CONSTANT TIME SORTING AND SEARCHING

A Thesis in

Computer Science

Presented to Faculty of University of
Missouri - Kansas City in partial fulfillment of
requirements for the degree

MASTER OF SCIENCE

By

Sai Swathi Kunapuli

B. Tech in Electrical Engineering
Vignan Institute of Technology & Sciences, Hyderabad, India, 2019

Kansas City, Missouri, United States of America

2022

CONSTANT TIME SORTING AND SEARCHING

Sai Swathi Kunapuli, Candidate for the Master of Science Degree

University of Missouri – Kansas City

ABSTRACT

To study the sorting of real numbers into a linked list on Parallel Random Access Machine model. To show that input array of n real numbers can be sorted into a linked list in constant time using $n^2/\log^c n$ processors for any positive constant c.

The searching problem studied is locating the interval of n sorted real numbers for inserting a query real number. Taking into account an input of n real numbers and organize them in the sorted order to facilitate searching.Initially, sorting the n input real numbers and then convert these real numbers into integers such that their relative order is preserved. Convert the query input real number to a query integer and then search the interval among these n integers for the insertion point of this query real number in constant time.

APPROVAL PAGE

The faculty listed below, appointed by Dean of School of Science and Engineering, must examine the thesis titled as “Constant Time Sorting and Searching” presented by Sai Swathi Kunapuli, candidate for the Master of Science degree, and certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Yijie Han, Ph. D., Chair
School of Science and Engineering

Gharibi Wajeb, Ph. D.
School of Science and Engineering

Sejun Song, Ph. D.
School of Science and Engineering

CONTENTS

TABLES

CHAPTER 1

INTRODUCTION

The requirement for parallel algorithms has become critical in these days and age. Our algorithm has a series of stages that takes many inputs from the input and can execute various instructions at the same time, combining all of the separate outputs to produce the final result. In this study, we use the PRAM (Parallel Random Access Machine) model[23,24] to sort n real numbers into a linked list. A Parallel Random Access Machine is a model that is used for the design of many parallel algorithms. In this model, n processors can conduct independent operations on n data sets in a unit of time. This may result in many CPU's accessing the same memory cells at the same time. This issue is resolved in various ways on the PRAM model: On the EREW (Exclusive Read Exclusive Write) PRAM[23,24], no two processors are allowed to read from or write to the same memory location at the same time, on the CREW (Concurrent Read Exclusive Write) PRAM[23,24], multiple processors are permitted to read from the same memory location at the same time but are not permitted to write to the same memory location at the same time, and in the CRCW (Concurrent Read Concurrent Write) PRAM[23,24], in which multiple processors are permitted to read from or write to the same memory location at the same time. Since, the CRCW PRAM allows multiple processors to read and write simultaneously into a memory cell arbitration schemes are used to resolve concurrent write conflict. On the Priority CRCW PRAM, the processor with the highest priority wins the write on the memory cell among the processors writing to the same memory cell. The processor's index can be used as the priority. On the Arbitrary CRCW PRAM, an arbitrary processor is chosen to win the write from among the processors to write on the same memory cell. On the Common CRCW PRAM, processors write to the same memory cell in a

step must write the same value, which is then written into the memory cell. Priority CRCW PRAM is the strongest of the three CRCW PRAM models; Arbitrary CRCW PRAM is weaker than Priority CRCW PRAM; and Common CRCW PRAM is the weakest of the three. In this study, we shall design algorithm on the Common CRCW PRAM. Because Common CRCR PRAM is weaker than Arbitrary and Priority CRCW PRAM and therefore our algorithm also runs on the Arbitrary and Priority CRCW PRAM.

Let Tp denote the complexity of a parallel algorithm with p processors. Let T_1 be the time complexity of the best serial algorithm for the identical issue. Then $pTp \geq T_1$. When $pTp=T_1$,this parallel algorithm is an optimal parallel algorithm.

When we have a T_P time algorithm that uses P processors, we can represent or translate the time as $T_PP/p+T_P$ when we employ p processors.

A parallel algorithm for a problem of size n that uses polynomial number processors (i.e., n^c processors for a constant c) and runs in polylog time (i.e., $O(\log^c n)$ time for a constant c) is considered to belong to the NC class[5], where NC is Nick's class.

NC algorithms, as well as fast and efficient parallel algorithms, are being developed by researchers in the field of parallel algorithms.

In this paper, we will study sorting real numbers into a linked list in constant time using $n^2/\log^c n$ processors. Previously it is known that n real numbers can be sorted into a linked listin constant time using n^2 processors [10,14,15,16].

On the CRCW PRAM with polynomial number of processors, it is known that sorting n real values into an array takes at least Ω(logn/loglogn) time [3]. If we want to sort them into a padded array, we need at least Ω(loglogn) time [8]. There are fast merging and sorting algorithms [23] but they do not achieve constant time. However, if we arrange them into a linked list, we can demonstrate that it is possible to do so in constant time. Thus, the lower bounds of Ω(logn/loglogn) [3] and Ω(loglogn) [8] are the bottom bounds for arranging integers in an array rather than "sorting" them.

There have been previous results for sorting integers into a linked list[4, 9]. It is known there that n numbers in the range of {0, 1, …, m-1} may be sorted into a linked list in constant timeusing nlogm processors. Functions of n cannot constrain m in this case. Except for prior resultsfor sorting real numbers into a linked list [10,14,15,16], we do not know any other results for parallel sorting real numbers into a linked list in constant time.

In [10,14,15] sorting integers and real numbers into a linked list is considered. The best resultto sort real numbers into a linked list in constant time used n^2 processors [14]. Although in [15] the number of processors is reduced to less than n^2 by using linked list contraction [1,18,19] but the time is not constant.

The traditional way of searching a query number among n numbers is to sort these n numbersand then use binary search to search for the query number. Traditionally the sorting is done bycomparison sort. This requires O(nlogn) time for the sorting and the binary search takes O(logn) time [6].

When the numbers are integers, the sorting can be done in O(nloglogn) time [11] and the searching can be done in min{O(logn), O(loglogm)} time [13] where logm is the number

of bits in the integers, i.e. the input integers are in {0, 1, …, m-1}.

We use the computation model which is basically the computation model used in computational geometry [26]. Besides normal arithmetic and index operations used in computational geometry, we also use some logical operations such as bitwise AND ($\wedge$) and XOR (exclusive-or) operation. These logical operations, when applied to integers and real numbers, enable our algorithms run better. However, these logical operations may not allowed in computational geometry. In the first version of our algorithm, we use the XOR operation on real numbers and use AND and XOR operations on integers. In the second version of our algorithm, we only use AND on integers. We do these because we want to keep the use of logical operations to minimum.

We consider the situation that the input numbers are real numbers, as this is assumed in computational geometry.

We show, by using Han's real number sorting algorithm [12] (which runs on the same computation model as used in computational geometry) , we can sort the input numbers in $O(n\sqrt{\log n})$ time. After sorting we convert these n real numbers into integers while preserving their order. These converted integers can then be packed into one word to facilitate search in constant time.

CHAPTER 2

SORTING IN CONSTANT TIME

2.1 Sorting real numbers into a linked list using n^2 processors in constant time.

We assume that the n input real numbers are distinct. This can be achieved by replacingevery real number a by a pair (a, i) where i is the index of the number a in the input array.

Firstly, let us discuss about the algorithm on how to sort the real numbers in linked list usingconstant time using n^3 processors. Let us say, A[0 n-1] be the input array of n real numbers and we have n^3 processors to achieve constant time.

Assign n processors to each element of the array to compare it with the other elements in the array. It will write as 1 for the elements that it greater than the given element and 0 for the elements if it is less than it. For example, we have the given input array elements as 4,2,5,1,6,3,9. Let us pick an element 5 from the array. As said above, it marks 1 to the elements greater than5 and 0 for the ones lesser than 5. So, the output is 0,0,0,0,1,0,1. We use the n^2 processors to theelements marked as 1 and find the smallest number among them (i.e., 6) in constant time [25,27]and link it to the element 5. So, here we have 6 and 9 out of which 6 is the minimum. So, 6 islinked to 5. This process is executed in parallel to all the elements in the array, and we get thefinal sorted linked list of elements. This algorithm can be done in constant time using n^3 processors.

Now, we let us show the algorithm on sorting the real numbers into a linked list using n^2 processors in O(loglogn) time on the Common CRCW PRAM. This algorithm is like the abovealgorithm where we assign n processors to compare a number to the rest of the elements

in thearray. Now, we need to compute the minimum of n numbers using n processors. This can be done in O(loglogn) time [25,27]. Let us say A[0. n-1] be the input array of n real numbers. As above, the comparison task of comparing one element A[i] to other elements takes constant time. Now, we need to find the minimum of elements in A that are larger than A[i]. Let us saym is the smallest element. Now, for each element in A[i] we will copy it into a new array A_i. This usually take constant time. We now compare A[i] with every element $A_i[j]$ in A_i. If $A[i] \geq A_i[j]$ then we will do $A_i[j]$ =MIN. Then we will find the smallest element $A_i[k]$ in A_i. This takes constant time using n^{1+e} processors (or O(loglogn) time with n processors) for A_i [22,24]. For all i=0, 1… n-1, this takes constant time with n^{2+e} processors (or O(loglogn) time with n^2 processors). $A_i[k]$ is the smallest element larger than A[i]. Thus, we can make a link from A[k]to A [i].

Now we show our new algorithm which allows to sort n real numbers into a linked list in constant time with n^2 processors. We divide the input numbers into $\sqrt{n}$ groups. So, now each group has $\sqrt{n}$ numbers. Assign $n^{3/2}$ processors for each group. So now the total number of processors to do this will be $\sqrt{n}$ x $n^{3/2}$ processors which is n^2 processors. We already know thatbuilding a sorted linked list with $n^{3/2}$ processors of $\sqrt{n}$ numbers take constant time. Now we have $\sqrt{n}$ groups with sorted linked lists. Since we have $\sqrt{n}$ groups there will be O(n) pairs of groups in total. Let us assign n processors for every pair of groups. So, we require n processors x O(n) pairs which is $O(n^2)$ processors total. So, for every number in the group, we can use $\sqrt{n}$ processors. So, we require n processors for each group. Now, let us say we have a number **A** inGroup 1. It finds the smallest number **B** larger than it in Group 2 by comparing with every number in group 2 and using the sorted linked list already built for group 2. This process is repeated for all the pairs of groups like Group 1, Group 3 and Group 1, Group 4

etc. We find $\sqrt{n}-1$ smallest numbers larger than **A**. In general, if we do it in parallel each number find $\sqrt{n}-1$ smallest numbers larger than it. Each number then uses n processors to find the minimum among these $\sqrt{n}-1$ smallest numbers in constant time [25,27]. So, in total the proposed algorithm uses n^2 processors to sort the n real numbers in a linked list in constant time.

Finally, let us discuss about the algorithm which is used to sort the real numbers in the linked list using less then n^2 processors. Divide n numbers into n/t groups with t numbers in each group. First sort the t numbers in each group into a linked list in constant time using $(n/t)t^2$ processors. Now for about every m nodes (between m and 2m nodes), we build a supernode. Initially we have n/t linked lists. Each linked list has t nodes. Combine about every consecutive m nodes to form a supernode. We have t nodes in linked list so we have O(t/m) super nodes. This can be down in $O(n/p+\log^{(c)}n\log t)$ time [1,14,15], where $\log^{(1)}n=\log n$ and $\log^{(c)}n = \log\log^{(c-1)}n$. The t/m supernodes for each sorted link of t nodes forms a sorted supernode linked list. Two supernode sorted linked lists with t/m nodes each can be merged into one lined list in constant time using $(t/m)^2$ processors. Let us say supernode s in one supernode linked list is to be inserted between supernode s_1 and supernode s_2 of the other supernode linked list. Then s uses O(m) processors to compare it with every nodes in s_1 and s_2 to find the exact position it needs to be inserted. Now merge every pair of about m nodes using m^2 processors in constant time.

Finally, let us discuss about the algorithm which is used to sort the real numbers in the linked list using less then n^2 processors. Divide n numbers into n/t groups with t numbers in each group. First sort the t numbers in each group into a linked list in constant time using $(n/t)t^2$ processors. Now for about every m node (between m and 2m nodes), we build a supernode.

Initially we have n/t linked lists. Each linked list has t nodes. Combine about every consecutive m node to form a supernode. We have t nodes in linked list, so we have O(t/m) super nodes. This can be down in $O(n/p+\log^{(c)}n\log t)$ time [1,8,9], where $\log^{(1)}n = \log n$ and $\log^{(c)}n = \log\log^{(c-1)}n$. The t/m supernodes for each sorted link of t nodes forms a sorted supernode linked list. Two supernode sorted linked lists with t/m nodes each can be merged into one linked list in constant time using $(t/m)^2$ processors. Let us say supernode s in one supernode linked list is to be inserted between supernode s_1 and supernode s_2 of the other supernode linked list. Then s uses O(m) processors to compare it with every node in s_1 and s_2 to find the exact position it needs to be inserted. Now merge every pair of about m nodes using m^2 processors in constant time.

Therefore there are $(n/t)^2$ pairs of linked lists. For every pair, we use $(t/m)^2$ processors to merge supernode linked lists. So, we use $(n/m)^2$ processors for merging the supernodes. For each supernode s we used nm/t processors (m processors for each of the n/t pairs) for comparingit with the nodes in other supernodes. Because we have n/m supernodes, therefore the processused is n^2/t processors. For merging the m nodes in one supernode list with m nodes in othersupernodes list we used $(n/t)^2(t/m)m=(n/m)(n/t)m=n^2/t$ processors and logm time. If we let m^2=tthen we used n^2/t processors and logt time.

The two extremes are t=1 which we use n^2 processors and sort real numbers into a linked list in constant time and when t=n where we use n processors and sort real numbers into a linked list in logn time.

2.2 Prepare for sorting real numbers into a linked list using $n^2/\log^c n$ processors.

A parallel algorithm for sorting n input real numbers into a linked list in constant time is described. This algorithm works by grouping input real numbers, let us say, splitting A[0…n-1] real numbers into $n/\sqrt{\log n}$ groups. We enumerate all permutations ~~of t~~he $\sqrt{\log n}$ numbersin every group. Among all these $\sqrt{\log n}!$ permutations there is only one permutation in which these $\sqrt{\log n}$ numbers are in sorted order (assuming that all input numbers are different). For each permutation of the numbers in a group we use $\sqrt{\log n}$ processor (one processor for each number) and therefore we used $\sqrt{\log n}! * \sqrt{\log n}$ processors for each group and for the n input real numbers we used $(n/\sqrt{\log n}) * \sqrt{\log n}! * \sqrt{\log n} = n\sqrt{\log n}!$ processors. For each group thepermutation with the sorted order of numbers is selected in constant time by verifying the $\sqrt{\log n}$ numbers are in sorted order using $\sqrt{\log n}$ processors. This is how internal sorting is carried out.

To continue the sorting process, each element e in a group G is compared to the elements in the next group G_i, $0 \le i < \sqrt{\log n}$, and fitted in a suitable position by determining its rank in G_i.This is do~~ne by~~ using $\sqrt{\log n}$ processors to compare it to every number in the (sorted) group G_1. e then enu~~mera~~tes ~~(1 +~~ $\sqrt{\log n})^{\sqrt{\log n}}$ possi~~bilitie~~s using ~~$\sqrt{\log}$~~ n base $\sqrt{\log n}$ digits. There are $(1 + \sqrt{\log n})^{\sqrt{\log n}}$ patterns in these digits. The pattern $a_0 a_1 \ldots a_{\sqrt{n}-1}$ denotes that e has rank a_i in G_i. Associated with pattern $a_0 a_1 \ldots a_{\sqrt{n}-1}$ is the pre-computed value $a_0 + a_1 + \cdots a_{\sqrt{n}-1}$which is the rank of e in $G_0 \cup G_1 \cup \ldots \cup G_{\sqrt{\log n}-1}$. For each permutation e then uses $\sqrt{\log n}$ processors with the i-th processor p_i to verify whether the rank of e in group G_i is a_i. If

the rank is not a_i then p_i will cancel this permutation by (concurrent) write to a predefined memory cell for this permutation. Thus only one permutation is not cancelled and the rank precomputed for this permutation is fetched. This determines the rank of e in $G_0 \cup G_1 \cup \ldots \cup G_{\sqrt{\log n}-1}$. e used $(1+\sqrt{\log n})^{\sqrt{\log n}} * \sqrt{\log n}$ processors. Thus for n real numbers the total number of processors used is $n * (1+\sqrt{\log n})^{\sqrt{\log n}} * \sqrt{\log n}$. The time complexity is constant time.

In the next step we again combine $\sqrt{\log n}$ groups into one group. This time we have, for each number e, $(1+\log n)^{\sqrt{\log n}}$ patterns because the rank of e in each group of $\log n$ numbers canbe from 0 to $\log n$. Thus we will use $n * (1+\log n)^{\sqrt{\log n}} * \log n$ processors.

For a positive integer c we will run the above process 2c times. Thus we will use O(c) steps and use $n * (1+(\log n)^c)^{\sqrt{\log n}} * (\log n)^c$ processors. We have sorted $(\log n)^c$ numbers in each of the $n/(\log n)^c$ groups.

EXAMPLE:

Let us now demonstrate our above approach using different numbers as an example. AssumeA[0,...,n-1] is the input array of n real values. Using $n * (1+\log n)^{\sqrt{\log n}} * \log n$ processors, we achieve this in constant time. For example, consider an input array of 2,3,8,6,12,19,5,4,1,0,9,7,10,18,16,13 where n=16.

In stage one of the process, we divide into groups depending on $\sqrt{\log n}$. When we solve, we get $\sqrt{\log 16}=2$, which represents two processors for each group. Following the procedure, we divided numbers into 8 groups with 2 numbers in one group, as indicated in

[2,3],[8,6],[12,19],[5,4],[1,0],[9,7],[10,18],[16,13]. Each group is solved by determining the proper order from the all potential $\sqrt{\log n}!$ (For our example, 2!=2.) permutations. Thus the first group will have two permutations: 2, 3, and 3, 2 and it determined that 2. 3 is in sorted order. The second group will have two permutations: 8, 6 and 6, 8 and it determined that 6, 8 is in the sorted order. And so on. Thus for each group we used 4 processors and the total number of processors used is 16/2*4=32. The time is constant. After this stage we get [2,3],[6,8],[12,19],[4,5],[0,1],[7,9],[10,18], [13,16].

As we move on to stage II of the process, after we have completed internal group sorting, we will combine $\sqrt{\log n}$=2 groups into one group. To combine [2,3] and [6,8] into one group each of the 2, 3, 6, 8 will use 4 processors to determine its rank in each group. For example, 3 will use 4 processors, use 2 processors to determine its rank in [2,3] as 1 and use 2 processors to determine its rank in [6,8] as 0. Then for each number we form $(\sqrt{\log n} + 1)^{\sqrt{\log n}} = 3^2 = 9$ permutations: p_0=00, p_1=01, p_2=02, p_3=10, p_4=11, p_5=12, p_6=20, p_7=21, p_8=22 and use $(\sqrt{\log n} + 1)^{\sqrt{\log n}}\sqrt{\log n}$=9*2=18 processors, two processors for each permutation. Thus 3 use 2 processors to check p_0 and finds that p_0 is incorrect as it indicates that 3 has rank 1 in [2,3] and rank 0 in [6,8]. Thus p_0 will be taken out of consideration (crossed out). The only permutation that is not crossed out is p_3=10 as it indicates that 3 has rank 1 in [2,3] and rank 0in [6,8]. Thus 3 picks the pre-computed rank of 0+1=1 for p_3. Thus at the end of this stage wegot [2,3,6,8],[4,5,12,19],[0,1,7,9],[10,13,16,18].

2.3 Sorting real numbers into a linked list with $n^2/\log^c n$ processors.

For a given total number of "n" inputs, dividing them into $n/(\log n)^c$ groups with $(\log n)^c$ numbers in each group. As described in the Section 2 we use $n * (1 + (\log n)^c)^{\sqrt{\log n}} * (\log n)^c$ processors to sorted $(\log n)^c$ numbers in each of the $n/(\log n)^c$ groups in constant time.

For each group of sorted $(\log n)^c$ numbers we sample every $(\log n)^d$-th number (with 0 <d< c) and thus we sampled $(\log n)^{c-d}$ numbers from each group and among the n input numbers we sampled $n/(\log n)^d$ numbers.

We now sort these $n/(\log n)^d$ numbers into a sorted linked list in constant time using $(n/(\log n)^d)^2$ processors using the algorithm in[14] .

Now for each number a we use $n/(\log n)^d$ processors to compare it to all the numbers on the sorted linked list to s: the largest number smaller than a and l: the smallest number larger thana among numbers in the sorted linked list. Because numbers in the linked list are sorted and therefore s and l can be found in constant time. s and l are neighboring elements on the sortedlinked list.

Now between two neighboring elements s and l on the sorted linked list there can be at-most $n/(\log n)^d$ numbers fell in between. This is because for each group of sorted $(\log n)^c$ numbersthere can be at most $(\log n)^d$ numbers fell between s and l. For otherwise if more than $(\log n)^d$ numbers fell in between s and l then there is at least sampled number fell in between s and l because we sampled every $(\log n)^d$-th number from these $(\log n)^c$ numbers.

But between s and l there is no another sampled number.Thus there are at most n/$(\log n)^d$ numbers fell in between s and l.

Now for all numbers fell in between s and l we sort them into a linked list. We use n/$(\log\ n)^d$ processor for each number (thus we used a total of n^2 /$(\log\ n)^d$ processors). Because there nomore than n/$(\log n)^d$ numbers between s and l and therefore we have at least m^2 processors for the m numbers between s and l. Thus we can sort the numbers between s and l into a sortedlinked list in constant time use[14] .

After the numbers in each interval between s and l are sorted into a linked list we can connectsthese linked lists into one sorted linked list.Because c is an arbitrarily large constant and d < c thus d can be an arbitrarily large constant.

CHAPTER 3

SEARCHING IN CONSTANT TIME

3.1 Converting real numbers to integers while preserving their order.

Let $a_0, a_1, \ldots, a_{n-1}$ be the n input real numbers. As our method is not convenient to deal with negative numbers, we will convert negative numbers to nonnegative numbers by finding the minimum number m among $a_0, a_1, \ldots, a_{n-1}$ and if m is negative then we will add –m to every a_i, i=0, 1, …, n-1.

First, we sort $a_0, a_1, \ldots, a_{n-1}$ in $O(n\sqrt{\log n})$ time [12]. Let $a_{i0}, a_{i1}, \ldots, a_{i(n-1)}$ be these real numbers in the sorted order. In the first version of our algorithm, we view these numbers as binary numbers and we do a_{ij} XOR $a_{i(j+1)}$, and then find the most significant bit b_{ij} of a_{ij} XOR $a_{i(j+1)}$ that is 1, for j=0, 1, n-2. The most significant bit can be found in multiple ways as we will mention below. Here the least significant integral bit is bit 0. If we cut a_{ij} and $a_{i(j+1)}$ at bit b_{ij} , i.e. take bits that are at least as significant as b_{ij} and discard bits that are less significant than b_{ij}, then the relative order a_{ij} and $a_{i(j+1}$ is preserved in the result integer a'_{ij} and $a'_{i(j+1)}$ (may need to shift or multiply by 2^{-bij} if $b_{ij} < 0$ to convert fraction part into integer). Here we got n-1 bits b_{ij}, j=0, 1, …, n-2, and we will find the smallest one b among them. We then cut all n real numbers $a_0, a_1,.., a_{n-1}$ at bit b. To cut real number we do int($a*2^{-b}$).

The way to find the most significant bit can be either by techniques presented in [7], or by taking the logarithm base 2 (then take the ceiling of it to make it an integer), or we can first scale all input real numbers by a factor to make the absolute value of every one of them to be less than 1. Say scaled value of $a_0, a_1, \ldots, a_{n-1}$ is $s_0, s_1, \ldots, s_{n-1}$. Then we take $scale_i = \lceil 1/|s_i| \rceil$,

i=0, 1, …, n-1. The largest $scale_i$ value $scale_{max}$ is chosen to scale all s_i's. That is we do $int(s_i*scale_{max})$ for i=0, 1, …, n-1.

Here is an example of these operations:

Now,let us follow this version with the use of XOR:

Remembering the truth table of XOR as described below:

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Table(i). XOR Table

Let me illustrate the procedure with the help of example. A set of real numbers are considered along with a query input number as shown below.

Let the real numbers be,

0 1 1 0

1 1 0 1

0 1 0 0

0 1 0 0

A query input q, 0

1

1

0

As mentioned earlier, initially we need to sort the input real numbers in smaller to larger orderor larger to smaller order. I have arranged them from smaller to larger and re-arranged input can be seen below.

Now, XOR is performed with each number with its adjacent number which let us find the bit position to cut. We will find the lowest bit position b among these found cut bit positions and cut all input integers and the query integer at bit b. And operation can be continued with bits that are at least as significant as the b-th bit.

Step I: 0 0 1 1 **0**

0 1 0 1 **1**

0 0 1 0 **1**

0 0 1 0 **0**

0 1 0

1 1 1

0 1 1

0 1 1

Thus, the first two number needs to be cut at bit 2 (bits are counted from the least significant integral bit at 0 with bits after decimal point being at negative positions). The second and the third number need to be cut at bit 3 and the third and the fourth, number need to be cut at bit

2. Thus bit 2 is the lowest bit for us to cut. Thus, all the input numbers will be cut at the second bit position as shown below.

Step II:

0	0	1	1	**0**
0	1	0	1	**1**
0	0	1	0	**1**
0	0	1	0	**0**

In the second version of our algorithm, we do not use XOR. Instead, we do subtraction for each number with its adjacent number which let us find the bit position to cut. This is shown here:

Step I:

0	0	1	1	**0**
0	1	0	1	**1**
0	0	1	0	**1**
0	0	1	0	**0**
0	0	0		
1	1	0		
0	1	0		
0	1	1		

And thus, in this version the bit for cutting all input real numbers is at bit position 0.

3.2 Our Algorithms

After these real numbers are cut, they become integers. Now we will perform Step III.

Step III: Now we will pack all the cut integers (not the cut query integer) into a word. When we pack them, we will add a sign bit which is 1 at the front of each integer. The idea of using the sign bit can be found in [2].

If the cut integers have t bits each after adding the sign bit each of them has t+1 bit. In the first version of our algorithm, we arranged the integer from larger to smaller and have them packed as shown here:

1 1 1 **1** 1 0 **1** 0 1 **1** 0 0

Here the bits in boldface are the sign bits.

In the second version of our algorithm, we will arrange integers from small to larger and thusthe packed word of the integers will be like this:

1 0 0 **1** 0 1 **1** 1 0 **1** 1 1

We will call this word A. Up so far; we have accomplished the preprocessing and it takes $O(n\sqrt{\log n})$ time.

To search for the query real number, we first cut it at the b-th bit position as we have the cut query integer here as $\begin{matrix}0\\1\end{matrix}$.

Note that if the minimum value m of a_0, a_1, …, a_{n-1} is negative then we need add -m to thisquery number before cutting it at the b-th bit.

We then make n copied of the cut query integer by multiply it with $(0^b1)^n$. This is a constant that can be prepared in O(logn) time in the preprocessing stage. What we got is:

```
      (0  0 1     0 0 1     0 0 1     0  0 1) * 0  1
=     0  0 1     0 0 1     0 0 1     0 0 1
```

The bits in boldface are sign bits. We will call this word B. Now we do A-B. In the firstversion of our algorithm, we got:

```
      (0  0 1     0 0 1     0 0 1     0  0 1) * 0  1
=     0  0 1     0 0 1     0 0 1     0 0 1
```

The bits in boldface are sign bits. We will call this word B. Now we do A-B. In the firstversion of our algorithm, we got:

```
      1  1 1     1 1 0     1 0 1     1 0 0
-     0  0 1     0 0 1     0 0 1     0 0 1
  ------------------------------------------
      1  1 0     1 0 1     1 0 0     0 1 1
```

In the difference C we got the first 3 sign bits are 1's and the last sign bit is 0. This says that the first three numbers are no less than 01 and the last number is less than 01. Thus 01 is fallingbetween the third and the fourth integers. We need to extract out this information. To do this we first extract the sign bits by AND C with a constant $(10^b)^n$. This constant, again, can be prepared in the preprocessing stage in O(logn) time. The result of this AND is shown here:

	1	1	0	**1**	0	1	**1**	0	0	**0**	1	1
AND	**1**	0	0	**1**	0	0	**1**	0	0	**1**	0	0
	1	0	0	**1**	0	0	**1**	0	0	**0**	0	0

We will call this result D_1.

In the second version the result of subtraction is:

	1	0	0	**1**	0	1	**1**	1	0	**1**	1	1
-	**0**	0	1	**0**	0	1	**0**	0	1	**0**	0	1
	0	1	1	**1**	0	0	**1**	0	1	**1**	1	0

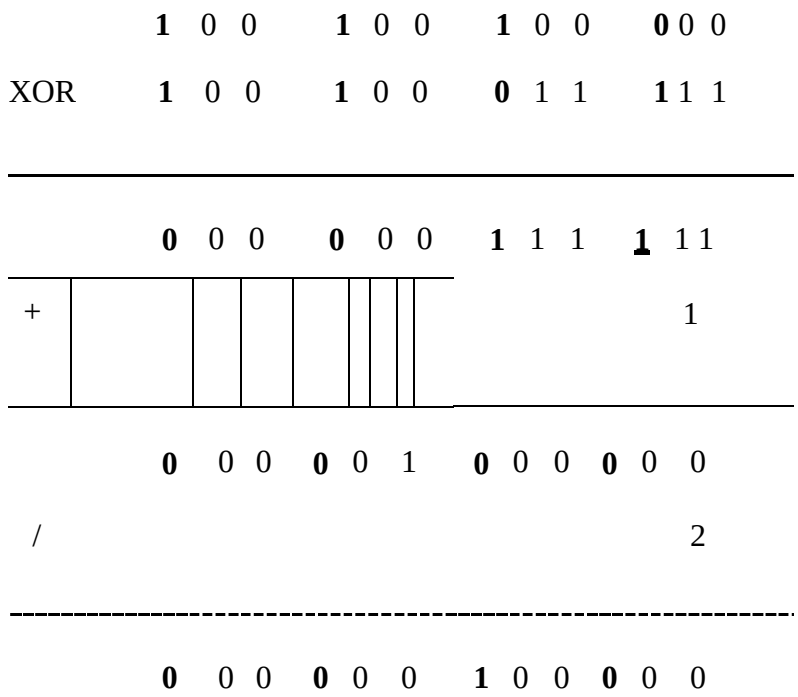
And the AND operation will give us:

	0	1	1	**1**	0	0	**1**	0	1	**1**	1	0
AND	**1**	0	0	**1**	0	0	**1**	0	0	**1**	0	0
	0	0	0	**1**	0	0	**1**	0	0	**1**	0	0

We will call this result D_2.

In the first version of our algorithm, we need to find the least significant bit that is 1 This isachieved by doing:

$((D_1 \text{ XOR } (D_1\text{-}1)) +1)/2$:



We will call this result E_1. After we find the most significant bit of E_1 that tells us that 01 is between the third and the fourth integer. Now we need to compare the cut query integer 01 with the third and the fourth integer and we found that 01 is equal to the third integer. At this point we need compare the input query real number q which is 0110 here with the real numberwhere the third integer is derived, i.e., 0100. Because 0110 > 0100 and therefore we know that0110 is between 0100 and 1011.

In the second version of our algorithm, we just need to find the most significant bit that is 1. This turn out to be the 1 shown in italic here:

0 0 0 ***1*** 0 0 **1** 0 0 **1** 0 0

This says that 01 is between the first integer and the second integer and thus we need to compare it with the first and the second integers. It turns out that 01 is equal to the second integer and thus we need to compare the real query number 0110 to the real number the second integer is derived from, i.e., 0100. Because 0110> 0100 and therefore we know that 0110 is between 0100 and 1011.

Note that in the second version we did not use the XOR operation.

CHAPTER 4

THEOREM

Theorem 1. n real numbers can be sorted into a linked list in constant time using n^2 processors on the Common CRCW PRAM.

We have been able to optimize the existing algorithms with less number processors and time. Earlier, we had algorithms like sorting of n real numbers into a linked list in constant time using n^3 processors and sorting of n real numbers into a linked list in O(loglogn) time using n^2processors [10,16].

Theorem 2. n real numbers can be sorted into a linked list in O(logt) time with n^2/t processors, where t can range from constant up to n.

Earlier, we had algorithms like sorting of n real numbers into a linked list in constant time usingn^3 and sorting of n real numbers in O(loglogn) time using n^2 processors [16]. We also came upwith an algorithm to sort the n real numbers in linked list using less than n^2 processors [16].

Theorems 1 and 2 are the results achieved before me[14,15].

Theorem 3. n Real Numbers can be sorted into a linked list in Constant Time Using $n^2/\log^c n$ Processors

Theorem 4.n real numbers can be preprocessed in $O(n\sqrt{\log n})$ time to support searching in constant time.

Theorems 3 and 4 are achieved by me together with my advisor[20,21].

CHAPTER 5

CONCLUSIONS

We discussed about sorting n real numbers into a linked list using o(n^2) processors in constant time. We have followed the approach to assign the processors by dividing the given input into groups. The approaches of solving correct order from potential permutations and finding rank made algorithm work efficiently to sort the given input of array.

Currently we do not know how to reduce the number of processors further to reach constant time for sorting ‘n’ real numbers into a linked list. The problem is that after we sorted real numbers into a linked list we cannot sample every k-th number in constant time because sortednumbers are on linked list.

We showed the approach to preprocess n real numbers to support search in constant time. We are exploring this approach for other prominent computational geometry problems.

REFERENCES

[1]. R. Anderson, G. Miller. “Deterministic parallel list ranking”. *Algorithmic*, Vol. 6, 859-868,1991.

[2]. A. Andersson, T. Hagerup, S. Nilsson, R. Raman. “Sorting in linear time?” *Proc. 1995 Symposium on Theory of Computing* STOC'1995, 427-436(1995). Also, in *Journal of Computer and System Science*, Vol. 57, 74-93,1998.

[3]. P. Beame, J. Hastad, “Optimal bounds for decision problems on the CRCW PRAM” , *Proc.1987 ACM Symp. On Theory of Computing* (STOC’1987), 83-93(1987).

[4]. P.C.P. Bhatt, K. Diks, T. Hagerup, V.C. Prasad, T. Radzik, S. Saxena, “Improved deterministic parallel integer sorting,” *Information and Computation*, Vol. 94, 29-47(1991).

[5]. S. A. Cook, “Towards a complexity theory of synchronous parallel computation,” *L’ Enseignement Mathématique*,Vol. 27, 99-124(1981).

[6]. T.H. Corman, C.E. Leiserson, R.L. Rivest, C. Stein. *Introduction to algorithms*. 3rd Edition, The MIT Press. 2009.

[7]. M. L. Fredman, D. E. Willard. “BLASTING through the information theoretic barrier with FUSION TREES”. *Proc.1990 ACM Symposium on Theorem of Computing* (STOC’1990),1-7(1990).

[8]. T. Goldberg, U. Zwick, “Optimal deterministic approximate parallel prefix sums and their applications”, *Proc. 3rd. Israel Symp. On Theory and Computing Systems,* 220-

228(1995).

[9]. T. Hagerup. “Towards optimal parallel bucket sorting”, *Information and Computation*. Vol. 75,39-51(1987).

[10]. Y. Han, N. Goyal, H. Koganti, “Sort Integers into a Linked List”, *Computer and Information Science*. Vol. 13, No. 1, 51-57(2020).

[11]. Y. Han. “Deterministic sorting in O (nloglog n) time and linear space”. *Journal of Algorithms*, Vol. 50, 96-105(2004).

[12] Y. Han. “Sorting real numbers in $O\ (n\sqrt{\log n})$ time and linear space”. *Algorithmic* Vol. 82, 966-978(2020).

[13] Y. Han, H. Koganti. “Searching in a sorted linked list”. *In Proceedings of the 17th Int. Conf. on Information Technology* (ICIT'2018).

[14] Y. Han, P. Kasani, “Sorting real numbers into a linked list on the PRAM model”, *Proc. of the 2021 Int. Conf. on List Science, Engineering and Technology*, 45-49(2021).

[15]. Y. Han, P. Kasani. “Time processor trade-off for sorting real numbers into a linked list”. *Proc. International Conference on Computation Structures and Algorithms*. 40-44(2021).

[16]. Y. Han, T. Sreevalli, “Parallel merging and sorting on linked list”, *International Journal*

of Computer, and Information Technology (IJCIT). Vol. 10, No. 2, (March 2021), to appear.

[17]. Y. Han, "Uniform linked list contraction", Paper 2002.05034 in arXiv.org.

[18]. Y. Han. "Matching partition a linked list and its optimization". *Proc. 1989 ACM Symposium on Parallel Algorithms and Architectures* (SPAA'89), 246-253 (June 1989).

[19]. Y. Han. "Parallel algorithms for computing linked list prefix". *Journal of Parallel and Distributed Computing* Vol. 6, 537-557(1989).

[20]. Y. Han, S.S Kunapuli. "Sorting real Numbers into a linked list in Constant Time Using $n^2/\log^c n$ Processors". *In IAARHIES Conference on IT & Computer Science. Edmonton, Canada* 21-22 May,2022.

[21]. Y. Han, S.S Kunapuli. "Preprocess of n real numbers in $O\ (n\sqrt{\log n}\ )$ time to support searching in constant time". To appear in *World Congress on Information Technology and Computer Science* (WCITSC), 2022.

[22]. P. Kasani. "Sorting real numbers into a linked list on the PRAM model". Master's Thesis. University of Missouri at Kansas City, 2022.

[23]. J. J´aJ´a. *An Introduction to Parallel Algorithms.* Addison Wesley, Reading, MA, 1992.

[24]. R. M. Karp, V. Ramachandran, "Parallel algorithms for shared-memory machines*". In Handbook of Theoretical Computer Science* (Vol. A): *Algorithms and Complexity*, J. van Leeuwen, Ed., New York, NY: Elsevier, 869-941(1991).

[25]. C. P. Kruskal. "Searching, merging, and sorting in parallel computation". *IEEE Trans.Compute.*, C-32, 942-946(1983).

[26] F.P. Preparata, M. Ian Shamos. *Computational Geometry, An Introduction.* Springer-Verlag, 1985.

[27]. L. G. Valiant. "Parallelism in comparison problems". SIAM *Journal on Computing*, Vol. 4.No.3, 348-355(1975).

VITA

Sai Swathi Kunapuli was born in Hyderabad, Telangana, India on October 4th, 1997. She attended elementary school named as Gautami Talent High School and graduated in the academic year 2013. Later, she got admitted into one of the top colleges and finished her bachelor's from "Vignan Institute of Technology & Sciences, Hyderabad" in 2019. In the year 2020, she was placed in a MNC called Wipro Limited and worked for a year and half. During Aug 2021, she entered University of Missouri Kansas City to complete his Master of Science Degree in Computer Science in Dec 2022.