1-1991

Optimal Parallel Lexicographic Sorting using a Fine-Grained Decomposition

Ramachandran Vaidyanathan

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

Pramod Varshney
Syracuse University, varshney@syr.edu

SU-CIS-91-01

Optimal Parallel Lexicographic Sorting using a Fine-Grained Decomposition

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

January 1991

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

(315) 443-2368

Optimal Parallel Lexicographic Sorting using a Fine-Grained Decomposition[1]

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 is with the Department of Electrical and Computer Engineering at 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 and Information Science at Syracuse University, Syracuse, NY 13244-4100. e-mail: hartmann@top.cis.syr.edu

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

Abstract: Though non-comparison based sorting techniques like radix sorting can be done with less "work" than conventional comparison-based methods, they are not used for long keys. This is because even though parallel radix sorting algorithms process the keys in parallel, the symbols in the keys are processed sequentially. In this report, we give an optimal algorithm for lexicographic sorting that can be used to sort n m-bit keys on an EREW model in $\Theta(\log n \log m)$ time with $\Theta(mn)$ "work". This algorithm is not only as fast as any optimal non-comparison based algorithm, but can also be executed with less work. We also use the proposed algorithm to show that if n $\Theta(\log n)$ unsigned binary numbers can be sorted optimally on an EREW PRAM than n unsigned binary numbers of unrestricted length can be sorted optimally on an EREW PRAM.

Keywords: ISR-PRAM, Lexicographic Sorting, Parallel Algorithms, Parallel Processing, PRAM, R-PRAM, Sorting.

Contents

List of Tables

List of Figures

1 Introduction

For years, sorting has drawn the interest of researchers, due both to the theoretical challenges that it poses and its practical applications. With the advent of parallel processing, this interest has been considerably enhanced. Several sorting algorithms have been proposed, one of the most important of which is the work of Ajtai, Komlós and Szemerédi (the AKS sorting network) [2]. This method led to the first algorithm that sorts n keys in $\Theta(\log n)$ comparison time with n processors. More recently, Cole [6] has developed an algorithm that also sorts n keys in $\Theta(\log n)$ time on an EREW PRAM with n processors. Several other sorting algorithms have also been proposed that sort n keys with a processor-time product of $\Theta(n \log n)$, but with a time that is a strictly higher order than $\log n$.

It is well known that the order of the processor-time product for any comparison-based algorithm for sorting n keys is lower bounded by $\Theta(n \log n)$ [9]. The atomic operation for comparison-based sorting is comparison. To compare two m-bit numbers with $\Theta(1)$ "gates", one needs $\Theta(m)$ time. Indeed, this is the best that can be done with $\Theta(1)$ gates, as it takes $\Theta(m)$ time to even scan the m-bit numbers. One could measure the efficiency of the comparison by the "*Gate-Time Product*" (GTP), a measure similar to the processor-time product. The idea of the GTP has been discussed further in § 2. For the comparison of two m-bit numbers the GTP is lower bounded by $\Theta(m)$. In fact, the most common parallel method of comparison, employed for comparing two m-bit numbers with a constant fan-in, constant fan-out circuit requires $\Theta(\log m)$ time (the information is fanned-in in a binary tree fashion). This method can be modified using Brent's theorem [5], so that two m-bit numbers can be compared in $\Theta(\log m)$ time, with $\Theta(\frac{m}{\log m})$ gates. On the other hand if a processor of "size m bits" is used (the idea of a processor of size b bits has been defined in § 2), the above comparison could possibly be done faster by using a look-up table. It turns out that if m is upper bounded by $\Theta(\log n)$ (where n is the size of the problem in which the m-bit comparison is being used) then the above comparison can be done in $\Theta(1)$ time, otherwise $\Theta(\log m)$ time is required. These ideas have been explained in more detail in § 2. For reasons explained later in this section, we will consider in this report, the problem of sorting n m-bit unsigned binary numbers (keys), where m is a strictly higher order than $\Theta(\log n)$. In the light of the above discussions one could say that the GTP for sorting n m-bit keys using a comparison-based sorting method is lower bounded by $\Theta(mn \log n)$, and the fastest algorithms [2, 6] that achieve this lower bound on the GTP, take a time of $\Theta(\log n \log m)$, when m is a strictly higher order than $\Theta(\log n)$. It should also be mentioned here that Azar and Vishkin [4] have shown that the lower bound on the time (measured in terms of comparison times) for any comparison-based sorting algorithm that performs a total of $\Theta(n \log n)$ comparisons is $\Theta(\log n)$. Restated in terms of the GTP, any comparison-based sorting algorithm for an EREW model that has a GTP of $\Theta(mn \log n)$ requires at least $\Theta(\log n \log m)$ time, if m is a strictly higher order than $\Theta(\log n)$. This in effect shows that the algorithms in [2, 6] are the fastest possible comparison-based EREW sorting algorithms that have a GTP of $\Theta(mn \log n)$.

One of the most common non-comparison based sorting techniques is called radix or distribution sorting [1, 9]. Methods based on this technique have keys that are represented by a q-tuple $<s_0, s_1, \ldots, s_{q-1}>$ where s_i $(0 \leq i < q)$ is a section of the key. Each section s_i is drawn from a set S, whose elements (the sections) are totally ordered by a relation $\preceq$. Consider two keys $k_x = <s_0^x, s_1^x, \ldots, s_{q-1}^x>$ and $k_y = <s_0^y, s_1^y, \ldots, s_{q-1}^y>$. We say that $k_x \sqsubseteq k_y$ if and only if $\exists i \in \{0, 1, \ldots, q-1\} \ni$ $s_i^x \preceq s_i^y$ and $\forall i', 0 \leq i' < i,$ $s_{i'}^x = s_{i'}^y$. The above ordering relation $\sqsubseteq$ on the set of keys is called a *lexicographic ordering* and a sorting of the keys based on this ordering is called *lexicographic sorting.*

The lexicographic sorting method given in [1] is for sorting n keys; each key being an ordered sequence of q sections, where each section is drawn from the set $\{0, 1, \ldots, r-1\}$. This algorithm requires $\Theta(q(n+r))$ time, if one processor is used. If each key is viewed as an m-bit number, where $m = q\lceil \log_2 r \rceil$, then the time required is $\Theta((\frac{m}{\log r})(n+r))$. However, it is assumed in the above algorithm that a pointer to a key's index can be accessed by the processor in constant time. Since the pointer is $\log n$ bits long, the processor used must at least of size $\log n$ bits. Thus, the GTP for the algorithm is at least $\Theta((\frac{m}{\log r})(n+r)\log n)$. If we group the m bits (where the order of m is at least $\Theta(\log n)$) in the keys to form groups of at most $\log n$ contiguous bits (i.e. $r = n$), then the above algorithm could be used to sort the numbers in $\Theta(\frac{mn}{\log n})$ time with one processor of size $\log n$ bits; the GTP is $\Theta(mn)$. Thus, an upper bound on the GTP for a non-comparison based sorting technique for n m-bit keys is $\Theta(mn)$. Since the number of bits in the input is mn, a lower bound on the GTP for the above problem is also $\Theta(mn)$. Thus the algorithm in [1] is optimal with respect to the GTP.

Even though radix sorting can be done with a lower order of GTP than conventional comparison-based sorting algorithms, it is not used for long keys as the time required grows linearly with the length of the keys. A natural question therefore is, "can n m-bit numbers be sorted on an EREW model in $\Theta(\log n \log m)$ time and a GTP of $\Theta(mn)$?"

Consider an algorithm to sort n m-bit numbers. If this algorithm is comparison-based then as mentioned earlier, the GTP is lower bounded by $\Theta(mn \log n)$. Therefore, any algorithm that achieves a time of $\Theta(\log n \log m)$ and a GTP of $\Theta(mn)$ on an EREW model must be a non-comparison based algorithm. One possibility is to use a lexicographic sorting algorithm in which the keys are assumed to be unsigned binary numbers. Unlike comparison-based algorithms which view the input as n indivisible objects, we will view the input as an $n \times m$ matrix of bits (henceforth referred to as the input matrix). The rows in the input matrix correspond to the keys. In order to sort this input fast and efficiently, one must use a model of computation that can manipulate the input elements with a certain amount of independence. In other words, the processors in this model must be able to access and manipulate the bits in the input matrix autonomously. For this purpose, we use a model of computation called the Reconfigurable PRAM (R-PRAM), that permits the use of "small processors". The R-PRAM is a variant of the PRAM that allows the problem to be decomposed very finely. More details of the R-PRAM appear in § 3 and in [12].

In this report, we propose a parallel lexicographic sorting algorithm for an EREW R-PRAM that sorts n m-bit numbers (where m is a strictly higher order than $\Theta(\log n)$), in $\Theta(\log n \log m)$ time with a GTP of $\Theta(mn)$ and a memory that is polynomial in mn. We note here that the above time complexity is the same as that of the fastest possible comparison-based parallel sorting algorithms. Moreover, this time is achieved with the lowest possible GTP. It has been shown in [7] that the lower bound for sorting n keys (by any method) on any CREW model is $\Theta(\log n)$. This result holds for both comparison-based and non-comparison based sorting algorithms. It has also been shown in [7] that the lower bound on finding the bitwise OR of an m-bit number on any CREW model is $\Theta(\log m)$. Since the bitwise OR of an m-bit number can be determined by comparing the number with an m-bit zero (i.e. m-bit number with all bits set to 0), the lower bound on comparing two m-bit numbers on any CREW model is $\Theta(\log m)$. Thus, we can conclude that our algorithm cannot be speeded up by more than a constant factor. It should be mentioned here that when m is $\Theta(\log n)$, the sorting can be done optimally on an EREW R-PRAM in $\Theta(\log n)$ time and with a GTP of $\Theta(n \log n)$ [11].

The contribution of this work is twofold:

– It gives an optimal EREW algorithm for sorting n m-bit numbers (where m is a strictly higher order than $\Theta(\log n)$) that has the speed of the fastest comparison-based sorting algorithms and the efficiency of the most efficient non-comparison based sorting algorithms. This efficiency cannot be achieved by any comparison-based algorithm.

– It proves that if n $\Theta(\log n)$-bit unsigned binary numbers can be sorted optimally on an EREW PRAM, then so can n unsigned binary numbers of unrestricted length. More details appear in § 6.

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 $i \geq r$. For any positive integer i, $N(i)$ is defined to be the set $\{0, 1, \cdots, i-1\}$. For any set S, $|S|$ denotes its cardinality. 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 discuss a few preliminaries and give a formal description of the sorting problem. § 5 is devoted to the description of our sorting algorithm and its complexity analysis. In § 6 we discuss how the proposed algorithm could be extended to the conventional PRAM model. 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 the 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. We will therefore not consider this representation of the problem instance, any further. 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-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. More details appear in [12]. Before we outline the R-PRAM, a few relevant details are discussed below.

Any computable function $f : \{0,1\}^n \longrightarrow \{0,1\}^m$ (that represents a problem of size n) 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 [10]) 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 now the addition of two b-bit numbers using a processor of "size b bits" (the idea of a processor of size b bits is formalized later in this section). The above addition cannot be done in time independent of b, without a table look-up. Even though some of the instructions in the minimal instruction set can be executed in $\Theta(1)$ time without a table look-up, for uniformity we will assume that all instructions are executed with a table look-up. The total size of the look-up tables for each processor is $\Theta(2^{\Theta(1)b})$ b-bit words, which by our earlier assumption is $O(n^{\Theta(1)})$; thus, b is $O(\log n)$. In fact, if b is $\Theta(\log n)$, then any instruction that has $O(\log n)$-bit operand(s) 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 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 [12]. 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. Other hardware necessary in a practical processor, like the memory and its ports, are not counted in our definition of computing hardware.

As mentioned in § 1, the comparison of two m bit numbers on an EREW model needs $\Theta(1)$ time, if m is $O(\log n)$ and $\Theta(\log n)$ time, otherwise. Consider the case where m is $\Theta((\log n)^{(1+\frac{\Theta(1)}{\log\log n})})$. Substituting this value of m in 2^m, the size of the look-up table turns out to be $\Theta(n^{\Theta(1)})$ bits. We note that for an asymptotically large value of n, $(\log n)^{(1+\frac{\Theta(1)}{\log\log n})}$ is $\Theta(\log n)$. Thus if m is $O(\log n)$, then 2^m is $O(n^{\Theta(1)})$ and the comparison time is $\Theta(1)$. If m is $\Theta((\log n)^{(1+\epsilon)})$ (where, ϵ is a constant greater than 0), then the size of the look-up table is $\Theta(n^{\Theta((\log n)^{\epsilon})})$, which is not polynomially bounded in mn. This, as mentioned earlier in this section, is not permitted. Therefore, if m is $\omega(\log n)$, we will only be able to use processors of size $\Theta(\log n)$ bits. Let us assume that we have an unbounded number of such processors. If an EREW model is used, then we will require $\Theta(\log(\frac{m}{\log n}))$ time to perform the comparison. The m bits in each of the input numbers are divided into $\lceil\frac{m}{\log n}\rceil$ sections, each at most $\log n$ bits long. Each section of the two numbers can be compared by a processor of size $\log n$ bits in $\Theta(1)$ time. Next, the $\lceil\frac{m}{\log n}\rceil$ partial results can be fanned in, in $\Theta(\log(\frac{m}{\log n}))$ time. The same order of time can be achieved with $\Theta(\frac{m}{\log n}/\log(\frac{m}{\log n}))$ processors.

Let m be $\Theta((\log n)^{1+f(n)})$, where $f(n)$ is any positive non-decreasing function of n. The time required for the comparison is $\Theta(f(n)\log\log n)$ which is $\Theta(\log m)$. Since $\log(\frac{m}{\log n})$ is $\Theta(\log m)$, the number of processors of size $\log n$ that are are used is $\frac{m}{\log n \log m}$; the resulting GTP is $\Theta(m)$, which is optimal.

The use of a table look-up by a processor of size $\log n$ bits, to execute instructions in constant time, involves the use of $\Theta(n^2 \log n)$ bits of memory for operations like $\log n$-bit addition. For an EREW model with p processors, each of size $\log n$ bits, the total size of the look-up tables is $\Theta(pn^2 \log n)$. As mentioned earlier, the R-PRAM models a fine-grained decomposition. At this level of abstraction, it is necessary to count the look-up table space when calculating the space complexity of an algorithm. Without the look-up tables, the operations cannot be done in $\Theta(1)$ time. On the other hand, in the conventional PRAM, the look-up tables may be disregarded as at this level of abstraction, each operation, by definition, can be performed in $\Theta(1)$ time. However, the processors in the PRAM will be assumed to be of size $\Theta(\log n)$ bits.

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 $\sum_{i=0}^{p-1} s_i$ 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 some aspects of communication and address decoding. In addition, the R-PRAM also abstracts the solution from some aspects of address generation and loop management. More details of these issues appear later in this section and in [12].

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, hence the name Reconfigurable PRAM. The reconfiguration is static; i.e. given the size of the problem, it can be decided *a priori*, which configuration the R-PRAM will assume at any point in the execution of the algorithm. We assume that the R-PRAM can be reconfigured in constant time. We also assume that each processor in any configuration of the R-PRAM that has p processors, each of size b bits, has n (the problem size), b (the processor size) and its index (a unique number between 0 and $p-1$) available to it. 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. As mentioned earlier, the R-PRAM can be configured as $\Theta(\frac{\mathcal{H}}{b})$ processors each of size b bits (where b is $O(\log n)$). In order to ensure that at least $\Theta(1)$ processors, each of size $O(\log n)$ bits is available, we will assume $\mathcal{H}$ to be $\Omega(\log n)$. This is similar to assuming that a PRAM at least $\Theta(1)$ processors. Like the PRAM, the R-PRAM can be EREW, CREW or CRCW. In this report, we use the EREW R-PRAM.

Since the address of the memory is $\Theta(\log n)$ bits long whereas the processors in the R-PRAM could be of size $o(\log n)$ bits, the address generation mechanism of the R-PRAM needs further elaboration. For this purpose, it is convenient to divide the variables used in an R-PRAM algorithm 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 are 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. This enables us to generate the address of a variable by accessing one or more bits in one or more index registers. In other words, all $\Theta(\log n)$ bits of the address need not be explicitly set every time a memory access is made.

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). A weaker variant of the R-PRAM called the ISR-PRAM accounts for all these overheads. More details appear in § 5.4 and in [12].

4 Preliminaries

Throughout this report, we will assume that there are n m-bit keys (unsigned binary numbers) to be sorted and that the sorting is to be done with respect to $\leq$. Before we proceed to the the sorting algorithm, a few definitions and terminology are necessary for the formal description of the sorting problem.

We denote by $\mathbf{K} = \{k_0, k_1, \ldots, k_{n-1}\}$ the set of keys and for all $i, j \in N(n)$, $k_i = a_{i,0}a_{i,1} \ldots a_{i,m-1}$, where $a_{i,j} \in \{0, 1\}$. In other words, the value of the number that k_i represents is $\sum_{j=0}^{m-1} a_{i,j}.2^{m-1-j}$. The input to the sorting algorithm is the $n \times m$ input matrix $[a_{i,j}]$. Let for some $q \in N(n)$, $\mathbf{P}' = \{B_0, B_1, \ldots, B_q\}$ be the partition of $\mathbf{K}$ with respect to *equality* (of the keys). $B_0, B_1, \ldots, B_q$ are the blocks of $\mathbf{P}'$, each of which contain keys that are equal in value. By imposing two relations α and β on the blocks of $\mathbf{P}'$ and on the elements of any block of $\mathbf{P}'$, respectively, we define an *ordered partition* $\mathbf{P}$ of $\mathbf{K}$. We define α and β as follows:

(*i*) $\forall i, j \in N(q)$, $B_i \alpha B_j$ if and only if $\forall k_{\ell_1} \in B_i$, $\forall k_{\ell_2} \in B_j$ $k_{\ell_1} < k_{\ell_2}$.

(*ii*) Let k_{ℓ_1} and k_{ℓ_2} be two distinct keys in some block B_i of $\mathbf{P}'$.

$k_{\ell_1} \beta k_{\ell_2}$ if and only if $\ell_1 < \ell_2$.

It is not difficult to see that the keys can be sorted by determining α. If β is also determined, then the sorting is stable [9][5]. Thus, the problem of sorting the elements of $\mathbf{K}$ stably, is the same as finding the ordered partition $\mathbf{P}$ of $\mathbf{K}$.

Let $A = \{a_1, a_2, \ldots, a_p\} \subseteq N(m)$. Given any key $k_i \in \mathbf{K}$ and the set A, we define the *section of* k_i *with respect to* A to be a quantity whose value is the p-bit number formed by the bits of k_i with indices in A. These bits are arranged in increasing order of their indices. We denote the section of k_i with respect to A by $k_i(A)$. For instance, if $m = 8, A = \{0, 3, 7\}$ and $k_i = 10110100$ (in binary representation), then the value of $k_i(A) = 110$ (also in binary representation). In a similar fashion, we define $\mathbf{K}_A = \{k_i(A) : k_i \in \mathbf{K}\}$, the set of sections of all the keys of $\mathbf{K}$ with respect to A. We now extend the idea of an ordered partition $\mathbf{P}$ of $\mathbf{K}$ to that of an ordered partition $\mathbf{P}_A$ of $\mathbf{K}$ with respect to a set $A \subseteq N(m)$. Let $\mathbf{P}'_A = \{B_0^A, B_1^A, \ldots, B_q^A\}$ be an unordered partition of $\mathbf{K}$ with respect to equality of the elements of $\mathbf{K}_A$. We define the relations α_A and β_A as follows:

(*i*) $\forall i, j \in N(q)$; $B_i^A \alpha_A B_j^A$ if and only if $\forall k_{\ell_1} \in B_i^A$, $\forall k_{\ell_2} \in B_j^A$ $k_{\ell_1}(A) < k_{\ell_2}(A)$.

(*ii*) Let k_{ℓ_1} and k_{ℓ_2} be two distinct elements of some block B_i^A of $\mathbf{P}'_A$.

$k_{\ell_1} \beta_A k_{\ell_2}$ if and only if $\ell_1 < \ell_2$.

It should be noted that the ordered partition $\mathbf{P}_A$ is a partition of $\mathbf{K}$ and not of $\mathbf{K}_A$. Even though the definition of a ordered partition $\mathbf{P}_A$ implies the existence of the relation β_A, we will often use the term *stable ordered partition* to emphasize this fact.

[5] $<k_1, k_2, \ldots, k_{n-1}>$ is sorted stably to form the list $<k_{j_1}, k_{j_2}, \ldots, k_{j_{n-1}}>$ if and only if $\forall j_q, j_s \in N(n)$, $(q < s) \Longrightarrow (k_{j_q} < k_{j_s})$ or $(k_{j_q} = k_{j_s}$ and $j_q < j_s)$

5 The Sorting Algorithm

In this section, we propose a sorting algorithm for an EREW R-PRAM that can sort n m-bit numbers with a GTP of $\Theta(mn)$ in $\Theta(\log n \log m)$ time. The basic idea of the algorithm is to construct an initial set of ordered partitions, based on disjoint sections of the keys, and successively refine them to obtain $\mathbf{P}_{N(m)}$, the solution to the sorting problem. In § 5.1 we outline the algorithm and a few of its essential features. In § 5.2 we discuss the leader finding problem which will be used subsequently in the description of the sorting algorithm. We describe the sorting algorithm with an example in § 5.3, and finally in § 5.4 we perform a complexity analysis of our algorithm.

5.1 A Brief Outline of the Sorting Algorithm

The sorting algorithm may be written as the following 3-step procedure:

Step 1: In this step we divide the m bits in the keys into $\lceil \frac{m}{\log n} \rceil$ sections, each section containing at most $\lceil \log n \rceil$ bits. Thus, each section may be considered to be a set of n $\lceil \log n \rceil$-bit numbers. In this step, we form $\lceil \frac{m}{\log n} \rceil$ ordered partitions based on the above sections. Recall that m is $\omega(\log n)$.

Step 2: The above ordered partitions are merged in a binary tree fashion in $\left\lceil \log \left(\lceil \frac{m}{\log n} \rceil \right) \right\rceil$ merge steps to construct $\mathbf{P}_{N(m)}$, the final ordered partition.

Step 3: The keys are relocated according to their order in $\mathbf{P}_{N(m)}$.

The variables used in the proposed algorithm will be termed *parallel variables*. A parallel variable is defined with respect to an index set S. It has $|S|$ components, one for each element of S. For example, a parallel variable named "*Order*" defined with respect to the set $N(n) = \{0, 1, \ldots, n-1\}$ of indices of the keys, will have a component $Order(i)$ for each $k_i \in \mathbf{K}$. Each component of a parallel variable could be a bit or even an array of bits. Normally, $S = N(n)$; however in Appendix A, we use parallel variables defined with respect to the set of processor indices.

Before we can understand Step 1 it is necessary for us to know how an ordered partition is represented. It is easy to see that an ordered partition of $\mathbf{K}$ can be uniquely specified by specifying for each key, the block to which it belongs and by specifying the order of the keys in the ordered partition. We do this by means of two parallel variables, *Block* and *Order*. *Block* and *Order* are both parallel variables of pointers to the keys.

Let $\mathbf{P}_J$ be a stable ordered partition based on the set J. We will use the components of *Block* and *Order* to represent $\mathbf{P}_J$. Let these n components be $Block(i, J)$ and $Order(i, J)$; $i \in N(n)$.

Consider a block B of $\mathbf{P}_J$. Let $k_i \in B$ be the key with the lowest index i among all the keys in B. Since $\mathbf{P}_J$ is stable, k_i must be the first element of the block B of $\mathbf{P}_J$ in the order imposed by the relation β_J. For this reason, k_i is called the *head* of the block B. For any block $B \in \mathbf{P}_J$ and any key $k_i \in B$, $Block(i, J)$ points to the head of B. $Order(i, J)$ is used to form a list of the keys in the order in which they occur in the ordered partition $\mathbf{P}_J$. Fig. 1 shows the representation of the ordered partition $\{\{k_0, k_3, k_5, k_7\}, \{k_1, k_2, k_4\}, \{k_6\}\}$. In Step 1 of the sorting algorithm we represent each of the $\lceil \frac{m}{\log n} \rceil$ sorted sections by the parallel variables *Block* and *Order*, as detailed above. Step 1 needs $\Theta(\frac{mn}{\mathcal{H}})$ time and $\Theta(\mathcal{H}n^2)$ space on an EREW R-PRAM with $\mathcal{H}$ bits of computing hardware, where $\mathcal{H}$ is $\Omega(\log n)$ and $O(\frac{mn}{\log n})$. Further details of Step 1 are discussed in § 5.3.

The most important part of the sorting algorithm is the *merge procedure* in Step 2. Given two ordered partitions $\mathbf{P}_X$ and $\mathbf{P}_Y$ based on the disjoint sets $X, Y \subseteq N(m)$, a step in the merge procedure (a merge step) constructs the ordered partition $\mathbf{P}_{X \cup Y}$ based on the set of $X \cup Y$. The sets X and Y are such that $\forall x \in X$, $\forall y \in Y$, $x < y$. Thus, for any key $k_i \in \mathbf{K}$, $k_i(X)$, the section of k_i based on X, has higher weight than $k_i(Y)$, the section of k_i based on Y, as the keys are assumed to be unsigned binary numbers. Therefore for any two keys $k_{i_1}, k_{i_2} \in \mathbf{K}$.

$$\begin{array}{rclcllll}
k_{i_1}(X \cup Y) & = & k_{i_2}(X \cup Y) & \Longleftrightarrow & (k_{i_1}(X) = k_{i_2}(X)) & \text{and} & (k_{i_1}(Y) = k_{i_2}(Y)) \\
k_{i_1}(X \cup Y) & < & k_{i_2}(X \cup Y) & \Longleftrightarrow & (k_{i_1}(X) < k_{i_2}(X)) & \text{or} & \\
 & & & & ((k_{i_1}(X) = k_{i_2}(X)) & \text{and} & (k_{i_1}(Y) < k_{i_2}(Y)))
\end{array}$$

The above observations are the essence of the merge step.

Each merge step can be completed in $\Theta(\frac{n \log n}{\mathcal{H}})$ time on an EREW R-PRAM with $\mathcal{H}$ bits of computing hardware (where, $\mathcal{H}$ is $\Omega(\log n)$ and $O(n)$) and $\Theta(\mathcal{H}n^2)$ bits of space. We show in § 5.4 that Step 2 needs $\Theta(\frac{mn}{\mathcal{H}})$ time and $\Theta(\mathcal{H}n^2)$ bits of space on an EREW R-PRAM with $\mathcal{H}$ bits of computing hardware, where $\mathcal{H}$ is $\Omega(\log n)$ and $O(\frac{mn}{\log n \log m})$.

Step 3, the relocation of the keys, can be done in $\Theta(\log n + \frac{mn}{\mathcal{H}})$ time and $\Theta(mn + \mathcal{H}n^2)$ bits of space on an EREW R-PRAM with $\mathcal{H}$ bits of computing hardware, where $\mathcal{H}$ is $\Omega(\log n)$ and $O(mn)$.

On the whole, the proposed sorting algorithm requires $\Theta(\frac{mn}{\mathcal{H}})$ time and $\Theta(mn + \mathcal{H}n^2)$ bits of space on an EREW R-PRAM with $\mathcal{H}$ bits of computing hardware, where $\mathcal{H}$ is $\Omega(\log n)$ and $O(\frac{mn}{\log n \log m})$. The GTP of the above algorithm is therefore $\Theta(mn)$, which is optimal.

5.2 The Leader Finding Problem

In order to make the description of our sorting algorithm easier, we discuss in this section the leader finding problem. This problem is used in our sorting algorithm to avoid concurrent reads in Step 1 and Step 2. The leader finding problem can be described formally as follows:

Let $N(n) = \{0, 1, \ldots, n-1\}$ be a set of n indices and let $\rho : N(n) \longrightarrow N(n)$ be the color function that gives $\rho(i)$, the color (a number from $N(n)$) of the element with index i. This function satisfies the following condition:

$$\forall i_1, i_2 \in N(n), \quad \rho(i_1) = \rho(i_2) \Longrightarrow \forall i_3 \in N(n) \ni i_1 < i_3 < i_2, \quad \rho(i_1) = \rho(i_2) = \rho(i_3) \tag{1}$$

In other words, there is no interleaving of the colors.

For any color $x \in N(n)$ that has at least one element $i_1 \in N(n)$ for which $\rho(i_1) = x$, we define i_1 to be the *leader* of x if and only if $\rho(i_1) = x$ and $\forall i_2 \in N(n) \quad ((\rho(i_2) = x) \Longrightarrow (i_1 \leq i_2))$. If the color x has no index $i \in N(n)$ such that $\rho(i) = x$, then the leader of x is undefined. Thus we can associate a leader with each $i \in N(n)$. It was mentioned earlier that the leader finding problem is used to avoid concurrent reads in our sorting algorithm. We denote by $Info(i)$ the component of the parallel variable $Info$ whose value is to be read by several other processors. More specifically, if $i \in N(n)$ is the leader of $\rho(i)$ then $Info(i)$ is to be read by all the processors associated with elements of color $\rho(i)$.

The solution to the leader finding problem is finding for each index $i \in N(n)$, such that $i' \in N(n)$ is the leader of $\rho(i)$, the information in $Info(i')$. This information is to be stored in the parallel variable $Dst\_Info(i)$. In all instances of the leader finding problem used in the proposed sorting algorithm, $Info(i)$ and $Dst\_Info(i)$ are $\Theta(\log n)$ bits long. In Appendix A, we show that the leader finding problem can be solved in $\Theta(\frac{n \log n}{\mathcal{H}})$ time and $\Theta(\mathcal{H}n^2)$ bits of space on an EREW R-PRAM with $\mathcal{H}$ bits of computing hardware. If an EREW PRAM is used, the space required is $\Theta(n \log n)$ (as we need not count the look-up tables). For both models $\mathcal{H}$ is $\Omega(\log n)$ and $O(n)$. An Example of the leader finding problem is shown in Appendix B.

5.3 An Example

In this section we describe the sorting algorithm with the aid of an example. Consider the following instance of a sorting problem where $n = 8$, $m = 12$, and $k_0 = 12$, $k_1 = 796$, $k_2 = 1018$, $k_3 = 12$, $k_4 = 796$, $k_5 = 12$, $k_6 = 3892$, $k_7 = 3$. The nm-bit input is shown in Table 1.

Step 1: In this step, we first divide the bits in the keys into sections, each containing $\lceil \log n \rceil$ (3 in our example) contiguous bits (see Table 2). This results in $\lceil \frac{m}{\log n} \rceil$ (4 for our example) independent problems of determining the ordered partitions of the above sections. For our example, the 4 ordered partitions turn out to be

$$\mathbf{P}_{\{0,1,2\}} = \{\{k_0, k_3, k_5, k_7\},\ \{k_1, k_2, k_4\},\ \{k_6\}\};$$
$$\mathbf{P}_{\{3,4,5\}} = \{\{k_0, k_3, k_5, k_7\},\ \{k_1, k_4, k_6\},\ \{k_2\}\};$$
$$\mathbf{P}_{\{6,7,8\}} = \{\{k_7\},\ \{k_0, k_3, k_5\},\ \{k_1, k_4\},\ \{k_6\},\ \{k_2\}\};$$
$$\text{and } \mathbf{P}_{\{9,10,11\}} = \{\{k_2\},\ \{k_7\},\ \{k_0, k_1, k_3, k_4, k_5, k_6\}\};$$

Before we proceed to discuss the details of Step 1, a brief overview of the integer sorting algorithm in [11] is necessary. This algorithm is for keys that are $\Theta(\log n)$ bits long and is based on Hagerup's method [8]. The integer sorting algorithm can be described as the following 4-step procedure.

(*i*) For each value $v \in N(n)$, find a list of all the keys with value v in ascending order of their indices. The beginning and end of each list is also known at the end of this step. Thus, for each $v \in N(n)$, there is a list of key indices. Some of these lists may be empty.

(*ii*) Concatenate the lists resulting from (*i*) in ascending order of the value v associated with each list.

(*iii*) Rank the concatenated list.

(*iv*) Relocate the keys according to their ranks.

The above algorithm requires $\Theta(\log n)$ time and $\Theta(n^3)$ bits of space on an EREW R-PRAM with n bits of computing hardware [11]. If an EREW R-PRAM with $\mathcal{H}$ bits of computing hardware (where $\mathcal{H}$ is $\Omega(\log n)$ and $O(n)$) is used, $\Theta(\frac{n \log n}{\mathcal{H}})$ time and $\Theta(\mathcal{H}n^2)$ bits of space is needed. The ranking in step (*iii*) uses the method due to Anderson and Miller [3], which is an EREW PRAM list ranking algorithm that uses $\frac{n}{\log n}$ processors to achieve a time of $\Theta(\log n)$. This algorithm is used at other portions of the proposed sorting algorithm and therefore a few words on its modification for the EREW R-PRAM are due. If an EREW R-PRAM with $\mathcal{H}$ bits of computing hardware (where $\mathcal{H}$ is $\Omega(\log n)$ and $O(n)$) is used, the time needed is $\Theta(\frac{n \log n}{\mathcal{H}})$ and the space needed is $\Theta(\mathcal{H}n^2)$ (the look-up table space for $\frac{\mathcal{H}}{\log n}$ processors, each of size $\log n$ bits, has been counted here).

We will use the steps of the integer sorting algorithm in [11] to determine $\mathbf{P}_J$ the ordered partition based on a $\lceil \log n \rceil$ element set J. We apply this method, in parallel, on the $\lceil \frac{m}{\log n} \rceil$ sections to obtain the ordered partitions based on them.

Consider an ordered partition $\mathbf{P}_J$ with $|J| = \lceil \log n \rceil$, for which we need to determine $Block(i, J)$ and $Order(i, J)$. This is easy if we use the integer sorting algorithm outlined above. Each non-empty list in step (i) of the above algorithm corresponds to a block of the ordered partition. The output of step (ii) is $Order$. To find $Block$ we proceed to step (iv) and relocate the sections after having ranked them. We now apply the leader finding problem algorithm. The relocated indices are used as indices for the leader finding problem, with the value of the section being the color of the associated index. For each $i \in N(n)$, that is the leader of a color (head of a block), $Info(i) = i$. The parallel variable Dst_Info in this case, is the same as $Block$. It is clear that condition (1) of § 5.2 is satisfied. Step 1 needs $\Theta(\frac{mn}{\mathcal{H}})$ time and $\Theta(\mathcal{H}n^2)$ bits of space on an EREW R-PRAM with $\mathcal{H}$ bits of computing hardware, where $\mathcal{H}$ is $\Omega(\log n)$ and $O(\frac{mn}{\log n})$. A detailed analysis appears in § 5.4. In Fig. 1, we illustrate the result of Step 1 of our algorithm for $\mathbf{P}_{\{0,1,2\}}$ of our example.

Step 2: Here we first merge $\mathbf{P}_{\{0,1,2\}}, \mathbf{P}_{\{3,4,5\}}$, to form $\mathbf{P}_{\{0,1,2,3,4,5\}}$ and $\mathbf{P}_{\{6,7,8\}}, \mathbf{P}_{\{9,10,11\}}$ to form $\mathbf{P}_{\{6,7,8,9,10,11\}}$. Both these merges may be performed concurrently. It may be verified that $\mathbf{P}_{\{0,1,2,3,4,5\}} = \{\{k_0, k_3, k_5, k_7\},\ \{k_1, k_4\},\ \{k_2\}, \{k_6\}\}$ and $\mathbf{P}_{\{6,7,8,9,10,11\}} = \{\{k_7\},\ \{k_0, k_3, k_5\},\ \{k_1, k_4\},\ \{k_6\},\ \{k_2\}\}$. In the next and final merge step $\mathbf{P}_{\{0,1,2,3,4,5\}}$ and $\mathbf{P}_{\{6,7,8,9,10,11\}}$ are merged to form $\mathbf{P}_{\{0,1,\ldots,11\}} = \mathbf{P}_{N(m)} = \{\{k_7\},\ \{k_0, k_3, k_5\},\ \{k_1, k_4\}$ $\{k_2\},\ \{k_6\}\}$, the solution to the sorting problem. We discuss below the details of a merge step and use the merge of $\mathbf{P}_{\{0,1,2,3,4,5\}}$ and $\mathbf{P}_{\{6,7,8,9,10,11\}}$ for our illustration.

The input to the merge step is $\mathbf{P}_X = \mathbf{P}_{\{0,1,2,3,4,5\}}$ represented by $Block(i, X)$ and

Order(i, X); and $\mathbf{P}_Y = \mathbf{P}_{\{6,7,8,9,10,11\}}$ represented by *Block*(i, Y) and *Order*(i, Y). (See Figs. 2, 3). The output of the merge step is $\mathbf{P}_{X \cup Y}$ represented by *Block*$(i, X \cup Y)$ and *Order*$(i, X \cup Y)$. The merge step will be done on an EREW R-PRAM with $\mathcal{H}$ bits of computing hardware, where $\mathcal{H}$ is $\Omega(\log n)$ and $O(n)$. Each merge step consists of four phases, which we describe below.

Phase 1: For any ordered partition $\mathbf{P}_J$, we define *Block_Order*(J) as a parallel variable that represents a list of the heads of the blocks of $\mathbf{P}_J$ in the order in which they appear in $\mathbf{P}_J$. In this phase we determine for $\mathbf{P}_X$ and $\mathbf{P}_Y$ the components of *Block_Order*(X) and *Block_Order*(Y) respectively. This can be done as follows. First the list given by *Order*(i, J) $(J \in \{X, Y\})$ is reversed to form the list *Rev_Order*(i, J). This can be done in $\Theta(\frac{n \log n}{\mathcal{H}})$ time and $\Theta(n \log n)$ bits of space. Next, the processor that is associated with k_i checks *Block*(*Rev_Order*$(i, J), J)$ (if *Rev_Order*(i, J) is not NIL). If *Block*$(i, J) \neq$ *Block*(*Rev_Order*$(i, J), J)$ then k_i is a head of a block of $\mathbf{P}_J$ and the head preceding it is *Block*(*Rev_Order*$(i, J), J)$. It is clear that this phase needs $\Theta(\frac{n \log n}{\mathcal{H}})$ time and $\Theta(n \log n)$ space. Fig. 4 shows the output of this phase, for our example.

Phase 2: Here we rank the elements of *Block_Order*(X) and *Block_Order*(Y). We denote these ranks by *Block_no*(i, J) $(J \in \{X, Y\})$. We now need to make *Block_no* known to all the elements of the blocks of $\mathbf{P}_J$. For this purpose we first rank the list given by *Order*(i, J) and temporarily reorder the keys according to this rank. If we consider *Block*(i, J) to be the color of k_i, then broadcasting *Block_no* from the head of a block to all the processors within a block, becomes an instance of the leader finding problem in which *Info* and *Dst_Info* are *Block_no*. Since $\mathbf{P}_X$ and $\mathbf{P}_Y$ are stable ordered partitions, condition (1) of § 5.2 is satisfied. Phase 2 can therefore be done in $\Theta(\frac{n \log n}{\mathcal{H}})$ time and $\Theta(\mathcal{H} n^2)$ space. Fig. 4 illustrates the *Block_no*(i, X) and *Block_no*(i, Y), for our example.

Phase 3: At this point, for each $k_i \in \mathbf{K}$, we have $Block\_no(i, J)$ ($J \in \{X, Y\}$) that have the following properties.
For any $k_{i_1}, k_{i_2} \in \mathbf{K}$,

$$\begin{aligned} k_{i_1}(J) &= k_{i_2}(J) \iff Block\_no(i_1, J) = Block\_no(i_2, J) \\ k_{i_1}(J) &< k_{i_2}(J) \iff Block\_no(i_1, J) < Block\_no(i_2, J) \end{aligned}$$

In other words, for any key k_i, $k_i(X)$ and $k_i(Y)$ (which may be $\omega(\log n)$ bits long) have been converted to $Block\_no(i, X)$ and $Block\_no(i, Y)$, that reflect the order of the elements of $\mathbf{K}_X$ and $\mathbf{K}_Y$. $Block\_no(i, X)$ and $Block\_no(i, Y)$ are each $\log n$ bits long.

Let $\kappa_i(X \cup Y)$ be the $2\log n$-bit number formed by concatenating $Block\_no(i, X)$ and $Block\_no(i, Y)$, in that order. Since the section X is of a higher weight than the section Y. For any $k_{i_1}, k_{i_2} \in \mathbf{K}$,

$$\begin{aligned} k_{i_1}(X \cup Y) &= k_{i_2}(X \cup Y) \iff \kappa_{i_1}(X \cup Y) = \kappa_{i_2}(X \cup Y) \\ k_{i_1}(X \cup Y) &< k_{i_2}(X \cup Y) \iff \kappa_{i_1}(X \cup Y) < \kappa_{i_2}(X \cup Y) \end{aligned}$$

Thus, $\mathbf{K}(X \cup Y)$ can be sorted by sorting $\{\kappa_i(X \cup Y) : i \in X \cup Y\}$. We use the stable integer sorting algorithm in [11] for this purpose. It should be noted that since $\mathbf{P}_X$ and $\mathbf{P}_Y$ are stable, so is $\mathbf{P}_{X \cup Y}$. The order of the elements of $\kappa_i(X \cup Y)$ gives $Order(i, X \cup Y)$. This phase requires $\Theta(\frac{n \log n}{\mathcal{H}})$ time and $\Theta(\mathcal{H}n^2)$ space. Fig. 5 shows the result of this phase for our example.

Phase 4: Here we obtain $Block(i, X \cup Y)$. This can be done by first determining the heads of the blocks of $\mathbf{P}_{X \cup Y}$ as in phase 1. Next we proceed as in Step 1 and obtain $Order(i, X \cup Y)$, by first ranking $Order(i, X \cup Y)$ and then applying the leader finding problem algorithm. Phase 4 needs $\Theta(\frac{n \log n}{\mathcal{H}})$ time and $\Theta(\mathcal{H}n^2)$ space. Fig. 5 shows the result of this phase for our example.

By means of the 4 phases described above, we have been able to obtain $\mathbf{P}_{X \cup Y}$ (represented by $Block(i, X \cup Y)$ and $Order(i, X \cup Y)$) from the ordered partitions $\mathbf{P}_X$ and $\mathbf{P}_Y$. The main result of this section is summarized by the following lemma.

Lemma 1 *: Given two stable ordered partitions,* $\mathbf{P}_X$ *and* $\mathbf{P}_Y$ *of a set of* n *keys such that* $\forall x \in X$, $\forall y \in Y$ $x < y$, *the stable ordered partition* $\mathbf{P}_{X \cup Y}$ *can be obtained on an EREW R-PRAM with* $\mathcal{H}$ *bits of computing hardware (where* $\mathcal{H}$ *is* $\Omega(\log n)$ *and* $O(n)$*) in* $\Theta(\frac{n \log n}{\mathcal{H}})$ *time and* $\Theta(\mathcal{H}n^2)$ *bits of space.*

We show in § 5.4 that Step 2 requires $\Theta(\frac{mn}{\mathcal{H}})$ time and $\Theta(\mathcal{H}n^2)$ bits of space on an EREW R-PRAM with $\mathcal{H}$ bits of computing hardware, where $\mathcal{H}$ is $\Omega(\log n)$ and $O(\frac{mn}{\log n \log m})$.

<u>Step 3:</u> The rank of each key can be obtained from the parallel variable *Order* of the stable ordered partition $\mathbf{P}_{N(m)}$. This requires $\Theta(\frac{n \log n}{\mathcal{H}})$ time and $\Theta(\mathcal{H}n^2)$ space. Once the ranks are obtained, the keys can be relocated in $\Theta(\frac{mn}{\mathcal{H}})$ time and $\Theta(mn)$ bits of space, using $\mathcal{H}$ bits of computing hardware. Therefore the time required for this step on an EREW R-PRAM with $\mathcal{H}$ bits of computing hardware is $\Theta(\log n + \frac{mn}{\mathcal{H}})$ and the space required is $\Theta(mn + \mathcal{H}n^2)$ bits; $\mathcal{H}$ is $\Omega(\log n)$ and $O(mn)$

5.4 Complexity Analysis of the Sorting Algorithm

In this section we will determine the time and space complexities of the three steps in the proposed sorting algorithm. We will use an EREW R-PRAM with $\mathcal{H}$ bits of computing hardware, where $\mathcal{H}$ is $\Omega(\log n)$ and $O(\frac{mn}{\log m \log n})$.

<u>Step 1:</u> As mentioned earlier, in this step we divide the keys into $\lceil \frac{m}{\lceil \log n \rceil} \rceil$ groups each of which are integer sorted and converted to stable ordered partitions. Recall that m is $\omega(\log n)$ and therefore $\frac{m}{\lceil \log n \rceil}$ is $\omega(1)$. As explained in § 5.3, this step also uses the leader finding problem algorithm. We show in Appendix A that the leader finding problem can be solved on an EREW R-PRAM with $\mathcal{H}$ bits of computing hardware (where $\mathcal{H}$ is $\Omega(\log n)$ and $O(n)$) in $\Theta(\frac{n \log n}{\mathcal{H}})$ time and $\Theta(\mathcal{H}n^2)$ space. These figures are the same as those for the integer sorting algorithm (see § 5.3). To analyze the complexity of Step 1 we consider two cases:

Case I: Here $\mathcal{H}$ is $O(m)$. To ensure that $\Omega(\log n)$ bits of computing hardware is available for each group of $\lceil \log n \rceil$ bits of the keys, we will process only $\frac{\mathcal{H}}{\log n}$ groups in parallel. We say that $\frac{\mathcal{H}}{\log n}$ processor columns, each column having $\Theta(\log n)$ bits of computing hardware are used. Therefore, each column sequentially processes $\Theta(\frac{m}{\mathcal{H}})$ groups. This requires $\Theta(\frac{m}{\mathcal{H}}\frac{n \log n}{\log n})$, which is $\Theta(\frac{mn}{\mathcal{H}})$ time. The space needed is $\Theta(\frac{\mathcal{H}}{\log n}n^2 \log n)$ which is $\Theta(\mathcal{H}n^2)$ bits.

Case II: Here $\mathcal{H}$ is $\Omega(m)$. We can now have $\frac{m}{\log n}$ processor columns, each having $\Theta(\frac{\mathcal{H} \log n}{m})$ (which is $O(n)$ as $\mathcal{H}$ is $O(\frac{mn}{\log n \log m})$) bits of computing hardware. Once again the time required is $\Theta(n \log n / \frac{\mathcal{H} \log n}{m})$, which is $\Theta(\frac{mn}{\mathcal{H}})$. The space required is $\Theta((\frac{m}{\log n})(\frac{\mathcal{H} \log n}{m})n^2)$ which is $\Theta(\mathcal{H}n^2)$ bits.

On the whole, the time required for Step 1 is $\Theta(\frac{mn}{\mathcal{H}})$ and the space required is $\Theta(\mathcal{H}n^2)$.

<u>Step 2:</u> In this step we merge the $\lceil \frac{m}{\lceil \log n \rceil} \rceil$ ordered partitions obtained in Step 1 in a binary tree. As shown in § 5.3, each merge requires $\Theta(\frac{n \log n}{\mathcal{H}})$ time and $\Theta(\mathcal{H}n^2)$ bits of space on an EREW R-PRAM with $\mathcal{H}$ bits of computing hardware where $\mathcal{H}$ is $\Omega(\log n)$ and $O(n)$. Before we proceed, we will define an operation called *Max_Order* as follows. For any set A of non-decreasing functions, $Max\_Order(A)$ is $\Theta(f)$ iff $\forall f_1 \in A$, f is $\Omega(f_1)$ and $\exists f_2 \in A \ni f_2$ is $\Theta(f)$. That is, $Max\_Order(A)$ gives the order of the element(s) of A that have the highest complexity. We consider four cases for Step 2:

Case I: Here $\mathcal{H}$ is $O(m)$ and $O(n)$. As in Case I of Step 1, initially there are $\frac{\mathcal{H}}{\log n}$ columns of processors, each having $\Theta(\log n)$ bits of computing hardware. Each column of processors handles $\Theta(\frac{m}{\mathcal{H}})$ merges, in $\Theta(\frac{mn}{\mathcal{H}})$ time. Let this take s_1 merge steps. At merge step $s_1 + i$ $(0 \leq i < \log(\frac{\mathcal{H}}{\log n}))$, there are $\frac{\mathcal{H}}{2^{i+1} \log n}$ merges to be performed. Thus, each merge in merge step $s_1 + i$ is done with $\Theta(2^{i+1} \log n)$ bits of computing hardware. It is easy to see that $2^{i+1} \log n \leq \mathcal{H}$, which is $O(n)$. We also note that the time required for merge step $s_1 + i$ is $\Theta(\frac{n \log n}{2^{i+1} \log n})$, which is $\Theta(\frac{n}{2^{i+1}})$. Therefore the time for Step 2 is

$$\Theta\left(\frac{mn}{\mathcal{H}} + \sum_{i=0}^{\log(\frac{\mathcal{H}}{\log n})-1} \frac{n}{2^{i+1}}\right),$$

which is $\Theta(\frac{mn}{\mathcal{H}} + n)$. Since $\mathcal{H}$ is $O(m)$, n is $O(\frac{mn}{\mathcal{H}})$. Therefore the time for Step 2 is $\Theta(\frac{mn}{\mathcal{H}})$.

The space required for Step 2 is $Max\_Order(\{\frac{\mathcal{H}}{\log n} n^2 \log n,\ \frac{\mathcal{H}}{2^{i+1} \log n}(2^{i+1} \log n) n^2 : 0 \leq i < \log(\frac{\mathcal{H}}{\log n})\})$ which is $\Theta(\mathcal{H}n^2)$.

Case II: Here $\mathcal{H}$ is $\Omega(n)$ and $O(m)$. This case is similar to Case I except that the merge step $s_1 + i$ now requires $\Theta(\lceil \frac{n}{2^{i+1} \log n} \rceil \log n)$ time. Let $2^{i_1} < \frac{n}{\log n} \leq 2^{i_1+1}$. Upto step $s_1 + i_1$ (where s_1 is as defined in Case I) the time will be $\Theta(\frac{n}{2^{i+1}})$. Beyond this point there will be $\Omega(n)$ bits of computing hardware available for each merge, which is more than what can be used. Therefore the time here will be $\Theta(\log n)$. Thus the time for Step 2 is

$$\Theta\left(\frac{mn}{\mathcal{H}} + \sum_{i=0}^{i_1} \frac{n}{2^{i+1}} + \sum_{i=i_1+1}^{\log(\frac{\mathcal{H}}{\log n})-1} \log n\right),$$

which is $O(\frac{mn}{\mathcal{H}} + n + \log(\frac{\mathcal{H}}{\log n}) \log n)$. As in Case I, n is $O(\frac{mn}{\mathcal{H}})$. Since $\mathcal{H}$ is $O(\frac{mn}{\log m \log n})$, $\frac{mn}{\mathcal{H}}$ is $\Omega(\log n \log m)$ and $\log(\frac{\mathcal{H}}{\log n}) \log n$ is $O(\log m \log n)$ (as $\mathcal{H}$ is $O(m)$). Therefore the time required for Step 2 is $\Theta(\frac{mn}{\mathcal{H}})$.

The space required for Step 2 is $Max\_Order(\{\frac{\mathcal{H}}{\log n} n^2 \log n,\ \frac{\mathcal{H}}{2^{i+1} \log n} 2^{i+1} n^2 \log n,\ \frac{\mathcal{H}}{2^{j+1} \log n} n^3 : 0 \leq i \leq i_1 < j < \log(\frac{\mathcal{H}}{\log n})\})$, which is $\Theta(\mathcal{H}n^2)$.

Case III: Here $\mathcal{H}$ is $\Omega(m)$ and $O(n)$. For this case, merge step s $(0 \leq s < \log(\frac{m}{\log n}))$ has $\frac{m}{2^{s+1} \log n}$ merges with $\frac{\mathcal{H}2^{s+1} \log n}{m}$ (which is $O(n)$) bits of computing hardware per merge. Proceeding as in Case I, we get the time for Step 2 to be

$$\Theta\left(\sum_{s=0}^{\log(\frac{m}{\log n})-1} \frac{mn}{\mathcal{H}2^{s+1}}\right),$$

which is $\Theta(\frac{mn}{\mathcal{H}})$. The space required is $Max\_Order(\{\frac{m}{2^{s+1}\log n}\frac{\mathcal{H}2^{s+1}\log n}{m}n^2 : 0 \leq s < \log(\frac{m}{\log n})\})$, which is $\Theta(\mathcal{H}n^2)$.

Case IV: Here $\mathcal{H}$ is $\Omega(m)$ and $\Omega(n)$. As in Case III, merge step s has $\frac{m}{2^{s+1}\log n}$ merges, each done with $\frac{\mathcal{H}2^{s+1}\log n}{m}$ bits of computing hardware. In fact, merge step s requires $\Theta(\lceil\frac{mn}{\mathcal{H}2^{s+1}\log n}\rceil\log n)$ time. Let $2^{s_2} < \frac{mn}{\mathcal{H}\log n} \leq 2^{s_2+1}$. Proceeding as in Case II, the time for Step 2 is

$$\Theta\left(\sum_{s=0}^{s_2}\frac{mn}{\mathcal{H}2^{s+1}} + \sum_{s=s_2+1}^{\log(\frac{mn}{\log n})-1}\log n\right),$$

which is $\Theta(\frac{mn}{\mathcal{H}} + \log(\frac{m}{\log n})\log n)$. As noted earlier, $\frac{mn}{\mathcal{H}}$ is $\Omega(\log n\log m)$ whereas $\log(\frac{m}{\log n})\log n$ is $O(\log n\log m)$. Therefore the time for Step 2 is $\Theta(\frac{mn}{\mathcal{H}})$.

The space needed for Step 2 is $Max\_Order(\{\frac{m}{2^{s+1}\log n}\frac{\mathcal{H}2^{s+1}\log n}{m}n^2, \frac{m}{2^{j+1}\log n}n^3 : 0 \leq s \leq s_2 < j < \log(\frac{m}{\log n})\})$ which is $\Theta(\mathcal{H}n^2)$.

On the whole, the time required for Step 2 is $\Theta(\frac{mn}{\mathcal{H}})$ and the space needed is $\Theta(\mathcal{H}n^2)$ bits.

Step 3: As explained in § 5.3, Step 3 needs $\Theta(\log n + \frac{mn}{\mathcal{H}})$ time, which is $\Theta(\frac{mn}{\mathcal{H}})$ as $\mathcal{H}$ is $O(\frac{mn}{\log n\log m})$. The space needed is $\Theta(mn + \mathcal{H}n^2)$ bits.

The following theorem summarizes the results of this section.

Theorem 1 *Given an EREW R-PRAM with $\mathcal{H}$ bits of computing hardware (where $\mathcal{H}$ is $\Omega(\log n)$ and $O(\frac{mn}{\log n\log m})$), n m-bit unsigned binary numbers (where m is $\omega(\log n)$) can be sorted in $\Theta(\frac{mn}{\mathcal{H}})$ time and $\Theta(mn + \mathcal{H}n^2)$ bits of space. The GTP of the above method is $\Theta(mn)$, which is optimal.*

Corollary 1 *Given and EREW R-PRAM with $\Theta(\frac{mn}{\log n\log m})$ bits of computing hardware, n m-bit unsigned binary numbers (where m is $\omega(\log n)$) can be sorted optimally in $\Theta(\log m\log n)$ time and $\Theta(mn + \frac{mn^3}{\log n\log m})$ bits of space.*

Before we close this section we briefly discuss how a weaker variant of the R-PRAM called the Iteration Sensitive R-PRAM (ISR-PRAM) can also be used for our lexicographic sorting algorithm. As mentioned in § 3, the R-PRAM abstracts the solution to a computational problem from some details of loop management. The ISR-PRAM accounts for these overheads. It has been shown in [12] that a loop whose loop variable goes from 0 to $Y-1$ has an overhead of $O(\log\log Y)$ in the bits of computing hardware, when executed on an ISR-PRAM. There is no overhead in time for the above loop. Since there are a total of $\Theta(\log n \log m)$ iterations in our algorithm When an EREW R-PRAM with $\frac{mn}{\log n \log m}$ bits of computing hardware are used, the degradation in the GTP for an EREW ISR-PRAM is only $O(\log(\log\log n + \log\log m))$. We generalize this in the following theorem.

Theorem 2 *Given an EREW ISR-PRAM with $\mathcal{H}(\log(\log\log n + \log\log m))$ bits of computing hardware (where $\mathcal{H}$ is $\Omega(\log n)$ and $O(\frac{mn}{\log n \log m})$), n m-bit unsigned binary numbers (where m is $\omega(\log n)$) can be sorted in $\Theta(\frac{mn}{\mathcal{H}})$ time and $\Theta(mn + \mathcal{H}n^2)$ bits of space.*

6 Sorting on an EREW PRAM

In this section we show how our sorting algorithm can be extended to the conventional PRAM model. Since our R-PRAM algorithm uses processors of size $\Theta(1)$ bits, we can easily simulate it with an overhead of $\Theta(\log n)$ in the bits of computing hardware on an EREW PRAM with processors of size $\Theta(\log n)$ bits. The resulting algorithm requires $\Theta(\log n \log m)$ time and has a GTP of $\Theta(mn \log n)$ which is as good as the best comparison-based sorting algorithms. It should be mentioned here that this is the first non-comparison based EREW PRAM sorting algorithm to achieve the above speed and GTP. However its space requirement is $\Theta(mn + \frac{mn^2}{\log n \log m})$. This is because the integer sorting algorithm in [11], when converted to a PRAM algorithm, requires $\Theta(n^2)$ bits of space, even when the look-up tables are not counted. In the above

PRAM sorting algorithm we have assumed that $\log\log m$ is $O(\log n)$. This is because our algorithm involves an iteration of size $\Theta(\log m)$. In order to manage the iteration variable for this iteration one needs the processors to be of size $\Omega(\log\log m)$ bits. The above assumption amounts to assuming that m is $O(2^{n^{\Theta(1)}})$, which is not a serious restriction, for if $\log m$ is $\omega(n^{\Theta(1)})$ the time required for the best algorithm would itself be exponential and a sequential algorithm would not be much slower.

We now show how an optimal integer sorting algorithm can be used to get an optimal sorting algorithm for unsigned binary numbers of length m (where $\log\log m$ is $O(\log n)$) that requires $\Theta(mn)$ bits of space. Suppose there is an EREW PRAM integer sorting algorithm that sorts n $\Theta(\log n)$-bit unsigned binary numbers in $\Theta(\log n)$ time with $\frac{n}{\log n}$ processors and $\Theta(n\log n)$ bits of space. Such an algorithm is theoretically possible. Before we proceed, we note that both list ranking and the leader finding problem can be solved in $\Theta(\log n)$ time on an EREW PRAM with $\frac{n}{\log n}$ processors and $\Theta(n\log n)$ bits of space. We also note that each merge step can be done in $\Theta(\log n)$ time on an EREW PRAM with $\frac{n}{\log n}$ processors and $\Theta(n\log n)$ bits of space (assuming that the above integer sorting algorithm exists). We now show that our lexicographic sorting algorithm can be used to sort n m-bit unsigned binary numbers (where m is $\omega(\log n)$ and $\log\log m$ is $O(\log n)$) in $\Theta(\log n\log m)$ time on an EREW PRAM with $\frac{mn}{\log^2 n\log m}$ processors and $\Theta(mn)$ bits of space.

In Step 1, each of the $\Theta(\frac{m}{\log n})$ groups is assigned $\frac{n}{\log n\log m}$ processors. Thus Step 1 requires $\Theta(\log n\log m)$ time and $\Theta(\frac{m}{\log n}n\log n)$ which is $\Theta(mn)$ bits of space. It is also clear that Step 3 can be done in $\Theta(\log n\log m)$ time and $\Theta(mn)$ bits of space.

We now consider Step 2. As discussed in § 5.4, step s ($0 \le s < \log(\frac{m}{\log n})$) has $\frac{m}{2^{s+1}\log n}$ merges. Therefore each of the above merges can use $\frac{mn}{\log^2 n\log m}\frac{2^{s+1}\log n}{m} = \frac{n2^{s+1}}{\log n\log m}$ processors. Let $\frac{n2^{s_1+1}}{\log n\log m} < \frac{n}{\log n} \le \frac{ns^{s_1+2}}{\log n\log m}$. Therefore, till step s_1 the time needed for a merge is $\Theta(\log n(\frac{n}{\log n})(\frac{\log n\log m}{n2^{s+1}}))$, which is $\Theta(\frac{\log n\log m}{2^{s+1}})$. Beyond step s_1 there are more processors per merge than can be used. The time for this case is therefore $\Theta(\log n)$. The time for Step 2 is therefore

$$\Theta\left(\sum_{s=0}^{s_1}\frac{\log n\log m}{2^{s+1}}+\sum_{s=s_1}^{\log(\frac{m}{\log n})-1}\log n\right),$$

which is $\Theta(\log n\log m)$. The space required is $Max\_Order(\{\frac{m}{2^{s+1}\log n}n\log n : 0\le s < \log(\frac{m}{\log n})\}$, which is $\Theta(mn)$.

Theorem 3 *If n $\Theta(\log n)$-bit unsigned binary numbers can be stably sorted in $\Theta(\log n)$ time on an EREW PRAM with $\frac{n}{\log n}$ processors and $\Theta(n\log n)$ bits of space, then n m-bit unsigned binary numbers (where m is $\omega(\log n)$ and $\log\log m$ is $O(\log n)$) can be sorted in $\Theta(\log n\log m)$ time on an EREW PRAM with $\frac{mn}{\log^2 n\log m}$ processors and $\Theta(mn)$ bits of space.*

Theorem 3 has very important implications. If the integer sorting algorithm referred to in the above theorem exists then there is a more efficient and as fast a method for sorting n m-bit unsigned binary numbers than the best conventional comparison based algorithms. However, we have conjectured in [11] that such an integer sorting algorithm does not exist. Nevertheless, this does not preclude the possibility of an optimal lexicographic sorting algorithm.

7 Concluding Remarks

We have shown in this report that n m-bit unsigned binary numbers can be sorted lexicographically in $\Theta(\log n\log m)$ time on an EREW R-PRAM with $\frac{mn}{\log n\log m}$ bits of computing hardware This algorithm is not only as fast (asymptotically) as the best conventional comparison-based sorting algorithms, but also has the lowest possible order of GTP. If a weaker variant of the R-PRAM called the ISR-PRAM is used, the degradation in the efficiency (GTP) is very small (a factor of $\Theta(\log(\log\log n+\log\log m))$). The speed of the algorithm is unaffected.

We have also shown how the proposed lexicographic sorting algorithm could be extended to the conventional PRAM model. An important result here is that if integer sorting can be solved optimally on an EREW PRAM then so can keys of unrestricted length.

Acknowledgment

The authors would like to thank Elaine Weinman and Margaret Brewer for their invaluable help in the preparation of this manuscript. Thanks are also due to Prof. Sanjay Ranka for his 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.

[2] M. Ajtai, J. Kómlos and E. Szemerédi, "An $O(n \log n)$ Sorting Network", in Proc. *15th 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", in Proc. *3rd 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. of Comput.*, Vol. 16, No. 3, June 1987, pp. 458-464.

[5] R. P. Brent, "The Parallel Evaluation of General Arithmetic Expressions", *JACM*, Vol. 21, 1974, pp. 201-208.

[6] R. Cole, "Parallel Merge Sort", *SIAM J. Comput.*, Vol. 17, No. 4, Aug. 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", *Info. and Comput.*, 75, 1987, pp. 39-51.

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

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

[11] R. Vaidyanathan, C. R. P. Hartmann and P. K. Varshney, "Optimal Parallel Solutions to the Neighbor Localization Problem and Integer Sorting: A Fine-Grained Approach", Technical Report CIS 89–11, School of Computer and Information Science, Syracuse University, Syracuse, NY 13244–4100, Oct. 1989.

[12] R. Vaidyanathan, C. R. P. Hartmann and P. K. Varshney, "The Effect of Processor and Memory Granularity on the Complexity of Parallel Algorithms", in preparation.

A Solution to the Leader Finding Problem

Here we present a solution to the leader finding problem (described in § 5.2) that requires $\Theta(\frac{n \log n}{\mathcal{H}})$ time and $\Theta(\mathcal{H}n^2)$ bits of space on an EREW model (R-PRAM or PRAM) with $\mathcal{H}$ bits of computing hardware, where $\mathcal{H}$ is $\Omega(\log n)$ and $O(n)$. This solution is used to avoid concurrent reads in Step 1 and phases 2 and 4 of the merge steps in Step 2 of our sorting algorithm, described in § 5.1. We also illustrate our solution to the leader finding problem with an example in Appendix B.

The solution to the leader finding problem given in this appendix, uses an EREW model with $\lceil \frac{n}{\log n} \rceil$ processors each of size $\lceil \log n \rceil$ bits. The solution can be scaled for a model with $\mathcal{H}$ bits of computing hardware. We denote the processors by c_i $(0 \leq i < \lceil \frac{n}{\log n} \rceil)$. The n elements are divided as equally as possible among the processors, so that each processor has at most $\lceil \log n \rceil$ elements. The elements assigned to processor c_i are collectively called group i. For convenience, we will assume $\log n$ and $\frac{n}{\log n}$ to be integers. Before we proceed, a few definitions are necessary.

Definitions: For any color $x \in N(n)$, the element $i_1 \in N(n)$ is called the *trailer* of x iff $\rho(i_1) = x$ and $\forall i_2 \in N(n) \;\; \rho(i_2) = x \Longrightarrow i_2 \leq i_1$. Recall that $\rho(i)$ is the color of i. The longest sequence of contiguous indices in a group that have the same color $x \in N(n)$ is called a *run* of x. For a given run $\mathcal{R}_x$ of x, if the group containing it has the leader (defined in § 5.2) of x then $\mathcal{R}_x$ is called a *known run* of x. If this run also has the trailer of x then it is called a *known complete run* of x or simply a *KC run* of x. If a known run of x does not have the trailer of x within the group then it is called a *known incomplete run* of x or simply a *KI run* of x. If $\mathcal{R}_x$ does not have the leader of x in the group then it is called an *unknown run* of x or simply a *U run* of x.

Observation: A given group can have at most one U run and at most one KI run. It may have more than one KC run.

Our leader finding problem algorithm has three steps, each of which require $\Theta(\log n)$ time. In other words the leader finding problem can be solved on an EREW model with $\mathcal{H}$ bits of computing hardware (where $\mathcal{H}$ is $\Omega(\log n)$ and $O(n)$) in $\Theta(\frac{n \log n}{\mathcal{H}})$ time. The space required will be shown to be $\Theta(\mathcal{H}n^2)$ for the EREW R-PRAM and $\Theta(n \log n)$ for the EREW PRAM. We note here that barring the parallel variables $Info$, Dst_Info and $Leader$, that are defined with respect to the set $N(n)$, all parallel variables used in the the leader finding problem algorithm are defined with respect to the set of processor indices.

In Step 1, each processor c_i scans group i and determines the KI, KC and U runs (if they exist). As observed earlier a group may have at most one U run and one KI run. If they exist, the parallel variables $U\_Flag(i)$ and $KI\_Flag(i)$ are set. In addition, for each element ℓ of group i, that belongs to a KI run or a KC run, $Dst\_Info(\ell)$ is set to the value of $Info(\ell')$, where ℓ' is the leader of $\rho(\ell)$.

In Step 2, the processors c_i that have a KI run with leader ℓ, send the value of $Info(\ell)$ (directly or via other processors) to the processors $c_{i'}$ that have a U run of the same color. This information is saved in the parallel variable $Cur\_Info(i')$.

In Step 3 the processors c_i that have a U run, set $Dst\_Info(\ell)$ to the value in $Cur\_Info(i)$ (obtained in Step 2), for each element ℓ that belongs to the above U run.

We provide below a pseudo code for the above three steps. Comments are provided wherever possible. In the following pseudo code for Step 1, Step 2 and Step 3, the actual code is in boldface, whereas the comments are in plain text. An example illustrating the pseudo code appears in Appendix B.

Procedure Step_1

```
/* Executed in parallel by processors $c_i$  $(0 \le i < \frac{n}{\log n})$ */
$First(i) \longleftarrow 1$   /* Flags the first iteration */
$Cur\_K(i) \longleftarrow 0$  /* set to 1 iff current element $i$ belongs to a known run */
for $j \longleftarrow 0$ to $(\log n)-1$ do
   $\ell(i) \longleftarrow i \log n + j$  /* index of the current element */
   $Leader(\ell(i)) \longleftarrow (\ell(i) = 0)$ or $(\rho(\ell(i)) \neq \rho(\ell(i)-1))$
      /* set to 1 iff element $\ell(i)$ is the leader of $\rho(\ell(i))$ */
   if $First(i) = 1$ then
      if $Leader(\ell(i)) = 1$ then
         $Cur\_Info(i) \longleftarrow Info(\ell(i))$
            /* $Cur\_Info$ contains the information to be written on $Dst\_Info$ of the
              currently processed known run */
         $Dst\_Info(\ell(i)) \longleftarrow Cur\_Info(i)$
         $Cur\_K(i) \longleftarrow 1$
      end
      $First(i) \longleftarrow 0$
   else
      if $Leader(\ell(i)) = 1$ then
         $Cur\_Info(i) \longleftarrow Info(\ell(i))$
         $Dst\_Info(\ell(i)) \longleftarrow Cur\_Info(i)$
         $Cur\_K(i) \longleftarrow 1$
      else
         if $Cur\_K(i) = 1$ then
            $Dst\_Info(\ell(i)) \longleftarrow Cur\_Info(i)$
         end
      end
   end
```

```
end
/* We now set $KI\_Flag(i)$ and $U\_Flag(i)$ that indicate whether group $i$ has a
    KI run or a U run in it */
$\ell_1(i) \longleftarrow (i+1)\log n - 1$ /* the last element of group $i$ */
$KI\_Flag(i) \longleftarrow (\ell_1(i) < n-1)$ and $(Cur\_K(i) = 1)$
            and $(\rho(\ell_1(i)) = \rho(\ell_1(i)+1))$
$\ell_2(i) \longleftarrow i \log n$ /* the first element of group $i$ */
$U\_Flag(i) \longleftarrow (\ell_2(i) \neq 0)$ and $\rho(\ell_2(i)) = \rho(\ell_2(i)-1))$
/* End Step 1 */
```

Procedure Step_2

```
/* Executed in parallel by processors $c_i$ $(0 \leq i < \frac{n}{\log n})$ */
if $i = \frac{n}{\log n} - 1$ then
    $Link(i) \longleftarrow$ NIL
else
    $Link(i) \longleftarrow i+1$
end
/* The above lines form a list of the processors in the order of their indices. The
pointers $Link(i)$ will be used subsequently to communicate information to processors
whose groups have a U run */
for $j \longleftarrow 0$ to $\lceil \log(\frac{n}{\log n}) \rceil - 1$ do
    if $Link(i) \neq$ NIL then
        if $KI\_Flag(i) = 1$ and $U\_Flag(Link(i)) = 1$ then
            $\ell_1(i) \longleftarrow (i+1)\log n - 1$ /* the last element of group $i$ */
            $\ell_2(i) \longleftarrow Link(i) \log n$ /* the first element of group $Link(i)$ */
            if $\rho(\ell_1(i)) = \rho(\ell_2(i))$ then
```

```
          /* Group $Link(i)$ has a U run of color $\rho(\ell_1(i))$ */
          $\boldsymbol{Cur\_Info}(\boldsymbol{Link}(\boldsymbol{i})) \longleftarrow \boldsymbol{Dst\_Info}(\boldsymbol{\ell}_1(\boldsymbol{i}))$
          $\boldsymbol{Link}(\boldsymbol{i}) \longleftarrow \boldsymbol{Link}(\boldsymbol{Link}(\boldsymbol{i}))$
        else
          $\boldsymbol{Link}(\boldsymbol{i}) \longleftarrow$ NIL
          /* Note: The *else* part of an *if-then-else* statement is executed only after
            the *if* part */
        end
      else
        $\boldsymbol{Link}(\boldsymbol{i}) \longleftarrow$ NIL
      end
    end
end
/* End Step 2 */
```

We note here that during any iteration j $(0 \leq j < \lceil \log(\frac{n}{\log n}) \rceil)$ and for any two processors c_i and $c_{i'}$ $(0 \leq i, i' < \frac{n}{\log n})$ $Link(i) = Link(i') \Longrightarrow Link(i) = Link(i') =$ NIL.

This ensures that all reads and writes during iteration j are exclusive.

```
Procedure Step_3
/* Executed in parallel by all processors $c_i$  $(0 \leq i < \frac{n}{\log n})$ */
$Active(i) \longleftarrow U\_Flag(i)$
    /* $Active(i)$ is set to 1 till $c_i$ has written the value of $Cur\_Info(i)$ on
      $Dst\_Info(\ell)$, for all elements $\ell$ in the U run in group $i$ */
for $j \longleftarrow 0$ to $\log n - 1$ do
   if $Active(i) = 1$ then
   $\ell(i) \longleftarrow i \log n + j$  /* currently processed element */
      if $(\ell(i) = i \log n)$ or $(\rho(\ell(i)) = \rho(\ell(i) - 1))$ then
        /* $c_i$ is still within the U run of group $i$ */
        $Dst\_Info(\ell(i)) \longleftarrow Cur\_Info(i)$
      end
        $Active(i) \longleftarrow 0$
      end
   end
end
/* End Step 3 */
```

Before we close this appendix we would like to point out that the memory requirement of our leader finding problem algorithm is $\Theta(n \log n)$ bits under conventional assumptions, where the look-up table space has been ignored. However, when a fine-grained decomposition is considered, the memory requirement is $\Theta(n^3)$, for an EREW R-PRAM with $\frac{n}{\log n}$ processors, each of which use a look-up table of size $\Theta(n^2 \log n)$ bits for $\log n$-bit addition. In general, the memory needed for an EREW R-PRAM with $\mathcal{H}$ bits of computing hardware (where $\mathcal{H}$ is $\Omega(\log n)$ and $O(n)$) is $\Theta(\mathcal{H} n^2)$.

B An Illustration of the Leader Finding Problem Algorithm

We illustrate in this appendix the leader finding problem algorithm with an example of $n = 64$ elements. We use an EREW PRAM with 8 processors, each of size 8 bits. Even though $\log 64 = 6$, we use 8 processors, only for ease of illustration. Therefore, the quantity $\log n$ in the pseudo code of Appendix A should be treated as 8 for this example. Also, we denote by τ_ℓ the value of $Info(\ell)$, where ℓ is the index of the leader of color $\rho(\ell)$.

Table 3 shows the input to our example. Tables 4 – 12 show the contents of the relevant parallel variables at the end of each of the 8 iterations of Step 1. The result of Step 1 are illustrated in Tables 13 and 14. Tables 15 – 18 illustrate the contents of the relevant parallel variables at the end of the 3 iterations of Step 2. Step 3 is illustrated in Tables 19 – 27. Table 28 shows the final result of the example. In all of the above tables an entry marked "–" denotes a *don't care* value.

i	k_i	Binary representation bit no.											
		0	1	2	3	4	5	6	7	8	9	10	11
0	12	0	0	0	0	0	0	0	0	1	1	0	0
1	796	0	0	1	1	0	0	0	1	1	1	0	0
2	1018	0	0	1	1	1	1	1	1	1	0	1	0
3	12	0	0	0	0	0	0	0	0	1	1	0	0
4	796	0	0	1	1	0	0	0	1	1	1	0	0
5	12	0	0	0	0	0	0	0	0	1	1	0	0
6	3892	1	1	1	1	0	0	1	1	0	1	0	0
7	3	0	0	0	0	0	0	0	0	0	0	1	1

Table 1: The input bits for the example of lexicographic sorting

index i	Section $\{0,1,2\}$		Section $\{3,4,5\}$		Section $\{6,7,8\}$		Section $\{9,10,11\}$	
	bits	$k_i(\{0,1,2\})$	bits	$k_i(\{3,4,5\})$	bits	$k_i(\{6,7,8\})$	bits	$k_i(\{9,0,11\})$
0	000	0	000	0	001	1	100	4
1	001	1	100	4	011	3	100	4
2	001	1	111	7	111	7	010	2
3	000	0	000	0	001	1	100	4
4	001	1	100	4	011	3	100	4
5	000	0	000	0	001	1	100	4
6	111	7	100	4	110	6	100	4
7	000	0	000	0	000	0	011	3

Table 2: Input to Step 1 of the lexicographic sorting Algorithm

i	$\rho(i)$	$Info(i)$
0	0	τ_0
1–7	0	–
8	1	τ_8
9–26	1	–
27	2	τ_{27}
28–31	2	–
32	3	τ_{32}
33	4	τ_{33}
34–44	4	–
45	5	τ_{45}
46	5	–
47	6	τ_{47}
48	7	τ_{48}
49–63	7	–

Table 3: The input to the leader finding problem example

Proc. index i	$First(i)$	$Cur\_K(i)$
0	1	0
1	1	0
2	1	0
3	1	0
4	1	0
5	1	0
6	1	0
7	1	0

Table 4: Leader finding problem algorithm Step 1; Initialization

i	$First(i)$	$Cur\_K(i)$	$\ell(i)$	$Leader(i)$	$Cur\_Info(i)$	$Dst\_Info(\ell(i))$
0	0	1	0	1	τ_0	τ_0
1	0	1	8	1	τ_8	τ_8
2	0	0	16	0	–	–
3	0	0	24	0	–	–
4	0	1	32	1	τ_{32}	τ_{32}
5	0	0	40	0	–	–
6	0	1	48	1	τ_{48}	τ_{48}
7	0	0	56	0	–	–

Table 5: Leader finding problem algorithm Step 1; Iteration 0

i	$First(i)$	$Cur\_K(i)$	$\ell(i)$	$Leader(i)$	$Cur\_Info(i)$	$Dst\_Info(\ell(i))$
0	0	1	1	0	τ_0	τ_0
1	0	1	9	0	τ_8	τ_8
2	0	0	17	0	–	–
3	0	0	25	0	–	–
4	0	1	33	1	τ_{33}	τ_{33}
5	0	0	41	0	–	–
6	0	1	49	0	τ_{48}	τ_{48}
7	0	0	57	0	–	–

Table 6: Leader finding problem algorithm Step 1; Iteration 1

i	$First(i)$	$Cur\_K(i)$	$\ell(i)$	$Leader(i)$	$Cur\_Info(i)$	$Dst\_Info(\ell(i))$
0	0	1	2	0	τ_0	τ_0
1	0	1	10	0	τ_8	τ_8
2	0	0	18	0	–	–
3	0	0	26	0	–	–
4	0	1	34	0	τ_{33}	τ_{33}
5	0	0	42	0	–	–
6	0	1	50	0	τ_{48}	τ_{48}
7	0	0	58	0	–	–

Table 7: Leader finding problem algorithm Step 1; Iteration 2

i	$First(i)$	$Cur\_K(i)$	$\ell(i)$	$Leader(i)$	$Cur\_Info(i)$	$Dst\_Info(\ell(i))$
0	0	1	3	0	τ_0	τ_0
1	0	1	11	0	τ_8	τ_8
2	0	0	19	0	–	–
3	0	1	27	1	τ_{27}	τ_{27}
4	0	1	35	0	τ_{33}	τ_{33}
5	0	0	43	0	–	–
6	0	1	51	0	τ_{48}	τ_{48}
7	0	0	59	0	–	–

Table 8: Leader finding problem algorithm Step 1; Iteration 3

i	$First(i)$	$Cur\_K(i)$	$\ell(i)$	$Leader(i)$	$Cur\_Info(i)$	$Dst\_Info(\ell(i))$
0	0	1	4	0	τ_0	τ_0
1	0	1	12	0	τ_8	τ_8
2	0	0	20	0	–	–
3	0	1	28	0	τ_{27}	τ_{27}
4	0	1	36	0	τ_{33}	τ_{33}
5	0	0	44	0	–	–
6	0	1	52	0	τ_{48}	τ_{48}
7	0	0	60	0	–	–

Table 9: Leader finding problem algorithm Step 1; Iteration 4

i	$First(i)$	$Cur\_K(i)$	$\ell(i)$	$Leader(i)$	$Cur\_Info(i)$	$Dst\_Info(\ell(i))$
0	0	1	5	0	τ_0	τ_0
1	0	1	13	0	τ_8	τ_8
2	0	0	21	0	–	–
3	0	1	29	0	τ_{27}	τ_{27}
4	0	1	37	0	τ_{33}	τ_{33}
5	0	1	45	1	τ_{45}	τ_{45}
6	0	1	53	0	τ_{48}	τ_{48}
7	0	0	61	0	–	–

Table 10: Leader finding problem algorithm Step 1; Iteration 5

i	$First(i)$	$Cur\_K(i)$	$\ell(i)$	$Leader(i)$	$Cur\_Info(i)$	$Dst\_Info(\ell(i))$
0	0	1	6	0	τ_0	τ_0
1	0	1	14	0	τ_8	τ_8
2	0	0	22	0	–	–
3	0	1	30	0	τ_{27}	τ_{27}
4	0	1	38	0	τ_{33}	τ_{33}
5	0	1	46	0	τ_{45}	τ_{45}
6	0	1	54	0	τ_{48}	τ_{48}
7	0	0	62	0	–	–

Table 11: Leader finding problem algorithm Step 1; Iteration 6

i	$First(i)$	$Cur\_K(i)$	$\ell(i)$	$Leader(i)$	$Cur\_Info(i)$	$Dst\_Info(\ell(i))$
0	0	1	7	0	τ_0	τ_0
1	0	1	15	0	τ_8	τ_8
2	0	0	23	0	–	–
3	0	1	31	0	τ_{27}	τ_{27}
4	0	1	39	0	τ_{33}	τ_{33}
5	0	1	47	1	τ_{47}	τ_{47}
6	0	1	55	0	τ_{48}	τ_{48}
7	0	0	63	0	–	–

Table 12: Leader finding problem algorithm Step 1; Iteration 7

i	$\ell_1(i)$	$\rho(\ell_1(i))$	$\rho(\ell_1(i)+1)$	$\ell_2(i)$	$\rho(\ell_2(i))$	$\rho(\ell_2(i)-1)$	$Cur\_K(i)$	$KI\_Flag(i)$	$U\_Flag(i)$
0	7	0	1	0	0	–	1	0	0
1	15	1	1	8	1	0	1	1	0
2	23	1	1	16	1	1	0	0	1
3	31	2	3	24	1	1	1	0	1
4	39	4	4	32	3	2	1	1	0
5	47	6	7	40	4	4	1	0	1
6	55	7	7	48	7	6	1	1	0
7	63	7	–	56	7	7	0	0	1

Table 13: Leader finding problem algorithm Step 1; $KI\_Flag(i)$ and $U\_Flag(i)$

Group i	Element j of group i							
	0	1	2	3	4	5	6	7
0	τ_0	τ_0	τ_0	τ_0	τ_0	τ_0	τ_0	τ_0
1	τ_8	τ_8	τ_8	τ_8	τ_8	τ_8	τ_8	τ_8
2	–	–	–	–	–	–	–	–
3	–	–	–	τ_{27}	τ_{27}	τ_{27}	τ_{27}	τ_{27}
4	τ_{32}	τ_{33}	τ_{33}	τ_{33}	τ_{33}	τ_{33}	τ_{33}	τ_{33}
5	–	–	–	–	–	τ_{45}	τ_{45}	τ_{47}
6	τ_{48}	τ_{48}	τ_{48}	τ_{48}	τ_{48}	τ_{48}	τ_{48}	τ_{48}
7	–	–	–	–	–	–	–	–

Table 14: $Dst\_Info(i \log n + j)$ after Step 1 of the leader finding problem algorithm

i	$KI\_Flag(i)$	$U\_Flag(i)$	$Link(i)$
0	0	0	1
1	1	0	2
2	0	1	3
3	0	1	4
4	1	0	5
5	0	1	6
6	1	0	7
7	0	1	NIL

Table 15: Leader finding problem algorithm Step 2; Initialization

i	$\ell_1(i)$	$\ell_2(i)$	$\rho(\ell_1(i))$	$\rho(\ell_2(i))$	$U\_Flag(Link(i))$	$Dst\_Info(\ell_1(i))$	$Cur\_Info(i)$	$Link(i)$
0	–	–	–	–	–	–	–	NIL
1	15	16	1	1	1	τ_8	–	3
2	–	–	–	–	–	–	τ_8	NIL
3	–	–	–	–	–	–	–	NIL
4	39	40	4	4	1	τ_{33}	–	6
5	–	–	–	–	–	–	τ_{33}	NIL
6	55	56	7	7	1	τ_{48}	–	NIL
7	–	–	–	–	–	–	τ_{48}	NIL

Table 16: Leader finding problem algorithm Step 2; Iteration 0

i	$\ell_1(i)$	$\ell_2(i)$	$\rho(\ell_1(i))$	$\rho(\ell_2(i))$	$U\_Flag(Link(i))$	$Dst\_Info(\ell_1(i))$	$Cur\_Info(i)$	$Link(i)$
0	–	–	–	–	–	–	–	NIL
1	15	24	1	1	1	τ_8	–	NIL
2	–	–	–	–	–	–	τ_8	NIL
3	–	–	–	–	–	–	τ_8	NIL
4	39	48	4	7	0	–	–	NIL
5	–	–	–	–	–	–	τ_{33}	NIL
6	–	–	–	–	–	–	–	NIL
7	–	–	–	–	–	–	τ_{48}	NIL

Table 17: Leader finding problem algorithm Step 2; Iteration 1

i	$\ell_1(i)$	$\ell_2(i)$	$\rho(\ell_1(i))$	$\rho(\ell_2(i))$	$U\_Flag(Link(i))$	$Dst\_Info(\ell_1(i))$	$Cur\_Info(i)$	$Link(i)$
0	–	–	–	–	–	–	–	NIL
1	–	–	–	–	–	–	–	NIL
2	–	–	–	–	–	–	τ_8	NIL
3	–	–	–	–	–	–	τ_8	NIL
4	–	–	–	–	–	–	–	NIL
5	–	–	–	–	–	–	τ_{33}	NIL
6	–	–	–	–	–	–	–	NIL
7	–	–	–	–	–	–	τ_{48}	NIL

Table 18: Leader finding problem algorithm Step 2; Iteration 2

i	$U\_Flag(i)$	$Active(i)$
0	0	0
1	0	0
2	1	1
3	1	1
4	0	0
5	1	1
6	0	0
7	1	1

Table 19: Leader finding problem algorithm Step 3; Initialization

i	$\ell(i)$	$i \log n$	$\rho(\ell(i))$	$\rho(\ell(i)-1)$	$Cur\_Info(i)$	$Dst\_Info(\ell(i))$	$Active(i)$
0	–	–	–	–	–	–	0
1	–	–	–	–	–	–	0
2	16	16	–	–	τ_8	τ_8	1
3	24	24	–	–	τ_8	τ_8	1
4	–	–	–	–	–	–	0
5	40	40	–	–	τ_{33}	τ_{33}	1
6	–	–	–	–	–	–	0
7	56	56	–	–	τ_{48}	τ_{48}	1

Table 20: Leader finding problem algorithm Step 3; Iteration 0

i	$\ell(i)$	$i \log n$	$\rho(\ell(i))$	$\rho(\ell(i)-1)$	$Cur\_Info(i)$	$Dst\_Info(\ell(i))$	$Active(i)$
0	–	–	–	–	–	–	0
1	–	–	–	–	–	–	0
2	17	16	1	1	τ_8	τ_8	1
3	25	24	1	1	τ_8	τ_8	1
4	–	–	–	–	–	–	0
5	41	40	4	4	τ_{33}	τ_{33}	1
6	–	–	–	–	–	–	0
7	57	56	7	7	τ_{48}	τ_{48}	1

Table 21: Leader finding problem algorithm Step 3; Iteration 1

i	$\ell(i)$	$i \log n$	$\rho(\ell(i))$	$\rho(\ell(i)-1)$	$Cur\_Info(i)$	$Dst\_Info(\ell(i))$	$Active(i)$
0	–	–	–	–	–	–	0
1	–	–	–	–	–	–	0
2	18	16	1	1	τ_8	τ_8	1
3	26	24	1	1	τ_8	τ_8	1
4	–	–	–	–	–	–	0
5	42	40	4	4	τ_{33}	τ_{33}	1
6	–	–	–	–	–	–	0
7	58	56	7	7	τ_{48}	τ_{48}	1

Table 22: Leader finding problem algorithm Step 3; Iteration 2

i	$\ell(i)$	$i \log n$	$\rho(\ell(i))$	$\rho(\ell(i)-1)$	$Cur\_Info(i)$	$Dst\_Info(\ell(i))$	$Active(i)$
0	–	–	–	–	–	–	0
1	–	–	–	–	–	–	0
2	19	16	1	1	τ_8	τ_8	1
3	27	24	2	1	–	–	0
4	–	–	–	–	–	–	0
5	43	40	4	4	τ_{33}	τ_{33}	1
6	–	–	–	–	–	–	0
7	59	56	7	7	τ_{48}	τ_{48}	1

Table 23: Leader finding problem algorithm Step 3; Iteration 3

i	$\ell(i)$	$i \log n$	$\rho(\ell(i))$	$\rho(\ell(i)-1)$	$Cur\_Info(i)$	$Dst\_Info(\ell(i))$	$Active(i)$
0	–	–	–	–	–	–	0
1	–	–	–	–	–	–	0
2	20	16	1	1	τ_8	τ_8	1
3	–	–	–	–	–	–	0
4	–	–	–	–	–	–	0
5	44	40	4	4	τ_{33}	τ_{33}	1
6	–	–	–	–	–	–	0
7	60	56	7	7	τ_{48}	τ_{48}	1

Table 24: Leader finding problem algorithm Step 3; Iteration 4

i	$\ell(i)$	$i \log n$	$\rho(\ell(i))$	$\rho(\ell(i)-1)$	$Cur\_Info(i)$	$Dst\_Info(\ell(i))$	$Active(i)$
0	–	–	–	–	–	–	0
1	–	–	–	–	–	–	0
2	21	16	1	1	τ_8	τ_8	1
3	–	–	–	–	–	–	0
4	–	–	–	–	–	–	0
5	45	40	5	4	–	–	0
6	–	–	–	–	–	–	0
7	61	56	7	7	τ_{48}	τ_{48}	1

Table 25: Leader finding problem algorithm Step 3; Iteration 5

i	$\ell(i)$	$i \log n$	$\rho(\ell(i))$	$\rho(\ell(i)-1)$	$Cur\_Info(i)$	$Dst\_Info(\ell(i))$	$Active(i)$
0	–	–	–	–	–	–	0
1	–	–	–	–	–	–	0
2	22	16	1	1	τ_8	τ_8	1
3	–	–	–	–	–	–	0
4	–	–	–	–	–	–	0
5	–	–	–	–	–	–	0
6	–	–	–	–	–	–	0
7	62	56	7	7	τ_{48}	τ_{48}	1

Table 26: Leader finding problem algorithm Step 3; Iteration 6

i	$\ell(i)$	$i \log n$	$\rho(\ell(i))$	$\rho(\ell(i)-1)$	$Cur\_Info(i)$	$Dst\_Info(\ell(i))$	$Active(i)$
0	–	–	–	–	–	–	0
1	–	–	–	–	–	–	0
2	23	16	1	1	τ_8	τ_8	1
3	–	–	–	–	–	–	0
4	–	–	–	–	–	–	0
5	–	–	–	–	–	–	0
6	–	–	–	–	–	–	0
7	63	56	7	7	τ_{48}	τ_{48}	1

Table 27: Leader finding problem algorithm Step 3; Iteration 7

Group i	Element j of group i							
	0	1	2	3	4	5	6	7
0	τ_0	τ_0	τ_0	τ_0	τ_0	τ_0	τ_0	τ_0
1	τ_8	τ_8	τ_8	τ_8	τ_8	τ_8	τ_8	τ_8
2	τ_8	τ_8	τ_8	τ_8	τ_8	τ_8	τ_8	τ_8
3	τ_8	τ_8	τ_8	τ_{27}	τ_{27}	τ_{27}	τ_{27}	τ_{27}
4	τ_{32}	τ_{33}	τ_{33}	τ_{33}	τ_{33}	τ_{33}	τ_{33}	τ_{33}
5	τ_{33}	τ_{33}	τ_{33}	τ_{33}	τ_{33}	τ_{45}	τ_{45}	τ_{47}
6	τ_{48}	τ_{48}	τ_{48}	τ_{48}	τ_{48}	τ_{48}	τ_{48}	τ_{48}
7	τ_{48}	τ_{48}	τ_{48}	τ_{48}	τ_{48}	τ_{48}	τ_{48}	τ_{48}

Table 28: $Dst\_Info(i \log n + j)$ after Step 3 of the leader finding problem algorithm

The values set during Step 3 are shown in large

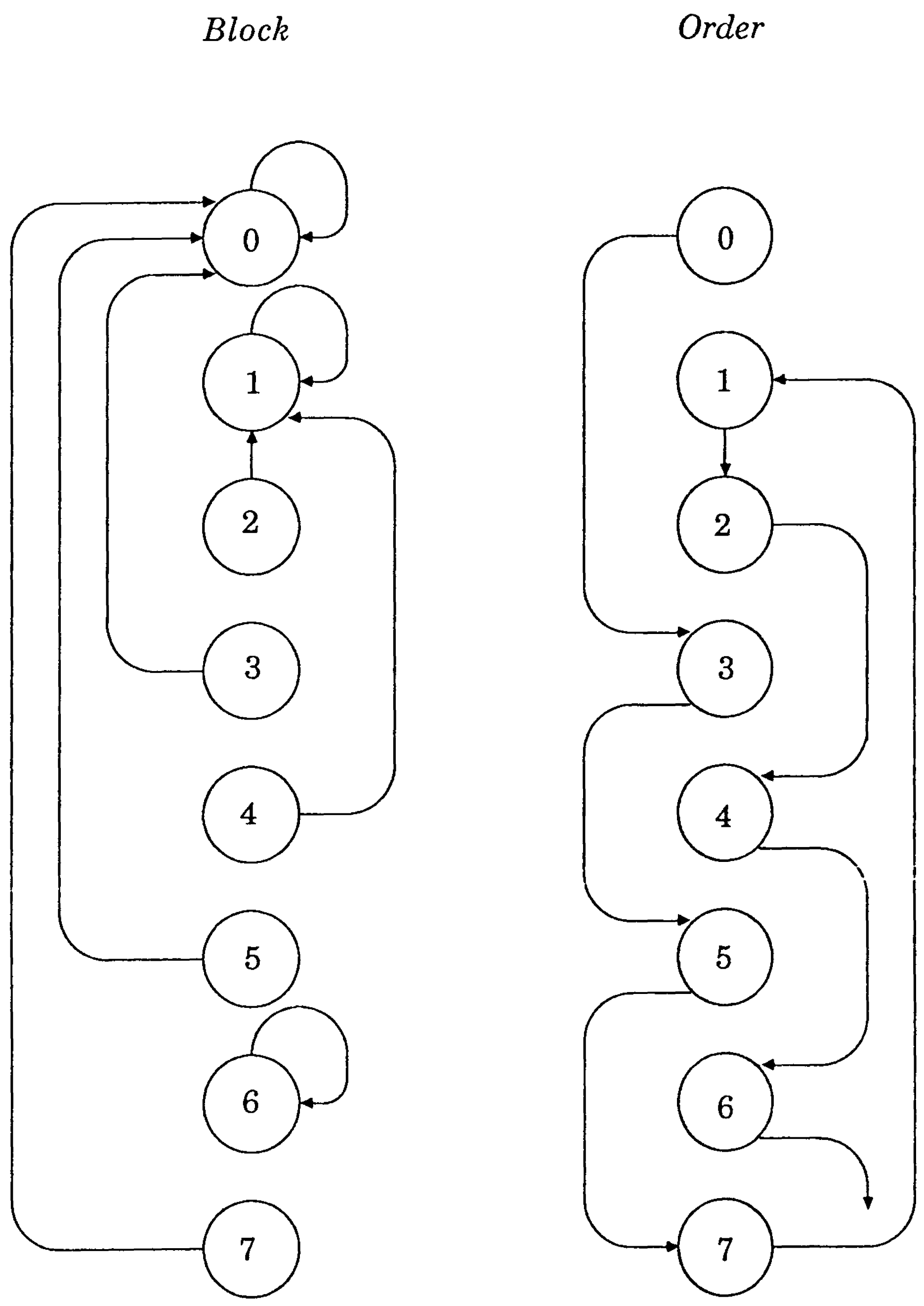

Figure 1: $\mathbf{P}_{\{0,1,2\}}$

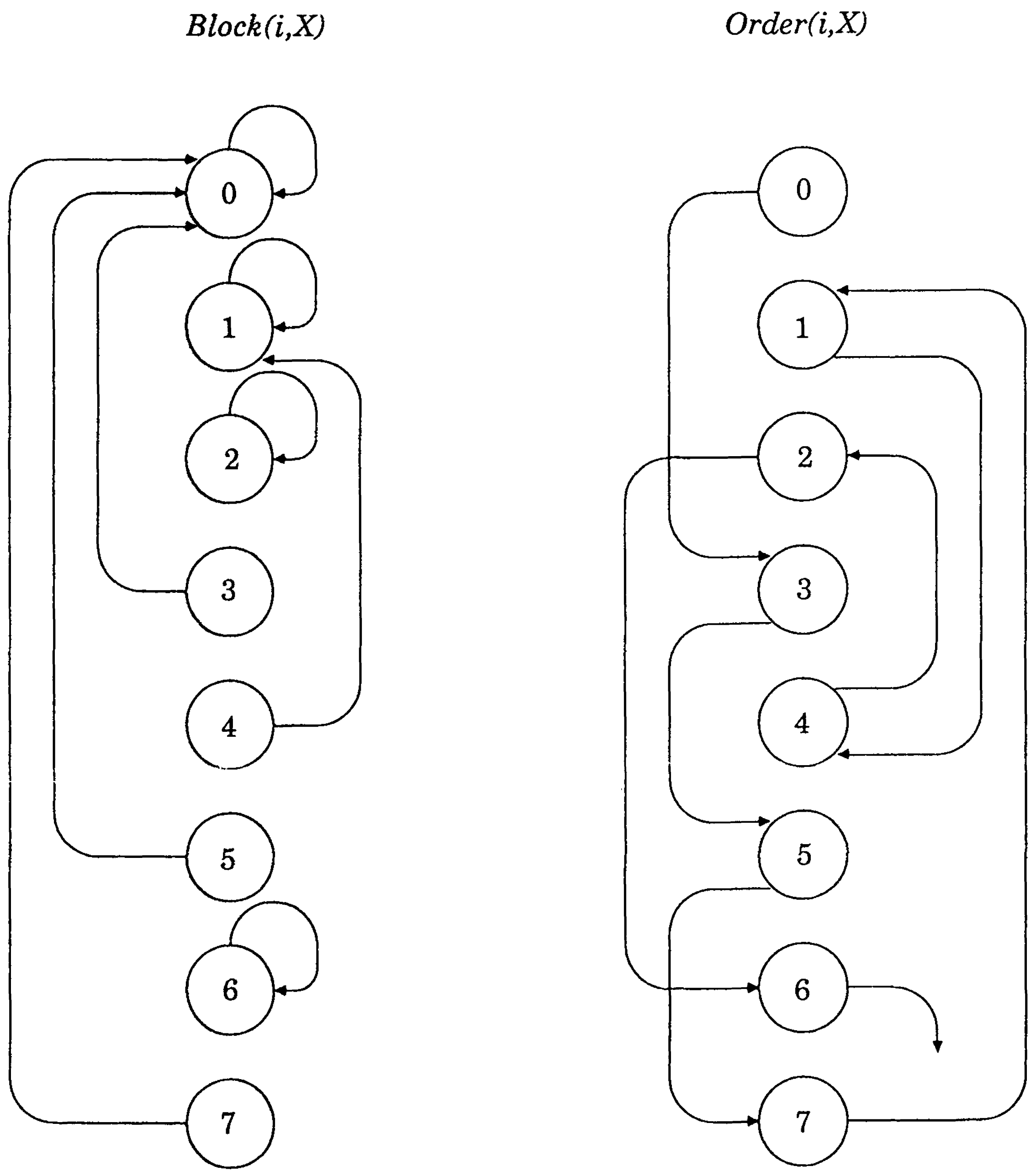

Figure 2: $\mathbf{P}_X = \mathbf{P}_{\{0,1,2,3,4,5\}}$

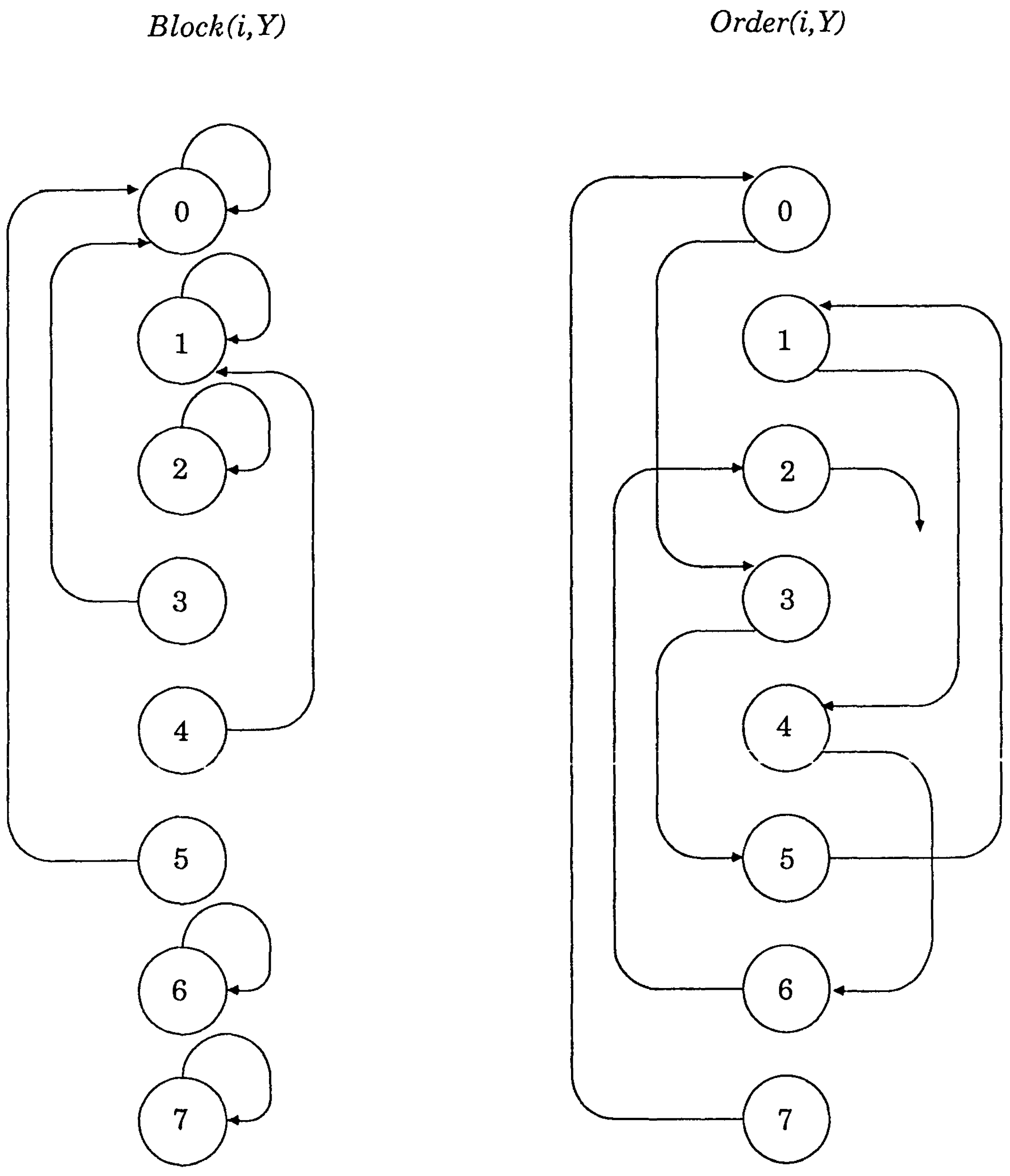

Figure 3: $\mathbf{P}_Y = \mathbf{P}_{\{6,7,8,9,10,11\}}$

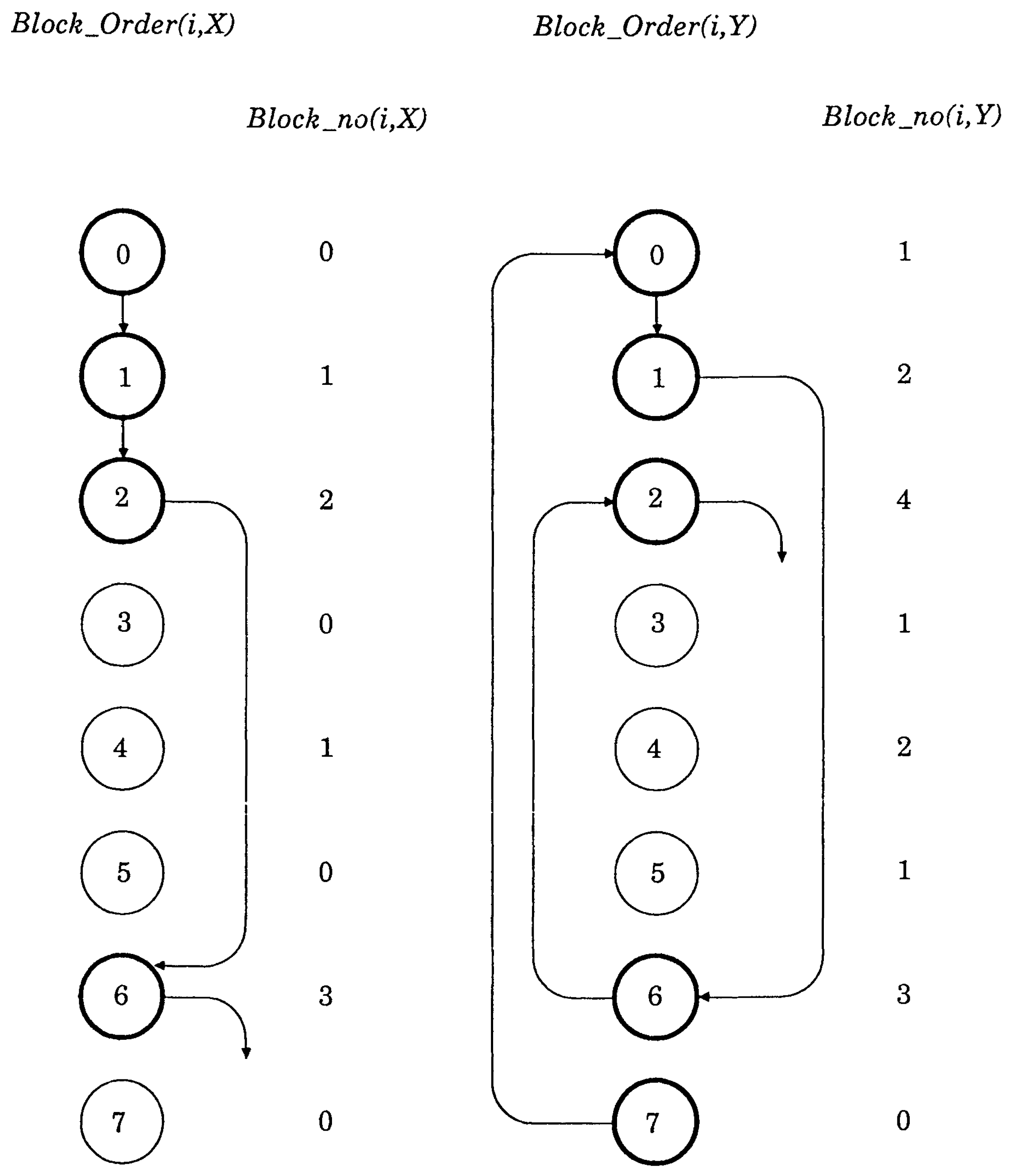

Figure 4: Phases 1 and 2

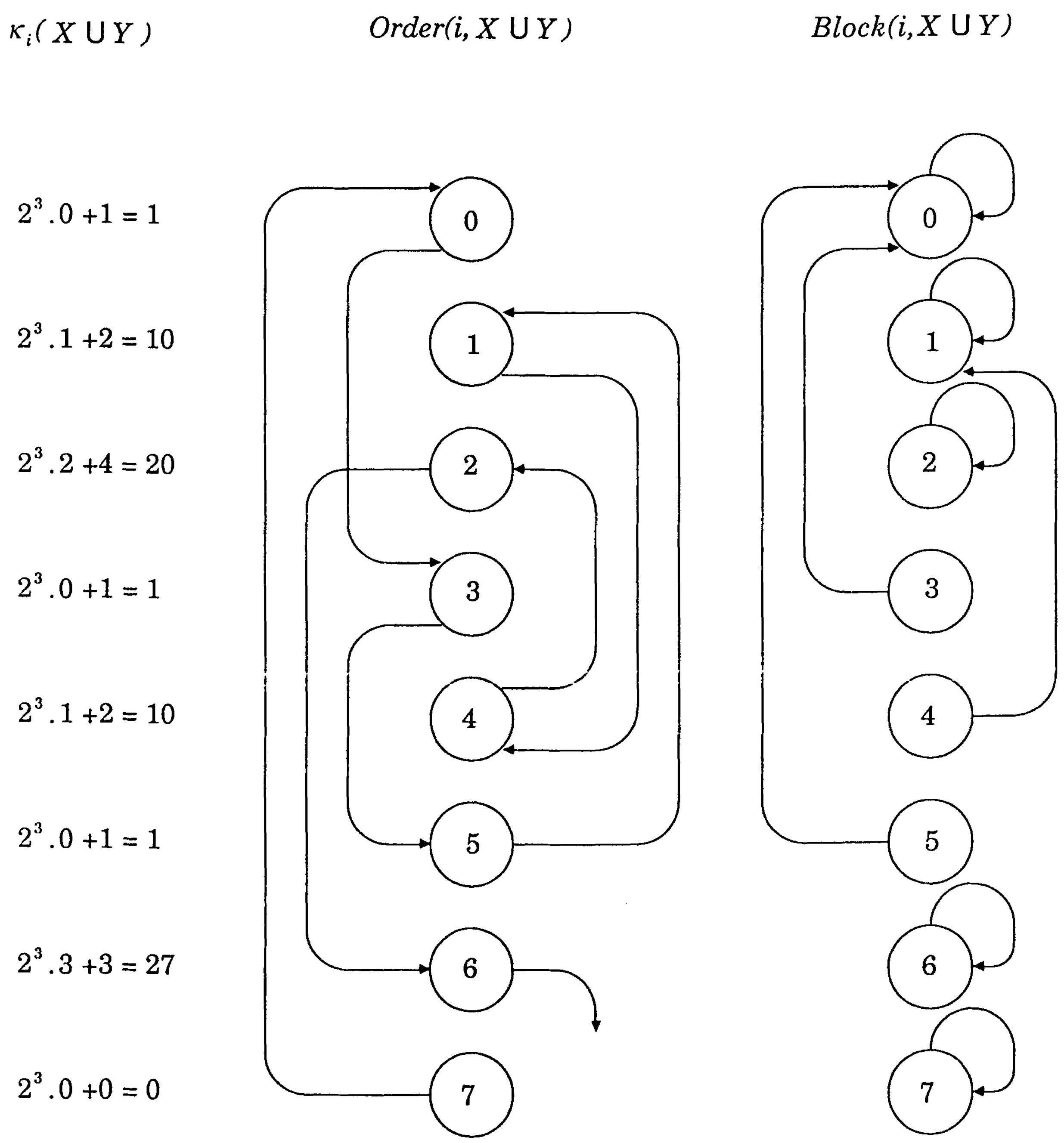

Figure 5: Phases 3 and 4

Errata

Page 15, § 5.2, paragraph 2, lines 1 and 2:

Let for any integer i, $N(i) = \{0, 1, \ldots, i-1\}$ and let $\rho : N(n) \longrightarrow N(n^2)$ be the color function that gives $\rho(i)$, the color (a number from $N(n^2)$) ...

Page 16, line 2:

For any color $x \in N(n^2)$ ...

Page 19, Phase 1, lines 7–11 should be replaced by

Next, the heads of $\mathcal{P}_J$ are detected using the following condition. A key k_i is a head of $\mathcal{P}_J$ iff $Block(i, J) = i$. For each head k_i of $\mathcal{P}_J$, $Block(Rev\_Order(i, J), J)$ is the head of $\mathcal{P}_J$ preceeding k_i (if $Rev\_Order(i, J) \neq$ NIL). If $Rev\_Order(i, J) =$ NIL then k_i is the first head of $\mathcal{P}_J$. It is clear that this phase needs $\Theta(\frac{n \log n}{\mathcal{H}})$ time and $\Theta(n \log n)$ space. Fig. 4 shows the output of this phase, for our example.

Page 20, Phase 4, lines 1–5 should be replaced by

Here we obtain $Block(i, X \cup Y)$. This can be done by first determining the heads of the blocks of $\mathcal{P}_{X \cup Y}$ as in phase 1, except that we use $\kappa_i(X \cup Y)$ instead of $Block(i, J)$. Next, we proceed as in Step 1 and obtain $Block(i, X \cup Y)$, by first ranking $Order(i, X \cup Y)$ and then applying the leader finding problem algorithm with $\kappa_i(X \cup Y)$ as the color of k_i. Phase 4 needs $\Theta(\frac{n \log n}{\mathcal{H}})$ time and $\Theta(\mathcal{H} n^2)$ space. Fig. 5 shows the result of this phase for our example.

Page 31, Appendix A, paragraph 3, line 1:

... For any color $x \in N(n^2)$ ...