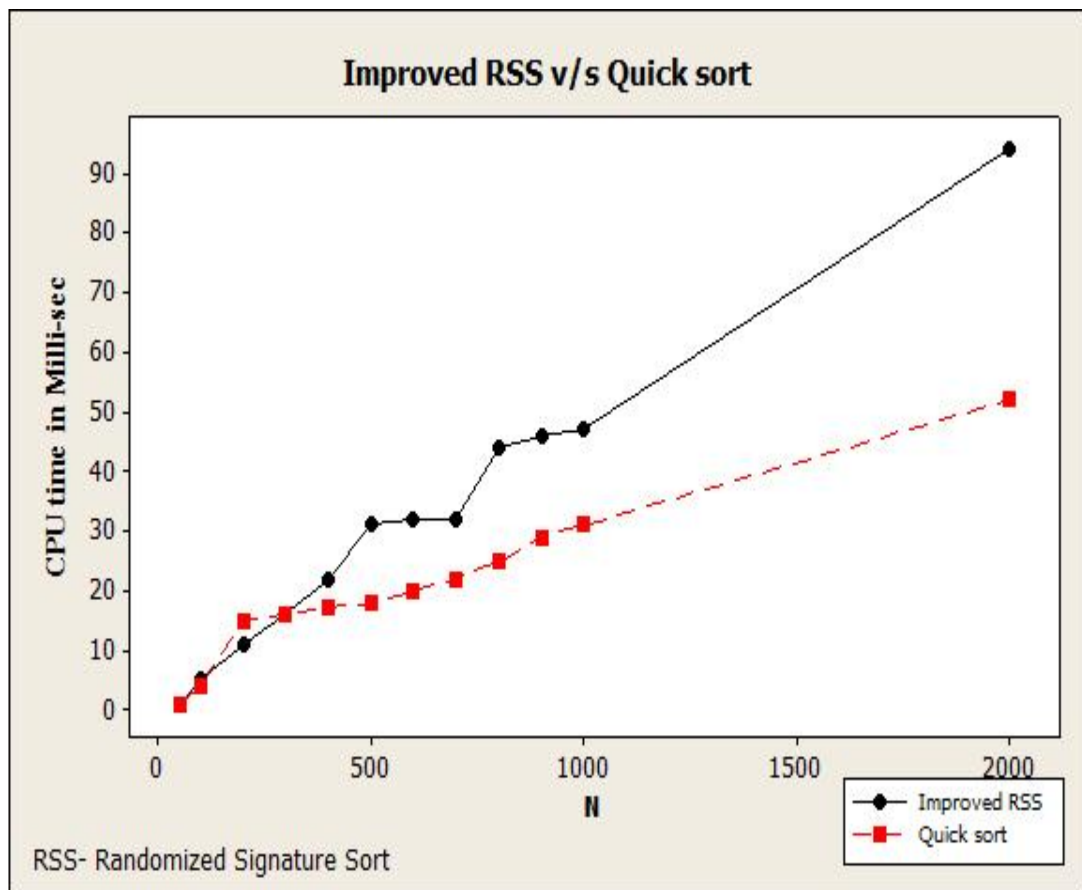


Figure 5.3: Runtime comparison: Improved randomized signature sort v/s quick sort

5.1.1.4 Overall Comparison

In figure 5.4, the difference can be easily observed between the performance of existing randomized signature sort and improved randomized signature sort. The running times for the same input size sequence length are larger than the improved one. The performance of quick sort is slightly better than the improved randomized signature sort but improved randomized signature sort produces good and close results than the existing one.

The figure 5.4 shows the overall comparison between all three algorithms. The graph clearly shows that the existing signature sort algorithm gives worst running time among all. Whereas, improved randomized signature sorting algorithm gives a running time near to quick sort. The running time of improved randomized signature sort algorithm is far better than the existing randomized signature sort algorithm.

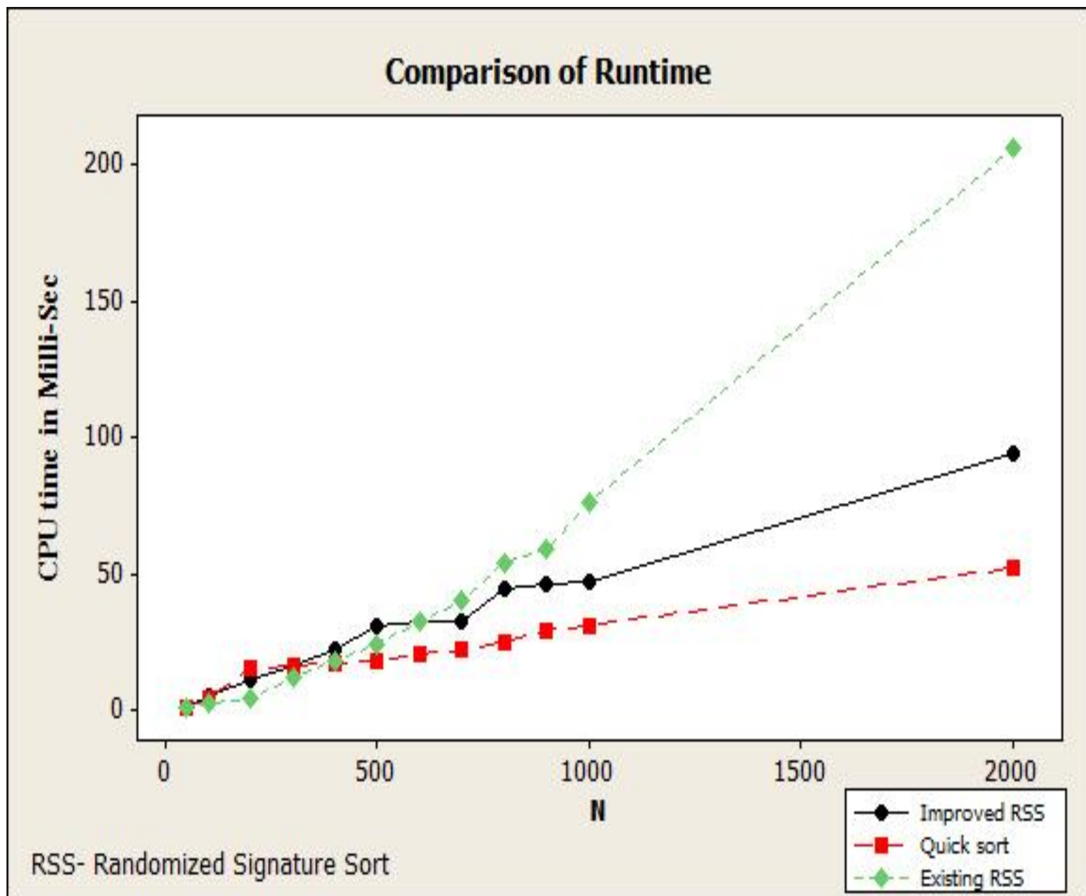


Figure 5.4: Runtime comparison

5.1.2 Memory Requirement

Quick sort has a space complexity of $O(log\ n)$, even in the worst case, when it is carefully implemented. As discussed above, the randomized signature sort algorithm has been run on a variety of input sequences to compare its performance of memory requirement to quick sort.

The memory used by the algorithm is measured by the Windows Task Manager. The algorithm is executed and the memory used is monitored and the maximum memory used by the algorithm during entire run is taken. Three runs are given and the maximum memory used is noted. The memory requirements are measured in KB (Kilo Bytes). Then in order to measure the memory consumed for different inputs, this can be done by looking the values corresponding to the process 'javaw.exe' as the algorithms is run in Java language.

The table 5.2 listed the data collected of memory required for both quick sort as well as improved randomized signature sort:

Table 5.2: Memory comparison details

N	Quick sort	Improved randomized signature sort
50	4192	4280k
100	4252	4300k
200	4768	4796k
300	4864	4808k
400	4884	4848k
500	4908k	4948k
600	4912k	4964k
700	4940k	4988k
800	4968k	5012k
900	5024k	5036k
1000	5052k	5060k
2000	5244k	5284k
4000	5644k	5692k

The figure-5.5 shows the memory requirement plot for the improved randomized signature sort algorithm which depicts that the graph has a linear slope. The memory requirement increases directly proportionally to number of integers to be sorted. The memory used by improved randomized signature sort using hashing and bitwise operators, the tasks including packing, comparison and unpacking tasks takes relatively takes lesser memory than the existing randomized signature sort.

Thus, the improved randomized signature sort algorithm has a good and optimized space requirement.

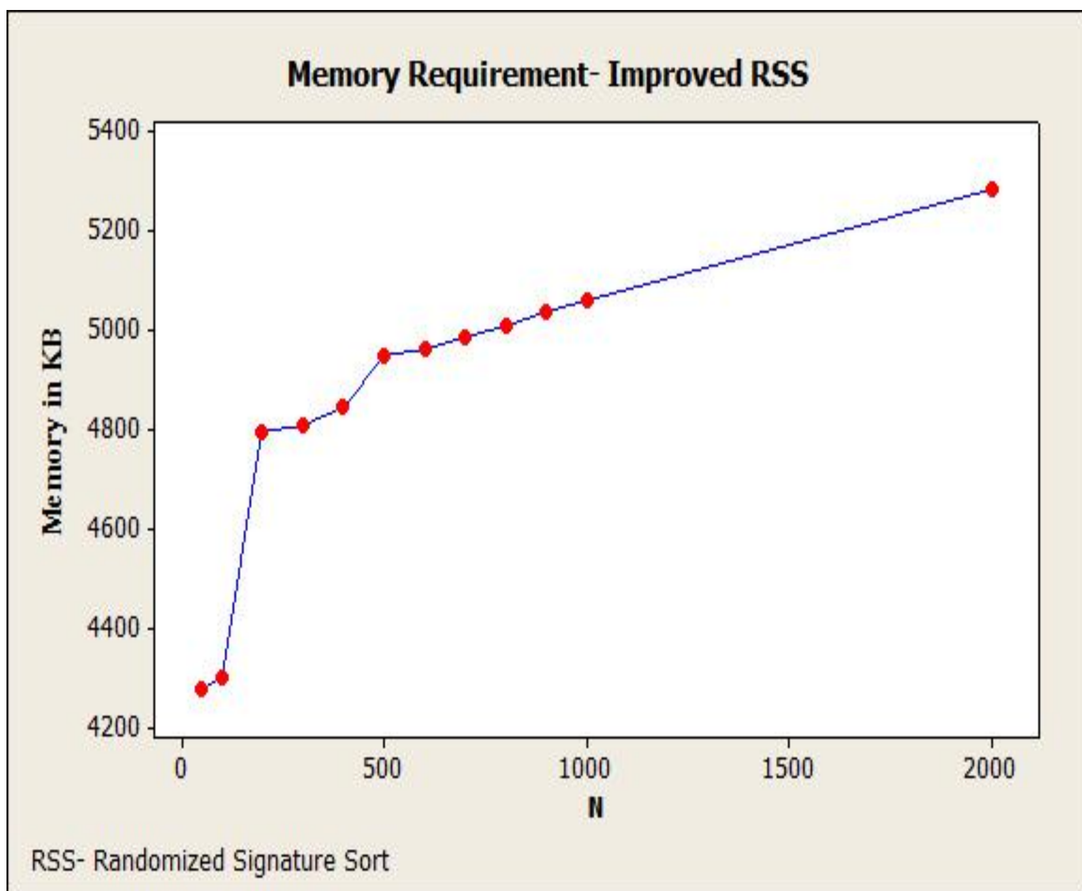


Figure 5.5: Memory requirement of improved randomized signature sort

The above graph depicts that as the input sequence lengths increases the amount of memory required is also increases considerably. As the input size grows, the graph for improved randomized signature sort appears to take nearly linear memory size.

The above figure 5.6 shows the performance of both improved randomized signature sort and the Quick sort. As the input size grows, the graph for both algorithms also grows close to each other. The difference between the two is minimal thus we can say that existing randomized signature sort has been improved considerably depleting the large difference of performance compared to quick sort.

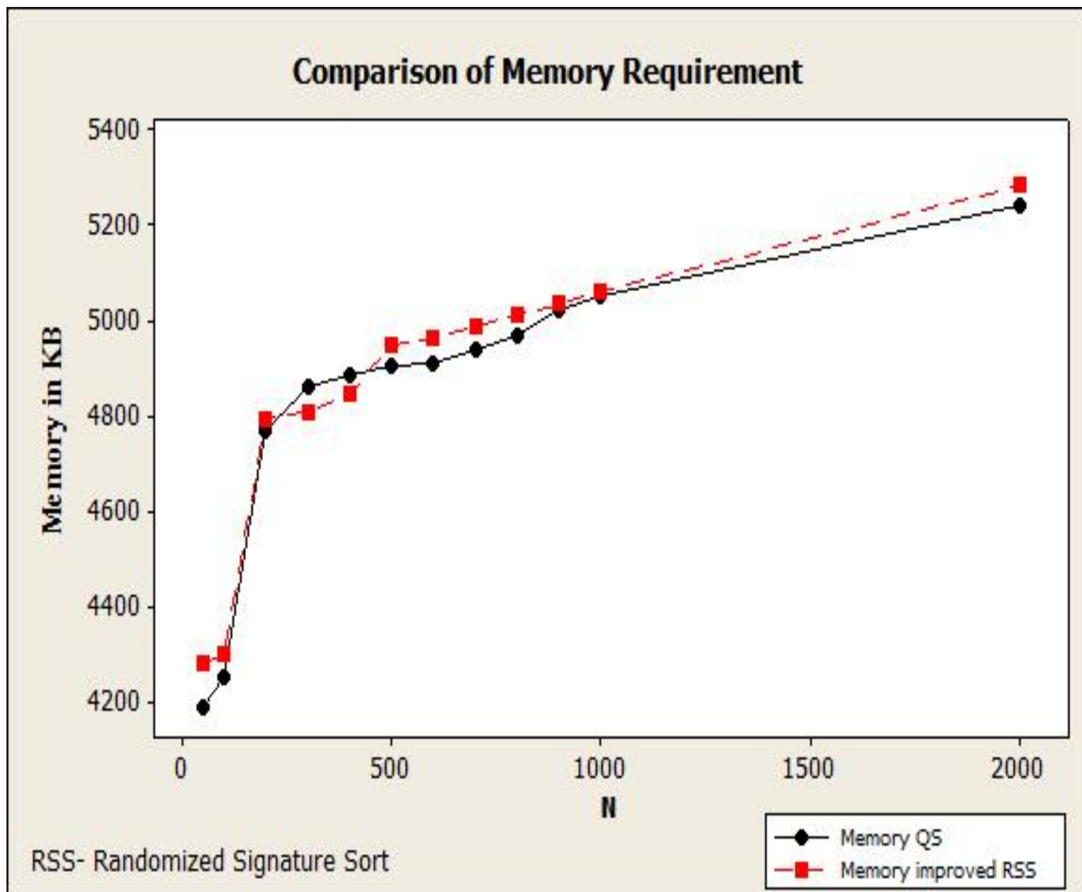


Figure 5.6: Comparison of memory requirement

The above figure 4 shows the performance of both improved randomized signature sort and the quick sort. As the input size grows, the graph for both algorithms also grows close to each other. The difference between the two is minimal thus it can be said that existing randomized signature sort has been improved considerably depleting the large difference of performance compared to quick sort.

5.2 Results

The implementation of algorithms is done in Java6.0 on Eclipse-IDE using OOP approach.

First task for running improved randomized signature sort is to input the random numbers. The input integers are generated using random function and stored in a file.

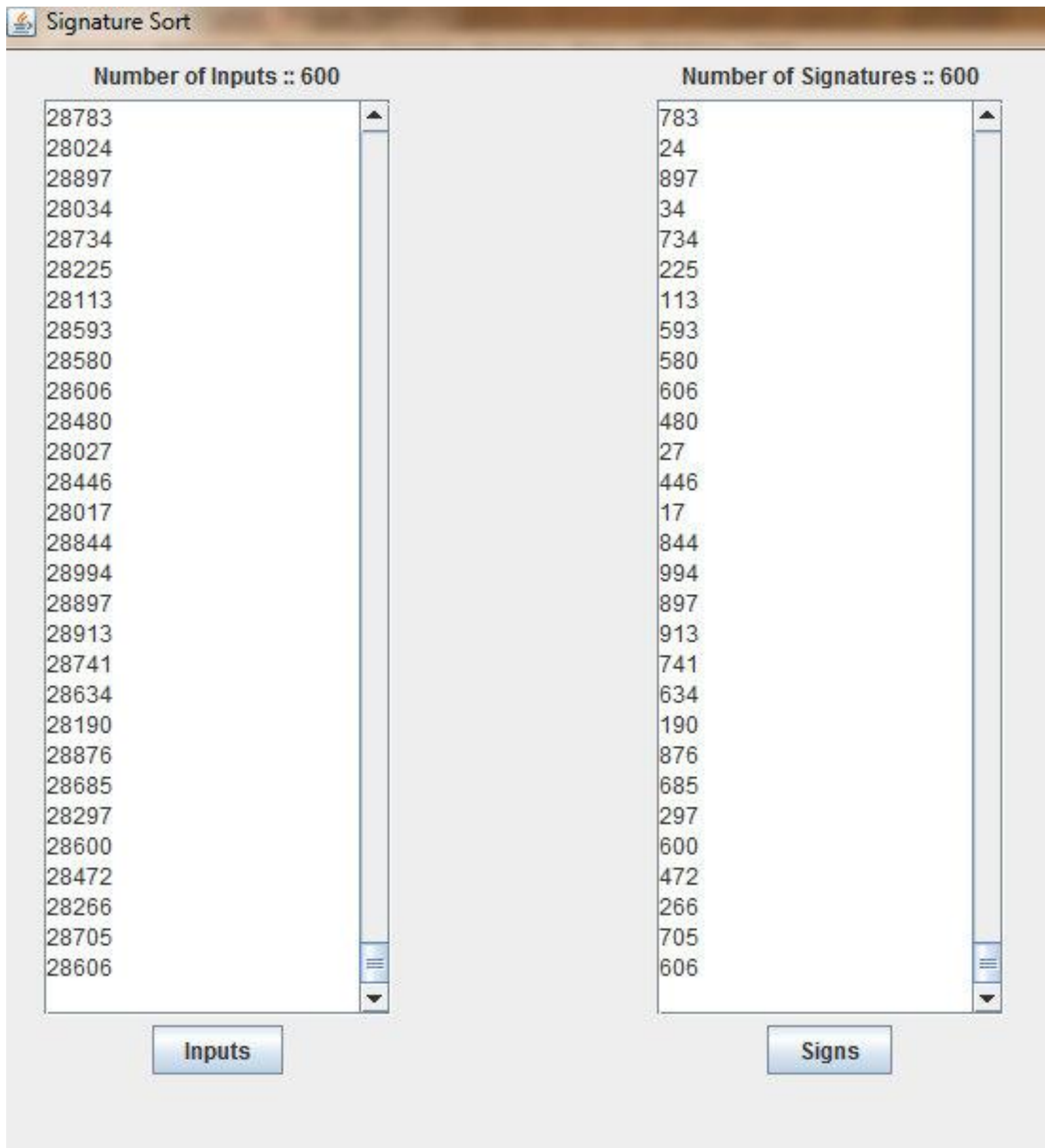


Figure 5.7: Random input and signature created for improved randomized signature sort

The number of input and the signatures formed corresponding to each integer is same i.e. signatures are created for each integer.

The signature is of $O(log\, n)$ bit size where n is the number of inputs. The hash function is applied on integers to reduce their size by creating signatures of $O(log\ n)$ bit size. The hash function given as:

$$h_a(x) = \ (ax\, mod\ 2^k)/2^{k-l}$$

Where, k is the number of bits in the input integer,

l is the number of bits in the signature which is $O(logn)$.

a is randomly chosen between zero and 2^k.

This function will take $O(n)$ time.

In figure 5.8, there are two more output windows. One consisting of the words created and second window displays the sorted output.

If w is word size of the machine, sb is the number of bits in the signature, wl is the word length, m is the number of words, l is the number of signatures in a word and n is the number of input integers then:

$$l = wl/sb$$

$$m = ceil(n/l)$$

There will $O(log\, n\, log \log n)$ number of words be created overall. The word formation phase will take $O(n)$ time.

Later, Comparison sorting is performed on words in order to get the sorted result. This is the phase where actual comparison will occur, which will result in sorting.

And lastly unpacking is done to get back the original input data corresponding to signatures. This will yield the final sorted output as shown in the figure 5.8.

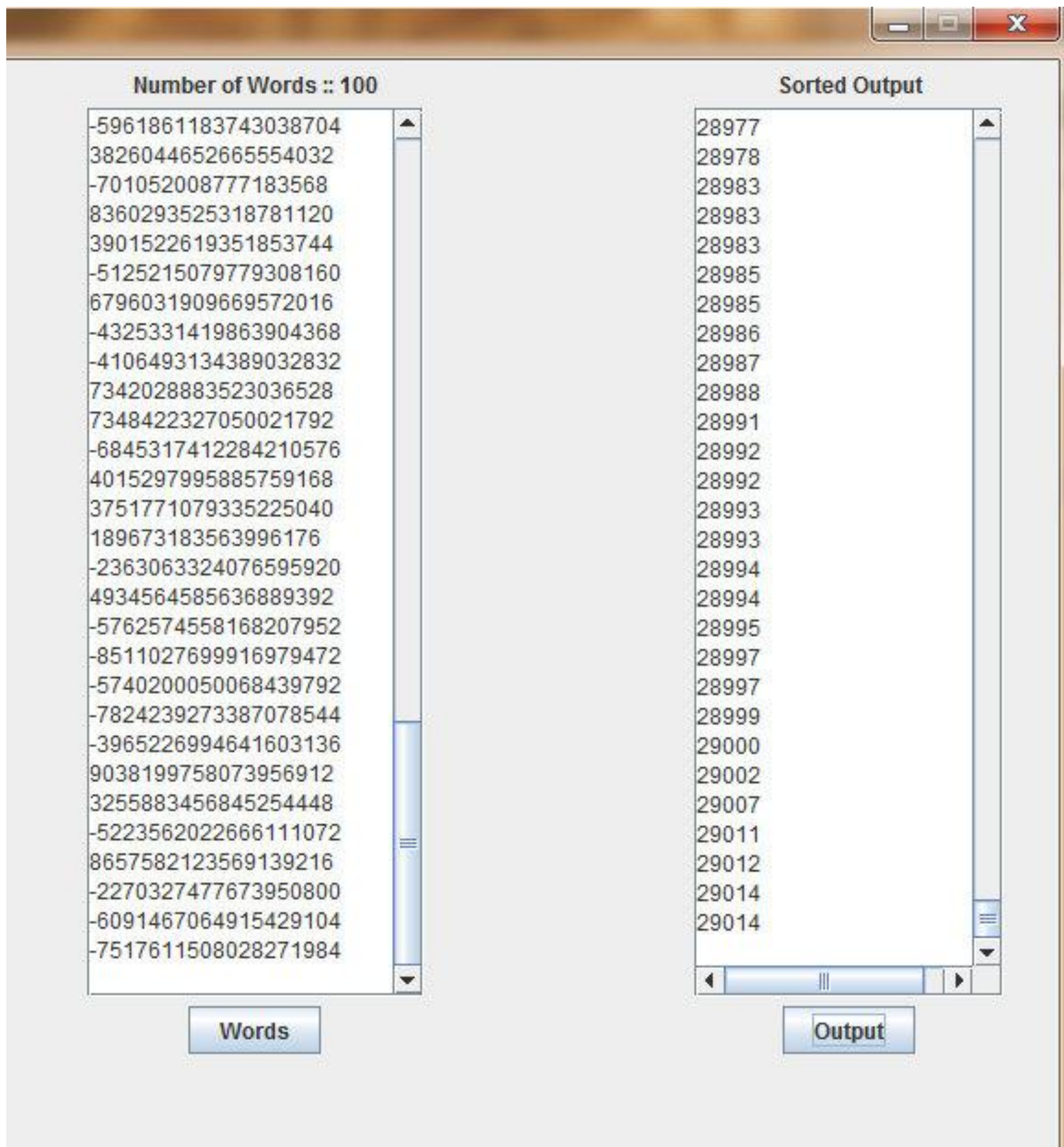


Figure 5.8: Words formed and sorted output of improved randomized signature sort

Therefore, improved randomized signature sort can sort n integers in $O(n)$ expected time using linear space with better performance.

Chapter- 6

Conclusion and Future Scope

This chapter summarizes and evaluates the contributions made by thesis, and speculates on fruitful avenues for future work in this area.

6.1 Conclusion

The comparison-base model is an elegant and general framework in which to study sorting problems and $O(n\, log\, n)$ complexity of sorting is one of the tenets of computer science. However many sorting problems of considerable interest can be cast as integer sorting problems. The complexity of integer sorting on word RAM-like model therefore is of great practical and theoretical significance. A fundamental question therefore is: How fast n integers can be sorted on a w-bit machine?

This thesis described and demonstrated improved randomized signature sort which performs integer sorting and runs under the word RAM model. This improved randomized signature sort shows better performance than existing randomized signature sort. The existing randomized signature sort which was given in 1988, and then 1995 was originally an idea of Arne Andersson [1] but still today many researchers are working on the topic to achieve a new efficient comparison based sorting algorithm as sorting is one of the challenging task. There are few completely new ideas in improved version of signature sort: Like new technique of packing which performs better mechanism to handle varying signature bit size that to be accommodated in the word of limited size. Secondly the implementation mechanism that holds efficient algorithm and faster way to run it with use of bitwise operators. These bitwise operators are considered to run faster than any other operators like addition, subtraction and so on. The use of bitwise operators and hashing has improved the performance of sorting algorithm significantly.

Apart from using the same packed sorting algorithm developed by Paul and Simon [16] for packing signatures into word and sorting, useful algorithm can be developed which deals with signature formation and packing of them into word in a more faster way.

The resultant algorithm after implementation provides more efficient improved randomized signature sort, which is not only stable but better in performance. The algorithm is designed in a way that it becomes easy to implement as well as lesser complex. The implementation of algorithm gives $O(n)$ expected time which uses only linear space. The actual running time of this variant is comparatively very low than existing signature sort.

Testing has been performed to measure the difference in the performance. And in order to measure its efficiency randomized quick sort has been chosen. Quick sort makes $O(n \log n)$ comparisons to sort the n integers and is considered faster in practice than the other algorithms. It is widely used and considered best among all. Therefore, it’s better to compare the improved randomized signature sort with quick sort to measure actual performance. The testing concluded that though randomized quick sort is better than improved randomized signature sort but the difference is so close that can be overcome.

For same, large input sequence length, CPU run-time is far less in case of improved randomized signature sort as compared to existing randomized signature sort. And similar result has been observed in case of space requirement. Thus, clearly improved signature sort is better than existing one.

With new algorithm to perform randomized signature sort, number of comparisons also have been reduced as there is no need to consider each integer for comparison for lesser number of time than existing randomized signature sort. As in the existing one, integers were divided into fields and each field had to be considered for comparison operation, which overall acted like an overhead on the algorithm. Hence by reducing the number of comparison to be performed, algorithm for improved randomized signature sort runs more efficiently.

6.2 Future scope

There are many interesting avenues to explore for future work. The most important issue to address will be algorithms consisting of efficient phases that result in faster computational speed. Also to achieve such an algorithm for the signature sort, that can sort n integers for all word lengths.

Unpacking also calls for new techniques like reverse hash function. Reverse hash function can make it faster as in order to retain the sorted list, there is no need to maintain ranks or indexing would be required. This task might be tedious but very useful.

Despite its excellent asymptotic expected runtime of $O(n)$, Signature sort performs poorly in practice because of most machines' limited word size and because of the very large constants. Therefore, an algorithm can be designed which has no limitation associated with limited word-size. It should work for all word lengths and whole task to be accomplished in linear expected time and lesser complexity.

Research also raises some theoretical questions to be answered like to find tight bounds on deterministic integer sorting. Can the performance of signature sort be matched by deterministic algorithm? Signature sort sorts n integers in $O(n)$ expected time with a word length $w = O(\log n)$ but how well this performs in case when $w \geq logn^{2+\epsilon}$ for arbitrary $\epsilon > 0$.

References

[1] A. Andersson, T. Hagerup, S. Nilsson, R. Raman, Sorting in Linear Time?, J. Comput. Syst.Sci., vol. 57, no. 1, pp. 74-93, 1998.

[2] Thorup M., Randomized sorting in $O(n \log \log n)$ time and linear space using addition, shift, and bit-wise Boolean operations, in "Proc. 8th ACM-SIAM Symp. on Discrete Algorithms (SODA'97)," pp. 352-359, 1997.

[3] Dessmark A., and Lingas A., Improved Bounds for Integer Sorting in the EREW PRAM Model, J. Parallel Distrib. Comput., vol. 48, pp. 64-70, 1998.

[4] Yijie Han and Mikkel Thorup. Integer sorting in $O(n\sqrt{\log \log n})$ expected time and linear space. In *IEEE Symp. on Foundations of Computer Science*, vol. 43, 2002.

[5] B. Vandiver, A.Rolfe, Exploiting sleight-of model to achieve super-luminal sorting, mit, Dec. 2003.

[6] Albers S., and Hagerup T., Improved parallel integer sorting without concurrent writing, Information and Comput., vol. 136, pp. 25-51, 1997.

[7] R.E Tarjan and U.Vishkin. An efficient parallel biconnectivity algorithm. SIAM J. Comput., vol. 14, pp. 862-874, 1985.

[8] H.Bast and T.Hagerup. Fast and reliable parallel hashing. In proc. 3rd Annual ACM Symp. On parallel algorithms and architectures, pp-50-61, 1991.

[9] C.P. Kruskal. Searching, merging and sorting in parallel computation. IEEE Trans. Comput., vol. 32, pp. 942-946, 1984.

[10] Han Y., and Shen X., Parallel integer sorting is more efficient than parallel comparison sorting on exclusive write PRAMs, in "Proc. 1999 Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99), Baltimore, Maryland, January 1999," pp. 419-428, 1999.

[11] F.E. Fich, P.Ragde, and A. Wigderson,. Simulations amoung concurrent -write PRAMs. Algorithmica, vol. 3, pp.43-51, 1988.

[12] R.Cole. Parallel merge sort. SIAM J. Comput., vol. 17, pp.770-785, 1988.

[13] P.Beame and J.Hastad. optimal bounds for decision problems on the CRCW PRAM. J.ACM, 36, pp. 643-670, 1989.

[14] Vaidyanathan R., Hartmann C. R. P., and Varshney P. K., Towards optimal parallel radix sorting, in “Proc. 7th International Parallel Processing Symposium,” pp. 193-197, 1993.

[15] R.W. Floyd and R.L. Rivest. Expected time bounds for selection. Comm. ACM, vol. 18, pp. 165-172, 1975.

[16] W. Paul and J. Simon. Decision trees and random access machines. In Proc. Symp. ¨uber Logik and Algoritmik, 1980.

[17] D. Kirkpatrick and S. Reisch. Upper bounds for sorting integers on random access machines, Theor. Comput. Sci., pp. 263-276, 1984.

[18] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: Introduction to Algorithms, Second Edition, The MIT Press and McGraw-Hill Book Company2001.

[19] M.Ajtai, J.Komlos, and E.Szemeredi. An $O(n \log n)$ sorting network. In Proc. 15th Annual ACM Symp. On Theory of Computing, pp. 1-9, 1983.

[20] A.M. Ben-Amram and Z.Galil. When can we sort in O(n log n) time? In Proc. 34th annual IEEE Symp. On Foundatiuons of computer Science, pp. 538-546, 1993.

[21] B. Cherkassky, A. Goldberg, and C. Silverstein. Buckets,heaps, lists, and monotone priority queues. SIAM J. Comp., vol. 28, no. 4, pp. 1326–1346, 1999.

[22] Andersson A., Fast deterministic sorting and searching in linear space, in “Proc. 1996 IEEE Symp. on Foundations of Computer Science,” pp. 135-141, 1996.

[23] A. Andersson and S. Nilsson. A new efficient radix sort. In proc. 35th Annual IEEE Symp. on Foundations of Computer Science, pp. 714-721, 1994.

[24] A. Andersson, P.Miltersen, andM. Thorup. Fusion trees can be implemented with AC 0 instructions only. Theor. Comput. Sc., 215(1-2):337–344, 1999.

[25] A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In Proc. 32nd STOC, pp. 335–342, 2000.

[26] Y. Han, Improved fast integer sorting in linear space, Inform. and Comput., vol. 170, no.1, pp. 81–94, 2001.

[27] P.C.P Bhatt, K.Diks, T.Hagerup, V.C. Prasad, T.Radzik and S.Saxena. Improved deterministic parallel integer sorting. Inform and Comput., vol. 94, pp.29-47,1991.

[28] A. I. Dumey., Indexing for rapid random access memory systems, Computers and Automation, AIEE-IRE '56 (Eastern), 1956.

[29] S. Cook and R. Reckhow. Time-bounded random access machines. J. Comp. Syst. Sc., vol. 10, no. 2, 1973.

[30] M. Ajtai, J. Komlos, and Szemeredi, Sorting in O(c log n) in parallel steps, Combinatorica, vol. 3, no. 1, pp. 1-19, 1983.

[31] M. Dietzfelbinger, T. Hagerup, J. Katajainen, M. Penttonen, A reliable randomized algorithm for the closest- pair problem, J. Algorithms, vol. 25, 1997.

List of Publications

[1] T. Pathak and D. Garg, Improving performance of randomized signature sort using hashing and bitwise operators, Journal of Global Research in Computer Science, vol 2, no 3, 2011.

[2] T. Pathak and D. Garg, Randomized signature sort: implementation & performance analysis (Communicated).