ソフトウェア基礎論　37－9
プログラミング言語　27－9
(1990. 12. 13)

再構成アレイ上の高速ソーティングアルゴリズム

中野浩嗣　　増澤利光　　都倉信樹

大阪大学基礎工学部

あらまし

再構成バスとは形状が動的に変化することのできるバスである．格子状に並べたプロセッサを再構成バスで接続したプロセッサアレイを再構成アレイという．本論文では，プロセッサ数 $N \times N\log^{(T)} N$ の再構成アレイ上で N 個の要素のソーティングを $O(T)$ 時間で行うアルゴリズムを示す．

A Fast Sorting Algorithm on a Reconfigurable Array

Koji NAKANO　　Toshimits MASUZAWA　　Nobuki TOKURA

Faculty of Engineering Science, Osaka University

Toyonaka-shi, Osaka 560, Japan

ABSTRACT

A bus whose configuration can be dynamically changed is called a reconfigurable bus. A reconfigurable array is a processor array on which processors arranged to a 2-dimensional grid and connected by reconfigurable buses. We show an algorithm which sorts N elements in $O(T)$ time on a reconfigurable array of size $N \times N\log^{(T)} N$.

1 Introduction

Sorting is one of the most important problems in computer science because there are many problems whose time complexity depends on that of sorting. It is well known[2] that any sequential sorting algorithm needs $\Omega(n \log n)$ time [†], and there exist optimal sorting algorithms. To speed-up sorting, parallel sorting algorithms have been investigated [4] [5] [7]. For example, the optimal parallel sorting algorithm which sorts N elements in $O(\log N)$ time using N processors is well known as the AKS sorting network[3]. A sorting network is a feasible model, but the coefficient of $\log N$ is so large that this algorithm seems to be impractical. Another optimal parallel algorithm is well known as the Cole's optimal merge sort which sorts N elements in $O(\log N)$ time on the CREW PRAM with N processors [9] [10]. The coefficient of $\log N$ is not so large, but the CREW PRAM is regarded as an impractical parallel machine. On the other hand, a logarithmic time sorting algorithm on more practical model is known [15]. The algorithm is based on *an enumeration scheme* for parallel sorting as follows: to sort N elements, each element is simultaneously compared to all the others in constant time by using $N(N-1)$ processors, and the rank of each element is computed in $O(\log N)$ time by enumerating elements whose values are smaller than that of it. Hence, N elements can be sorted in $O(\log N)$ time using $N(N-1)$ processors. In this paper, we present a sub-logarithmic time sorting algorithm based on an enumeration scheme. Sub-logarithmic time is achieved by computing the rank of elements faster.

Recently many processor arrays with buses attract considerable attention. It is known that many problems can be solved fast on processor arrays with buses because buses decrease the diameter of networks and enhance the communication capabilities. For example, finding maximum [1] [8] [12] [16], finding median [16], sorting [12] [16], image processing [8] [12] [17] and component labeling [11] have been efficiently solved. The bus system used these algorithms is *static* in the sense that the configuration of buses never change during the execution of the algorithms.

In this paper we deal with *a reconfigurable array* that consists of processors arranged to a 2-dimensional grid and connected by *reconfigurable buses*(Fig. 1). A reconfigurable bus is a bus whose configuration can be dynamically changed. Some algorithms on a reconfigurable array are known [13] [14] [20]. For example, it is known [20] that N elements can be sorted by an enumeration scheme on a 3-dimensional reconfigurable array of size $N \times N \times N$. This algorithm can be executed on a 2-dimensional reconfigurable array of size $N \times N^2$. In this paper, we reduce the number of processors and improve this algorithm. Firstly, we show an algorithm for summing up N binary values in constant time on a reconfigurable array of size $N \times \log^2 N$. Secondary,

[†]Throughout this paper, the log to the base 2 is used.

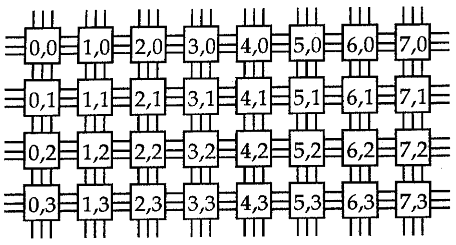

Figure 1: Reconfigurable array

by using this algorithm, we show that N elements can be sorted in constant time on a reconfigurable array of size $N \times N \log^2 N$. Lastly, we obtain more generalized algorithm which sorts N elements in $O(T)$ time on a $N \times N \log^{(T)} N$ reconfigurable array [‡]. This implies that N elements can be sorted in constant time on a reconfigurable array of size $N \times N \log^{(O(1))} N$ and in $O(\log^* N)$ time on a reconfigurable array of size $N \times N$.

2 Models and Notations

A reconfigurable array (shortly, *an array*) consists of processors arranged to a 2-dimensional grid and connected by reconfigurable buses. An array is formalized as follows. Processors on an array of size $N \times M$ are denoted by $\mathbf{PE}(i,j)$ $(0 \leq i \leq N-1, 0 \leq i \leq M-1)$. As shown in Fig. 1, it is considered that $\mathbf{PE}(i,0)$ $(0 \leq i \leq N-1)$ is located at the top row of an array and $\mathbf{PE}(0,j)$ $(0 \leq j \leq M-1)$ is located at the leftmost column of an array. Each processor on an array is the RAM (random access machine) extended by the instructions for changing configuration of buses, sending data to buses and receiving data from buses. Each processor has several ports denoted by $\mathbf{U}(k)$, $\mathbf{D}(k)$, $\mathbf{L}(k)$ and $\mathbf{R}(k)$ $(0 \leq k \leq P-1)$. The ports face to each other are connected by buses, that is, $\mathbf{D}(k)$ on $\mathbf{PE}(i,j)$ and $\mathbf{U}(k)$ on $\mathbf{PE}(i,j+1)$ are connected. Similarly, $\mathbf{R}(k)$ on $\mathbf{PE}(i,j)$ and $\mathbf{L}(k)$ on $\mathbf{PE}(i+1,j)$ are connected. It is assumed that each processor on an array can have the constant number of ports, that is, P is constant. All processors on an array work synchronously and execute the following instructions in order in a unit time:

Phase 1 Connecting its own ports by local buses(Fig. 2).

Phase 2 Sending at most one piece of data to each port.

Phase 3 Receiving data from each port. The data sent at the previous phase are received at this phase.

[‡]Let $\log^{(k)} = \underbrace{\log\log\cdots\log}_{k \text{ times}}$ and $\log^* n$ be the smallest k such $\log^{(k)} n \leq 1$.

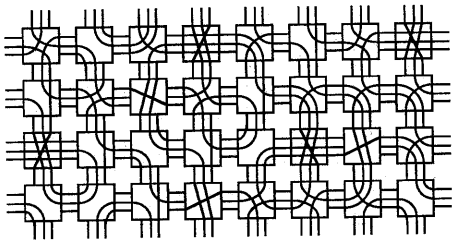

Figure 2: Example of connecting ports

Phase 4 Executing the constant number of instructions of the RAM.

In an array, all processors execute these phases synchronously, that is, no processor executes a phase before all processors finishing the previous phase. As shown in Fig. 2, after connecting ports of each processor by local buses, it can be regarded that the processors are connected by static buses. Several bus models are proposed [11] in terms of simultaneous sending on a static bus system.

Exclusive It is prohibited that two or more processors simultaneously attempt to send to the same bus.

Common If two or more processors simultaneously attempt to send to the same bus, they must be sending the same value.

Arbitrary If several processors simultaneously attempt to send to the same bus, then one of them succeeds and sends its value actually.

Throughout this paper, we use the common model as the model of a reconfigurable bus. For a cleaner presentation of the algorithms, we will omit the floor and ceiling operators necessary to ensure that all values are integers.

3 Basic Property and Basic Algorithms

In this section, we show a basic property and basic algorithms for our sorting algorithm. At the end of this section, we show an algorithm that sums up binary values in constant time.

3.1 Basic Property

The following lemma implies that the difference within the constant factor of the number of processors can be ignored.

Lemma 3.1 *Any execution in a unit time on an array of size $O(N) \times O(M)$ can be simulated in a unit time on an array of size $N \times M$.*

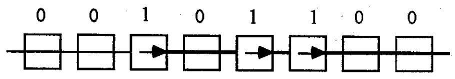

Figure 3: Leftmost finding

Proof. Let **A** and **B** be an array of size $c_1N \times c_2M$ (where c_1 and c_2 are constant positive numbers) and an array of size $N \times M$, respectively. Since both c_1 and c_2 are constant, it can be assumed that each processor on **B** has $\max\{c_1, c_2\}$ times as many ports as **A**. Then, each $\mathbf{PE}(x, y)$ on **B** can simulates any execution of all $\mathbf{PE}(i, j)$ $(c_1x \leq i < c_1(x+1), c_2y \leq j < c_2(y+1))$ on **A**. Therefore, any execution on **A** in a unit time can be simulated on **B** in a unit time. □

Even if we regard that the time complexity is affected by the number of the local computation, any execution in a unit time on **A** can be simulated in $O(c_1c_2)$ time on **B**. Since c_1c_2 is constant, **B** can simulate **A** in constant time.

3.2 Leftmost Finding

We consider the problem to find the leftmost element whose value is 1, when a binary sequence of length N is given. The problem to finding the leftmost element on an array of size $N \times 1$ is defined as follows.

Input Let $B = < a(0), a(1), \ldots, a(N-1) >$ be a binary sequence. Each $a(i)$ $(0 \leq i \leq N-1)$ is given to $\mathbf{PE}(i, 0)$.

Output All processors know m such that $m = \min\{i | a(i) = 1\}$. If there does not exist i such that $a(i) = 1$, all processors know $m (= N)$.

The leftmost finding algorithm follows (Fig. 3).

[Algorithm for Leftmost Finding]

```
Step1 if $a(i) = 0$, then $\mathbf{L}(0) - \mathbf{R}(0) : \mathbf{PE}(i, 0)$. {means
          that if $a(i) = 0$, then $\mathbf{PE}(i, 0)$ connects $\mathbf{L}(0)$
          and $\mathbf{R}(0)$}
      if $a(i) = 1$, then $\mathbf{R}(0) \leftarrow 1 : \mathbf{PE}(i, 0)$. {means that
          if $a(i) = 1$, then $\mathbf{PE}(i, 0)$ sends 1 to $\mathbf{R}(0)$.}
      $\mathbf{L}(0) \rightarrow c(i) : \mathbf{PE}(i, 0)$ $(0 \leq i \leq N-1)$. {means
          that each $\mathbf{PE}(i, 0)$ receives the data from $\mathbf{L}(0)$,
          and stores it to $c(i)$ that is a local memory cell
          of $\mathbf{PE}(i, 0)$.}
      if $\mathbf{PE}(i, 0)$ cannot receive any data, let $c(i) \leftarrow 0$.
Step2 if $a(i) = 1$ and $c(i) = 0$, then $\mathbf{PE}(i, 0)$ broad-
          casts $i$ to all processors.
      if $a(N-1) = 0$ and $c(N-1) = 0$, then $\mathbf{PE}(N-1, 0)$
          broadcasts $N$ to all processors.
```

[end of algorithm]

The next lemma holds.

Lemma 3.2 *Leftmost finding can be done in constant time on an array of size $N \times 1$.*

Proof. If $c(i) = 1$ then there exists $j < i$ such that $a(j) = 1$ and vice versa. Thus, if both $a(i) = 1$ and $c(i) = 0$ hold, $a(i)$ is the leftmost element. If such i does not exist, the value of all elements is 0. This completes the proof. □

3.3 Compression

We consider the procedure that compresses a sequence of elements. Compression on an array of size $N \times M$ is defined as follows.

Input Let $B =< a(0), a(1), \ldots, a(N-1) >$ be a sequence of elements. Each element in the sequence may take value NULL. Each $a(i)$ $(0 \le i \le N-1)$ is given to $\mathbf{PE}(i,0)$.

Output Let $B' =< a(i_0), a(i_1), a(i_2), \ldots >$ be the subsequence of B such that an element in B is in B' if and only if its value is not NULL. Each $\mathbf{PE}(0,j)$ $(0 \le j \le M-1)$ knows $a(i_j)$ if exists.

In the compression algorithm, each column on an array works as a stack and each processor on the top row works as the top of the stack. From the rightmost to the leftmost column, if an element given to a column is not NULL, then the processors on the column push the element on a stack. Otherwise, the processors do nothing to the stack. After that, each processor on the leftmost column knows the element whose value is not NULL. The compression algorithm follows (Fig. 4).

[Compression Algorithm]

```
Step1 Each $\mathbf{PE}(i,0)$ $(0 \le i \le N-1)$ broadcasts $a(i)$ to
    the processors on the same column, $\mathbf{PE}(i,j)$ $(0 \le j \le M-1)$.
Step2 if $a(i) = $ NULL
        $\mathbf{L}(0) - \mathbf{R}(0) : \mathbf{PE}(i,j)$ $(0 \le j \le M-1)$.
    if $a(i) \ne$ NULL
        $\mathbf{D}(0) - \mathbf{R}(0) : \mathbf{PE}(i,j)$ $(0 \le j \le M-1)$
        $\mathbf{U}(0) - \mathbf{L}(0) : \mathbf{PE}(i,j)$ $(0 \le j \le M-1)$
        $\mathbf{U}(0) \leftarrow a(i) : \mathbf{PE}(i,0)$.
    $\mathbf{L}(0) \rightarrow c(j) : \mathbf{PE}(0,j)$ $(0 \le j \le M-1)$. {$c(j)$
        contains $a(i_j)$.}
                                        [end of algorithm]
```

The following lemma holds.

Lemma 3.3 *Compression can be done in constant time on an array.*

Proof. The proof can be done by induction easily. □

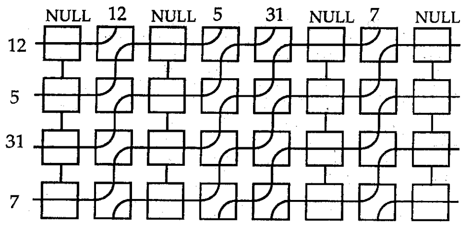

Figure 4: Compression

3.4 Prefix Remainder Computation

The prefix remainder computation of a binary sequence on an array of size $N \times M$ is defined as follows.

Input Let $< a(0), a(1), \ldots, a(N-1) >$ be a binary sequence of length N. Each $a(i)$ $(0 \le i \le N-1)$ is given to $\mathbf{PE}(i,0)$.

Output Each $\mathbf{PE}(i,0)$ $(0 \le i \le N-1)$ know $(\sum_{j=0}^{i} a(j)) \bmod M$.

In the algorithm for computing the prefix remainder, each column on an array works as a cyclic shift register of size M. It is considered that the cyclic shift register is placed vertically. On the leftmost column, only the top element of the cyclic shift register is 1. On each column, if the element given to the column is 1, then the processors on the column shift the cyclic shift register. Otherwise, the processors do nothing to the cyclic shift register. Then the prefix remainder is equal to the position where 1 places on the cyclic shift register. The algorithm for computing the prefix remainder is described as follows(Fig. 5).

[Algorithm for Computing Prefix Remainder]

```
Step1 Each $\mathbf{PE}(i,0)$ $(0 \le i \le N-1)$ broadcasts $a(i)$
    to the processors on the same column.
Step2 if $a(i) = 0$,
        $\mathbf{L}(0) - \mathbf{R}(0) : \mathbf{PE}(i,j)$ $(0 \le j \le N-1)$.
    if $a(i) = 1$,
        $\mathbf{L}(0) - \mathbf{D}(0) : \mathbf{PE}(i,j)$ $(0 \le j \le M-2)$
        $\mathbf{U}(0) - \mathbf{R}(0) : \mathbf{PE}(i,j)$ $(1 \le j \le M-1)$
        $\mathbf{U}(1) - \mathbf{D}(1) : \mathbf{PE}(i,j)$ $(1 \le j \le M-2)$
        $\mathbf{D}(1) - \mathbf{R}(0) : \mathbf{PE}(i,0)$
        $\mathbf{L}(0) - \mathbf{U}(1) : \mathbf{PE}(i,M-1)$.
    $\mathbf{L}(0) \leftarrow 1 : \mathbf{PE}(0,0)$.
    $\mathbf{R}(0) \rightarrow c(i,j) : \mathbf{PE}(i,j)$ $(0 \le j \le M-1)$.
Step3 if $c(i,j) = 1$, $\mathbf{PE}(i,j)$ broadcasts $j$ to the pro-
    cessors on the same column.
                                        [end of algorithm]
```

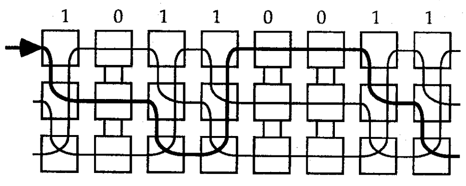

Figure 5: Prefix remainder

Lemma 3.4 *The prefix remainder of a binary sequence can be computed in constant time on an array.*

Proof. It is sufficient to prove that for all i, $c(i,j) = 1$ iff $\sum_{k=0}^{i} a(k) \equiv j \pmod M$ holds. This can be easily proved by induction. □

The CRCW PRAM with the polynomial number of processors cannot compute even the exclusive-or of the binary sequence in constant time[6]. Hence, an array is more powerful than the PRAM with regard to the prefix remainder computation.

3.5 Remainder Computation

We show an algorithm for computing the remainder of the sum of a binary sequence on an array. The w-remainder computation on an array is defined as follows.

Input Let $< a(0), a(1), \ldots, a(N-1) >$ be a binary sequence of length N. Each $a(i)$ $(0 \le i \le N-1)$ is given to $\mathbf{PE}(i,0)$.

Output Let $x = \sum_{i=0}^{N-1} a(i)$. All processors know r in the congruence $x \equiv r \pmod w$.

From Lemma 3.4, M-remainder can be computed in constant time on an array of size $N \times M$. We will show that the remainder from a lager modulus can be computed in constant time. The basic idea is as follows. Consider that an array of size $N \times M$ is divided into $\sqrt{M}$ subarrays of size $N \times 1, N \times 2, \cdots, N \times \sqrt{M}$. To express x by the RNS (residue number system), we apply the algorithm in Lemma 3.4 to each subarray, and then $c_1, c_2, \ldots, c_{\sqrt{M}}$ can be computed such that $x \equiv c_i \pmod i$. If $c_1, c_2, \ldots, c_{\sqrt{M}}$ are known, the remainder from a larger modulus than M can be computed. We utilize the Chinese remainder theorem and the prime number theorem, famous theorems in discrete mathematics, for analyzing how large modulus come to be by using this method.

Theorem 3.5 (Chinese remainder theorem) *Let $p_1, p_2, \ldots, p_m$ be mutually prime positive integers. If $b_1, b_2, \ldots, b_m$ are known as follows,*

$$\begin{cases} x \equiv b_1 \pmod{p_1} \\ x \equiv b_2 \pmod{p_2} \\ \quad \vdots \\ x \equiv b_m \pmod{p_m} \end{cases}$$

then r can be computed such that $x \equiv r \pmod{p_1 p_2 \cdots p_m}$. □

From the Chinese remainder theorem, the following corollary holds.

Corollary 3.6 *Let $\mathrm{lcm}(m)$ be the L.C.M. (least common multiple) of $\{1, 2, \ldots, m\}$. If the following congruences hold for integers x and y,*

$$\begin{cases} x \equiv y \pmod 1 \\ x \equiv y \pmod 2 \\ \quad \vdots \\ x \equiv y \pmod m \end{cases}$$

then the congruence $x \equiv y \pmod{\mathrm{lcm}(m)}$ holds. □

To analyze how large $\mathrm{lcm}(m)$ is, we use the prime number theorem.

Theorem 3.7 (Prime number theorem) *Let $\pi(n)$ be the number of prime numbers less than or equal to n. The following equality holds:*

$$\lim_{n\to\infty} \frac{\pi(n)\ln n}{n} = 1,$$

where $\ln$ is the natural logarithm. □

Thus, $\pi(n) = \Theta(n/\log n)$ holds. From the prime number theorem, the following lemma holds.

Lemma 3.8 *The equality $\mathrm{lcm}(n) = 2^{\Theta(n)}$ holds.*

Proof. Let $\{p_1, p_2, \ldots, p_m\}$ $(p_1 = 2 < p_2 < \cdots < p_m \le n, m = \pi(n))$ be the set of primes less than or equal to n and $a_1, a_2, \ldots, a_m$ be the integers such that ${p_i}^{a_i} \le n < {p_i}^{a_i+1}$ holds. From the prime number theorem, $m = \Theta(n/\log n)$. Thus,

$$\begin{aligned} \mathrm{lcm}(n) &= {p_1}^{a_1}{p_2}^{a_2}\cdots{p_m}^{a_m} \\ &< n^m \\ &= n^{\Theta(n/\log n)} \\ &= 2^{\Theta(n)}. \end{aligned}$$

Let m' be the integer such that $p_{m'} < n/2 \le p_{m'+1}$. From the prime number theorem, $m - m' = \Theta(n/\log n)$. Thus,

$$\begin{aligned} \mathrm{lcm}(n) &> p_{m'+1}p_{m'+2}\cdots p_m \\ &> (n/2)^{\Theta(n/\log n)} \\ &= 2^{\Theta(n)}. \end{aligned}$$

Therefore, $\mathrm{lcm}(n) = 2^{\Theta(n)}$ holds. □

The algorithm for computing the remainder from a lager modulus is described as follows.

[Algorithm for Computing $\mathrm{lcm}(\sqrt{M})$-Remainder] Consider that an array of size $N \times M$ is partitioned into $\sqrt{M}$ subarrays of size $N \times 1, N \times 2, \ldots, N \times \sqrt{M}$. $\mathbf{PE}(i,j)$ on an subarray of size $N \times k$ is demoted by $\mathbf{PE}_k(i,j)$.

Step1 On each subarray of size $N \times k$, computing r_k such that $x \equiv r_k \pmod{k}$ by computing the prefix remainder. When the computation of prefix remainder completed, $\mathbf{PE}_k(N-1,0)$ broadcasts r_k to all processors on the kth subarray.

Step2 Computing the prefix remainder of the sequence $< 1, 1, \ldots, 1 >$ on each subarray of size $N \times k$. After that, each $\mathbf{PE}_k(i,0)$ knows $r'_{k,i}$ in the congruence $i \equiv r'_{k,i} \pmod{k}$.

Step3 On each ith column, comparing r_k and $r'_{k,i}$ and examining whether $r_k = r'_{k,i}$ holds for all k $(1 \le k \le \sqrt{M})$.

Step4 Finding the minimum i provided that $r_k = r'_{k,i}$ for all k $(1 \le k \le \sqrt{M})$ by executing the leftmost finding algorithm. That is, computing $r = \min\{i | r_k = r'_{k,i} \text{ for all } k\}$. And broadcasting r to all processors on an array. After that, all processors know r, the solution of $\mathrm{lcm}(\sqrt{M})$-remainder.

Step5 Finding the minimum $i > 0$ provided that $r'_{k,i} = 0$ for all k $(1 \le k \le \sqrt{M})$ by leftmost finding. That is, computing $w = \min\{i | r'_{k,i} = 0 \text{ for all } k\}$. If such w $(= \mathrm{lcm}(\sqrt{M}))$ exists, broadcasting w.

[end of algorithm]

For computing $\mathrm{lcm}(\sqrt{M})$-remainder only, we do not have to execute step5. But, for the application to the following algorithm that computes the sum of a binary sequence, $\mathrm{lcm}(\sqrt{M})$ is computed in step 5.

Lemma 3.9 *The problem* $\mathrm{lcm}(\sqrt{M})$*-remainder can be solved in constant time on an array of size* $N \times M$.

Proof. For all k $(1 \le k \le \sqrt{M})$, the congruence $x \equiv r \pmod{k}$ holds. Therefore, from Corollary 3.6, the congruence $x \equiv r \pmod{\mathrm{lcm}(\sqrt{M})}$ holds. This completes the proof. □

From Lemma 3.9, $\mathrm{lcm}(\sqrt{kM})$-remainder can be computed in constant time on an array of size $N \times kM$. Hence, from Lemma 3.1, the following lemma holds.

Lemma 3.10 *The problem* $\mathrm{lcm}(\Theta(\sqrt{M}))$*-remainder can be solved in constant time on an array of size* $N \times M$. □

From Lemma 3.8, there exists a constant number k such that $\mathrm{lcm}(\sqrt{k \log^2 N}) > N$. Therefore, the following corollary holds.

Corollary 3.11 *The sum of a binary sequence of size* N *can be computed in constant time on an array of size* $N \times \log^2 N$. □

Proof. To compute the sum of a binary sequence, the $\mathrm{lcm}(\sqrt{k \log^2 N})$-remainder computation is repeated for $k = 1, 2, \ldots$ in order until $\mathrm{lcm}(\sqrt{k \log^2 N}) > N$ holds. At the end of the iteration, $x = (x \bmod \mathrm{lcm}(\sqrt{k \log^2 N}))$ holds because x is at most N. Since k is constant, the number of the iteration is constant. Therefore, the sum of the binary sequence can be computed in constant time. □

A more efficient algorithm can be obtained if the algorithm in Lemma 3.9 is modified as follows: for the prime numbers $p_1, p_2, \ldots$ defined in the proof of Lemma 3.8, an array is divided into subarrays of size $N \times p_1, N \times p_2, \ldots$ and $x \bmod p_1, x \bmod p_2, \ldots$ are computed on them. In this algorithm, the modulus come to be $2^{\Theta(\sqrt{M \log M})}$. Hence, in Corollary 3.11, the size of an array is reduced to $N \times \log^2 N / \log\log N$. But this modification makes algorithm more complicated and does not lead asymptotical improvement of our sorting algorithm.

4 New Sorting Algorithm

Sorting on an array is defined as follows.

Input Let $< a(0), a(1), \ldots, a(N-1) >$ be a sequence of elements. Each $a(i)$ $(0 \le i \le N-1)$ is given to $\mathbf{PE}(i,0)$.

Output Let $< a(r_0), a(r_1), \ldots, a(r_{N-1}) >$ be the sorted sequence of the input sequence, that is, $a(r_i) \le a(r_{i+1})$ for all i. Each $\mathbf{PE}(i,0)$ $(0 \le i \le N-1)$ knows $a(r_i)$.

Without loss of generality, it is assumed that the elements are distinct, that is, for all i and j $(i \ne j)$, $a(i) \ne a(j)$ holds. *The rank of* $a(i)$, the number of smaller elements than $a(i)$, is denoted by r_i.

4.1 Constant Time Sorting

At first we show a constant time sorting algorithm on an array of size $N \times N \log^2 N$. Sorting is done along an enumeration scheme, that is, by computing the rank of each element. To compute the rank of each element, we use the algorithm for computing the sum of a binary sequence. From Corollary 3.11, the sum of a binary sequence of length N can be computed in constant time on an array of size $N \times \log^2 N$. Therefore, the ranks of all elements can be computed in constant time on an array of size $N \times N \log^2 N$. The algorithm is described as follows.

[Constant Time Sorting Algorithm] Consider that an array of size $N \times N \log^2 N$ is divided horizontally into N subarrays of size $N \times \log^2 N$. We denote $\mathbf{PE}(i,j)$ on the kth $(0 \le k \le N-1)$ subarray by $\mathbf{PE}_k(i,j)$, that is, $\mathbf{PE}_k(i,j)$ means $\mathbf{PE}(i, k \log^2 N + j)$.

Step1 Each $\mathbf{PE}(i,0)$ broadcasts $a(i)$ to the processors on the same column. And each $\mathbf{PE}_k(k,0)$ broadcasts $a(k)$ to all processors on the same row.

Step2 Each $\mathbf{PE}_k(i,0)$ compares $a(k)$ and $a(i)$. If $a(k) > a(i)$, let $r_{k,i} \leftarrow 1$. Otherwise, let $r_{k,i} \leftarrow 0$.

Step3 Computing the rank of $a(k)$ i.e. $r_k = \sum_{j=0}^{N-1} r_{k,j}$ on each kth subarray in constant time by computing the sum of the binary sequence.

Step4 Each $\mathbf{PE}_k(r_k, 0)$ sends r_kth element, $a(k)$, to $\mathbf{PE}(r_k, 0)$.

[end of algorithm]

The following theorem holds.

Theorem 4.1 *N elements can be sorted in constant time on an array of size $N \times N \log^2 N$.* □

4.2 General Sorting Algorithm

We show a sorting algorithm on an array of size less than $N \times N\log^2 N$. On an array of size $N \times M$ ($M < N\log^2 N$), the rank of each elements cannot be computed in constant time. But the remainder of the rank of each element can be computed by the remainder computation, so the elements are classified by the remainder of their ranks and are partitioned into the groups. And the sorting algorithm is applied to each group recursively so that the rank of each element in its group can be computed. After that, the rank of each element can be computed from the remainder of the rank and the rank in its group.

The sorting algorithm is described as follows.

[Algorithm for Sorting] Similarly to the constant time sorting, consider that an array of size $N \times M$ is divided into N (horizontal) subarrays of size $N \times (M/N)$. And let $w = \text{lcm}(\sqrt{M/N})$. Regardless of the horizontal partition, if $w \leq N$, consider that an array of size $N \times M$ is divided into w vertical subarrays of size $N/w \times M$. Each $\mathbf{PE}(i,j)$ on the kth vertical subarray is denoted by $\mathbf{PE}'_k(i,j)$ ($0 \leq i \leq N/w - 1, 0 \leq j < M$).

Step1,Step2 Executing the same as step1 and step 2 of the constant time sorting.

Step3 Computing w-remainder of $< r_{k,0}, r_{k,1}, \ldots, r_{k,N-1} >$ on each the kth horizontal subarray, that is, computing r'_k such that $r_k \equiv r'_k \pmod{w}$. In case $w > N$, executing step 4 of the constant time sorting algorithm, because $r_k = r'_k$ holds. Otherwise, executing the following steps to gather elements whose ranks take the same remainder.

Step4 Each $\mathbf{PE}'_j(0, kM/N)$ ($0 \leq j \leq w-1, 0 \leq k \leq N-1$) let its local variable $d_j(k) \leftarrow$ NULL. Since for all k each $\mathbf{PE}'_{r'_k}(0, kM/N)$ knows $a(k)$, let $d_{r'_k}(k) \leftarrow a(k)$. After that, $< d_j(0), d_j(1), \ldots, d_j(N-1) >$ contains all elements whose rank takes the remainder j

Step5 Compressing $< d_j(0), d_j(1), \ldots, d_j(N-1) >$ for each j on each vertical subarray. After this, each $\mathbf{PE}'_j(i,0)$ knows one of the elements whose rank takes the remainder j. Let $s_{j,i}$ be the index such that $\mathbf{PE}'_j(i,0)$ knows $a(s_{j,i})$.

Step6 Sorting each sequence $<$ $a(s_{j,0})$, $a(s_{j,1})$, $\ldots$, $a(s_{j,N/w-1})$ $>$ for all j on each vertical subarray recursively. Let the rank of $a(s_{j,i})$ in the sequence $<$ $a(s_{j,0})$, $a(s_{j,1})$, $\ldots$, $a(s_{j,N/w-1})$ $>$ be $r_{j,i}$. Then the rank of $a(s_{j,i})$ in $<$ $a(0)$, $a(1)$, $\ldots$, $a(N-1)$ $>$ is $r_{j,i}w + j$.

Step7 Similarly to step4 of the constant time sorting, sending $a(s_{j,i})$ to $\mathbf{PE}(r_{j,i}w + j, 0)$.

[end of algorithm]

Theorem 4.2 *N elements can be sorted in time $O(T)$ on an array of size $N \times N\log^{(T)} N$.*

Proof. The correctness of the algorithm can be proved easily by induction. We have to analyze the computation time required in this algorithm. Let $u_i \times M$ be the size of vertical subarrays after the ith iteration of the recursive procedure. Thus, $u_0, u_1, \ldots$ are the decreasing sequence and $u_0 = N$, $u_1 = N/\text{lcm}(\sqrt{M/N})$ hold. In general,$u_{i+1} = u_i/\text{lcm}(\sqrt{M/u_i})$ holds. From Lemma 3.8, there exists constant c such that $\text{lcm}(n) \geq 2^{n/c}$ for all $n > 2$. Hence, $u_{i+1} \leq u_i/2^{\sqrt{M/u_i}/c}$ holds. Thus, $u_{c(i+1)} \leq u_{ci}/2^{\sqrt{M/u_{ci}}}$ holds. Let $M = N(\log^{(T)} N)^2$, then $u_c \leq N/\log^{(T-1)} N$ and $u_{2c} \leq N/(\log^{(T-1)} N)^2$. In general, $u_{2ci-c} \leq N/\log^{(T-i)} N$ and $u_{2ci} \leq N/(\log^{(T-i)} N)^2$ hold. This can be proved by induction on i. In case $i > T$, $u_{2ci} \leq 1$ holds, so the number of iterations of the recursive procedure is at most $2cT = O(T)$. Because each iteration can be done in constant time, the computation time is $O(T)$. Furthermore, since $(\log^{(T)} N)^2 \leq \log^{(T-1)} N$ holds, the computation time is $O(T)$ in case $M = N\log^{(T)} N$. □

Theorem 4.2 implies the following corollaries.

Corollary 4.3 *N elements can be sorted in constant time on an array of size $N \times N\log^{(O(1))} N$.* □

Corollary 4.4 *N elements can be sorted in $O(\log^* N)$ time on an array of size $N \times N$.* □

Let us estimate the area-time complexity of our sorting algorithm in Corollary 4.3. On a reconfigurable array of size $N \times N\log^{(O(1))} N$, processors at the top row of each horizontal subarray must be connected by horizontal buses that transfer $\log N$ bits in one unit time because the elements and the rank of elements have to be transferred. Similarly, every vertical bus is required to transfer $\log N$ bits in a unit time. But the other horizontal buses are required to transfer only 1 bit in one unit time, because these buses are used only for summing up the binary sequence. Hence, the sorting algorithm needs area $A = O(N\log N) \times O(N\log N + N\log^{(O(1))} N) =$

$O(N^2 \log^2 N)$ and time $T = O(1)$. Therefore, we get $AT^2 = O(N^2 \log^2 N)$ that is slightly closed to the lower bound $AT^2 = \Omega(N^2 \log N)$[18][19].

5 Conclusions

In this paper, we have presented a fast sorting algorithm on reconfigurable arrays. As mentioned before, our sorting algorithm needs the smaller number of processors than the previous algorithm [20]. But the previous algorithm requires reconfigurable buses whose model is exclusive. Hence the previous algorithm is executed on the weaker model than ours.

It remains open whether N elements can be sorted in constant time on a reconfigurable array of size $N \times N$ or not.

References

[1] A. Aggarwal, *Optimal Bounds for Finding Maximum on Array of Processors with k Global Buses*, IEEE Trans. Comput., C-35, 1, pp.62–64, 1986.

[2] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Welsey,1974.

[3] M. Ajtai, J. Kolmos and E. Szemeredi, *An $O(n \log n)$ Sorting Network*, Proc. 15th ACM Symp. on Theory of Computing, pp.1–9, 1983.

[4] S. G. Akl, *Parallel Sorting Algorithms*, Academic Press, 1985.

[5] S. G. Akl, *The Design and Analysis of Parallel Algorithms*, Prentice-Hall, 1989.

[6] P. Beame and J. Hastad, *Optimal Bounds for Decision Problems on the CRCW PRAM*, Proc. 19th STOC, pp.83–93, 1987.

[7] D. Bitton, D. J. Dewitt, D. K. Hsiao and J.Menon, *A Taxonomy of Parallel Sorting*, ACM Computing Surveys, 16, 3, pp.287–318, 1984

[8] S.H.Bokhari, *Finding Maximum on an Array Processor with a Global Bus*, IEEE Trans. Comput., C-33, 2, pp.133–139, 1984.

[9] R. Cole, *Parallel Merge Sort.* Proc. 27th FOCS, pp.511–516, 1986.

[10] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms*, Cambridge University Press, 1989.

[11] K. Iwama and Y. Kambayashi, *An $O(\log n)$ Parallel Connectivity Algorithm on the Mesh of Buses*, Information Processing, 11, pp.305–310, 1989.

[12] V. K. P. Kumar and C. S. Raghavendra, *Array Processor with Multiple Broadcasting*, J. of Parallel and Distributed Comput., 4, pp.173–190,1987.

[13] R. Miller and Q. F. Stout, *Efficient Parallel Convex Hull Algorithms*, IEEE Trans. Comput., C-37, 12, pp.1605–pp.1618, 1988.

[14] R. Miller, V. K. P. Kumar, D. I. Reisis and Q. F. Stout, *Data Movement Operations and Applications on Reconfigurable VLSI Arrays*, Proc. of ICPP, 1, pp.205–208, 1988.

[15] D. E. Muller and F. P. Preparata, *Bounds to Complexities of Networks for Sorting and for Switching*, J.ACM, 22, 2, pp.195–201, 1975.

[16] Q. F. Stout, *Mesh-connected Computers with Multiple Broadcasting*, IEEE Trans. Comput., C-32, 9,pp.826–830,1983.

[17] Q. F. Stout, *Meshes with Multiple Buses*, Proc. 27th FOCS, pp.264–272, 1986.

[18] C. D. Thompson, *The VLSI Complexity of Sorting*, IEEE Trans. Comput., C-32, 12,pp.1171–1184,1983.

[19] J. D. Ullman, *Computational Aspects of VLSI* Computer Science Press, 1984.

[20] B. F. Wang, G. H. Chen and F. C. Lin, *Constant Time Sorting on a Processor Array with a Reconfigurable Bus System*, Information Processing Letters, 34, pp.187–192, 1990.