10-1990

Optimal Parallel Solutions to the Neighbor Localization Problem and Integer Sorting: A Fine Grained Approach

Ramachandran Vaidyanathan

Carlos R.P. Hartmann
Syracuse University, chartman@syr.edu

Pramod K. Varshney
Syracuse University, varshney@syr.edu

SU-CIS-89-11

Optimal Parallel Solutions to the Neighbor Localization Problem and Integer Sorting: A Fine Grained Approach

Ramachandran Vaidyanathan, Carlos R.P. Hartmann, and Pramod K. Varshney

Revised October 1990

School of Computer and Information Science
Suite 4-116
Center for Science and Technology
Syracuse, New York 13244-4100

(315) 443-2368

SU-CIS-89-11

Optimal Parallel Solutions to the Neighbor Localization Problem and Integer Sorting: A Fine-Grained Approach

Ramachandran Vaidyanathan, Carlos R.P. Harmann, and Pramod K. Varshney

October 1989*

School of Computer and Information Science
Syracuse University
Suite 4-116
Center for Science and Technology
Syracuse, NY 13244-4100

(315) 443-2368

***Revised October 1990**

CIS 89-11
October 1989

Optimal Parallel Solutions to the Neighbor Localization Problem and Integer Sorting: A Fine-Grained Approach [1]

(Revised Version)

Ramachandran Vaidyanathan[2]

Carlos R. P. Hartmann[3]

Pramod K. Varshney[4]

[1]This work was partially supported by The Northeast Parallel Architectures Center (NPAC) at Syracuse University, Syracuse, NY 13244 and The Rome Air Development Center, under contract F30602-88-D-0027.

[2]R. Vaidyanathan was with the Electrical & Computer Engineering Department of Syracuse University and is currently with the Electrical & Computer Engineering Department of Louisiana State University, Baton Rouge, LA 70803-5901. e-mail: vaidy@max.ee.lsu.edu

[3]C. R. P. Hartmann is with the School of Computer & Information Science at Syracuse University, Syracuse, NY 13244-4100. e-mail: hartmann@top.cis.syr.edu

[4]P. K. Varshney is with the Electrical & Computer Engineering Department of Syracuse University, Syracuse, NY 13244-1240.e-mail: varshney@sunrise.acs.syr.edu

Abstract

In this report, a fine-grained decomposition approach is used to obtain an optimal parallel solution to the Neighbor Localization Problem, which in turn is œ used to sort n $\Theta(\log n)$-bit numbers optimally on an EREW model. The model of computation used is the EREW Reconfigurable PRAM (R-PRAM) that permits the use of "very small" processors. The main result of this report is a parallel EREW R-PRAM algorithm that sorts n $\Theta(\log n)$-bit numbers in $\Theta(\log n)$ time with $\Theta(n \log n)$ "work". The proposed algorithm is asymptotically optimal in time and efficiency. If a weaker variant of the R-PRAM (called the ISR-PRAM) is used, the efficiency suffers only a slight degradation.

Keywords: Integer Sorting, ISR-PRAM, Model of Computation, PRAM, Parallel Processing, R-PRAM, Sorting.

Contents

List of Figures

List of Tables

1 Introduction

It is well known that n numbers (keys) can be sorted sequentially in $\Theta(n \log n)$ time, where each unit of time is the time required to compare two keys. Considerable work has been done towards solving this problem in parallel. The AKS sorting network [2] and a parallel merge sorting algorithm due to Cole [6], sort n keys in $\Theta(\log n)$ time with $\Theta(n)$ processors. Azar and Vishkin [4] have proved that the optimal processor-time product of $\Theta(n \log n)$ for comparison-based sorting of n keys cannot be achieved with a time that is a lower order than $\Theta(\log n)$. Thus, the AKS network and Cole's algorithm are optimal.

The above results are for the general sorting problem where no assumption is made about the length of the keys to be sorted. In particular, if the keys are restricted to assume values from $\{0, \ldots . n^{\Theta(1)}\}$, the n keys can be sorted sequentially in $\Theta(n)$ time [9]. This restricted sorting problem is generally referred to as the Integer Sorting Problem. Since the n keys in the above problem are drawn from $\{0, \ldots, n^{\Theta(1)}\}$, their length is at most $\Theta(\log n)$ bits. In this report, we consider unsigned binary numbers that are $\Theta(\log n)$ bits long. Considering that the input to this problem consists of $\Theta(n \log n)$ bits, one could say that the total work, expressed at the bit level, (from now on referred to as Gate-Time Product (GTP); the GTP has been discussed in § 2) needed to solve the Integer Sorting Problem of size n is lower bounded by $\Theta(n \log n)$.

This lower bound on GTP has not been achieved with a time of $\Theta(\log n)$, except in the case of a sorting network [2, 10]. The best known deterministic Integer Sorting algorithm that sorts n $\log n$-bit keys in $\Theta(\log n)$ time is due to Bhatt *et al* [5], and it needs $\Theta(\frac{\log n}{\log \log n})$ time and a GTP of $\Theta(n \log n \log \log n)$ on an ARBITRARY CRCW PRAM[5].

For any CREW model it has been proved [7] that n 1-bit numbers (and hence n $\Theta(\log n)$-bit numbers) need at least $\Theta(\log n)$ time to be sorted. Furthermore, since the input to the Integer Sorting Problem consists of $\Theta(n \log n)$ bits, the GTP of any solution to it is lower bounded by $\Theta(n \log n)$. A logical question therefore is "can Integer Sorting be solved in $\Theta(\log n)$ time and a GTP of $\Theta(n \log n)$ on a CREW model ?" We conjecture that this cannot be done if a lower order than $\Theta(n)$ processors are used. If our conjecture is correct, one can hope to solve the Integer Sorting Problem in $\Theta(\log n)$ time and a GTP of $\Theta(n \log n)$ only if "processors of size $\Theta(1)$ bits" are used. In order to achieve the above bounds on time and GTP, we use in this report a new model of computation called the Reconfigurable PRAM (R-PRAM), which permits the use of small processors. More details of the R-PRAM appear in § 3 and in [13].

In this report, we present a deterministic EREW R-PRAM algorithm that solves the Integer Sorting Problem optimally in $\Theta(\log n)$ time with a GTP of $\Theta(n \log n)$.

The above algorithm is based on a method due to Hagerup [8], which uses a PRIORITY CRCW PRAM with $\frac{n \log \log n}{\log n}$ processors, each of word-size $\log n$ bits, to

[5]The result presented in [5] is more general than what is stated here

sort n $\Theta(\log n)$-bit numbers in $\Theta(\log n)$ time. The bottleneck of Hagerup's algorithm is the Neighbor Localization Problem, to solve which in $\Theta(\log n)$ time, a PRIORITY CRCW PRAM with n processors, each of word-size $\log n$ bits, is required. We show here that the Neighbor Localization Problem can be solved deterministically on an EREW model in $\Theta(\log n)$ time with a GTP of $\Theta(n \log n)$. We use the above result with Hagerup's algorithm to show that n $\Theta(\log n)$-bit unsigned binary numbers can be sorted optimally on an EREW model in $\Theta(\log n)$ time with a GTP of $\Theta(n \log n)$.

Before we proceed, we would like to explain some of the notation used in this report. Let $f(n)$ and $g(n)$ be two non-decreasing functions of a variable n. We say

- $f(n)$ is $\Theta(g(n))$ iff $f(n)$ and $g(n)$ have the same order of complexity.
- $f(n)$ is $O(g(n))$ iff the complexity of $f(n)$ is the same as or lower than that of $g(n)$.
- $f(n)$ is $\Omega(g(n))$ iff $g(n)$ is $O(f(n))$.
- $f(n)$ is $o(g(n))$ iff $f(n)$ is $O(g(n))$ and $f(n)$ is not $\Theta(g(n))$.
- $f(n)$ is $\omega(g(n))$ iff $g(n)$ is $o(f(n))$.

Barring the "ω" notation, the rest of the above complexity notation is commonly used in the literature. For any real number r, $\lceil r \rceil$ denotes the smallest integer i such that $i \geq r$. All logarithms used are to the base 2.

In the next section we briefly describe the idea of a fine-grained problem decomposition which is necessary before we describe our model of computation in § 3. In § 4 we outline the Neighbor Localization Problem, and Hagerup's algorithm. In § 5, we discuss our solution to the Neighbor Localization Problem and explain how it can be used to solve the Integer Sorting Problem. In § 6 we explain the basis for our conjecture that n $\Theta(\log n)$-bit numbers cannot be sorted by any "oblivious" CREW algorithm in $\Theta(\log n)$ time and a GTP of $\Theta(n \log n)$, unless a fine-grained decomposition is used. Finally, in § 7 we summarize our results and make some concluding remarks.

2 Fine-Grained Problem Decomposition

Any computational problem can be viewed as a computable function $f : A \longrightarrow B$ where A and B are the sets representing the input and the output domains. If nothing more is specified about sets A and B, one has to work at a level of abstraction in which any input $a \in A$ and $f(a) \in B$ are treated as atomic entities and one cannot say much about how the computation is performed. Usually, the input and the output are assumed to consist of several smaller entities and A and B may be expressed as $A_1 \times A_2 \times \cdots \times A_N$ and $B_1 \times B_2 \times \cdots \times B_M$, respectively. A slightly lower level of abstraction views the input and output as N and M atomic entities, respectively. At this level of abstraction, one could conceivably parallelize the problem, as there is more than one entity to manipulate. Proceeding in a similar fashion one could view the input as a sequence of n bits and the output as a sequence of m bits, each of which can be processed individually. At this level of abstraction the problem may

be highly parallelizable. Any level of abstraction that views the input and output as entities that are smaller than the elements of $A_1, A_2, \ldots, A_N$ and $B_1, B_2, \ldots, B_M$, will be referred to as a *fine level of abstraction.* A problem decomposition at a fine level of abstraction is called a *fine-grained decomposition.* The granularity of the decomposition is intimately associated with the size of the objects that a processor considers atomic, i.e. the "word-size" of the processor. For many problems, a fine-grained decomposition could result in better solutions. More details appear in [13]. Before we outline the R-PRAM, a few relevant details are discussed below.

Any computable function $f : \{0,1\}^n \longrightarrow \{0,1\}^m$ can be computed trivially in $\Theta(1)$ time using a look-up table with 2^n m-bit entries. The address decoding time has been ignored as is the case for the rest of the discussion in this report. We will therefore assume that the memory used to solve a computational problem of size n is $O(n^{\Theta(1)})$ bits; i.e. memory is polynomially upper-bounded in the size of the input. Similarly, we will also assume that the total number of processors used and their word-size are $O(n^{\Theta(1)})$ bits.

For most non-trivial computational problems of size n, each processor used in its solution has an address space that is $\Omega(n)$ bits (and $O(n^{\Theta(1)})$) bits as discussed earlier). Therefore, the length of an address is $\Theta(\log n)$ bits. This makes it necessary for the processors to be of size $\Omega(\log n)$ bits, if memory addressing is not ignored and is required to take $\Theta(1)$ time. This lower-bounds the size of the processors and hence limits the granularity of the problem decomposition.

The R-PRAM is a variant of the PRAM. Like the PRAM, the model will abstract the solution to a problem from the communication and synchronization details. It is also generally assumed that the PRAM can execute any instruction from its instruction set in $\Theta(1)$ time. To make this assumption reasonable, the instruction set is restricted to include only "simple" operations. One such restricted class of instructions (called the minimal instruction set in [11]) includes data movement, addition, subtraction, and shifting by one bit. One could also include comparison and bitwise and global logical operations in this instruction set. Consider an instruction chosen from this class that uses a b-bit operand. It is clear that data movement, 1-bit shifting, and bitwise logical operations can be done in constant time using a "processor of size b bits." (The notion of a processor of size b bits is defined later). Address generation is assumed to require no time here. Consider now the addition of two b-bit numbers using a processor of size b bits. If we assume that the internal gates of the processors have constant fan-in and fan-out, the above addition cannot be done in time independent of b, unless a table look-up is used. The same holds for comparison and global logical operations. Since each of the above instructions need at most two b-bit operands, and the instruction set contains a constant number of instructions, the total size of the look-up tables for each processor is $\Theta(2^{\Theta(1)b})$ b-bit words. By our earlier assumption $\Theta(2^{\Theta(1)b})$ is $O(n^{\Theta(1)})$. Thus, b is $O(\log n)$. In fact, if b is $O(\log n)$, then any instruction that requires x operands, each of size $\Theta(y)$ bits such that xy is $O(b)$, can be executed in $\Theta(1)$ time by a "processor of size b bits." Therefore any

step in a computation may be viewed as a set of concurrent memory accesses. This motivates the following definition.

Definition: A processor is said to be of size b bits iff the largest number of contiguous memory bits that it can access in unit time is b, where unit time is defined to be the time required by a processor of any size to access a single bit of the memory.

In the above definition it is assumed that no other processor is making an access and that the address for the memory access is known. These assumptions are only for the purpose of a precise definition and do not reflect on the capabilities of the model. More details appear in [13]. The above definition is consistent with the assumption that the instructions from the instruction set of a processor of size b bits (b is $O(\log n)$) can be executed in constant time. We also note that since the size of a processor has been defined in terms of its memory accessing capability and to access b bits of memory in constant time one needs $\Theta(b)$ bits of hardware (not counting the memory, the memory port etc.), we will say that a processor of size b bits has $\Theta(b)$ bits of computing hardware. Conversely, $\Theta(b)$ bits of computing hardware is sufficient to construct $p \leq b$ processors, each of size $\Theta(\frac{b}{p})$ bits. We do not count other hardware necessary in a practical processor, like the memory and its ports, as computing hardware.

If p processors $c_0, c_1, \ldots, c_{p-1}$, with processor c_i of size s_i bits, are used to solve a problem of size n in time $T(n)$, then under the assumptions made earlier we say that the problem can be solved in time $T(n)$ with $\left(\sum_{i=0}^{p-1} s_i\right)$ bits of computing hardware. We measure the efficiency of this solution by the quantity *Gate Time Product* (GTP) which is the product of the bits of computing hardware used and the time taken. The GTP is a measure of computational efficiency, analogous to the commonly used processor time product.

3 The Model of Computation

As mentioned earlier, the model used in this report is the Reconfigurable Parallel Random Access machine (R-PRAM). This model captures the idea of a fine-grained problem decomposition and like the PRAM, abstracts the solution from details of communication and address decoding. In addition, the R-PRAM also abstracts the solution from details of address generation and loop management. More details of these issues appear in [13].

The R-PRAM consists of $\mathcal{H}$ bits of computing hardware that may be configured as $\Theta(p)$ processors, each of size $\Theta(\frac{\mathcal{H}}{p})$ bits, for any p that is $\Omega(1)$, such that $\frac{\mathcal{H}}{p}$ is a non-decreasing function. For each value of p we have a different processor configuration of the $\mathcal{H}$ bits of computing hardware. The reconfiguration is static; i.e. it can be

decided *a priori*, which configuration the R-PRAM will assume at any point in the execution of the algorithm. Like the PRAM, the R-PRAM has $\mathcal{M}$ bits of global memory that could be accessed by all the processors in a given configuration. If a configuration has $\Theta(\frac{\mathcal{H}}{b})$ processors, each of size b bits, then each processor views the global memory as $\Theta(\frac{\mathcal{M}}{b})$ words, each of which consists of b contiguous bits. We note here that a processor of size b bits can only access one b-bit memory word at a time. If a processor of size b-bits accesses ℓ contiguous bits of the memory, then it is assumed to require $\Theta(\lceil\frac{\ell}{b}\rceil)$ time. In this report, we use two configurations for the R-PRAM. The first one has $\Theta(\mathcal{H})$ processors, each of size $\Theta(1)$ bits and the second one has $\Theta(\frac{\mathcal{H}}{\log n})$ processors, each of size $\Theta(\log n)$ bits. In order to ensure that at least $\Theta(1)$ processors, each of size $\Theta(\log n)$ bits is available, we will assume $\mathcal{H}$ to be $\Omega(\log n)$. This is similar to assuming that a PRAM used for the solution has at least $\Theta(1)$ processors.

Like the PRAM, the R-PRAM can be EREW, CREW or CRCW. In this report, we mainly use the EREW R-PRAM.

As mentioned earlier, the R-PRAM could assume a configuration that consists of processors of size $o(\log n)$ bits. Since the address of the memory is $\Theta(\log n)$ bits long, the address generation mechanism of the R-PRAM needs further elaboration. For this purpose, it is convenient to divide the variables into two broad classes; *local variables* and *shared variables*. As the name indicates, the local variables are local to a processor. Since there are a constant number of them, they may be addressed by a processor of size $\Theta(1)$ bits in constant time. On the other hand, a shared variable in general could have the form $Array(x_1)(x_2)\cdots(x_c)$, where c is a constant. These variables are addressed with an additional level of indirection. The indices $x_1, x_2, \cdots x_c$ of the array are treated as the contents of the index registers $R_1, R_2, \cdots R_c$. These index registers themselves could be treated as local variables. Addressing the above array involves first accessing the index registers and setting their values appropriately and the using these values as the address of the array. Thus the above address generation takes as much time as is needed to set the index registers.

The R-PRAM has a weaker variant called the Iteration Sensitive R-PRAM (also called the ISR-PRAM). As mentioned earlier, the R-PRAM assumes that a processor of size b bits can access ℓ contiguous bits of the memory in $\Theta(\lceil\frac{\ell}{b}\rceil)$ time. In other words, the processor executes $\Theta(\lceil\frac{\ell}{b}\rceil)$ iterations, accessing $\Theta(b)$ bits at a time. The overheads in managing the above iterations are ignored (i.e. incrementing the loop variable and deciding when to exit the loop). The ISR-PRAM accounts for all these overheads. More details appear in [13].

4 Preliminaries

We give in § 4.1 a description of the Neighbor Localization Problem that is somewhat different from the description given in [8]. Since the essential idea of the problem is the same we will continue to use the term "Neighbor Localization Problem" in this

report. In § 4.2 we describe Hagerup's Integer Sorting algorithm.

4.1 The Neighbor Localization Problem

Our version of the Neighbor Localization Problem may be described formally as follows. As mentioned earlier, we use the solution to the Neighbor Localization Problem to solve the Integer Sorting Problem.
Let $k_0, k_1, \ldots, k_{n-1}$ be n unsigned binary numbers whose values are drawn from the set $\{0, 1, \ldots, n-1\}$. Let $\rho(k_i)$ denote the value of k_i; $0 \leq i < n$. The solution to the Neighbor Localization Problem is to determine for each number k_i; $0 \leq i < n$, the index j; $i < j < n$ such that $\rho(k_i) = \rho(k_j)$ and for all indices j'; $i < j' < j$, $\rho(k_i) \neq \rho(k_{j'})$. The number k_j is said to be the *neighbor* of k_i. The solution is represented as a set of pointers. The pointer of k_i is set to its neighbor. If k_i has no neighbor, then its pointer is set to a value not in $\{0, 1, \ldots, n-1\}$, which we denote by NIL. It should be mentioned here that a *pointer* is a variable that can assume values from $N(n) \cup \{NIL\}$. It is represented by $\lceil \log n \rceil + 1$ bits. The $\lceil \log n \rceil$ bits represent the value of the pointer (if it is not NIL); the extra tag bit is used to ascertain whether the pointer is NIL or not. It should be noted that a pointer can be tested for a NIL value by examining just one bit.

4.2 Hagerup's Integer Sorting Algorithm

Hagerup's Integer Sorting algorithm for sorting n $\log n$-bit numbers may be described by the following four-step procedure.

Step A: Find the neighbor of each number (if the neighbor exists).

Step B: Concatenate the lists formed in Step A in the order imposed by the function ρ. It is assumed that for each of the lists in Step A the beginning and end may be accessed in constant time, using a processor of size $\log n$ bits. Since list concatenation is an associative operation, it is not difficult to see that Step B can be carried out in $\Theta(\log n)$ time with $\frac{n}{\log n}$ processors, each of size $\log n$ bits by fanning in the lists in a binary tree fashion.

Step C: Rank the elements of the list generated in Step B. This can be done in $\Theta(\log n)$ time on an EREW PRAM with $\frac{n}{\log n}$ processors, each of size $\log n$ bits [3].

Step D: The rank generated in Step C is used to relocate the n $\log n$-bit numbers. The $\frac{n}{\log n}$ processors, each of size $\log n$ bits can achieve this in $\Theta(\log n)$ time.

We show in the next section that the Neighbor Localization Problem can be solved in $\Theta(\log n)$ time on an EREW R-PRAM with $\Theta(n)$ bits of computing hardware.

5 The Proposed Algorithm

This section is divided into two parts. In the first part, we present an algorithm that solves the Neighbor Localization Problem on an EREW R-PRAM with $\Theta(n)$ bits of computing hardware in $\Theta(\log n)$ time. In § 5.2 we explain how this algorithm may be used to sort n $\Theta(\log n)$-bit unsigned binary numbers optimally in $\Theta(\log n)$ time using an EREW R-PRAM with $\Theta(n)$ bits of computing hardware.

5.1 Optimal Solution to the Neighbor Localization Problem

Before we proceed, we remark that the input to the Neighbor Localization Problem is a set of n unsigned binary numbers whose values are from the set $\{0, \ldots, n-1\}$. In other words, the input is of size $(n \log n)$ bits and hence the GTP of any solution to the Neighbor Localization Problem is $\Omega(n \log n)$. Also as mentioned earlier, n numbers cannot be sorted on any CREW model in $o(\log n)$ time [7]. This implies that the Neighbor Localization Problem cannot be solved on any CREW model in $o(\log n)$ time. Thus, a parallel solution that uses $\Theta(n)$ bits of computing hardware and takes $\Theta(\log n)$ time (so that the GTP is $\Theta(n \log n)$) is indeed optimal. We present such a solution in this section.

A naive EREW approach for the Neighbor Localization Problem would fan-in the indices of the processors in a binary tree fashion. This would require n processors, each of size $\Theta(\log n)$ bits to achieve a time of $\Theta(\log n)$, as the processor indices (and hence the result pointers) are of length $\Theta(\log n)$, and the GTP is $\Theta(n \log^2 n)$. It should be pointed out that if $o(n)$ processors, each of size $\log n$ bits are used, the (worst case) time becomes $\omega(\log n)$ and the GTP is still $\Theta(n \log^2 n)$. One way of reducing the GTP is by decreasing the number of data bits in each step of the fan-in. In our method we fan-in the information about the presence (or absence) of the neighbor of a given number. This implies the fanning-in of $\Theta(1)$-bit information in a binary tree, which we will call the fan-in tree. Thus, n processors, each of size $\Theta(1)$ bits can perform the fan-in in $\Theta(\log n)$ time. However, we cannot determine the neighbor of a given number by this method; only a subtree of the fan-in tree in which the neighbor lies can be identified. Subsequently, this subtree will be systematically searched for the neighbors. We note that the nodes of the fan-in tree correspond to a set of processors. At the leaves these sets are singleton sets and could also be taken to represent the n numbers (keys).

Definition: Let k_i be a number whose neighbor is k_j; $0 \leq i < j < n$. The *neighbor tree* of i, denoted by $\mathcal{T}_i$, is the smallest subtree of the fan-in tree that has both k_i and its neighbor k_j as leaves. For numbers that have no neighbors, the neighbor tree is undefined.

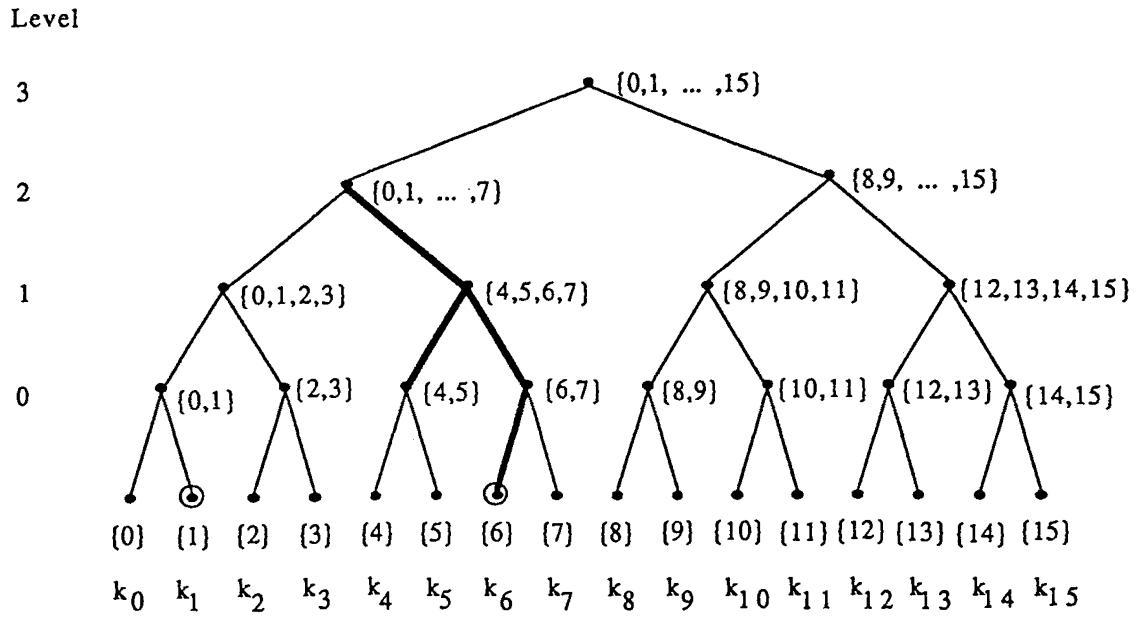

Figure 1: The Fan-in tree for the Example

As mentioned earlier, our fan-in step identifies the numbers that have a neighbor and for each such number k_i, $\mathcal{T}_i$ $(0 \leq i < n)$ is determined. We use this information to search $\mathcal{T}_i$ in $\Theta(\log n)$ time using n processors, each of size $\Theta(1)$ bits. The following example illustrates our approach.

Example : Consider 16 unsigned binary numbers in which the neighbor of k_1 is k_6. The Fan-in tree is illustrated in Fig. 1 with the nodes represented as sets of processor indices. Also shown are the levels of the non-leaf nodes of the fan-in tree. For brevity, we will refer to a non-leaf node by the highest processor index in its representative set and its level. For instance, the node labeled $\{0, 1, \ldots, 7\}$ in Fig. 1 is represented as $< 7, 2 >$. A leaf node will be referred to by the processor (element) index associated with it. As an example, the leaf node labeled $\{1\}$ in Fig. 1 is referred to simply as node 1.

It is clear that $\mathcal{T}_1$ is rooted at the node $< 7, 2 >$ (the node at level 2 with 7 as the highest index in its representative set). We know from this that the subtree rooted at node $< 7, 1 >$ has the neighbor of k_1. The processor 1 therefore searches the node $< 5, 0 >$ for the presence of its neighbor. Since its neighbor is not a leaf of the subtree rooted at $< 5, 0 >$, processor 1 does not detect it and decides to search $< 7, 0 >$, the right child of $< 7, 1 >$. At the next and final step, processor 1 searches first the left child (node 6) of $< 7, 0 >$ and finds its neighbor. Otherwise, node 7 (right child of $< 7, 0 >$) would have been its neighbor.

In our algorithm we use n processors, each of size $\Theta(1)$ bits, indexed 0 to $n-1$ and we assign processor i to the number k_i. We assume that n is an integer power of 2. This is purely for convenience and will in no way affect the complexity of our algorithm. The variables used in the algorithm will be termed *parallel variables*. A parallel variable has n components, one for each processor. For example, a parallel variable named "*List*" will have a component $List(i)$ corresponding to each processor i; $0 \leq i < n$. The component $List(i)$ will be referred to as the i^{th} component of *List*. A parallel variable V whose i^{th} component is accessed only by the processor i may be treated as a local variable. All other parallel variables are accessed indirectly as discussed in § 3. Each component of a parallel variable could be a bit or even an array. For brevity, when we talk of information stored in the i^{th} component of some parallel variable, we will say that the information is in processor i. Also, the processor i and the number k_i assigned to it will be used interchangeably where there is no danger of ambiguity. We now describe our algorithm as a 3-step procedure.

Step 1: For each number k_i we set a flag "$Flag(i)$" which is 1 if and only if k_i has a neighbor. For the number k_i that has $Flag(i) = 1$, we also determine $\mathcal{T}_i$, it's neighbor tree. We note here that $\mathcal{T}_i$ can be uniquely specified by the level of its root. For instance, $\mathcal{T}_1$ in our example, can be specified as simply 2, knowing that only the node $< 7, 2 >$ or $\{0, 1, 2, 3, 4, 5, 6, 7\}$ can have k_1 as its leaf. We represent the level information in the component $Level(i)$ of a parallel variable *Level*. $Level(i)$ is a $\log n$-bit vector, each bit of which denotes a level of the fan-in tree. The least significant bit is numbered 0 and the most significant bit is numbered $\log n - 1$. If $\mathcal{T}_i$ is rooted at a node at level h of the fan-in tree; $0 \leq h < \log n$, then bits 0 to h of $Level(i)$ are set to 1; the remaining bits are set to 0. In our example, $Level(i)$ is a 4-bit vector and $Level(1) = 0111$. For the numbers that have *Flag* set to 0, *Level* does not matter.

Step 2: We use $Level(i)$ to search $\mathcal{T}_i$, as was illustrated by our example. The output of this step is the parallel variable *Link*. If $Flag(i) = 1$, then $Link(i)$ points to the neighbor of k_i. If $Flag(i) = 0$ then $Link(i)$ has a "*don't-care*" value. The search in this step is performed only for those numbers k_i that have $Flag(i) = 1$.

Step 3: For each number k_i we set the pointer $Nbr(i)$ to point to its neighbor. If k_i has no neighbor, then $Nbr(i)$ is set to NIL.

Inputs		Outputs			
i	$\rho(i)$	$Level(i)$	$Flag(i)$	$Link(i)$	$Nbr(i)$
0	5	0 1 1	1	2	2
1	2	1 1 1	1	7	7
2	5	0 0 1	1	3	3
3	5	1 1 1	0	–	NIL
4	4	0 1 1	1	6	6
5	7	1 1 1	0	–	NIL
6	4	1 1 1	0	–	NIL
7	2	1 1 1	0	–	NIL

Table 1: An illustration of the Neighbor Localization Problem

All of the above three steps will need $\Theta(\log n)$ time on an EREW R-PRAM with n bits of computing hardware. Table 1 shows the values of the relevant parallel variables for a small example of eight numbers. For instance, consider element 0. The value of k_0 is 5 and the smallest index $i > 0$ so that $\rho(i) = 5$, is 2. Thus, k_2 is the neighbor of k_0 and $Nbr(0) = 2$. At the first iteration of Step 1 of our algorithm, processor 0 searches the index 1 for a neighbor. Since $\rho(0) \neq \rho(1)$, a neighbor is not detected. In the next iteration, processor 0 searches the indices 2 and 3 and detects a neighbor. For the remainder of Step 1 processor need not look for a neighbor. This is reflected by $Level(i)$, which is 1 (starting from the lsb) till a processor i detects the neighbor of k_i. For processor 0, $Level(0) = 011$ as the neighbor is detected in the second iteration. $Flag(0) = 1$ as k_0 has a neighbor. In contrast k_5 has no neighbor and $Flag(5) = 1$ and $Level(5) = 111$. The output of Step 2 is $Link(i)$ which points to the neighbor of k_i (if the neighbor exists); otherwise $Link(i)$ has a *don't care* value, shown as "–" in Table 1. The only difference between $Link(i)$ and $Nbr(i)$ is that the *don't care* values in $Link(i)$ are replaced by NIL in $Nbr(i)$.

Steps 1 and 2 use a parallel variable called the Fan_in_Array. The component $Fan\_in\_Array(i)$ is itself an array of n bits, one for each possible value of a number. We use the Fan_in_Array to fan-in the neighbor information in Step 1 and to search the subtrees in Step 2. The basic operation of Step 1 is a merge of the neighbor information. For some value of i_1 and $0 \leq h < \log n$, let $S_\ell = \{i_1 + j : 0 \leq j < 2^h\}$ and $S_r = \{i_1 + 2^h + j : 0 \leq j < 2^h\}$, be subsets of $N(n) = \{0, 1, \ldots, n-1\}$. The elements of these sets are to be taken as indices of the processors or the numbers (keys).

Definitions: Consider a number k_i with $i \in S \subseteq N(n)$. If k_i has a neighbor whose index is in S, then k_i is said to be *known* with respect to S or simply a *known element* of S; Otherwise k_i is said to be an *unknown element* of S. If the neighbors of all the known elements of S have been detected, then S is said to be *solved.* The unknown elements of S are called the *last elements* of S. An element of S that is not a neighbor of any other element of S is called a *first element* of S.

The non-leaf nodes of the fan-in tree are called merge steps. For a merge step, the input is the sets S_ℓ and S_r, which are assumed to be solved. The output is the solved set $S_\ell \cup S_r \subseteq N(n)$. When $S_\ell \cup S_r = N(n)$, Step 1 has been completed. The sets S_ℓ and S_r are called the *Left* and *Right Sets* of the merge step, respectively. Each element of S_ℓ (or S_r) has a common destination index $D(S_\ell)$ (or $D(S_r)$) associated with it. In fact, $D(S_\ell)$ (or $D(S_r)$) is the largest index in S_ℓ (or S_r). The non-leaf nodes of the Fan-in tree of Fig. 1 represent the merge steps and the sets used to represent them are the sets $S_\ell \cup S_r$ resulting from the merge step. In fact, if $S \subseteq N(n)$ and if <*max_index*, *level*> represents the node (merge step), then $max\_index = D(S)$. During a merge step we use $Fan\_in\_Array(D(S_r))$ to check if any of the unknown elements of S_ℓ have neighbors in S_r. This is done as follows.

Each last element (unknown element) i of S_ℓ initializes $Fan\_in\_Array(D(S_r))(\rho(i))$. Next, each first element j of S_r marks $Fan\_in\_Array(D(S_r))(\rho(j))$ with a 1. Finally, each *last element* i of S_ℓ checks $Fan\_in\_Array(D(S_r))(\rho(i))$ for a mark. If a mark is found, then the existence of a neighbor of the last element in S_r is established. At the end of the merge $D(S_\ell \cup S_r) = D(S_r)$. We use the parallel variable Dst to represent the destination index of a processor. At the beginning of Step 1 the left sets are $\{i\}$; $0 \leq i < n$ and i is even, and the right sets are $\{i\}$; $0 \leq i < n$ and i is odd; $D(\{i\}) = i$. The parallel variables Fan_in_Array and Dst are used for similar purposes in Step 2, as is illustrated later. We provide below a simple algorithmic description of the above steps. A detailed pseudo-code is given in the Appendix A.

In the following description, processor i will be called c_i; $0 \leq i < n$; and will be assumed to be associated with k_i the i^{th} input number. Where there is no ambiguity, we will use c_i and k_i interchangeably. The following algorithm is executed by each processor c_i.

Step 1

Initialize k_i to be both a first and a last element of $\{i\}$. Initialize $Level(i)$ to $00\ldots0$;
i.e. set each bit of $Level(i)$ to 0;

for $h \longleftarrow 0$ to $\log n - 1$ do

Compute $Dst(i)$ the address of the buffer area (component of Fan_in_Array)
through which c_i will exchange information;

/* Initialize Step: This ensures that garbage values are not read in the subsequent Check Step */

If k_i is a last element and part of a Left Set then

c_i initializes $Fan\_in\_Array(Dst(i))(\rho(i))$ to 0;

/* Set Step: Here the first elements of each Right Set declare their presence (for the last elements of the corresponding Left Sets) */

If k_i is a first element and part of a Right Set then

c_i sets $Fan\_in\_Array(Dst(i))(\rho(i))$ to 1;

/* Check Step */

If k_i is a last element and part of a Left Set then

c_i checks $Fan\_in\_Array(Dst(i))(\rho(i))$;

If the value checked is a 1 then the existence of a neighbor of k_i in a subtree rooted at level h has been established;

If k_i has a neighbor in a subtree rooted at level h then

$Level(i)$ and $Last(i)$ are appropriately adjusted. $Last(i)$ is a flag which is 1 iff k_i is a last element;

$First(i)$ is adjusted if necessary. $First(i)$ is a flag which is 1 iff k_i is a first element;

end

If k_i did not find a neighbor then set $Flag(i)$ to 0; otherwise set it to 1;

Step 2

$CST(i) = \mathcal{T}_i$; /* $CST(i)$ is the current search tree of c_i; this is initialized to $\mathcal{T}_i$, the fan-in tree of k_i, that was obtained in Step 1 */

for $h \longleftarrow \log n - 2$ down to 0 do

if $Flag(i) = 1$ and $Level(i) + 1 = 1$ then

Search the left subtree of $CST(i)$ for the neighbor of k_i;

If a neighbor is detected then

$CST(i) =$ left subtree of $CST(i)$;

else

$CST(i) =$ right subtree of $CST(i)$;

end

At this point $CST(i)$ is rooted at a leaf, which is the neighbor of k_i.

At the end of Step 2, $Flag(i) = 1$ iff k_i has a neighbor and for those elements that have $Flag(i) = 1$, a parallel variable called $Link$ is set to point to the neighbor. Step 3 sets $Nbr(i)$ to point to the neighbor of k_i, if k_i has a neighbor. Otherwise, $Nbr(i)$ is set to NIL.

Step 3

if $Flag(i) = 1$ then

$Nbr(i) \longleftarrow Link(i)$

else $Nbr(i) \longleftarrow$ NIL

Each of steps 1, 2 and 3 need $\Theta(\log n)$ time. The memory used is $\Theta(n^2)$ bits (for Fan_in_Array).

We now illustrate our solution to the Neighbor Localization Problem with a more detailed explanation for the instance in Fig. 1. The fan-in tree for this example is shown in Fig. 2. The nodes of the fan-in tree are numbered 0 to 14 with the leaves corresponding to the indices of the input numbers. The values of the numbers are also shown in Fig 2. In the following description processor i is denoted by c_i and is assumed to be associated with the input k_i. The neighbor tree of k_i is denoted by $\mathcal{T}_i$ and $\mathcal{T}(j)$ represents the subtree of the fan-in tree rooted at node j. For instance, $\mathcal{T}(14)$ denotes the entire fan-in tree. We will also assume that each node j that is searched by a processor c_i has all the information about the leaves of $\mathcal{T}_j$.

Step 1: This step has $\log n$ iterations (3 for the example).

Iteration 0

- Processors c_0, c_2, c_4 and c_6 search $\mathcal{T}(1), \mathcal{T}(3), \mathcal{T}(5)$ and $\mathcal{T}(7)$ respectively.
- Only c_2 finds a match; $\mathcal{T}_2 = \mathcal{T}(9)$.

Iteration 1

- c_0 and c_1 search $\mathcal{T}(9)$; c_4 and c_5 search $\mathcal{T}(11)$.
- c_0 and c_4 find matches; $\mathcal{T}_0 = \mathcal{T}(12)$ and $\mathcal{T}_4 = \mathcal{T}(13)$.

Iteration 2

- c_1 and c_3 search $\mathcal{T}(13)$.

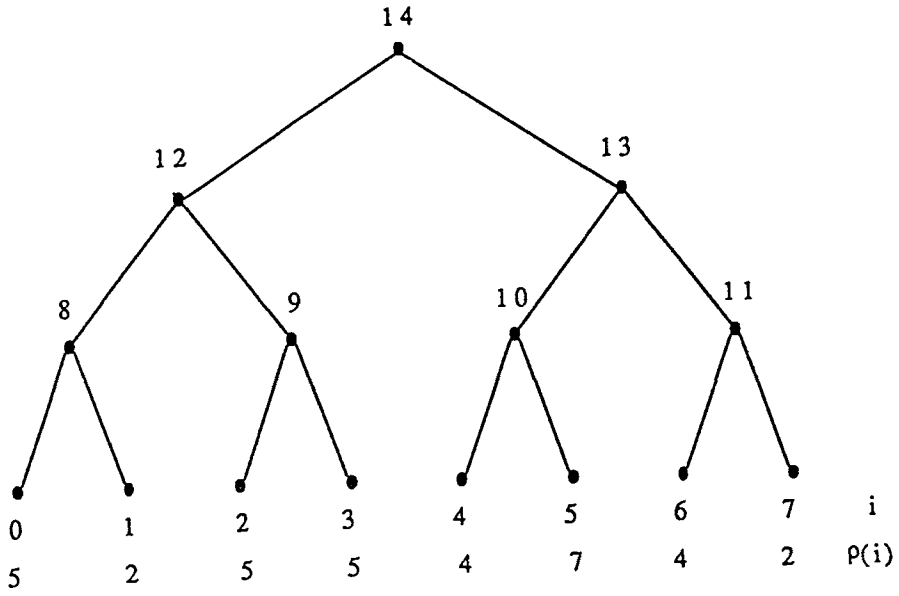

Figure 2: Fan-in tree for the example in Table 1

- c_0 and c_2 do not participate in the search as the neighbors of k_0 and k_2 have been detected.
- c_1 detects a neighbor while c_3 doesn't; $\mathcal{T}_1 = \mathcal{T}(14)$.

At the end of Step 1, $Flag(0) = Flag(1) = Flag(2) = Flag(4) = 1$ as the neighbors of k_0, k_1, k_2 and k_4 have been detected. The remaining elements i (that have $Flag(i) = 0$) do not participate in the search in Step 2.

Step 2: This step has $(\log n) - 1$ iterations (2 for the example).

Iteration 1

- The neighbor tree of k_1 is $\mathcal{T}(14)$. From this it is obvious that the neighbor of k_1 is a leaf of $\mathcal{T}(13)$. c_1 therefore searches $\mathcal{T}(10)$ the left subtree of $\mathcal{T}(13)$. After having failed to detect the neighbor in $\mathcal{T}(10)$, c_1 decides to search $\mathcal{T}(11)$, the right subtree of $\mathcal{T}(13)$.

Iteration 0

- c_1 now searches $\mathcal{T}(6)$ the left subtree of $\mathcal{T}(11)$ and having failed to detect the neighbor deduces that $\mathcal{T}(7)$ = node 7 is the neighbor.
- c_0 and c_4 join in the search during this iteration. c_0 searches $\mathcal{T}(2)$ and finds the neighbor. c_4 searches $\mathcal{T}(6)$ and finds the neighbor.
- It should be noted that c_2 does not participate in Step 2 as it can directly deduce that $\mathcal{T}(3)$ = node 3 is the neighbor.

Step 3: For each index i, $Nbr(i)$ is set to the value of $Link(i)$ (obtained in Step 2), if $Flag(i) = 1$; otherwise $Nbr(i)$ is set to NIL.

In Appendix A we provide pseudo code for Step 1 and Step 2 of the Neighbor Localization Problem algorithm. An explicit illustration of Steps 1–3 appears in Appendix B. It is clear from Procedure Step_1 (Appendix A) that $Level(i)$ and $Flag(i)$ are appropriately set. It is not difficult to show that the reads and writes on $Fan\_in\_Array(Dst(i))(\rho(i))$ are exclusive. This is because for any given values of $Dst(i)$ and $\rho(i)$ there is no more than one processor (the one corresponding to the last element of value $\rho(i)$ in the Left Set) that initializes the above location, checks it for a mark and resets the mark. Similarly, the only processor that marks this location and checks for a reset mark is the one corresponding to the first element of value $\rho(i)$ in the Right Set. Again, for Procedure Step_2 (Appendix A) it is evident that the search is performed as illustrated in the earlier examples. The reason for using $Half\_Level(i)$ is that the search really begins at the level of the subtrees of $\mathcal{T}_i$. For searching a subtree rooted at a node x, the unknown elements of the set (corresponding to the node x in Step 1) are used to reconstruct Fan_in_Array. Thus, all reads and writes can be proved to be exclusive in Step 2, by virtue of the fact that $Level(i)$ and hence $Half\_Level(i)$ are based on the access pattern seen in Step 1. An important point to note is that the parallel variables Dst and $Link$ are set 1 bit at a time.

We note here that the only shared variable used in our algorithm is Fan_in_Array. When processor i accesses a component of the above parallel variable, the address has the form $Fan\_in\_Array(x)(\rho(i))$, where x is either $Dst(i)$ or $Link(i)$, both of which are local variables. As mentioned earlier, x and $\rho(i)$ are to be treated as contents of index registers and the time required to access $Fan\_in\_Array(x)(\rho(i))$ is the time required to generate the values of x and $\rho(i)$. The value of $\rho(i)$ can be generated once at the start of the algorithm as a part of the initialization procedure. This value does not change subsequently. Since the above value can be generated in $\Theta(\log n)$ time by a processor of size $\Theta(1)$ bits, it does not affect the time complexity of the algorithm. The variables $Dst(i)$ and $Link(i)$ are either changed outside the loops or are changed only one bit at a time (inside the loops). Hence they too do not affect

the time complexity of the algorithm. In other words, the effective access time for *Fan_in_Array* is $\Theta(1)$.
We summarize the results of this section in the following lemma.

Lemma 1 *The Neighbor Localization Problem for n elements can be solved on an EREW R-PRAM with* $\Theta(n)$ *bits of computing hardware in* $\Theta(\log n)$ *time and* $\Theta(n^2)$ *bits of space.*

5.2 An Optimal Solution to Integer Sorting

As mentioned earlier, our Integer Sorting Algorithm is based on Hagerup's method. We replace Step A of Hagerup's algorithm by our Neighbor Localization Problem algorithm (see § 4.2). This makes Step A optimal and requiring a EREW model. For Step B of Hagerup's algorithm requires that the beginning and end of each list generated by Step A be available for access by a processor of size $\log n$ bits in constant time. This can be done as shown in Appendix A.

Step C requires $\Theta(\log n)$, $\log n$-bit addition steps. As mentioned in § 2, each addition step requires a non-constant time, unless a look-up table is used. The size of the look-up table for each of the $\Theta(\frac{n}{\log n})$ processors used in this step is $\Theta(n^2 \log n)$. Thus unless a CREW model is used, each processor needs a look-up table and the total size of the look-up tables is $\Theta(n^3)$. If a CREW R-PRAM is used the memory requirement is $\Theta(n^2 \log n)$ bits.

So far, we have considered only n $\log n$-bit unsigned binary numbers. If the unsigned binary numbers (keys) are $(c \log n)$-bits long, where c is any constant, the sorting can be done with an additional time factor of $\lceil c \rceil$, as follows.[6] Divide the $(c \log n)$ bits into $\lceil c \rceil$ sections of contiguous bits, each at most $\log n$ bits long. We proceed in $\lceil c \rceil$ steps over the sections (starting from the least significant section), Integer Sorting the current section in $\Theta(\log n)$ time. This sorting is used to reorder the keys for the next iteration. This method is very similar to the lexicographic sorting in [1]. If an ISR-PRAM (a weaker variant of the R-PRAM) is used there is a slight degradation of the GTP caused by the overheads of managing a loop of $\Theta(\log n)$ iterations. The speed of the algorithm is not affected. Our results are summarized in the following theorems.

Theorem 1 *Given n* $\Theta(\log n)$*-bit unsigned binary numbers, they can be sorted stably in* $\Theta(\log n)$ *time on an EREW R-PRAM with n bits of computing hardware, and with* $\Theta(n^3)$ *bits of space.*
The space requirement can be reduced to $\Theta(n^2 \log n)$ *bits if a CREW R-PRAM is used.*

We note here that the time and GTP of the above algorithm are optimal.

[6] It should be noted that the sorting method discussed so far is stable [9].

It has been shown in [13] that a loop whose loop variable goes from 0 to $Y-1$ has an overhead of $\Theta(\log\log Y)$ in the bits of computing hardware needed, when executed on an ISR-PRAM. There is no overhead in time for the above loop. Since the loops for our Neighbor Localization Problem algorithm have $\Theta(\log n)$ iterations, the corresponding overheads in the bits of computing hardware, when an ISR-PRAM is used, is $\Theta(\log\log\log n)$. Thus we have,

Theorem 2 *Given n $\Theta(\log n)$-bit unsigned binary numbers, they can be sorted stably in $\Theta(\log n)$ time on an EREW ISR-PRAM with $n\log\log\log n$ bits of computing hardware, and with $\Theta(n^3)$ bits of space.*
The space requirement can be reduced to $\Theta(n^2\log n)$ bits if a CREW ISR-PRAM is used.

Though the GTP of the ISR-PRAM solution is suboptimal, the degradation in the GTP is by a very small order. In any case, this GTP is an improvement over the conventional EREW PRAM algorithm that has a GTP of $\Theta(n\log^2 n)$.

6 Integer Sorting and Fine-Grained Decomposition

In this section we address the issue of how important fine-grained problem decomposition is for Integer Sorting. Before we can attempt to discuss this let us examine the Matching Value Problem that is described below.

Consider a function $f : \{0,1,\ldots,n-1\}^n \longrightarrow \{0,1\}^n$ for which $f(\alpha_1,\alpha_2,\ldots,\alpha_{n-1}) =< \beta_1,\beta_2,\ldots,\beta_{n-1} >$, where $\beta_i = 1$ iff $\exists j \in \{0,1,\ldots,n-1\} - \{i\} \ni \alpha_i = \alpha_j$; $0 \leq i < n$. Computing the above function is the solution to the Matching Value Problem.

For this section we consider a special case of the Matching Value Problem in which at most 2 of the n input elements have the same value. We call this the Restricted Matching Value Problem. Before we proceed any further, a few definitions and observations are useful.

An algorithm is said to be *oblivious* if it is possible to choose an input for which the performance of the algorithm is the worst possible.

Consider now an oblivious CREW algorithm for the Restricted Matching Value Problem. It is easy to see that even if only β_0 need be computed, the above algorithm would need $\Theta(\log n)$ time. Thus, a lower bound on time needed to solve the Matching Value Problem is $\Theta(\log n)$. A lower bound on the GTP needed to solve the Restricted Matching Value Problem (and hence the Matching Value Problem) is $\Theta(n\log n)$, the number of bits in the input. If n processors, each of size $\log n$ bits is used, it is easy to design a CREW algorithm that achieves the above lower bound on time. However the GTP is $\omega(n\log n)$. We now pose the following question. Is is possible to design an

oblivious CREW algorithm that uses $o(n)$ processors to solve the Restricted Matching Value Problem in $\Theta(\log n)$ time and a GTP of $\Theta(n \log n)$?

Before we address this question, the following observations about the Restricted Matching Value Problem are important.

- If any processor finds a matching pair of values the the Restricted Matching Value Problem is solved.
- Till a pair of matching values is found (or the algorithm terminates with $\beta_i = 0$, for all $i \in \{0, 1, \ldots, n-1\}$), none of the inputs elements may be ignored. If any input element is ignored, an input to the problem may be chosen so that the ignored input has a matching value. This would make the algorithm slow or worse still, incorrect.

Consider now an oblivious CREW algorithm for the Restricted Matching Value Problem that uses p processors, each of size $\Theta(n/p)$ bits, where p is $o(n)$. The size of the processors is therefore $\omega(1)$. Since there are n input elements to be considered by these processors, each processor has n/p (which is $\omega(1)$) input elements associated with it. One way of representing the information in the input elements is by their values. Each value is $\Theta(\log n)$ bits long. Since no input element may be ignored, each step in the algorithm actually needs $\Theta((n/p)\lceil \frac{\log n}{(n/p)} \rceil)$ time (in the worst case), which is $\omega(1)$. Since there are $\Omega(\log n)$ steps in any CREW algorithm for the Restricted Matching Value Problem, the time taken is $\omega(\log n)$, if $o(n)$ processors, each of size $\Theta(n/p)$ bits are used.

We conjecture that no representation of the information in the n/p arbitrary input elements assigned to each processor would lead to an oblivious CREW algorithm for the Restricted Matching Value Problem that uses $o(n)$ processors and achieves a time of $\Theta(\log n)$ and a GTP of $\Theta(n \log n)$.

Lemma 2 *If the Restricted Matching Value Problem cannot be solved by an oblivious CREW algorithm that uses $o(n)$ processors and achieves a time of $\Theta(\log n)$ and a GTP of $\Theta(n \log n)$, then n $\log n$-bit numbers cannot be sorted by an oblivious CREW algorithm that uses $o(n)$ processors and achieves a time of $\Theta(\log n)$ and a GTP of $\Theta(n \log n)$.*

Proof: Suppose there is an oblivious CREW algorithm $\mathcal{A}$ that sorts n $\log n$-bit numbers in $\Theta(\log n)$ time and with a GTP of $\Theta(n \log n)$, using $o(n)$ processors. We now show how the above oblivious CREW algorithm $\mathcal{A}$ can be used to solve the Matching Value Problem (and hence the Restricted Matching Value Problem) using $o(n)$ processors, in $\Theta(\log n)$ time and with a GTP of $\Theta(n \log n)$.

First the input numbers $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$ are sorted using $\mathcal{A}$ to form the sorted list $\gamma_0, \gamma_1, \ldots, \gamma_{n-1}$. Let $\kappa(i)$ be the position of the input α_i in the sorted list (i.e. $\alpha_i = \gamma_{\kappa(i)}$). The index $\kappa(i)$ can be obtained for each i; $(0 \leq i < n)$ in $\Theta(\log n)$ time. Also let ρ_i denote the value of α_i. The output bit β_i can now be set as follows:

$$\beta_i = 1 \text{ iff } \rho_{\kappa(i)} = \rho_{\kappa(i)-1} \text{ or } \rho_{\kappa(i)} = \rho_{\kappa(i)+1}$$

We define $\gamma_{-1} = \gamma_n = \text{NIL}$, a value not in $\{0, 1, \ldots, n-1\}$. Therefore the algorithm $\mathcal{A}$ can be used to solve the Matching Value Problem in $\Theta(\log n)$ time and a GTP of $\Theta(n \log n)$ with $o(n)$ processors.

□

Thus, if our conjecture about the Restricted Matching Value Problem is true, Integer Sorting of n $\Theta(\log n)$-bit numbers cannot be done by an oblivious CREW algorithm in $\Theta(\log n)$ time and with a GTP of $\Theta(n \log n)$, without a fine-grained decomposition. Our Integer Sorting algorithm proves that Integer Sorting can be solved in $\Theta(\log n)$ time and with a GTP of $\Theta(n \log n)$ with fine-grained decomposition, on an EREW model.

7 Concluding Remarks

We have shown in this report that by using a fine-grained decomposition, the Neighbor Localization Problem can be solved very efficiently. As a consequence of this result we find that n $\Theta(\log n)$-bit unsigned binary numbers can be sorted optimally in $\Theta(\log n)$ time and a GTP of $\Theta(n \log n)$ on an EREW R-PRAM. If a weaker variant of the R-PRAM called the ISR-PRAM [13] is used, the degradation in the efficiency (GTP) is very small (a factor of $\Theta(\log \log \log n)$). The speed of the algorithm is unchanged. It should be noted that the ISR-PRAM accounts for all overheads. Though our algorithm, when run on an ISR-PRAM, results in a sub-optimal GTP, it is a big improvement over the GTP of conventional EREW PRAM algorithms.

Our algorithm illustrates the power of a fine-grained problem decomposition in solving the Integer Sorting Problem very efficiently. We have conjectured that such an efficient (and fast) solution is not possible unless a fine-grained problem decomposition is used. We have outlined our reasons for making this conjecture.

We would like to mention that the memory requirement of $\Theta(n^2)$ does not really affect the complexities of our algorithm, as all initializations have been accounted for. Also this memory requirement is reasonable as is evident from the following discussion. Suppose there is a CREW PRAM algorithm for integer sorting that uses $\Theta(\frac{n}{\log n})$ processors, each of size $\Theta(\log n)$ bits to achieve a time of $\Theta(\log n)$. This algorithm cannot be a comparison-based algorithm as its GTP is $o(n \log^2 n)$ (it has been shown in [12] that the GTP of a comparison-based sorting algorithm used to sort n m-bit numbers is $\Omega(mn \log n)$). Hence, it would in all probability require some operation like $\log n$-bit addition that needs a look-up table. From the discussion in § 5.2, it is clear that the memory required for this algorithm is $\Omega(n^2)$. In other words, for our model of computation, it is the ranking step and not the Neighbor Localization Problem that decides the space complexity of our algorithm. Thus our Integer Sorting

algorithm is not only optimal in time and GTP, but also has a reasonable memory requirement.

Acknowledgment

The authors would like to thank Elaine Weinman for her invaluable help in the preparation of this manuscript. Thanks are also due to Prof. Sanjay Ranka and Prof. Torben Hagerup for their useful suggestions.

References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley Publishing Company, 1974, pp. 76-80.

[2] M. Ajtai, J. Komlós and E. Szemerédi, "An $O(n \log n)$ Sorting Network", *Proc.* $15^{\underline{th}}$ *ACM Symp. on Theory of Computation*, 1983, pp. 1-9.
"Sorting in $c \log n$ parallel steps", *Combinatorica* 3(1), 1983, pp. 1-19.

[3] R. J. Anderson and G. L. Miller, "Deterministic Parallel List Ranking", *Proc.* $3^{\underline{rd}}$ *Aegean Workshop on Computing*, Springer Verlag Lecture Notes in Computer Science, Vol. 319, 1988, pp. 81–90.

[4] Y. Azar and U. Vishkin, "Tight Bounds on the Complexity of Parallel Sorting", *SIAM J. Computing*, Vol. 16, No. 3, June 1987, pp. 458-464.

[5] P. C. P. Bhatt, K. Diks, T. Hagerup, V. C. Prasad, T. Radzik and S. Saxena, "Improved Deterministic Parallel Integer Sorting", Technical Report 15/1989, Fachbereich Informatik, Universität des Saarlandes, D–6600 Saarbrücken, West Germany.

[6] R. Cole, "Parallel Merge Sort", *SIAM J. Computing*, Vol. 17, No. 4, August 1988, pp. 770-785.

[7] S. Cook, C. Dwork and R. Reischuk, "Upper and Lower Time Bounds for Parallel Random Access Machines without Simultaneous Writes", *SIAM J. Comput.*, Vol. 15, No. 1, Feb. 1986, pp. 87–97.

[8] T. Hagerup, "Towards Optimal Parallel Bucket Sorting", *Information and Computation*, 1987, pp. 39-51.

[9] D. E. Knuth, "The Art of Computer Programming Vol. 3, Sorting and Searching", Addison-Wesley Publishing Company, 1973.

[10] T. Leighton, "Tight Bounds on the Complexity of Parallel Sorting", *IEEE Trans. on Computers*, Vol. C-34, No. 4, April 1985, pp. 344-354.

[11] I. Parberry, "Parallel Complexity Theory", John Wiley and Sons, Inc., New York, 1987.

[12] R. Vaidyanathan, C. R. P. Hartmann and P. K. Varshney, "Optimal Parallel Lexicographic Sorting using a Fine-Grained Decomposition", in preparation.

[13] R. Vaidyanathan, C. R. P. Hartmann and P. K. Varshney, "The R-PRAM: A Fine-Grained PRAM Model", in preparation.

A Pseudo Code for the Neighbor Localization Problem

In this appendix we give pseudo codes for Steps 1 and 2 of the Neighbor Localization Problem. We provide comments (enclosed in "/*" and "*/") wherever possible. We also give an explicit illustration of the algorithm in the Appendix B. We suggest that this example be read together with the pseudo code.

Procedure Step_1 /* Find the Level vectors and set $Flag$ */
/* Executed in parallel by all processors indexed i */
begin
/* Initialization */
$\boldsymbol{Dst(i)} \longleftarrow \boldsymbol{i}$ /* The initial Left and Right Sets are $\{i\}$ */
$\boldsymbol{First(i)} \longleftarrow \mathbf{1}$ /* $First(i) = 1$ iff k_i is a first element of the current set */
$\boldsymbol{Last(i)} \longleftarrow \mathbf{1}$ /* $Last(i) = 1$ iff k_i is a last element of the current set */
/* During a merge step only the last (or first) elements of the Left (or Right) Sets participate. This ensures exclusive reads and writes. */
/* Initialize the Level vector bits to 0. */
for $\boldsymbol{h} \longleftarrow \mathbf{0}$ **to** $\log(\boldsymbol{n}) - \mathbf{1}$ **do**
$\boldsymbol{Level(i)}|_h \longleftarrow \mathbf{0}$ /* bit h of $Level(i)$ is set to 0 */
end
/* End Initialization */
/* Fan-in the neighbor information. Each iteration is a merge step */
for $\boldsymbol{h} \longleftarrow \mathbf{0}$ **to** $\log(\boldsymbol{n}) - \mathbf{1}$ **do**
/* Set $Left\_Set(i)$, a Boolean variable which is 1 iff i is a member of a Left Set of the current merge step */
if $\boldsymbol{Dst(i)}|_h = \mathbf{0}$ **then** /* $Dst(i)|_h$ denotes bit h of $Dst(i)$ */
$\boldsymbol{Left\_Set(i)} \longleftarrow \mathbf{1}$
else $\boldsymbol{Left\_Set(i)} \longleftarrow \mathbf{0}$
end
/* Initialize Step:
For elements i of a Left Set, set $Dst(i)$ to the destination of the correspond -ing Right Set and initialize the appropriate locations of $Fan\_in\_Array(Dst(i))$. This ensures that in the Check Step, garbage values are not read. */
if $\boldsymbol{Left\_Set(i)} = \mathbf{1}$ **and** $\boldsymbol{Last(i)} = \mathbf{1}$ **then**
$\boldsymbol{Dst(i)}|_h \longleftarrow \mathbf{1}$
$\boldsymbol{Fan\_in\_Array(Dst(i))(\rho(i))} \longleftarrow \mathbf{0}$
/* $Dst(i)$ and $\rho(i)$ may be thought of as index registers that are used to access $Fan\_in\_Array(Dst(i))(\rho(i))$. The value of $Dst(i)$ is changed only one bit at a time and once the value of $\rho(i)$ is fixed (in $\Theta(\log n)$ time), it is never changed. */
end

```
        /* Set Step: Mark the appropriate locations of Fan_in_Array(Dst(i)) */
        if Left_Set(i) = 0 and First(i) = 1 then
            Fan_in_Array(Dst(i))(ρ(i)) ⟵ 1
        end
        /* Check Step: Check Fan_in_Array(Dst(i)) for marks */
        if Left_Set(i) = 1 and Last(i) = 1 then
            if Fan_in_Array(Dst(i))(ρ(i)) = 1 then
              /* a mark has been found */
              Level(i)|_h ⟵ 1 /* bit h of Level(i) set to 1 */
              Last(i) ⟵ 0  /* k_i is no longer a last element */
              Fan_in_Array(Dst(i))(ρ(i)) ⟵ 0
              /* This reinitialization of Fan_in_Array(Dst(i)) is done so that
                 the elements of the Right Sets may adjust First(i) */
            end
        end
        /* Adjust First(i). This is done by the first elements k_i of the Right Set.
           If a last element k_i' of the Left Set for which ρ(k_i') = ρ(k_i), detects a
           neighbor in the Right Set, then k_i must be its neighbor. Thus k_i is
           no longer a first element for the next iteration and First(i) must be
           set to 0 */
        if Left_Set(i) = 0 and First(i) = 1 then
            if Fan_in_Array(Dst(i))(ρ(i)) = 0 then
            /* A last element of the Left Set has detected a mark. */
              First(i) ⟵ 0
            end
        end
    end
    /* End of Iterations */
    /* At this point Level(i)|_h = 1 iff the root of T_i is at level h. We have to set
       Level(i)|_j to 1 for all j ≤ h. Also Flag(i) has to be set.
       Recall that Flag(i) = 1 iff k_i has a neighbor */
    Flag(i) ⟵ 0   /* initialization */
    for h ⟵ 0 to log(n) − 1  do
        if Flag(i) = 0 then
            if Level(i)|_h = 1 then
              Flag(i) ⟵ 1 /* Level(i) is not changed any more */
            else
              Level(i)|_h ⟵ 1
            end
        end
    end
end      /* End of Step 1 */
```

```
Procedure Step_2  /* Search T_i */
/* Executed in parallel by all processors indexed i */
/* In this procedure each processor i searches T_i for the neighbor of k_i */
   from the root of T_i */
begin
   /* Initialization */
   Link(i) ⟵ Dst(i)      /* This is the root of the current subtree of the fan-in
   tree that is being searched. Initially, it is the root of T_i */
   /* Initialize Dst(i) to the destination processor indices at level log n − 2 */
   for h ⟵ 0 to log(n) − 2 do
       Dst(i)|_h ⟵ 1
   end
   Dst(i)|_{log n−1} ⟵ i|_{log n−1}   /* i|_{log n−1} denotes the msb of i */
   Half_Level(i) ⟵ Level(i) shifted right by 1 bit.
       /* Half_Level(i), as the name indicates, is Level(i) div 2 and is needed to
          determine the levels of the Fan-in tree that processor i searches if necessary.
          Level(i) is used to reconstruct the Fan_in_Array for the searches.
          The above assignment can be done in Θ(log n) time. */
   /* End Initialization */
   /* Determine the neighbor. Each iteration searches one level of the Fan-in tree */
   for h ⟵ log(n) − 2 down to 0 do
       if Flag(i) = 1 then
          if Half_Level(i)|_h = 1 then
                /* Half_Level(i)|_h = 1 iff Level(i)|_{h+1} = 1 */
             Link(i)|_h ⟵ 0   /* search the left subtree */
             /* Initialize Fan_in_Array */
             Fan_in_Array(Link(i))(ρ(i)) ⟵ 0
          end
       end
       Dst(i)|_h ⟵ i|_h
       /* Reconstruct Fan_in_Array */
       if Level(i)|_h = 1 then
          Fan_in_Array(Dst(i))(ρ(i)) ⟵ 1
       end
       /* Check Fan_in_Array */
       if Flag(i) = 1 then
          if Half_Level(i)|_h = 1 then
                if Fan_in_Array(Link(i))(ρ(i)) = 0 then
             /* Left subtree does not have a neighbor. Therefore we set Link to
                the root of the right subtree */
                Link(i)|_h ⟵ 1
             end
```

```
        end
    end
    /* End Iterations */
end   /* End Step 2 */
```

As mentioned earlier, Step B of Hagerup's algorithm requires that the beginning and end of each list generated by Step A be available for access by a processor of size $\log n$ bits in constant time. In our algorithm, the end of each list is given by the processor that has $Nbr(i)$ set to NIL. To find the beginning we reverse the Nbr list (i.e. generate a list represented by Rev_Nbr) and look for the processor that has $Rev\_Nbr(i)$ set to NIL. We use two Arrays, the $Begin\_Array$ and the End_Array, each containing n pointer locations to store the above information. The following pseudo code uses n processors, each of size $\log n$ bits and achieves a time of $\Theta(1)$. It is straight forward to modify the pseudo code for $\frac{n}{\log n}$ processors, each of size $\log n$ bits and a time of $\Theta(\log n)$.

```
Procedure Find_Begin_and_End_of_Lists
/* Executed in parallel by all processors indexed i */
begin
        /* Initialize Begin_Array and End_Array */
        Begin_Array(i) ⟵ NIL
        End_Array(i) ⟵ NIL
        if Nbr(i) = NIL then
            End_Array(ρ(i)) ⟵ i
        end
        /* Reverse the Nbr list */
        Rev_Nbr(i) ⟵ NIL   /* Initialization */
        Rev_Nbr(Nbr(i)) ⟵ i
        /* Set End_Array */
        if Rev_Nbr(i) = NIL then
            Begin_Array(ρ(i)) ⟵ i
        end
end
```

B An Illustration of the Neighbor Localization Problem Algorithm

We now illustrate the steps of the Neighbor Localization Algorithm with an example where $n = 8$. The values of the 8 numbers are given below; $\rho(0) = 5, \rho(1) = 2, \rho(2) = 5, \rho(3) = 5, \rho(4) = 4, \rho(5) = 7, \rho(6) = 4$ and $\rho(7) = 2$. We now give below the values in the various memory locations at each step of the algorithm. Locations that are left blank contain garbage (undefined) values. We suggest that this portion be read with the algorithmic descriptions given in § 5.

i	$\rho(i)$	$Dst(i)$	$First(i)$	$Last(i)$	$Level(i)$
0	5	0	1	1	0 0 0
1	2	1	1	1	0 0 0
2	5	2	1	1	0 0 0
3	5	3	1	1	0 0 0
4	4	4	1	1	0 0 0
5	7	5	1	1	0 0 0
6	4	6	1	1	0 0 0
7	2	7	1	1	0 0 0

Table 2: Step 1; Initialization

i	$\rho(i)$	$Dst(i)$	$First(i)$	$Last(i)$	$Level(i)$	$Left\_Set(i)$
1	2	1	1	1	0 0 0	0
2	5	3	1	0	0 0 1	1
3	5	3	0	1	0 0 0	0
4	4	5	1	1	0 0 0	1
5	7	5	1	1	0 0 0	0
6	4	7	1	1	0 0 0	1
7	2	7	1	1	0 0 0	0

Table 3: Step 1, Iteration 0; Variables

	0	1	2	3	4	5	6	7
0								
1						0		
2								
3						0		
4								
5					0			
6								
7					0			

Table 4: Step 1, Iteration 0; *Fan_in_Array* after initialization

	0	1	2	3	4	5	6	7
0								
1			1			0		
2								
3						1		
4								
5					0			1
6								
7			1		0			

Table 5: Step 1, Iteration 0; *Fan_in_Array* after marking

	0	1	2	3	4	5	6	7
0								
1			1			0		
2								
3						0		
4								
5					0			1
6								
7			1		0			

Table 6: Step 1, Iteration 0; *Fan_in_Array* after resetting marks

i	$\rho(i)$	$Dst(i)$	$First(i)$	$Last(i)$	$Level(i)$	$Left\_Set(i)$
0	5	3	1	0	0 1 0	1
1	2	3	1	1	0 0 0	1
2	5	3	0	0	0 0 1	0
3	5	3	0	1	0 0 0	0
4	4	7	1	0	0 1 0	1
5	7	7	1	1	0 0 0	1
6	4	7	0	1	0 0 0	0
7	2	7	1	1	0 0 0	0

Table 7: Step 1, Iteration 1; Variables

	0	1	2	3	4	5	6	7
0								
1								
2								
3			0			0		
4								
5								
6								
7					0			0

Table 8: Step 1, Iteration 1;*Fan_in_Array* after initialization

	0	1	2	3	4	5	6	7
0								
1								
2								
3			0			1		
4								
5								
6								
7			1		1			0

Table 9: Step 1, Iteration 1;*Fan_in_Array* after marking

	0	1	2	3	4	5	6	7
0								
1								
2								
3			0			0		
4								
5								
6								
7			1		0			0

Table 10: Step 1, Iteration 1;*Fan_in_Array* after resetting marks

i	$\rho(i)$	$Dst(i)$	$First(i)$	$Last(i)$	$Level(i)$	$Left\_Set(i)$
0	5	3	1	0	0 1 0	1
1	2	7	1	0	1 0 0	1
2	5	3	0	0	0 0 1	1
3	5	7	0	1	0 0 0	1
4	4	7	1	0	0 1 0	0
5	7	7	1	1	0 0 0	0
6	4	7	0	1	0 0 0	0
7	2	7	0	1	0 0 0	0

Table 11: Step 1, Iteration 2; Variables

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7			0			0		

Table 12: Step 1, Iteration 2;*Fan_in_Array* after initialization

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7			1		1	0		1

Table 13: Step 1, Iteration 2;*Fan_in_Array* after marking

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7			0		1	0		1

Table 14: Step 1, Iteration 2;*Fan_in_Array* after resetting marks

	Initially		$h = 0$		$h = 1$		$h = 2$	
i	$Level$	$Flag$	$Level$	$Flag$	$Level$	$Flag$	$Level$	$Flag$
0	0 1 0	0	0 1 1	0	0 1 1	1	0 1 1	1
1	1 0 0	0	1 0 1	0	1 1 1	0	1 1 1	1
2	0 0 1	0	0 0 1	1	0 0 1	1	0 0 1	1
3	0 0 0	0	0 0 1	0	0 1 1	0	1 1 1	0
4	0 1 0	0	0 1 1	0	0 1 1	1	0 1 1	1
5	0 0 0	0	0 0 1	0	0 1 1	0	1 1 1	0
6	0 0 0	0	0 0 1	0	0 1 1	0	1 1 1	0
7	0 0 0	0	0 0 1	0	0 1 1	0	1 1 1	0

Table 15: Step 1; Setting $Flag$ and $Level$

i	$\rho(i)$	$Flag(i)$	$Level(i)$	$Dst(i)$	$Link(i)$	$Half\_Level(i)$
0	5	1	0 1 1	3	3	0 0 1
1	2	1	1 1 1	3	7	0 1 1
2	5	1	0 0 1	3	3	0 0 0
3	5	0	1 1 1	3	7	0 1 1
4	4	1	0 1 1	7	7	0 0 1
5	7	0	1 1 1	7	7	0 1 1
6	4	0	1 1 1	7	7	0 1 1
7	2	0	1 1 1	7	7	0 1 1

Table 16: Step 2; Initialization

i	$\rho(i)$	$Flag(i)$	$Level(i)$	$Dst(i)$	$Link(i)$	$Half\_Level(i)$
0	5	1	0 1 1	1	3	0 0 1
1	2	1	1 1 1	1	7	0 1 1
2	5	1	0 0 1	3	3	0 0 0
3	5	0	1 1 1	3	7	0 1 1
4	4	1	0 1 1	5	7	0 0 1
5	7	0	1 1 1	5	7	0 1 1
6	4	0	1 1 1	7	7	0 1 1
7	2	0	1 1 1	7	7	0 1 1

Table 17: Step 2, Iteration 1; Variables

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5			0					
6								
7								

Table 18: Step 2, Iteration 1; Fan_in_Array after initialization

	0	1	2	3	4	5	6	7
0								
1			1			1		
2								
3						1		
4								
5			0		1			1
6								
7			1		1			

Table 19: Step 2, Iteration 1; Fan_in_Array after marking

i	$\rho(i)$	$Flag(i)$	$Level(i)$	$Dst(i)$	$Link(i)$	$Half\_Level(i)$
0	5	1	0 1 1	0	2	0 0 1
1	2	1	1 1 1	1	7	0 1 1
2	5	1	0 0 1	2	3	0 0 0
3	5	0	1 1 1	3	7	0 1 1
4	4	1	0 1 1	4	6	0 0 1
5	7	0	1 1 1	5	7	0 1 1
6	4	0	1 1 1	6	7	0 1 1
7	2	0	1 1 1	7	7	0 1 1

Table 20: Step 2, Iteration 0; Variables

	0	1	2	3	4	5	6	7
0								
1								
2						0		
3								
4								
5								
6			0		0			
7								

Table 21: Step 2, Iteration 0; *Fan_in_Array* after initialization

	0	1	2	3	4	5	6	7
0						1		
1			1					
2						1		
3						1		
4					1			
5								1
6			0		1			
7			1					

Table 22: Step 2, Iteration 0; *Fan_in_Array* after marking

i	$\rho(i)$	$Flag(i)$	$Link(i)$	$Nbr(i)$
0	5	1	2	2
1	2	1	7	7
2	5	1	3	3
3	5	0	7	NIL
4	4	1	6	6
5	7	0	7	NIL
6	4	0	7	NIL
7	2	0	7	NIL

Table 23: Step 3; Variables