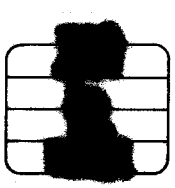
PARALLEL ALGORITHMS ON PROCESSOR ARRAYS WITH BUSES

Koji Nakano

January 1992



PARALLEL ALGORITHMS ON PROCESSOR ARRAYS WITH BUSES

Koji Nakano

January 1992

Dissertation submitted to the Faculty of Engineering Science
of Osaka University in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy
in Engineering

Abstract

Processor networks connected by buses have attracted considerable attention. Since buses enhance communication capabilities compared with one-to-one communication links, algorithms on bus-connected networks run faster than on usual link-connected networks. Several conventions concerning simultaneous sending to the same bus have been proposed: *exclusive* (simultaneous sending is prohibited), *common* (simultaneous sending is permitted only if the same value is sent) *arbitrary* (simultaneous sending is permitted and one of the processors trying to send to the same bus succeeds), and *priority* (simultaneous sending is permitted and the rightmost processor trying to send to the same bus succeeds). The power of these models become stronger as going backward in this order and become more practical as going forward. It has been open whether the models differ properly with respect to the power. In Chapter 2, we present two practical methods to realize a priority bus using common buses. From this result, it should be concluded that the difference of power among common, arbitrary, and priority is not proper.

Since the computation time of many problems depends on that of sorting, it is very important to develop fast sorting algorithms. In Chapter 3, we present the following parallel algorithms on bus-connected processor arrays: for every fixed $\epsilon > 0$, N elements can be sorted in $O(N/W)$ time on a 1-dimensional processor array of size N with W $(W \leq N^{1-\epsilon})$ buses and N^2 elements can be sorted in $O(N/W)$ time on a 2-dimensional processor array of size $N \times N$ with W $(W \leq N^{1-\epsilon})$ buses at each row and each column. Since the computation time of these algorithms attains the obvious lower bound $\Omega(N/W)$, these algorithms are optimal.

The bus system in Chapters 2 and 3 is referred to as a *static bus system* in the sense that the configuration of buses cannot be changed during the execution of the algorithms. *A reconfigurable bus system* is a bus system whose configuration can be dynamically changed.

A reconfigurable array is a processor array that consists of processors arranged to a 2-dimensional grid with a reconfigurable bus system. Since a reconfigurable array is more powerful than a PRAM and more practical, a reconfigurable array becomes the focus of attention. In Chapter 3 we show that for every T, $(1 \leq T \leq \log^* N)$, N elements can be sorted in $O(T)$ time on a reconfigurable array with $N \times N \log^{(T)} N$ processors. This result implies that N elements can be sorted in $\log^* N$ time on a reconfigurable array of size $N \times N$.

Chapter 2, 3, and 4 are based on the results in [1,4], [3,5], and [2,6], respectively.

List of Publications

[1] K. Nakano, T. Masuzawa, and N. Tokura, *Mutual Exclusion for Simultaneous Sending to Buses*, Technical Report SIGAL16–12, IPS Japan, in Japanese (1990–7).

[2] K. Nakano, T. Masuzawa, and N. Tokura, *A Fast Sorting Algorithm on a Reconfigurable Array*, Technical Report COMP90–69, IEICE (1990–12).

[3] K. Nakano, T. Masuzawa, and N. Tokura, *Optimal Sorting Algorithm on Processor Arrays with Multiple Buses*, Technical report COMP 91–7, IEICE (1991–4).

[4] K. Nakano, T. Masuzawa, and N. Tokura, *Methods to realize a Priority Bus System*, Transactions IEICE ,D-I, J74-D-I, 6, pp.345-351, in Japanese (1991–6).

[5] K. Nakano, T. Masuzawa, and N. Tokura, *Fast Sorting on Processor Arrays with Buses*, '91 LA Summer Symposium(1991–7).

[6] K. Nakano, T. Masuzawa, and N. Tokura, *A Sub-logarithmic Time Sorting Algorithm on a Reconfigurable Array*, IEICE Transactions on Communication and Systems, E-74, 11, pp.3894– 3901 (1991-11).

Contents

List of Figures

Chapter 1

Introduction

Most sequential algorithms have been developed on the basis of a RAM (random access machine)[2]. Since a RAM reflects practical sequential computers and is amenable to theoretical analysis, a RAM is regarded as the most suitable theoretical model for sequential algorithms. On the contrary, there is no general consensus concerning the best parallel theoretical model. In fact, many formal models of parallel computation have appeared in the literature. Among many kinds of parallel formal models, a PRAM (parallel random access machine)[5][15] have been studied most frequently and parallel algorithms on it have been actually developed. One of the reason why a PRAM is major is that a PRAM is easy to deal with. However, the parallel algorithms on a PRAM seem to be impractical, because it is considered to be impossible to realize the shared memory employed by a PRAM.

On the other hand, parallel algorithms on processor arrays connected with one-to-one communication links such as a mesh connected machine, a hypercube machine, etc, have been investigated. Since the one-to-one communication link is more practical than the shared memory, these processor arrays are more feasible than a PRAM. Actually, many parallel machines based on mesh and hypercube topologies have been developed. But, non-trivial problems require computation time at least as large as the diameter of the network. Most of the algorithms on processor arrays with one-to-one communication links are rather slower than those on a PRAM. To overcome the limit of computation time bounded by the diameter of networks, processor arrays with capability of one-to-many communication have attracted 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] [9] [21] [31], finding median [31], sorting [21] [31], image processing [9] [21] [29] [32], and component labeling [16] [17] [28] have been efficiently solved.

Several models concerning simultaneous sending to the same bus have been proposed [16]. For example, *exclusive* (simultaneous sending is prohibited), *common* (simultaneous sending is permitted only if the same value is sent), *arbitrary* (simultaneous sending is permitted and one of the processors trying to send to the same bus succeeds), and *priority* (simultaneous sending is permitted and the rightmost processor trying to send to the same bus succeeds) have been proposed. Obviously, the power of these models become stronger as going backward in this order and become more practical as going forward. It has been open whether the differences of the power and the practicalness among these models are proper. One of the main interests of researchers is to develop faster algorithms using less powerful buses.

The bus system mentioned above is *static* in the sense that the configuration of buses cannot be changed during the execution of the algorithms. To speed up the computation, a reconfigurable bus system which has capability of changing bus configuration dynamically has been proposed. Such a dynamic bus system is referred to as *a reconfigurable bus system. A reconfigurable array* is a processor array that consists of processors arranged to a 2-dimensional grid with a reconfigurable bus system. Recently, several algorithms on a reconfigurable array have been investigated. For example, sorting [36] [40], graph problems [35] [38], computational geometry problems [23] [36], image processing [24] [25], basic arithmetic operations [38] [39], generating computation tree forms [37] and simulating the PRAM [34] have been solved. Furthermore, it is known that the reconfigurable bus is at least as powerful as a PRAM if enough processors are available [34]. And there exists a problem which can be solved faster on a reconfigurable array than on a PRAM; the logical exclusive OR of a binary vector of size N can be computed in constant time on a reconfigurable array of size N, while this problem requires $\Omega(\log N/\log\log N)$ time[†] on a PRAM with a polynomial number of processors.

In this dissertation, we will discuss some topics on processor arrays with buses. In Chapter 2, we discuss the feasibility of a bus whose model is priority. In other words, we

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

present two practical methods to realize a priority bus using common buses. Thus, it should be concluded that a priority bus is a practical model. In Chapter 3, we present optimal sorting algorithms on processor arrays with multiple buses. That is, we will show that for every fixed $\epsilon > 0$, N elements can be sorted in $O(N/W)$ time on a 1-dimensional processor array of size N with W $(W \leq N^{1-\epsilon})$ buses and N^2 elements can be sorted in $O(N/W)$ time on a 2-dimensional processor array of size $N \times N$ with W $(W \leq N^{1-\epsilon})$ buses at each row and each column. Since the computation time of these algorithms attains the obvious lower bound $\Omega(N/W)$, these algorithms are optimal. But this algorithm is at most as fast as $O(N^\epsilon)$ time. Even if a polynomial number of processors are available, sorting of N elements requires $\Omega(\log N/\log\log N)$ time not only on a processor array with static buses but also on a PRAM. In order to sort N elements in $o(\log N/\log\log N)$ time, we have to deliver more powerful model than a PRAM. In Chapter 4, we consider a reconfigurable array, which is more powerful than a PRAM but more practical. we will show that for every T, $(1 \leq T \leq \log^* N)$, N elements can be sorted in $O(T)$ time on a reconfigurable array with $N \times N \log^{(T)} N$ processors[‡]. This sorting algorithm is based on an algorithm computing the number of 1's in a given binary sequence. The algorithm to compute the number of 1's is useful for the other problems.

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

Chapter 2

Methods to realize the priority bus

A PRAM employs processors which have capability to access any memory cell in the shared memory(Fig. 2.1). Several models of a PRAM have been proposed with regard to simultaneous reading and writing to the same memory cell as follows:

- EREW (exclusive read exclusive write)
  Both simultaneous reading and writing are prohibited,
- CREW (concurrent read exclusive write)
  Only simultaneous reading is permitted,
- CRCW (concurrent read concurrent write)
  Both simultaneous reading and writing are permitted.

Furthermore, a CRCW model is subdivided as follows:

- common
  All processors trying to write into a same memory cell must be writing the same value,
- arbitrary
  If several processors simultaneously try to write into a same memory one of them succeeds and writes its value, but there is no rule assumed to govern the selection of the successful processor.

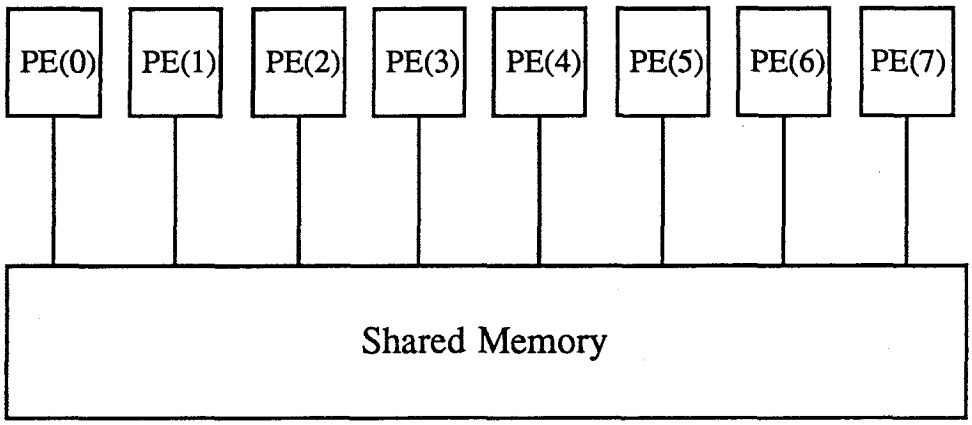


Figure 2.1: PRAM

- priority

  If several processors simultaneously try to write to a same cell, then the lowest-numbered processor among them succeeds.

Obviously, a priority model has the strongest power of the three.

Similarly to simultaneous writing on a PRAM, several bus models have been proposed with regard to simultaneous sending to the same bus as follows:

- exclusive

  Two or more processors cannot send to the same bus.

- common

  All processors trying to send to the same bus must be sending the same value,

- arbitrary

  If several processors simultaneously try to send to the same bus, one of them succeeds and transfers its value, but there is no rule assumed to govern the selection of the successful processor,

- priority

  If several processors simultaneously try to send to the same bus, then the rightmost processor among them succeeds. (the rightmost processor means the processor nearest to one end of the bus.)

A priority model is most powerful and a exclusive model is weakest of the four. Indeed, there is a problem which can be solved faster using an arbitrary model than using a common model[16][17].

In this chapter, we present practical methods to realize (or simulate) communication through a bus whose model is priority. To present feasible methods, we have to be clear what devices are available. We assume that only a bus whose model is common and which can transfer one bit value in a time unit (shortly *a 1-bit common bus*) is only available as a communication device. This assumption is reasonable because it is not so difficult to develop a 1-bit common bus by recent technology.

It is easy to realize a w-bit common bus (i.e. a bus whose model is common and which can transfer w bits value in a time unit). If w 1-bit common buses are arranged, we can realize w-bit common bus; to transfer each bit of w bits value, a 1-bit common bus is used. Since the constraint of a w-bit common bus concerning simultaneous sending holds, different values are never sent to each 1-bit common bus. On the other hand, if we use this method to realize a w-bit priority bus (i.e. a bus whose model is priority and which can transfer w bits value in a time unit), different values may be sent to a 1-bit common bus. This does not meet the constraint of a common model. To realize a w-bit priority bus, we will show methods to find the rightmost processor among processors trying to send to the bus using 1-bit common buses: after finding the rightmost processor, it actually sends the data. For example, the following method will be the first idea: the rightmost processor trying to send to the bus can be found in $O(\log n)$ time using a 1-bit common bus by means of a binary search method, where n is the number of processors connected with the bus. By this method, a w-bit priority bus can be realized in $O(\log n)$ time. But, since most algorithms on processor networks with buses take logarithmic or sub-logarithmic time, the overhead of the logarithmic factor is fatal. It is important to realize a w-bit priority bus in low constant time even if extra 1-bit common buses are used.

In this chapter, we will present two methods to realize a w-bit priority bus in low constant time. In these methods, to find the rightmost processor, 1-bit common buses arranged to $O(\log n)$ layers are used. The arrangements are called *a binary tree layout* (Fig. 2.3)and *an extended binary tree layout*(Fig. 2.4), respectively. In Section 2.2, we will show that the rightmost processor can be found in one time unit on the binary tree layout. But in this method, each processor has to send to $\log n$ buses simultaneously. In Section 2.3, we will

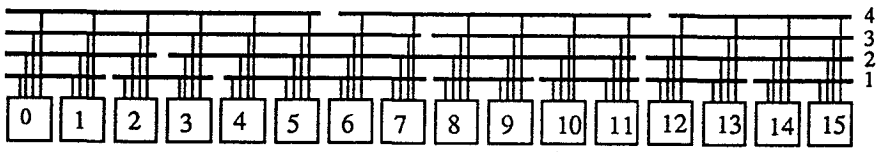


Figure 2.2: A 1-dimensional processor array with buses

improve this method and show that the rightmost processor can be found in low constant time on the extended binary tree layout. In this method, each processor will access at most two buses in a time unit. We will compare these two methods in Section 2.4.

2.1 Model and definition

A 1-dimensional processor array with buses consists of processors PE(0), PE(1), ..., PE($n-1$) arranged to a 1-dimensional grid with multiple buses. Figure 2.2 illustrates an example of a processor array with buses. On processor arrays, each processor is a RAM extended by communication commands and works synchronously. Assume that only 1-bit common buses are arranged to a processor array. As shown in Fig 2.2, buses on a processor array arranged to layers. Layers are called the 1st layer, the 2nd layer, ..., from the bottom. In Fig. 2.2, the 3rd layer consists of two buses: a bus over PE(0), ..., PE(7) and a bus over PE(8), ... PE(15). These buses are denoted as $[0, 7]$ and $[8, 15]$, respectively. Furthermore, buses on a layer is denoted as a set of buses. For example the buses on the 3rd layer in Fig. 2.2 are denoted as $\{[0, 7], [8, 15]\}$.

In Sections 2.3 and 2.4, in order to simulate communication through a priority bus by 1-bit common buses, we will present two methods to find the rightmost processor among processors trying to send to a priority bus. To formalize finding the rightmost processor, we define *the rightmost finding* as follows:

Definition 1 (the rightmost finding)

Input. Let $(inp(0), inp(1), \ldots inp(n-1))$ be an n bit binary vector. Each $inp(i)$ $(0 \leq i \leq n-1)$ is given to PE(i). This means that PE(i) try to send data to a priority bus iff $inp(i) = 1$.

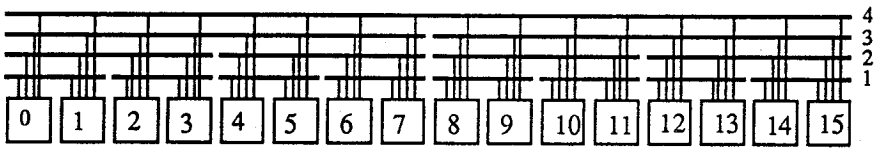


Figure 2.3: Binary tree layout

Output. Let $(out(0), out(1), \ldots, out(n-1))$ be an n bit binary vector defined as follows:

$$out(i) = \begin{cases} 1 & \text{if } i = \max\{k | inp(k) = 1\} \\ 0 & \text{otherwise} \end{cases}$$

Each PE(i) knows the value of $out(i)$. Furthermore, PE(i) such that $out(i) = 1$ is called *the rightmost processor.* □

2.2 Method 1 for rightmost finding

Figure 2.3 illustrates a processor array with buses arranged to *a binary tree layout.* In this section, we will show a method to find the rightmost processor in a time unit on processor arrays with 1-bit common busses arranged to the binary tree layout. The binary tree layout can be defined formally as follows:

Definition 2 (binary tree layout)

- the bus layout of the jth layer $1 \leq j \leq \log n$ is:

$$\{[0, 2^j - 1], [2^j, 2 \cdot 2^j - 1], \ldots, [n - 2^j, n - 1]\}.$$

- Each processor is connected to $\log n$ buses over it on all layers *.

□

On the binary tree layout, concerning the bus $[s,t]$ which connects PE(s), PE($s+1$),..., PE(t), we call that PE(s), PE($s+1$),..., PE($(s+t-1)/2$) is connected by *the left side of the bus* $[s,t]$ and PE($(s+t+1)/2$),..., PE(t) is connected by *the right side.*

*Throughout this dissertation, for cleaner presentation, we will omit the floor or ceiling operators necessary to ensure that all values are integers.

We now present a rightmost finding algorithm on the binary tree layout by means of *a parallel binary search*. The similar algorithm on the PRAM was presented in [10].

The idea of the rightmost finding algorithm is as follows. Consider the bus, $[0, n-1]$, on the $\log n$th layer. PE(i) connected by the right side of $[0, n-1]$ sends 1 to $[0, n-1]$ if $inp(i) = 1$. And PE(i) connected by the left side of $[0, n-1]$ tries to receive data from $[0, n-1]$. If the processors connected by the left side receives 1, then these processors will know that they are not the rightmost processor. Otherwise, these processors can know that they are candidates of the rightmost processor. Assume that there is a processor PE(i) ($n/2 \leq i \leq n-1$) connected by the right side of $[0, n-1]$ and whose input value, $inp(i)$, is 1. Then by using the bus $[n/2, n-1]$, the processors can find whether the rightmost processor belongs to $[n/2, 3n/4-1]$ or $[3n/4, n-1]$ similarly. By iterating similarly, the rightmost processor can be found in $\log n$ steps. Since this method takes $O(\log n)$ time, we reduce the computation time by simultaneous communication as follows:

[Method 1] PE(i) with $inp(i) = 0$ does nothing in this algorithm. If $inp(i) = 1$ then PE(i) executes the following steps.

Step 1 For each bus on the jth layer ($1 \leq j \leq \log n$), executes the following commands simultaneously.

1. If the processor is connected by the right side of the bus, it sends 1 to the bus.
2. If the processor is connected by the left side of the bus, it tries to receive data from the bus.

Step 2 If PE(i) receives no data from every bus in Step 1, then PE(i) lets $out(i) := 1$. Otherwise lets $out(i) := 0$. □

In this algorithm, each processor executes $\log n$ communication commands simultaneously. We have:

Theorem 2.1 *Method 1 finds the rightmost processor in a time unit on the binary tree layout.*

Proof. Let $d = \log n$ and we prove the theorem by induction on d. If $d = 1$, the rightmost processor can be found obviously. We assume that the rightmost processor can be found in case $d-1$, and prove that the rightmost finding can be completed in case d.

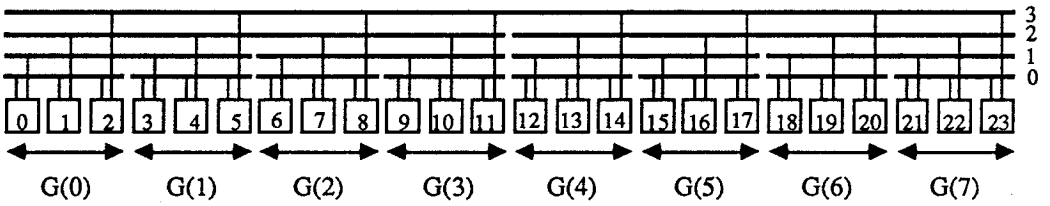


Figure 2.4: Extended binary tree layout

The binary tree layout of size n consists of two binary tree layout of size $n/2$ and a bus $[0, n]$. Let PE(r) be the rightmost processor. If $0 \leq r \leq n/2-1$, the rightmost processor can be found correctly on the left binary tree layout of size $n/2$ from the induction hypothesis. If $n/2 \leq r \leq n-1$, processors connected by the left side of $[0, n-1]$ receives 1 from PE(r) through the bus $[0, n-1]$, PE(i) $(0 \leq i \leq n/2-1)$ cannot become the rightmost processor. Furthermore, from the induction hypothesis, the rightmost processor can be found on the right binary tree layout of size $n/2$. This completes the proof. □

2.3 Methods 2 for rightmost finding

In this section, we present another rightmost finding algorithm. In the previous algorithm, each processor has to execute communication commands simultaneously. In the algorithm in this section, each processor execute at most one send and one receive command in a time unit. This algorithm is executed on *the extended binary tree layout* depicted by Fig. 2.4.

Definition 3 (extended binary tree layout)

- For simplicity, processors on the extended binary tree layout are indexed as follows:

  $$\text{PE}\langle i, j\rangle \qquad (0 \leq i \leq 2^d - 1, 1 \leq j \leq d)$$

  where d is an integer such that $n = d \cdot 2^d$. In other words, PE$\langle i, j\rangle$ corresponds to PE$(i \cdot d + j - 1)$.

- The bus layout of the kth layer $(0 \leq k \leq d)$ is

  $$\{[0, d \cdot 2^k - 1], [d \cdot 2^k, 2 \cdot d \cdot 2^k - 1], \ldots [n - d \cdot 2^k, n - 1]\}.$$

  Note that layers are indexed by $0, 1, \ldots, d$, for simplicity.

- Each PE$\langle i, j\rangle$ is connected by two buses on the 0th layer and the jth layer only. □

On the extended binary layout, for each i $(0 \leq i \leq 2^d - 1)$, $\mathrm{PE}\langle i,1\rangle, \mathrm{PE}\langle i,2\rangle, \ldots, \mathrm{PE}\langle i,d\rangle$ forms *the ith group* denoted by $G(i)$. If we consider that processors in a group merged into a processor, the extended binary tree layout can be regarded as a binary tree layout.

Consider a bus $[s,t]$ on the jth layer on the extended bus layout. This bus connects $\mathrm{PE}\langle s,j\rangle, \mathrm{PE}\langle s+1,j\rangle, \ldots, \mathrm{PE}\langle t,j\rangle$. Similarly to the binary tree layout, $\mathrm{PE}\langle s,j\rangle, \mathrm{PE}\langle s+1,j\rangle, \ldots, \mathrm{PE}\langle (s+t-1)/2,j\rangle$ are connected by the left side of the bus and $\mathrm{PE}\langle (s+t+1)/2,j\rangle, \ldots, \mathrm{PE}\langle t,j\rangle$ are connected by the right side of the bus.

First, we will show the outline of the rightmost finding method on the extended binary tree layout.

[Outline of Method 2]

Step 1 For each group $G(i)$, determine whether there exists a processor in the group whose input value is 1. In other words, for each i $(0 \leq i \leq 2^d - 1)$

$$u(i) \quad = \quad inp\langle i,1\rangle \vee inp\langle i,2\rangle \vee \cdots \vee inp\langle i,d\rangle$$

is computed where $inp\langle i,j\rangle$ denotes input variable of $\mathrm{PE}\langle i,j\rangle$.

Step 2 Find the rightmost element of $u(0), u(1), \ldots, u(2^d - 1)$ whose value is 1. In other words, compute

$$R \quad = \quad \max\{s | u(s) = 1\}.$$

After Step 2, the rightmost processor belongs to $G(R)$.

Step 3 Find the rightmost element among input values $inp\langle R,1\rangle$, $inp\langle R,2\rangle$, $\ldots$, $inp\langle R,d\rangle$. In other words, compute

$$r \quad = \quad \max\{s | inp\langle R,s\rangle = 1\}$$

After Step 3, $\mathrm{PE}\langle R,r\rangle$ is the rightmost processor. □

Step 1 can be easily performed by buses on the 0th layer. Similarly to Theorem 1, Step 2 can be performed by the parallel binary search method. Step 3 can be performed as follows: Assume that $inp\langle R,j_1\rangle = inp\langle R,j_2\rangle = \cdots = inp\langle R,j_k\rangle = 1$ $(j_1 < j_2 < \cdots < j_k)$, and the other input values are 0. Then $\mathrm{PE}\langle R,j_k\rangle$ is the rightmost processor. Let $B(j_1), B(j_2), \ldots, B(j_k)$ be buses which connects $\mathrm{PE}\langle R,j_1\rangle, \mathrm{PE}\langle R,j_2\rangle, \ldots, \mathrm{PE}\langle R,j_k\rangle$ on the

j_1th, j_2th, ... j_kth layers, respectively. Concerning these buses, $B(j_k)$ is longest, and includes $B(j_1), B(j_2), \ldots, B(j_{k-1})$. From this fact $B(j_k)$ can be found, and then the rightmost processor PE$\langle R, j_k\rangle$ can be computed.

We now show the method on the extended binary tree layout precisely.

[Method 2] Each PE$\langle i,j\rangle$ employs local variables $u(i), v\langle i,j\rangle, w\langle i,j\rangle$, and $w(i)$. Initially each variable takes value 0.

Step 1 For each PE$\langle i,j\rangle$, if $inp\langle i,j\rangle = 1$, PE$\langle i,j\rangle$ sends 1 to the bus on the 0th layer. Each PE$\langle i,j\rangle$ tries to receive data from the bus on the 0th layer, and if it receives 1, then let $u(i) := 1$, otherwise $u(i) := 0$.

Step 2 Each PE$\langle i,j\rangle$ with $u(i) = 1$ executes the following substeps. If $u(i) = 0$, PE$\langle i,j\rangle$ skips Step 2.

1. Each PE$\langle i,j\rangle$ connected by the right side of the bus on the jth layer, sends 1 to the bus. Each PE$\langle i,j\rangle$, which is connected by the left side of the bus on the jth layer, receives data from the bus. If PE$\langle i,j\rangle$ receives 1, $v\langle i,j\rangle$ will be set to 1, otherwise set to 0.
2. If $v\langle i,j\rangle = 1$, PE$\langle i,j\rangle$ sends 1 to the bus on the 0th layer. Each PE$\langle i,j\rangle$ tries to receive data from the bus on the 0th layer, if PE$\langle i,j\rangle$ cannot receive 1 from the bus, $u(i)$ is the rightmost element taking value 1 among $u(0), u(1), \ldots, u(2^d - 1)$. Let R be the index such that $u(R)$ is the rightmost element.

Step 3
1. Each PE(R,k) belonging to $G(R)$ sends 1 to the bus on the kth layer if $inp\langle R,k\rangle = 1$. Every processor PE$\langle i,j\rangle$ tries to receive data from the bus on the jth layer. If it receives 1 then $w\langle i,j\rangle := 1$, otherwise $w\langle i,j\rangle := 0$
2. If $w\langle i,j\rangle = 1$, PE$\langle i,j\rangle$ sends 1 to the bus on the 0th layer. Every processor PE$\langle i,j\rangle$ tries to receive data from the bus on the 0th layer, and if it receives 1 then $w(i) := 1$. In other words, compute

$$w(i) \quad := \quad w\langle i,1\rangle \vee w\langle i,2\rangle \vee \cdots \vee w\langle i,d\rangle$$

3. Find the rightmost element taking value 1 among $w(0)$, $w(1)$, ..., $w(2^d - 1)$. This can be performed by the similar scheme to Step 2. Then, let $w(RG)$ be the rightmost element, that is, $RG = \max\{i | w(i) = 1\}$.

4. Find the leftmost element taking value 1 among $w(0), w(1), \ldots, w(2^d - 1)$. This can be performed by the similar scheme to Step 2. Then let $w(LG)$ be the leftmost element, that is, $LG = \min\{i | w(i) = 1\}$.

5. Each $\mathrm{PE}\langle RG, j\rangle$ $(1 \leq j \leq d)$ connected by the right side of the bus on the jth layer sends 1 to this bus. Each $\mathrm{PE}\langle R, k\rangle$ $(1 \leq k \leq d)$ tries to receive data from the bus on the kth layer.

6. Each $\mathrm{PE}\langle LG, j\rangle$ $(1 \leq j \leq d)$ connected by the left side on the jth layer send 1 to this bus. Each $\mathrm{PE}\langle R, k\rangle$ $(1 \leq j \leq d)$ tries to receive data from the bus on the kth layer.

7. The processor succeeding to receive 1 in both 5. and 6. is the rightmost processor. Let this processor be $\mathrm{PE}(R, r)$. Let $out(R, r) := 1$ and the other output values $out(i, j)$ be 0.

The following theorem holds:

Theorem 2.2 *Method 2 finds the rightmost processor in constant time on the extended binary tree layout.*

Proof. Each step in Method 2 can be performed in constant time. So we just prove the correctness of the method. Similarly to Method 1, it is easy to prove that $G(R)$ includes the rightmost processor. Hence, we will prove that the $\mathrm{PE}\langle R, r\rangle$ is exactly the rightmost processor.

As said before, let $inp\langle R, j_1\rangle = inp\langle R, j_2\rangle = \cdots = inp\langle R, j_k\rangle = 1$ $(j_1 < j_2 < \cdots < j_k)$ and the other input values be 0. Let $B(j_1), B(j_2), \ldots, B(j_k)$ be buses which connects $\mathrm{PE}(R, j_1), \mathrm{PE}(j_2), \ldots, \mathrm{PE}(j_k)$ on the $j_1, j_2, \ldots j_k$th layers, respectively. Let s and t be indices such that $\mathrm{PE}\langle s, j_k\rangle, \mathrm{PE}\langle s+1, j_k\rangle, \ldots, \mathrm{PE}\langle t, j_k\rangle$ are connected by $B(j_k)$. Then,

$$w(s) = w(s+1) = \cdots = w(t)$$

holds, and the other values of w's are 0. Hence, $s = LG$ and $t = RG$ hold. $\mathrm{PE}\langle RG, j_k\rangle$ and $\mathrm{PE}\langle LG, j_k\rangle$ are connected by the right side of $B(j_k)$ and the left side of $B(j_k)$, respectively. Therefore, $\mathrm{PE}\langle R, j_k\rangle$ receives 1 in both 5. and 6. Since $\mathrm{PE}\langle LG, j\rangle$ and $\mathrm{PE}\langle RG, j\rangle$ are connected by the same side of the bus on the jth layer when $j > j_k$, $\mathrm{PE}\langle R, j\rangle$ can receive 1 only once in 5. and 6. Since $\mathrm{PE}\langle LG, j\rangle$ and $\mathrm{PE}\langle RG, j\rangle$ are connected by the different bus

on the jth layer when $j < j_k$, PE$\langle R, j\rangle$ cannot receive 1 in both 5. and 6. Therefore, only PE$\langle R, j_k\rangle$ receives 1 in both 5. and 6. This completes the proof. □

2.4 Comparison between Methods 1 and 2

The w-bit priority bus can be realized by w 1-bit common buses and the bus layout for rightmost finding. The bus layout for rightmost finding needs $O(\log n)$ layers in both Methods 1 and 2. Hence, the w-bit priority bus can be realized on the common bus layout with $w + O(\log n)$ layers in constant time. Therefore, if $w = \Omega(\log n)$, the w-bit priority bus can be realized on the common buses with increasing constant factors of hardware and time complexity. This assumption is acceptable because most of the algorithms on the bus network presented in the literature require buses sending $\log n$ bits in a time unit.

Let us compare Methods 1 and 2. In Method 1, each processor is connected by $\log n$ buses and executes send or receive commands to each bus simultaneously. In Method 2, each processor is connected by at most two buses and executes send or receive commands to at most one bus in a time unit. Though Method 1 completes the rightmost finding in a time unit, Method 2 requires approximately 10 steps.

Let us consider that a processor network with several priority buses are realized by Method 1 and 2, respectively. In this network, some processors are connected by several priority buses. Focus on a processor which is connected by k priority buses. To realize communication through k priority buses by Method 1 and 2, this processor is connected k binary tree layouts and k extended binary tree layouts, respectively. If we use Method 1, the processor has to execute communication commands to the binary layouts corresponding to priority buses to which the processor sends. If we use Method 2, the processor has to execute communication commands to all the extended binary layouts, even if the processor executes no communication command to the priority buses. In other words, in Method 2, the processor has to execute k communication commands to 1-bit common buses simultaneously. Consequently, in both Method 1 and 2, some kinds of parallel execution within a processor are required.

But, in both Method 1 and 2, the algorithm for each processor can be expressed by a small finite state machine. Hence, the priority bus can be realized in constant time using 1-bit common buses, as long as a small circuit to execute the rightmost finding algorithm is

added to every input/output port.

Since Method 1 and 2 solve the rightmost finding problem, an arbitrary bus can be realized similarly by these methods.

2.5 Concluding remarks

In this chapter, we have presented feasible methods to realize communication through priority buses by 1-bit common buses. It is known [5][6] that the graph connectivity problem can be solved on a 2-dimensional processor array with buses arranged to each row and column as follows:

1. If the buses are common, it can be solved in $O(\log^2 n)$ time,
2. If the buses are common, it can be solved in $O(\log n)$ expected time,
3. If the buses are arbitrary, it can be solved in $O(\log n)$ time.

By applying our methods to the third algorithm, the graph connectivity problem can be solved in $O(\log n)$ time in spite of using common buses.

Chapter 3

Sorting on processor arrays with buses

Sorting is one of the most fundamental problems with rich theory and wide practical applications. It is widely known in [2] that any sequential sorting requires $\Omega(N \log N)$ time and there exist optimal sequential sorting algorithms. To speed-up sorting, parallel sorting algorithms have been investigated [4]. For example, both AKS sorting network [3] [15] and Cole's optimal merge sort [11] [15], sort N elements in $O(\log N)$ time using N processors. Since the product of time and the number of processors is equal to the time complexity of the best known sequential algorithm, these algorithms are called *optimal speed-up*. But they seem to be impractical, because AKS sorting network has a large constant factor, and Cole's optimal merge sort is executed on a shared memory machine, a PRAM.

On the other hand, more feasible algorithms on practical parallel machines have been studied. We focus on the processor array where processors are arranged to a 1-dimensional or a 2-dimensional grid. Let *MESH(N)* denote a processor array which consists of N processors arranged to a 1-dimensional grid with neighbor links(Fig. 3.1). Each processor can communicate with its neighbors via neighbor links. Let *MESH($N \times N$)* denote a processor array which consists of $N \times N$ processors arranged to a 2-dimensional grid with neighbor links(Fig. 3.2). It is known that N elements given one at each processor on MESH(N) can be sorted in $O(N)$ time by *the odd-even transposition sort* [8] which is a parallel version of the serial *bubble sort*. The odd-even transposition sort is optimal because the time complexity is

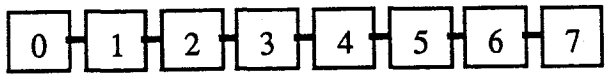


Figure 3.1: MESH(8)

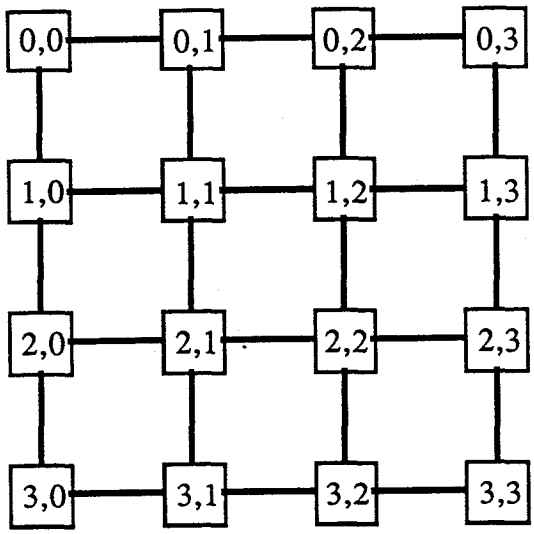


Figure 3.2: MESH(4×4)

equal to the obvious lower bound obtained from the diameter of MESH(N). Many sorting algorithms have been presented which sort N^2 elements in $O(N)$ time on MESH($N \times N$) [33]. Similarly, these algorithms are optimal. The research target is to reduce the constant factor of the time complexity [30].

In order to speed up sorting, we consider processor arrays with multiple buses. Let *GRID(N, W)* denote a processor array which consists of N processors arranged to a 1-dimensional grid with W buses(Fig. 3.3). And let *GRID($N \times N, W$)* denote a processor array which consists of $N \times N$ processors arranged to a 2-dimensional grid with W buses at each row and at each column(Fig. 3.4). We will present the following sorting algorithms on these processor arrays:

- For every fixed $\epsilon > 0$, N elements can be sorted in $O(N/W)$ time on GRID(N, W) if $W \leq N^{1-\epsilon}$.

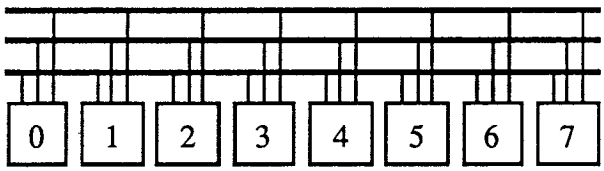


Figure 3.3: GRID(8, 3)

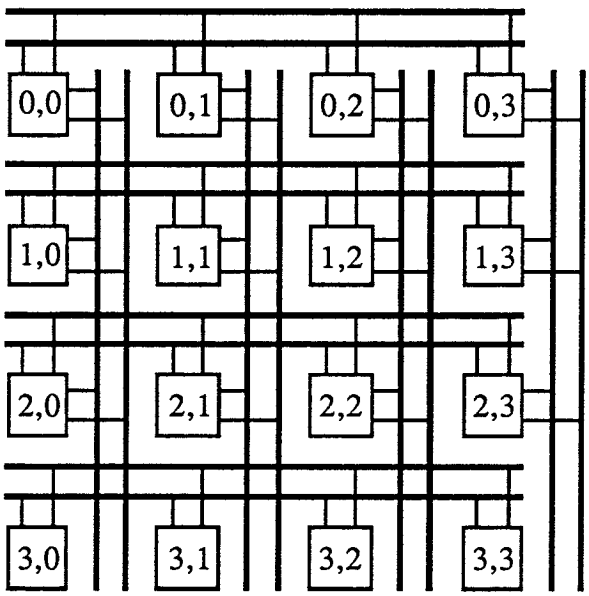


Figure 3.4: GRID($4 \times 4, 2$)

- For every fixed $\epsilon > 0$, N^2 elements can be sorted in $O(N/W)$ time on GRID($N \times N, W$) if $W \leq N^{1-\epsilon}$.

These algorithms are optimal because the time complexity is equal to the obvious lower bound $\Omega(N/W)$.

It is known [13] that N elements can be sorted in $O(N/W + N^{3/4} \log^{1/4} N)$ time on GRID(N, W) with neighbor links, i.e., on a 1-dimensional processor array with communication capability of both GRID(N, W) and MESH(N). Recently, the computation time has been improved to $O(N/W + N \log N)$ [14] without neighbor links. This algorithm is optimal for $W = O(N^{1/4}/\log^{1/4} N)$. Hence our algorithm is optimal for a wider range of W. Furthermore a randomized algorithm which sorts N^2 elements in $O(N)$ time on

GRID($N \times N, 1$) was presented [18]. Since our sorting algorithm sorts N^2 elements in $O(N)$ time on GRID($N \times N, 1$) deterministically, thereby our algorithm sort faster than this algorithm in the worst case.

The optimal sorting algorithm on a 1-dimensional processor array with complicated bus layout was presented in [27]. This sorting algorithm is optimal if the number of layers is at most $O(N/\log^2 N)$. Therefore, if a bus layout suitable for sorting is available, sorting can be performed optimally with the larger number of layers. But if enough large number of layers is not available, our sorting algorithm performs sorting optimally despite using a simple bus layout.

The sorting algorithms we will present are based on Leighton's *Columnsort* [22]. Columnsort sorts the elements of a matrix by the following operations to the matrix.

- Sort the elements within each column or within each column except the leftmost column;
- Permute the elements along a certain permutation.

These two operations are repeated alternately constant times. In Section 3.2, we will describe Columnsort precisely. In Section 3.3, we will present the two basic algorithms: an $O(N)$ time sorting algorithm on GRID($N, 1$) and an $O(N/W)$ time permutation routing algorithm on GRID(N, W). These algorithms are used for applying Columnsort to GRID(N, W) and GRID($N \times N, W$). Sections 3.4 and 3.5 give the new sorting algorithms on GRID(N, W) and GRID($N \times N, W$), respectively.

3.1 Model

The processors on GRID(N, W) and GRID($N \times N, W$) are indexed by PE(0), PE(1) ,..., PE($N-1$) and PE$(0,0)$, PE$(0,1)$, ..., PE$(N-1, N-1)$, respectively(Figs. 3.3 and 3.4). Each processor is a uniform-cost random access machine (RAM) with usual operations and instructions. The arithmetic operations (addition and subtraction), bitwise logical operations, equality predicate and so on require constant time. The processors execute the same program synchronously. Although performing the same instructions, processors can work on different data. In one step each processor can access one bus. As mentioned in Chapter 2, models differ in simultaneous access of the same bus by two or more processors.

In this chapter we use exclusive buses.

3.2 Columnsort

Sorting algorithms presented in this chapter are based on a simple sorting algorithm called *Columnsort*. In this section, we describe Columnsort. Though our description of Columnsort is different from Leighton's on a minor point, they are same essentially.

Let A be an $r \times s$ matrix of elements where $s|r$ (r is divisible by s) and $r \geq 2(s-1)^2$. The (i,j) element of A is denoted as $a(i,j)$. Columnsort is described by simple operations to the matrix A. After completion of Columnsort, the (i,j) element $(0 \leq i < r, 0 \leq j < s)$ of A will contain the $i+jr$th sorted element of A. In other words, Columnsort sorts the elements of A in column major order.

Columnsort takes eight steps. Let the jth column of A be $A_j = a(0,j), a(1,j), \ldots, a(r-1,j)$. In Steps 1, 3 and 5 the elements within each column of A are sorted. In Step 7, the elements within each column of A except the leftmost column A_0 are sorted. In Steps 2, 4, 6 and 8, the elements of A are permuted. The permutation in Step 2 corresponds to *transposing* of A(Fig. 3.5). The elements are picked up column by column and then deposited row by row. More precisely, transposing is represented by a one-to-one mapping $trans : \{0,\ldots,r-1\} \times \{0,\ldots,s-1\} \rightarrow \{0,\ldots,r-1\} \times \{0,\ldots,s-1\}$ defined as follows:

$$trans(i,j) = (\lfloor (i+jr)/s \rfloor, i \bmod s).$$

In Step 2, the (i,j) element, $a(i,j)$, is transferred to the $trans(i,j)$ element. The permutation in Step 4 corresponds to *untransposing*, the inverse of transposing. Step 6 involves a cyclic shift of A for $\lfloor r/2 \rfloor$ positions along column major order. That is, each $i+jr$th element is transferred to the $(i+jr+\lfloor r/2 \rfloor) \bmod rs$th element. Hence, the permutation in Step 6 corresponds to *shifting* of A defined as follows(Fig. 3.6).

$$shift(i,j) = ((i+\lfloor r/2 \rfloor) \bmod r, (\lfloor (i+\lfloor r/2 \rfloor)/r \rfloor + j) \bmod s).$$

The permutation in Step 8 corresponds to *unshifting*, the inverse of shifting. Summarize Columnsort as follows:

[Columnsort]

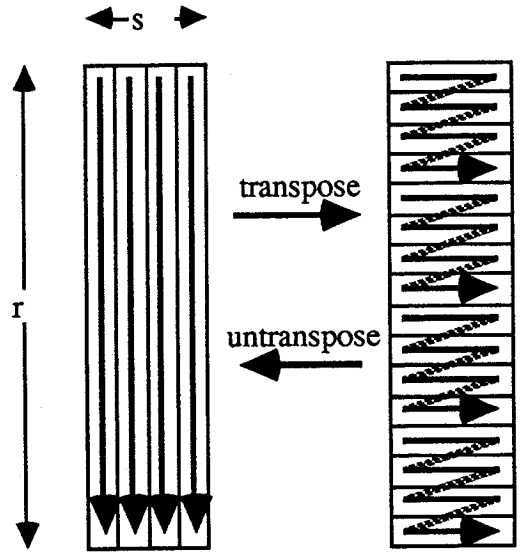


Figure 3.5: Transpose and untranspose

Step1 Sort the elements within each column of A.

Step2 Transpose the elements of A.

Step3 Sort the elements within each column of A.

Step4 Untranspose the elements of A.

Step5 Sort the elements within each column of A.

Step6 Shift the elements of A.

Step7 Sort the elements within each column of A except the leftmost column.

Step8 Unshift the elements of A. □

An example of the execution process of Columnsort on 4×16 for a zero-one input is depicted in Fig. 3.7. Though this example does not satisfy the constraint that $r \geq 2(s-1)^2$, it illustrates the essence of Columnsort. The following theorem holds.

Theorem 3.1 [22] *Columnsort sorts the elements of a matrix A if $s|r$ and $r \geq 2(s-1)^2$ hold.*

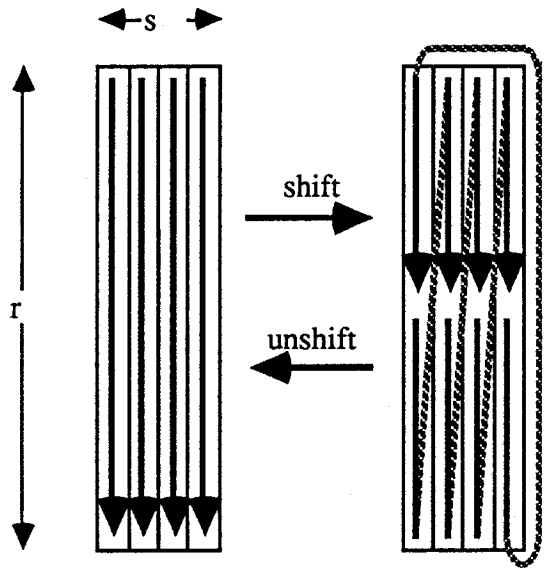


Figure 3.6: Shift and unshift

Proof. Though Leighton have proved the correctness of Columnsort [22], we show a proof of the correctness by zero-one principle [15] [20] which is simpler than Leighton's one. In other words, we will prove Columnsort sorts A whose elements are either 0 or 1. After Step 1, each column begins with 0's followed by 1's. Then after transposing, the element within each column are transferred to an $r/s \times s$ sub-matrix. In each sub-matrix, the upper rows are filled with 0, the lower rows are filled with 1 and at most one row consists of both 0 and 1. Then after Step 3, the upper rows of A are filled with 0, the lower rows are filled with 1 and at most s rows contain both 0 and 1. In the s rows, each row begins with 0's followed by 1's and each column begins with 0's followed by 1's. Since $r \geq 2(s-1)^2$ holds, the elements in the s rows are transferred to at most two column by untransposing. In other words, after Step 4, the left columns of A are filled with 0, the right columns of A are filled with 1, and at most two columns between them consist of both 0 and 1. Suppose there are exactly two columns which consist of both 0 and 1. The right column of them contains 0. Thus, in the s rows before untransposing, each row which is transferred to the left column of them by untransposing contains 0. Therefore, the left column of them contains at most $(s-1)^2$ 1's. Similarly, the right column of them contains at most $(s-1)^2$ 0's. After Step 5, each column of length r begins with 0's followed by 1's. From $r \geq 2(s-1)^2$, after shifting at most one column except the leftmost column contains both 0 and 1. After Step 7, the

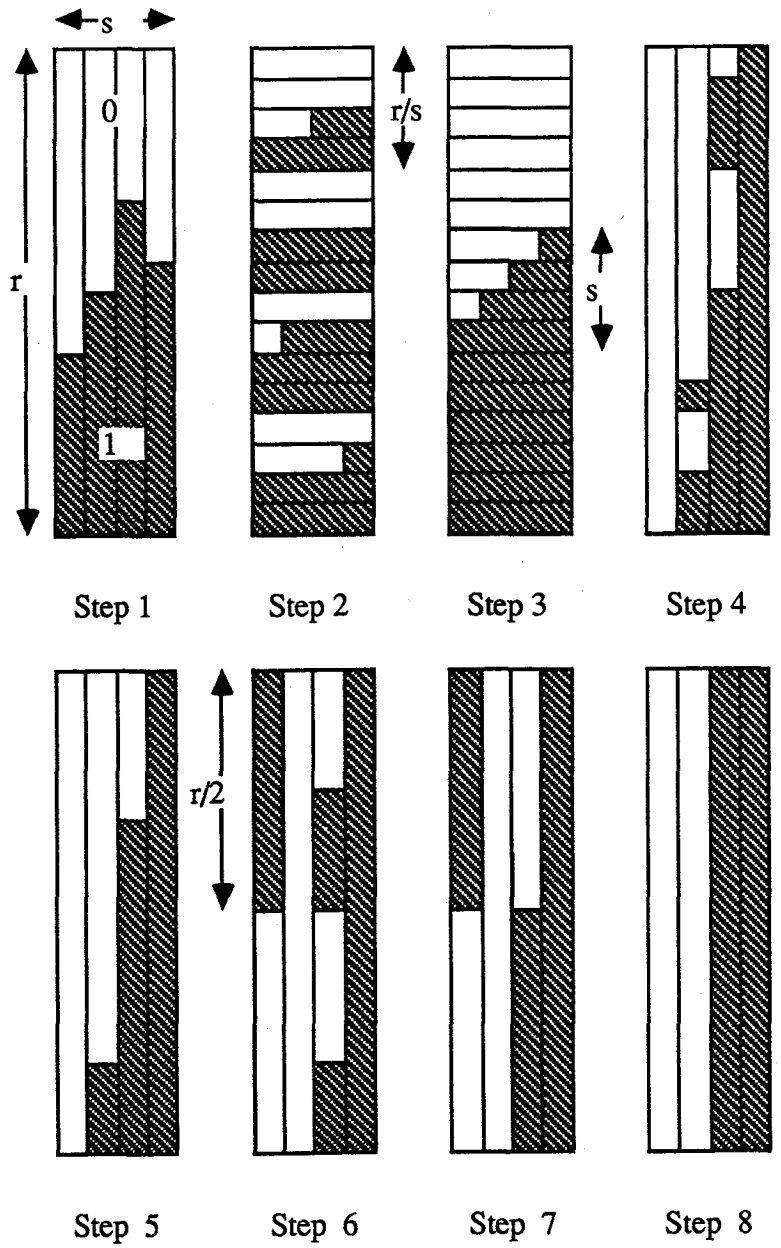


Figure 3.7: An execution process of Columnsort

elements except the leftmost column are already sorted. By unshifting, all the elements of A are sorted. □

Assume that $s|r$ does not hold. If there are exactly two columns which contain both 0 and 1 after Step 4, the left column of them contains at most $s(s-1)$ 1's and the right column contains at most $s(s-1)$ 0's. To sort the elements, the constraint $r \geq 2s(s-1)$ is sufficient. Hence, the following corollary holds.

Corollary 3.2 *Columnsort sorts the elements of a matrix A if $r \geq 2s(s-1)$.* □

Therefore, the constraint $s|r$ is not essential for Columnsort.

3.3 Basic algorithms

Before showing the sorting algorithm on GRID(N, W), we present basic algorithms used for the sorting algorithm. Sorting on GRID(N, W) is defined precisely as follows:

[Sorting on GRID$(N, 1)$]

INPUT Let $A = a(0), a(1), \ldots, a(N-1)$ be a sequence of elements. Each $a(i)$ is given to PE(i).

OUTPUT Each PE(i) knows the ith element of the sorted sequence of A.

First, we show a simple parallel sorting algorithm on GRID$(N, 1)$ by *an enumeration scheme* [8]. Sorting on GRID$(N, 1)$ is used for sorting the elements within each column in Columnsort.

[Sorting algorithm on GRID$(N, 1)$] Let $c(i)$ be a local memory cell of PE(i). After completion of the algorithm, the rank of $a(i)$ is stored to $c(i)$. Initially, $c(i) = 0$.

Step1 Step 1 consists of N sub-steps. In each ith sub-step ($i = 0, 1, \ldots, N-1$), PE(i) sends $a(i)$ to the bus and all the processors receive it from the bus. During each sub-step, when receiving $a(i)$, PE(j) compares $a(i)$ and $a(j)$. If $(a(i), i) < (a(j), j)$ holds in lexicographical order, PE(j) increases the value of $c(j)$ by 1.

Step2 Step 2 consists of N sub-steps. In each ith sub-step ($i = 0, 1, \ldots, N-1$), if $c(j) = i$, PE(j) sends $a(j)$ to PE(i) via the bus, and PE(i) memorizes $a(j)$, the ith sorted elements of A. □

Obviously, the following lemma holds.

Lemma 3.3 *N elements can be sorted in $O(N)$ time on GRID(N, 1).* □

Second, we consider *the permutation routing* on GRID(N, W) defined as follows.

[Permutation routing on GRID(N, W)]

INPUT Let $p : \{0, 1, \ldots, N-1\} \rightarrow \{0, 1, \ldots, N-1\}$ be a one-to-one mapping and $A = a(0), a(1), \ldots, a(N-1)$ be a sequence of elements. Each $a(i)$ and $p(i)$ are given to PE(i).

OUTPUT Each PE$(p(i))$ knows $a(i)$.

The permutation routing algorithm is used for implementation of transposing and untransposing in Columnsort. Permutation routing can be performed on GRID(N, W) efficiently as follows.

[Permutation routing algorithm on GRID(N, W)] The algorithm has $\lceil N/W \rceil$ steps. In the ith step $(0 \leq i < \lceil N/W \rceil)$, for all $iW \leq j < (i+1)W$, PE(j) with $p(j) = i$ sends $a(j)$ to PE(i) in parallel. Then, after completion of the whole steps, each PE$(p(i))$ knows $a(i)$. □

Obviously, we have,

Lemma 3.4 *Any permutation routing on GRID(N, W) can be performed in $O(N/W)$ time.* □

It can be considered that Step 2 of the sorting on GRID$(N, 1)$ executes the permutation routing along the permutation c.

In this section, we have presented the sorting algorithm on GRID$(N, 1)$ and the permutation algorithm on GRID(N, W). Main sorting algorithms presented in this chapter exploit these algorithms.

3.4 Sorting on GRID(N, W)

Sorting on GRID(N, W) can be performed efficiently by adapting Columnsort with a matrix of size $N/W \times W$. It is assumed that N/W is an integer. Each PE$(i + jN/W)$ $(0 \leq i < N/W, 0 \leq j < W)$ maintains the (i, j) element of the matrix in Columnsort.

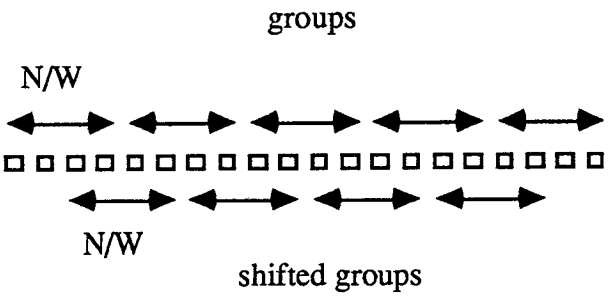


Figure 3.8: Groups and shifted groups on GRID(N, W)

[Sorting algorithm on GRID(N, W) ($W \leq N^{1/4}$)] Consider that the processors are partitioned into W groups with consecutive N/W processors(Fig. 3.8) and one bus can be assigned to each group. Then, each group can be regarded as GRID($N/W, 1$). We apply Columnsort to GRID(N, W) such that each group corresponds to a column in Columnsort. Only in Step 6, the processors PE(i) ($N/(2W) \leq i < N - N/(2W)$) are divided into $W - 1$ groups with consecutive N/W processors. These groups are called *shifted groups*.

Step1 Sort the elements within each group in parallel.

Step2 Transpose the elements.

Step3 Sort the elements within each group in parallel.

Step4 Untranspose the elements.

Step5 Sort the elements within each group in parallel.

Step6 Sort the elements within each shifted group in parallel. □

From simple observation of Columnsort, Step 6 of the algorithm implements shifting, sorting each column and unshifting in Columnsort. From Lemma 3.3, sorting the elements within each group and within each shifted group can be performed in $O(N/W)$ time. From Lemma 3.4, transposing and untransposing can be completed in $O(N/W)$ time. Furthermore, since $W \leq N^{1/4}$ holds, the constraint of Columnsort, $N/W \geq 2W(W-1)$, holds. Therefore, the following lemma holds.

Lemma 3.5 *N elements can be sorted in $O(N/W)$ time on GRID(N, W) if $W \leq N^{1/4}$.* □

Similarly to Step 6, Steps 2 and 4 can be omitted by modifying Step 3. That is, in Step 3, the elements are sorted within each "transposed" group.

Consider the case that N/W is not an integer. Let $N' = W \cdot \lceil N/W \rceil$. From Lemma 3.5, N' elements can be sorted in $O(N'/W)$ time on GRID(N', W). Furthermore, any execution within one step on GRID(N', W) can be simulated by GRID(N, W) in constant time because $N' = O(N)$. Hence, N' elements which consists of N input elements and $N' - N$ dummy elements can be sorted in $O(N/W)$ time on GRID(N, W). Therefore, for every integer N, N elements can be sorted in $O(N/W)$ time on GRID(N, W). By means of this method, we can ignore the cases such that N/W, $N^{1/4}$, and so on, are not integers.

We now apply the above sorting algorithm with constant depth recursion and get the optimal sorting algorithm on GRID(N, W) with large number of buses.

[Sorting algorithm on GRID(N, W)] If $W > N^{1/4}$, it is considered that the processors are partitioned into $N^{1/4}$ groups with consecutive $N^{3/4}$ processors. Since $W/N^{1/4}$ buses can be assigned to each group, each group is regarded as GRID$(N^{3/4}, W/N^{1/4})$. We apply Columnsort to GRID(N, W) such that each group is seen to be a column in Columnsort. Only in Step 6, the processors PE(i) $(N^{3/4}/2 \leq i < N - N^{3/4}/2)$ are divided into $N^{1/4} - 1$ groups with consecutive $N^{3/4}$ processors. These groups are called *shifted groups*.

Step1 If $W \leq N^{1/4}$, sort all the elements and terminate the algorithm. Otherwise, execute the following steps.

Step2 Sort the elements within each group recursively in parallel.

Step3 Transpose the elements.

Step4 Sort the elements within each group recursively in parallel.

Step5 Untranspose the elements.

Step6 Sort the elements within each group recursively in parallel.

Step7 Sort the elements within each shifted group recursively in parallel. □

Similarly to Lemma 3.5 the following theorem holds.

Theorem 3.6 *For every fixed $\epsilon > 0$, N elements can be sorted in $O(N/W)$ time on GRID(N, W) if $W \leq N^{1-\epsilon}$.*

Proof. We will estimate the computation time of the algorithm. Let $T(N, W)$ be the computation time required in case the algorithm is executed on GRID(N, W). In the algorithm, sorting on GRID$(N^{3/4}, W/N^{1/4})$ are executed recursively four times. Therefore,

$$T(N,W) = \begin{cases} O(N/W) & \text{if } W \leq N^{1/4} \\ 4T(N^{3/4}, W/N^{1/4}) + O(N/W) & \text{otherwise.} \end{cases}$$

Assume that $W = N^{1-\epsilon}$ for fixed $\epsilon > 0$. Then,

$$T(N,N^{1-\epsilon}) = \begin{cases} O(N^{\epsilon}) & \text{if } W \leq N^{1/4} \\ 4T(N^{3/4}, N^{3/4-\epsilon}) + O(N^{\epsilon}) & \text{otherwise.} \end{cases}$$

Consider the depth-k recursion, we have,

$$T(N,N^{1-\epsilon}) = 4^k T(N^{(3/4)^k}, N^{(3/4)^k-\epsilon}) + O(4^k N^{\epsilon}).$$

Since ϵ is independent from N, the depth k of recursion is constant. Therefore, $T(N, W) = O(N/W)$ if $W \leq N^{1-\epsilon}$. □

The GRID(N, W) can be simulated simply by the CREW-PRAM with N processors and W shared memory cells. Therefore, the following corollary holds.

Corollary 3.7 *For every fixed $\epsilon > 0$, N elements can be sorted in $O(N/W)$ time on the CREW-PRAM with N processors and W shared memory cells if $W \leq N^{1-\epsilon}$.* □

3.5 Sorting on GRID$(N \times N, W)$

Sorting on GRID$(N \times N, W)$ is defined precisely as follows:

[Sorting on GRID$(N \times N, W)$]

INPUT Let A be an $N \times N$ matrix of elements and $a(i, j)$ $(0 \leq i, j < N)$ be the (i, j) element of A. Each $a(i, j)$ is given to PE(i, j) on GRID$(N \times N, W)$.

OUTPUT Each PE(i, j) knows the $i + jN$th sorted elements of A.

In other words, A is sorted in column major order.

The permutation routing on GRID($N \times N, W$) can be defined similarly. Since the sorting algorithm on GRID($N \times N, W$) is based on Columnsort, the permutation routing on GRID($N \times N, W$) will be used. Any permutation routing on GRID($N \times N, W$) can be performed by iterating the permutation routings on GRID(N, W) at each row and at each column. That is, it can be performed by iterating Steps 1 and 2 alternately.

Step1 Permute the elements within each column on GRID($N \times N, W$) in parallel.

Step2 Permute the elements within each row on GRID($N, \times N, W$) in parallel.

Obviously, if we select appropriate permutations at each step, any permutation requires at most $O(N)$ iterations. But some "simple" permutation routings on GRID($N \times N, W$) requires merely constant number of iterations. Though we omit the precise description of a permutation routing algorithm on GRID($N \times N, W$), the following lemma can be convinced intuitively.

Lemma 3.8 *Any "simple" permutation routing can be done in $O(N/W)$ time on GRID($N \times N, W$).* □

The "simple" permutations include the permutations used for Columnsort (transposing, untransposing and so on) on GRID($N \times N, W$). It is useless to clarify what "simple" permutation routings are, however it seems to be interesting. The reason why it is useless will be shown after Corollary 3.11.

Now we show a sorting algorithm on GRID($N \times N, W$).

[Sorting algorithm on GRID($N \times N, W$)] Consider that A is partitioned into $N^{3/5}$ groups of size $N \times N^{2/5}$(Fig. 3.9). Sorting can be performed by Columnsort in which each groups are seen to be a column. Furthermore, consider that $a(i,j)$ $(N^{2/5}/2 \leq j < N - N^{2/5}/2)$, a part of A, is divided into $N^{3/5} - 1$ groups of size $N \times N^{2/5}$. These groups are called *shifted groups*.

Step1 Sort the elements within each group in column major order in parallel.

Step2 Transpose the elements of A.

Step3 Sort the elements within each group in column major order in parallel.

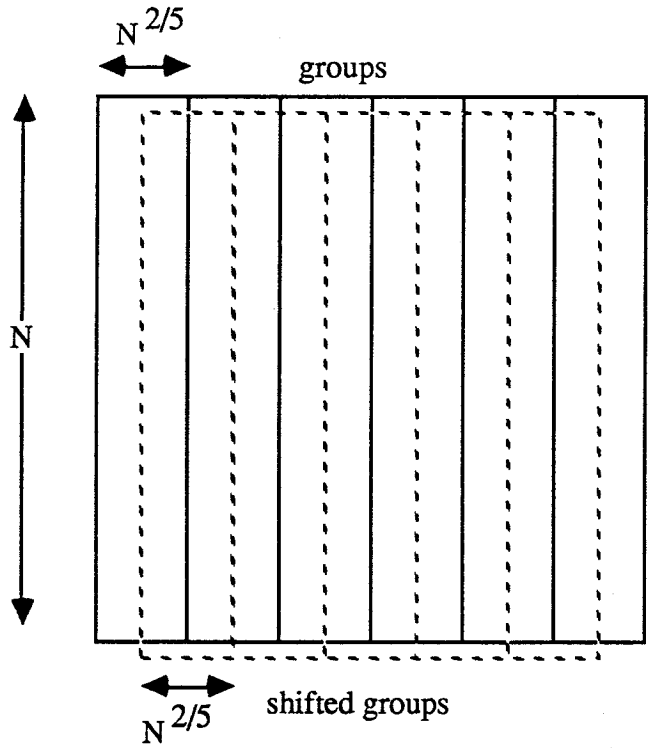


Figure 3.9: Groups and shifted groups on GRID($N \times N, W$)

Step4 Untranspose the elements of A.

Step5 Sort the elements within each group in column major order in parallel.

Step6 Sort the elements within each shifted group in column major order in parallel. □

Since the constraint of Columnsort, $N^{7/5} \geq 2N^{3/5}(N^{3/5}-1)$, holds, sorting is completed correctly. Thus, if each step can be completed in $O(N/W)$ time, the algorithm sorts $N \times N$ elements in $O(N/W)$ time. It remains how to sort the elements within each group and within each shifted group, transpose A and untranspose A in $O(N/W)$ time.

First, we will confirm transposing the elements of A can be performed by means of permutations at each column and at each row on GRID($N \times N, W$), though it is convinced from Lemma 3.8. Sorting the elements within each column is performed just after transposing. Hence, it is considered that transposition is completed, if each $i+jN$th element within each group is transferred to the $((i+jN) \bmod N^{3/5})$th group, that is, the $(i \bmod N^{3/5})$th group. To ensure the permutation *trans2* which transposes the elements is a one-to-one mapping,

we select the following permutation:

$$trans2(i,j) = \\ ((i+j) \bmod N, (i \bmod N^{3/5})N^{2/5} + i \bmod N^{2/5}).$$

It is easy to find out that the permutation *trans2* implements transposing. We will show that the permutation *trans2* can be performed by the permutations at each column and row. The permutation *trans2* can be performed by two permutations $t1$ and $t2$ defined as follows:

$$\begin{array}{c} (i,j) \\ \downarrow t1 \\ ((i+j) \bmod N, j) \\ \downarrow t2 \\ ((i+j) \bmod N, (i \bmod N^{3/5})N^{2/5} + i \bmod N^{2/5}) \end{array}$$

The permutation $t2$ means that the $((i+j) \bmod N, j)$ element is transferred to PE$((i+j) \bmod N, (i \bmod N^{3/5})N^{2/5} + i \bmod N^{2/5})$. Let us verify both $t1$ and $t2$ are one-to-one mappings at each column and at each row, respectively. Assume j is fixed. Since the mapping $f(i) = (i+j) \bmod N$ defined over $\{0, \ldots, N-1\}$ is a one-to-one mapping, $t1$ is a one-to-one mapping for every fixed j. Hence, from Lemma 3.4, the permutation $t1$ can be performed in $O(N/W)$ time by a permutation within each column. Let $k = (i+j) \bmod N$. The permutation $t2$ can be rewritten as follows:

$$\begin{array}{c} (k,j) \\ \downarrow t2 \\ (k, ((k-j) \bmod N^{3/5})N^{2/5} + (k-j) \bmod N^{2/5}) \end{array}$$

Since the mapping $g(j) = ((k-j) \bmod N^{3/5})N^{2/5} + (k-j) \bmod N^{2/5}$ defined over $\{0, \ldots, N-1\}$ is a one-to-one mapping, $t2$ is a one-to-one mapping for every fixed k. Hence, from Lemma 3.4, the permutation $t2$ can be performed in $O(N/W)$ time by a permutation within each row. Therefore, transposing can be completed in $O(N/W)$ time. Similarly, untransposing can be implemented.

Now, we show how to sort the elements within each group. They are sorted by Columnsort.

[Sorting elements within each group] Consider each group is a matrix of size $N \times N^{2/5}$.

Sorting can be performed by applying Columnsort to each matrix.

Step1 Sort the elements within each column.

Step2 Transpose the elements within each group.

Step3 Sort the elements within each column.

Step4 Untranspose the elements within each group.

Step5 Sort the elements within each column.

Step6 Shift the elements within each group.

Step7 Sort the elements within each column except the leftmost column within each group.

Step8 Unshift the elements within each group. □

Since the constraint of Columnsort, $N \geq 2N^{2/5}(N^{2/5}-1)$, holds, sorting can be completed correctly. From Lemma 3.3, each column can be sorted in $O(N/W)$ time in case $W \leq N^{1-\epsilon}$. Transposing and untransposing can be performed in the same way. We will confirm shifting and unshifting can be completed in $O(N/W)$ time. Sorting the elements within each column is performed just after shifting. Hence, shifting can be considered to be completed, if the elements in the lower $N/2$ row are transferred to the next column and in the last column are transferred to the first column. Let us focus on the elements within the first group, $a(i,j)$ $(0 \leq i < N, 0 \leq j < N^{2/5})$. In the first group, the elements can be transposed along the permutation *shift2* defined as follows:

$$shift2(i,j) = \begin{cases} (i,j) & \text{if } i < N/2 \\ (i,(j+1) \bmod N^{2/5}) & \text{otherwise.} \end{cases}$$

Hence shifting can be performed by a permutation within each row. Therefore, we have,

Theorem 3.9 *For every fixed $\epsilon > 0$, N^2 elements can be sorted in $O(N/W)$ time on GRID($N \times N, W$) if $W \leq N^{1-\epsilon}$.* □

In general, the following corollary holds.

Corollary 3.10 *For every fixed $\epsilon > 0$, NM elements can be sorted in $O((N+M)/W)$ time on GRID($N \times M, W$) if $W \leq (N+M)^{1-\epsilon}$.* □

The sorting algorithm on GRID($N \times N, W$) is represented simply as follows.

Repeat the following steps constant times alternately:

Step1 Sort the elements within each column (sometimes sort except some columns).

Step2 Permute the elements along the appropriate permutations.

The elements are permuted along ten permutations: (1)transposing, (2)untransposing, (3)transposing within each (shifted) group, (4)untransposing within each (shifted) group, (5)shifting within each (shifted) group, and (6) unshifting within each (shifted) group. The order of the permutations and the implementation of each permutation can be computed easily.

Obviously, from Corollary 3.10, we have

Corollary 3.11 *Any permutation routing can be performed in* $O((N+M)/W)$ *time on the GRID*$(N \times M, W)$ *if* $W \leq (N+M)^{1-\epsilon}$. □

Therefore, not only "simple" permutation routings but also any other permutation routing can be performed effectively.

3.6 Concluding remarks

In this chapter, optimal parallel sorting algorithms on processor arrays with multiple buses are presented. Obviously, by using AKS sorting network [3] [15], N elements can be sorted in $O(\log N)$ time on GRID(N, W) where $W = \Omega(N)$. This method is optimal because sorting N elements needs at least $\Omega(\log N)$ time as long as at most N processors are available. Therefore it remains open to find an optimal sorting algorithm on GRID(N, W) where $W \in [\omega(N^{1-\epsilon}), o(N)]$ (e.g. $W = N/\log N$).

Chapter 4

Sorting on a reconfigurable array

A logarithmic time sorting algorithm on a practical model is known [26]. 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 value is smaller than that of it. Hence, N elements can be sorted in $O(\log N)$ time using $N(N-1)$ processors. In this chapter, we will present a sub-logarithmic time sorting algorithm on a reconfigurable array (shortly RE-ARRAY) on the basis of an enumeration scheme. Sub-logarithmic time is achieved by computing the rank of elements faster.

It is known [36] [40] that N elements can be sorted in constant time by an enumeration scheme both on an $N \times N^2$ RE-ARRAY (means RE-ARRAY with $N \times N^2$ processors) and on an $N^{3/2} \times N^{3/2}$ RE-ARRAY. We will reduce the number of processors and improve this algorithm. Firstly, we show an algorithm summing up N binary values in constant time on $N \times \log^2 N$ RE-ARRAY. Secondly, by using these algorithms, we show that N elements can be sorted by an enumeration scheme in constant time on an $N \times N \log^2 N$ RE-ARRAY. Lastly, we obtain more generalized algorithm which sorts N elements in $O(T)$ time on an $N \times N \log^{(T)} N$ RE-ARRAYwhere $1 \leq T \leq \log^* N$. This implies that N elements can be sorted in constant time on an $N \times N \log^{(\alpha)} N$ RE-ARRAY for any constant integer $\alpha \geq 1$, and in $O(\log^* N)$ time on an $N \times N$ RE-ARRAY.

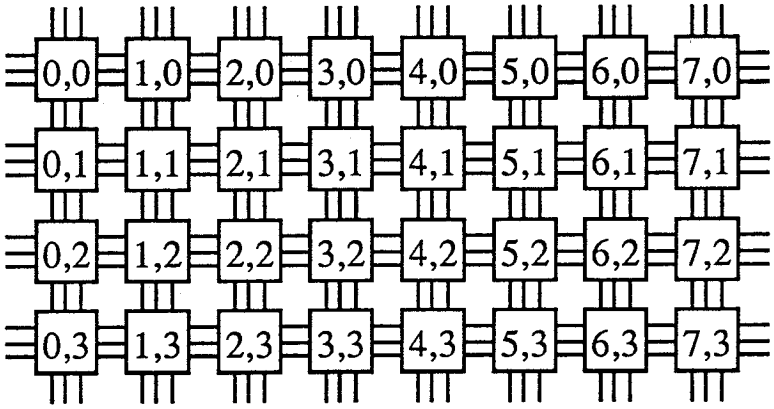


Figure 4.1: RE-ARRAY

4.1 Model and notation

A RE-ARRAY is formalized as follows. Processors on an $N \times M$ RE-ARRAY are denoted by PE(i,j) $(0 \leq i \leq N-1, 0 \leq j \leq M-1)$. As shown in Fig. 4.1, it is considered that PE$(i,0)$ $(0 \leq i \leq N-1)$ is located at the top row of a RE-ARRAY and PE$(0,j)$ $(0 \leq j \leq M-1)$ is located at the leftmost column of a RE-ARRAY. Each processor on a RE-ARRAY is the RAM (random access machine) with an extended set of instructions for changing configuration of buses, sending data to buses and receiving data from buses. As shown in Fig. 4.2, 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 facing to each other are connected by static buses, that is, $\mathbf{D}(k)$ on PE(i,j) and $\mathbf{U}(k)$ on PE$(i,j+1)$ are connected by a static bus. Similarly, $\mathbf{R}(k)$ on PE(i,j) and $\mathbf{L}(k)$ on PE$(i+1,j)$ are connected by a static bus. It is assumed that each processor may have the constant number of ports, that is, P is constant. All processors work synchronously and execute the following phases in a time unit:

Phase 1 Changing configuration of a reconfigurable bus systems by connecting or disconnecting its own ports by buses locally. The data sent at Phase 2 is transferred through the locally connected buses and static buses between processors.

Phase 2 Sending data to each port.

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

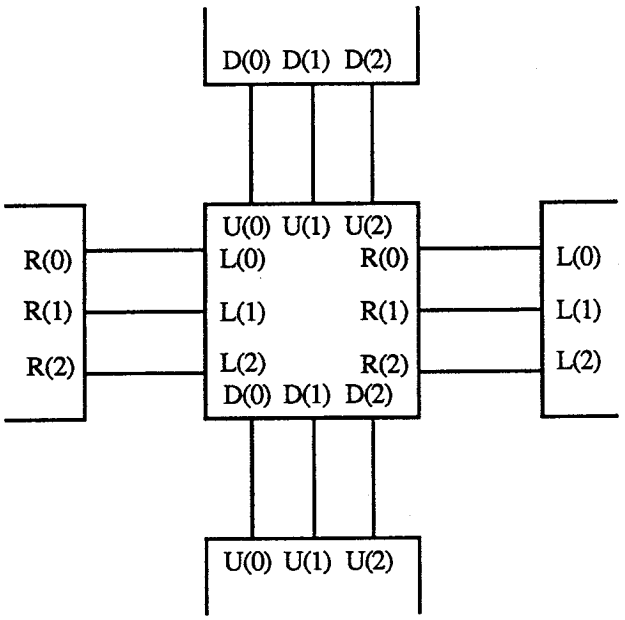


Figure 4.2: Ports of RE-ARRAY

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

On a RE-ARRAY, all processors execute the phases synchronously, that is, no processor executes a phase before all processors finishing the previous phase. The computation time of an algorithm is evaluated by counting the number of iterations of these four phases during the execution of it. Hence, our model is *the unit-time delay model* because it is assumed that communication completes within a time unit.

In algorithms presented in this chapter, we use the following notation:

- The connection of two ports, say, $\mathbf{L}(0)$ and $\mathbf{R}(0)$ is denoted by $\mathbf{L}(0) \cdot \mathbf{R}(0)$.
- Sending the value x to the port, say, $\mathbf{L}(0)$ is denoted by $\mathbf{L}(0) \leftarrow x$.
- Receiving value from a port and storing it to a local memory cell, say, receiving value from the port $\mathbf{L}(0)$ and storing it to the local memory cell c is denoted by $\mathbf{L}(0) \rightarrow c$.

From Phase 2 to Phase 4, it can be regarded that the processors are connected by static buses, because the configuration of bus system is not changed. Several bus models are proposed [16] with respect to simultaneous sending on a static bus system. All algorithms

presented in this chapter and the previously known sorting algorithms [36] [40] are designed for the exclusive model.

4.2 Basic property and algorithms

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

4.2.1 Basic property

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

Lemma 4.1 *Any execution in a time unit on an $O(N) \times O(M)$ RE-ARRAY can be simulated in a time unit on an $N \times M$ RE-ARRAY.*

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

Even if we regard that the time complexity is affected by the number of the local computation, any execution in a time unit 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.

4.2.2 Leftmost finding

We consider the problem to find the leftmost element whose value is 1, when a binary sequence of length N is given. *Leftmost Finding* on an $N \times 1$ RE-ARRAY is defined as follows.

Input Let $B = \langle a(0), a(1), \ldots, a(N-1) \rangle$ be a binary sequence of length N. Each $a(i)$ $(0 \leq i \leq N-1)$ is given to 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 $m(= N)$.

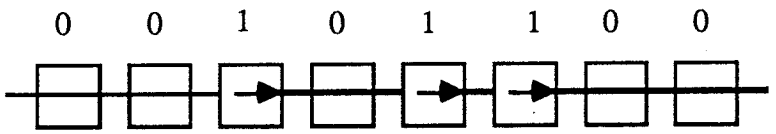


Figure 4.3: Leftmost finding

The leftmost finding algorithm on an $N \times 1$ RE-ARRAY follows (Fig. 4.3).

[Leftmost Finding Algorithm]

Step 1 If $a(i) = 0$, then $\mathbf{L}(0) \cdot \mathbf{R}(0) : \mathrm{PE}(i,0)$. {means that if $a(i) = 0$, then $\mathrm{PE}(i,0)$ connects $\mathbf{L}(0)$ and $\mathbf{R}(0)$.}

If $a(i) = 1$, then $\mathbf{R}(0) \leftarrow 1 : \mathrm{PE}(i,0)$. {means that if $a(i) = 1$, then $\mathrm{PE}(i,0)$ sends 1 to $\mathbf{R}(0)$.}

$\mathbf{L}(0) \rightarrow c(i) : \mathrm{PE}(i,0)$ $(0 \leq i \leq N-1)$. {means that each $\mathrm{PE}(i,0)$ receives data from $\mathbf{L}(0)$, and stores it to $c(i)$ that is a local memory cell of $\mathrm{PE}(i,0)$.}

If $\mathrm{PE}(i,0)$ receives no data, let $c(i) \leftarrow 0$.

Step 2 If $a(i) = 1$ and $c(i) = 0$, then $\mathrm{PE}(i,0)$ broadcasts i to all processors.

If $a(N-1) = 0$ and $c(N-1) = 0$, then $\mathrm{PE}(N-1,0)$ broadcasts N to all processors.

[end of algorithm]

The following lemma holds.

Lemma 4.2 *The leftmost element can be found in constant time on an $N \times 1$ RE-ARRAY.*

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. Obviously the algorithm completes in constant time. □

4.2.3 Logical OR

The logical OR is the problem to determine whether there is an element whose value is 1, when a binary sequence of length N is given. The logical OR on an $N \times 1$ RE-ARRAY is defined as follows.

Input Let $B = \langle a(0), a(1), \ldots, a(N-1) \rangle$ be a binary sequence of length N. Each $a(i)$ $(0 \leq i \leq N-1)$ is given to PE$(i, 0)$.

Output All processors know the logical OR of the input sequence, that is,know m such that $m = \max\{a(i)\}$.

The logical OR can be simply computed by using the leftmost finding algorithm. To compute the logical OR, after executing the leftmost finding algorithm, if there is a leftmost element, then the result of the logical OR is 1, otherwise, the result is 0. Therefore, the following lemma holds.

Lemma 4.3 *The logical OR can be computed in constant time on an $N \times 1$ RE-ARRAY.*

4.2.4 Compression

We consider the procedure that compresses a sequence of elements. *Compression* on an $N \times M$ RE-ARRAY is defined as follows.

Input Let $A = \langle a(0), a(1), \ldots, a(N-1) \rangle$ be a sequence of elements. In addition to the domain of elements, each element in the sequence may take value NULL. Each $a(i)$ $(0 \leq i \leq N-1)$ is given to PE$(i, 0)$.

Output Let $A' = \langle a(i_0), a(i_1), a(i_2), \ldots \rangle$ be the subsequence of A such that an element in A is in A' if and only if its value is not NULL. The processors on the leftmost column knows the first M elements in A', that is, each PE$(0, j)$ $(0 \leq j \leq M-1)$ knows $a(i_j)$ if exists.

In the compression algorithm on an RE-ARRAY, each column 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 the stack. Otherwise, the processors hold the stack. Then each processor on the leftmost column knows the element whose value is not NULL. The compression algorithm on a RE-ARRAY follows (Fig. 4.4).

[Compression Algorithm]

Step 1 Each PE$(i, 0)$ $(0 \leq i \leq N-1)$ broadcasts $a(i)$ to the processors on the same column, PE(i, j) $(0 \leq j \leq M-1)$.

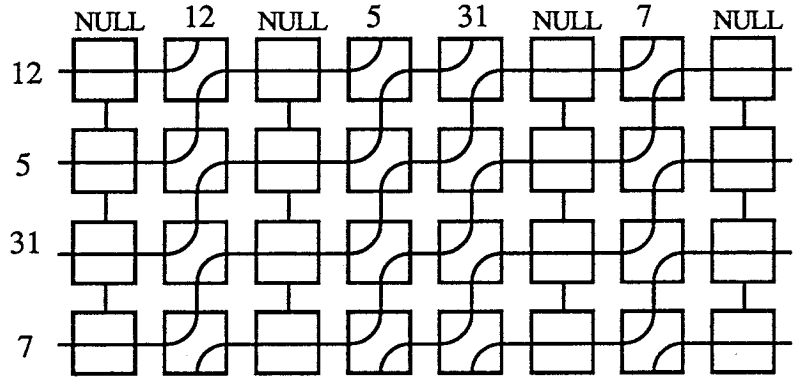


Figure 4.4: Compression

Step 2 If $a(i) = \text{NULL}$, then

$$\mathbf{L}(0) \cdot \mathbf{R}(0) : \text{PE}(i,j)\ (0 \leq j \leq M-1).$$

If $a(i) \neq \text{NULL}$, then

$$\mathbf{D}(0) \cdot \mathbf{R}(0) : \text{PE}(i,j)\ (0 \leq j \leq M-1)$$

$$\mathbf{U}(0) \cdot \mathbf{L}(0) : \text{PE}(i,j)\ (0 \leq j \leq M-1)$$

$$\mathbf{U}(0) \leftarrow a(i) : \text{PE}(i,0).$$

$$\mathbf{L}(0) \rightarrow c(j) : \text{PE}(0,j)\ (0 \leq j \leq M-1).\ \{c(j) \text{ contains } a(i_j).\}$$

[end of algorithm]

The following lemma holds.

Lemma 4.4 *Compression can be done in constant time on an RE-ARRAY.*

Proof. The correctness of the algorithm can be proved by induction easily. □

4.2.5 Prefix remainder computation

The prefix w-remainder of a binary sequence on an $N \times M$ RE-ARRAY is defined as follows.

Input Let $\langle a(0), a(1), \ldots, a(N-1) \rangle$ be a binary sequence of length N. Each $a(i)$ $(0 \leq i \leq N-1)$ is given to $\text{PE}(i,0)$.

Output Let x_i be the prefix remainder at position i, that is, $x_i = (\sum_{j=0}^{i} a(j)) \bmod w$. Each $\text{PE}(i,0)$ $(0 \leq i \leq N-1)$ knows x_i.

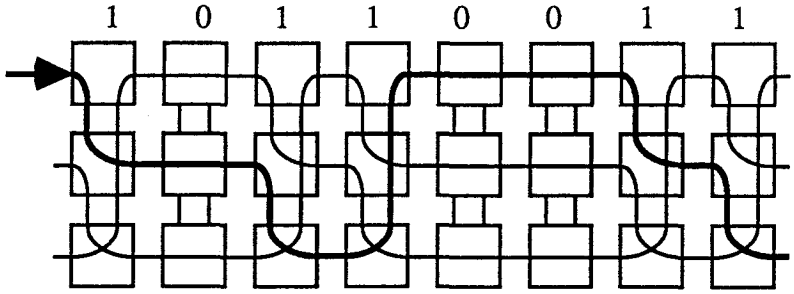


Figure 4.5: Prefix remainder

We show the algorithm for computing the prefix M-remainder on an $N \times M$ RE-ARRAY. In the algorithm each column works as a cyclic shift register of size M. 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 hold 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 M-remainder on an $N \times M$ RE-ARRAY is described as follows(Fig. 4.5).

[Prefix Remainder Algorithm]

Step 1 Each PE$(i,0)$ $(0 \leq i \leq N-1)$ broadcasts $a(i)$ to the processors on the same column.

Step 2 if $a(i) = 0$, then

$$\mathbf{L}(0) \cdot \mathbf{R}(0) : \mathrm{PE}(i,j)\ (0 \leq j \leq N-1).$$

if $a(i) = 1$, then

$$\mathbf{L}(0) \cdot \mathbf{D}(0) : \mathrm{PE}(i,j)\ (0 \leq j \leq M-2)$$

$$\mathbf{U}(0) \cdot \mathbf{R}(0) : \mathrm{PE}(i,j)\ (1 \leq j \leq M-1)$$

$$\mathbf{U}(1) \cdot \mathbf{D}(1) : \mathrm{PE}(i,j)\ (1 \leq j \leq M-2)$$

$$\mathbf{D}(1) \cdot \mathbf{R}(0) : \mathrm{PE}(i,0)$$

$$\mathbf{L}(0) \cdot \mathbf{U}(1) : \mathrm{PE}(i,M-1).$$

$$\mathbf{L}(0) \leftarrow 1 : \mathrm{PE}(0,0).$$

$$\mathbf{R}(0) \rightarrow c(i,j) : \mathrm{PE}(i,j)\ (0 \leq j \leq M-1).$$

Step 3 if $c(i,j) = 1$, PE(i,j) broadcasts j $(= x_i)$ to the processors on the same column.

[end of algorithm]

The following lemma holds.

Lemma 4.5 *The prefix M-remainder of a binary sequence of length N can be computed in constant time on an $N \times M$ RE-ARRAY.*

Proof. It is sufficient to prove that for all i, $c(i,j) = 1$ iff $x_i = j$ 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, a RE-ARRAY with the exclusive buses is more powerful than the CRCW PRAM with regard to the prefix remainder computation.

4.2.6 Remainder computation

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

Input Let $\langle a(0), a(1), \ldots, a(N-1)\rangle$ be a binary sequence of length N. Each $a(i)$ $(0 \leq i \leq N-1)$ is given to PE$(i,0)$.

Output Let x is the sum of the binary sequence, that is, $x = \sum_{i=0}^{N-1} a(i)$. All processors know r such that $x \equiv r \pmod{w}$.

From the definition, the M-remainder can be easily computed by means of the prefix M-remainder. We will show that the remainder from a larger modulus can be computed in constant time on a RE-ARRAY. The basic idea is as follows. Consider that an $N \times M$ RE-ARRAY is divided into $\sqrt{M}$ subarrays of sizes $N \times 1, N \times 2, \cdots, N \times \sqrt{M}$. To express x by the RNS (residue number system), we apply the algorithm in Lemma 4.5 to each subarray, and then $r_1, r_2, \ldots, r_{\sqrt{M}}$ can be computed such that $x \equiv r_i \pmod{i}$. If $r_1, r_2, \ldots, r_{\sqrt{M}}$ are known, the remainder from a larger modulus than M can be computed. We analyze how large the modulus is by using the Chinese remainder theorem and the prime number theorem, famous theorems in number theory.

Theorem 4.6 (Chinese remainder theorem) *Let $p_1, p_2, \ldots, p_m$ be pairwise relatively prime positive integers . For unknown x, 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 4.7 *Let $\mathrm{lcm}(m)$ be the L.C.M. (least common multiple) of $\{1, 2, \ldots, m\}$. If the following congruence holds 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 4.8 (Prime number theorem) *Let $\pi(n)$ denote 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 4.9 *The equality $\mathrm{lcm}(n) = 2^{\Theta(n)}$ holds.*

Proof. Let $\{p_1, p_2, \ldots, p_m\}$ $(p_1 = 2 < p_2 < \cdots < p_m \leq n, m = \pi(n))$ be the set of primes less than or equal to n and $a_1, a_2, \ldots, a_m$ be the integers such that ${p_i}^{a_i} \leq n < {p_i}^{a_i+1}$ 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 $p_{m'} < n/2 \leq p_{m'+1}$. 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 larger modulus on a RE-ARRAY is described as follows.

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

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

Step 2 Compute the prefix k-remainder of the sequence $\langle 0,1,1,1,\ldots,1\rangle$ on each subarray of size $N \times k$. After Step 2, each PE$_k(i,0)$ knows $r'_{k,i}$ such that $i \equiv r'_{k,i} \pmod{k}$.

Step 3 On each ith column, compare r_k and $r'_{k,i}$ and examine whether the condition $r_k = r'_{k,i}$ holds for all k $(1 \leq k \leq \sqrt{M})$. To examine whether the condition holds for all k on each column, the logical OR is used.

Step 4 Find the minimum i such that $r_k = r'_{k,i}$ for all k $(1 \leq k \leq \sqrt{M})$ by using the leftmost finding algorithm. That is, compute $r = \min\{i | r_k = r'_{k,i} \text{ for all } k\}$. And broadcast r to all processors on an array. After Step 4, all processors know r, the solution of lcm($\sqrt{M}$)-remainder.

Step 5 Find the minimum $i > 0$ such that $r'_{k,i} = 0$ for all k $(1 \leq k \leq \sqrt{M})$ by leftmost finding. That is, compute $w = \min\{i | r'_{k,i} = 0 \text{ for all } k\}$. If such w $(= \text{lcm}(\sqrt{M}))$ exists, broadcast w.

[end of algorithm]

To compute $\text{lcm}(\sqrt{M})$-remainder only, we do not have to execute step 5. But $\text{lcm}(\sqrt{M})$ has been computed in step 5 for the following algorithm that computes the sum of a binary sequence by using $\text{lcm}(\sqrt{(M)})$-remainder.

Lemma 4.10 *The problem* $\text{lcm}(\sqrt{M})$*-remainder can be solved in constant time on an* $N \times M$ *RE-ARRAY.*

Proof. For all k $(1 \leq k \leq \sqrt{M})$, the congruence $x \equiv r \pmod{k}$ holds. Therefore, from Corollary 4.7, $x \equiv r \pmod{\text{lcm}(\sqrt{M})}$ holds. This completes the proof. □

From Lemma 4.10, $\text{lcm}(\sqrt{kM})$-remainder can be computed in constant time on an $N \times kM$ RE-ARRAY. Hence, from Lemma 4.1, the following lemma holds.

Lemma 4.11 *The problem* $\text{lcm}(\Theta(\sqrt{M}))$*-remainder can be solved in constant time on an* $N \times M$ *RE-ARRAY.* □

From Lemma 4.9, there exists a fixed k such that $\text{lcm}(\sqrt{k \log^2 N}) > N$. Therefore, the following corollary holds.

Corollary 4.12 *The sum of a binary sequence of length* N *can be computed in constant time on a RE-ARRAY of size* $N \times \log^2 N$. □

Proof. To compute the minimum k such that $\text{lcm}(\sqrt{k \log^2 N}) > N$, the $\text{lcm}(\sqrt{k \log^2 N})$-remainder computation is repeated for $k = 1, 2, \ldots$ until $\text{lcm}(\sqrt{k \log^2 N}) > N$ holds. At the end of the iteration, $x = (x \bmod \text{lcm}(\sqrt{k \log^2 N}))$ 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 4.10 is modified as follows: for the prime numbers $p_1, p_2, \ldots$ defined in the proof of Lemma 4.9, a RE-ARRAY is divided into subarrays of sizes $N \times p_1, N \times p_2, \ldots$ and $x \bmod p_1, x \bmod p_2, \ldots$ are computed on them. In this algorithm, we can compute the remainder from the modulus

of size $2^{\Theta(\sqrt{M \log M})}$. Hence, in Corollary 4.12, the size of a RE-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.

4.3 New sorting algorithm

Sorting on a RE-ARRAY is defined as follows.

Input Let $\langle a(0), a(1), \ldots, a(N-1)\rangle$ be a sequence of elements. Each $a(i)$is given to PE$(i, 0)$ $(0 \leq i \leq N-1)$.

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

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

4.3.1 Constant time sorting algorithm

First, we show a constant time sorting algorithm on RE-ARRAY of size $N \times N \log^2 N$. The sorting algorithm is 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 4.12, the sum of a binary sequence of length N can be computed in constant time on an $N \times \log^2 N$ RE-ARRAY. Therefore, the ranks of all elements can be computed in constant time on an $N \times N \log^2 N$ RE-ARRAY. Figure 4.6 illustrates the constant time sorting algorithm.

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

Step 1 Each PE$(i, 0)$ broadcasts $a(i)$ to all processors on the same column. And each PE$_k(k, 0)$ broadcasts $a(k)$ to all processors on the same row.

Step 2 Each PE$_k(i, 0)$ compares $a(k)$ and $a(i)$. If $a(k) > a(i)$, let $r_{k,i} \leftarrow 1$. Otherwise, let $r_{k,i} \leftarrow 0$.

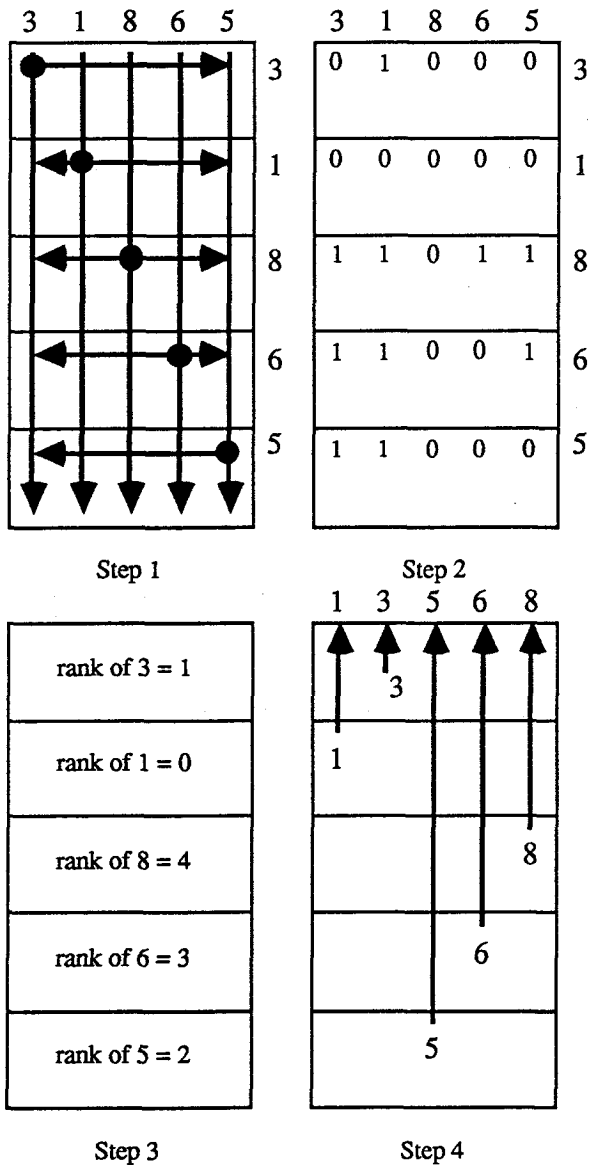


Figure 4.6: Constant time sorting

Step 3 Compute 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.

Step 4 Each $\mathrm{PE}_k(r_k, 0)$ sends r_kth element, $a(k)$, to $\mathrm{PE}(r_k, 0)$.

[end of algorithm]

The following theorem holds.

Theorem 4.13 *Sorting of N elements can be performed in constant time on an $N \times N \log^2 N$ RE-ARRAY.* □

4.3.2 General sorting algorithm

We show a sorting algorithm on a RE-ARRAY of size less than $N \times N \log^2 N$. The algorithm in Corollary 4.12 cannot be used to compute the rank of all elements on $N \times M$ RE-ARRAY($M < N \log^2 N$). 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 the range of its group can be computed. Then the rank of each element can be computed from the remainder of the rank and the rank in the range of its group. Figure 4.7 illustrates the general sorting algorithm.

[General Sorting Algorithm] Similarly to the constant time sorting, consider that $N \times M$ RE-ARRAY is divided into N (horizontal) subarrays of size $N \times (M/N)$. And let $w = \mathrm{lcm}(\sqrt{M/N})$. If $w \leq N$, consider that an $N \times M$ RE-ARRAY is divided into w vertical subarrays of size $N/w \times M$. Each $\mathrm{PE}(i, j)$ on the kth vertical subarray is denoted by $\mathrm{PE}'_k(i, j)$ $(0 \leq i \leq N/w - 1, 0 \leq j < M)$. The algorithm has seven steps. As shown in Fig. 4.7, the remainder of the rank of each element is computed on each horizontal subarray in Steps 1,2, and 3. In Steps 4 and 5, the elements whose rank takes the remainder j is transferred to the jth vertical subarray. In Step 6, sorting the elements within each vertical subarray is performed recursively and then the rank of each elements is computed. In Step 7, each element is transferred to the correct position.

Step 1, Step 2 Execute the same as step 1 and step 2 of the constant time sorting.

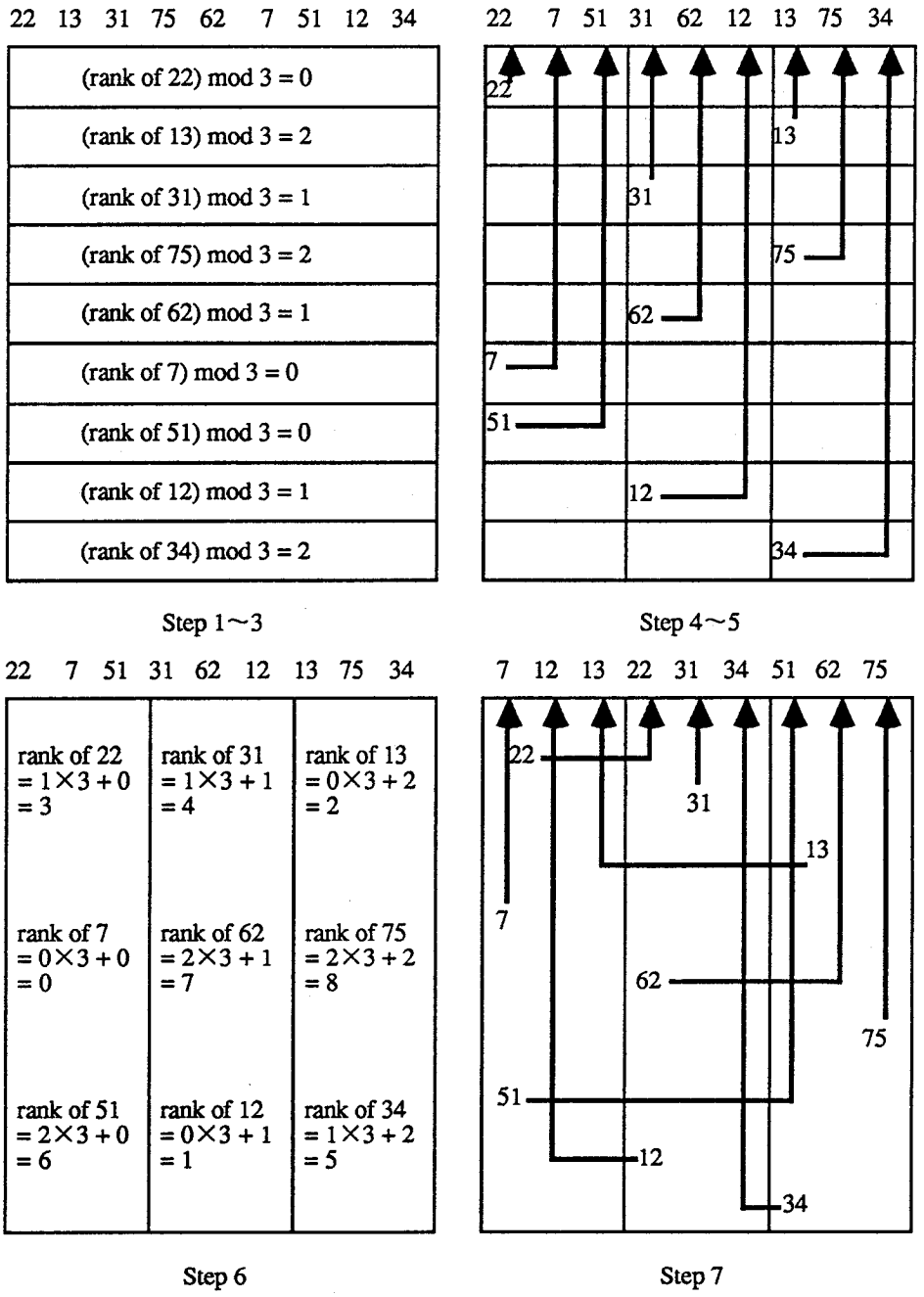


Figure 4.7: General sorting

Step 3 Compute w-remainder of $\langle r_{k,0}, r_{k,1}, \ldots, r_{k,N-1}\rangle$ on each the kth horizontal subarray, that is, compute r'_k such that $r_k \equiv r'_k \pmod{w}$. From Lemma 4.10, this can be done in constant time. In case $w > N$, execute step 4 of the constant time sorting algorithm, because $r_k = r'_k$ holds. Otherwise, execute the following steps to gather elements whose rank takes the same remainder.

Step 4 Each $\mathrm{PE}'_j(0, kM/N)$ $(0 \le j \le w-1, 0 \le k \le N-1)$ let its local variable $d_j(k) \leftarrow$ NULL. Since for all k each $\mathrm{PE}'_{r'_k}(0, kM/N)$ knows $a(k)$, let $d_{r'_k}(k) \leftarrow a(k)$. After Step 4, $\langle d_j(0), d_j(1), \ldots, d_j(N-1)\rangle$ contains all elements whose rank takes the remainder j.

Step 5 Compress $\langle d_j(0), d_j(1), \ldots, d_j(N-1)\rangle$ for each j on each vertical subarray. After Step 5, each $\mathrm{PE}'_j(i, 0)$ knows one of the elements whose rank takes the remainder j. Let $s_{j,i}$ be the index such that $\mathrm{PE}'_j(i, 0)$ knows $a(s_{j,i})$.

Step 6 Sort each sequence $\langle a(s_{j,0}), a(s_{j,1}), \ldots, a(s_{j,N/w-1})\rangle$ for all j on each vertical subarray recursively. Let the rank of $a(s_{j,i})$ in the sequence $\langle a(s_{j,0}), a(s_{j,1}), \ldots, a(s_{j,N/w-1})\rangle$ be $r_{j,i}$. Then the rank of $\langle a(s_{j,i})$ in $< a(0), a(1), \ldots, a(N-1)\rangle$ is $r_{j,i}w + j$.

Step 7 Similarly to step 4 of the constant time sorting, send $a(s_{j,i})$ to $\mathrm{PE}(r_{j,i}w + j, 0)$.

[end of algorithm]

Theorem 4.14 *N elements can be sorted in $O(T)$ time on $N \times N \log^{(T)} N$ RE-ARRAY for every $1 \le T \le \log^* N$.*

Proof. The correctness of the algorithm can be proved easily by induction on the size of a RE-ARRAY. We have to analyze the computation time required in this algorithm. Let $t(N, M)$ be the computation time required if this algorithm sorts N elements on $N \times M$ RE-ARRAY. In step 6, this algorithm sorts N/w elements on each $N/w \times M$ horizontal subarray recursively. Hence the following equality holds.

$$t(N, M) = \begin{cases} t(N/2^{\Theta(\sqrt{N/M})}, M) + O(1) & \text{if } M < N \log^2 N \\ O(1) & \text{if } M \ge N \log^2 N \end{cases}$$

Therefore, $t(N, N \log^{(T)} N) = O(T)$. □

Theorem 4.14 implies the following corollaries.

Corollary 4.15 *N elements can be sorted in constant time on an $N \times N \log^{(\alpha)} N$ RE-ARRAY for any fixed integer $\alpha \geq 1$.* □

Corollary 4.16 *N elements can be sorted in $O(\log^* N)$ time on an $N \times N$ RE-ARRAY.* □

4.4 Concluding remarks

In this chapter, we have presented a fast sorting algorithm on reconfigurable arrays. Our sorting algorithm requires a smaller number of processors than the previous algorithm [40]. Recently, more efficient algorithm has developed: N elements can be sorted in constant time on an $N \times N$ reconfigurable array[7][19]. However a method presented in this chapter, *the remainder method* is effective with regard to counting 1s and has lots of applications. The author will report the applications at some other time.

Chapter 5

Conclusions

In this dissertation, we have shown three topics with regard to parallel algorithms on bus-connected machines. In Chapter 2, we presented practical methods to realize the priority buses using common buses. In Chapter 3, we showed optimal sorting algorithms on bus-connected processor arrays. In Chapter 4, we presented a sub-logarithmic time sorting algorithm on a reconfigurable array. Furthermore, in previous papers, the author have been studying and showed the following interesting results:

- N elements can be sorted in $(N/W + \log^2 N)$ time on a 1-dimensional processor array of size N with multiple buses arranged to W layers [27].
- A 2-dimensional processor array of size $N \times N$ with buses arranged to a single layer can be simulated in $O(\log N)$ time on a 2-dimensional processor array with buses arranged to an orthogonal tree [28].
- Image component labeling of size $N \times N$ can be performed in $O(\log^2 N)$ time on a 2-dimensional processor array with buses arranged to an orthogonal tree [29].

The author intend to clarify the computation power of bus-connected processor arrays and reconfigurable arrays.

Acknowledgments

The author wishes to express his gratitude for guidance and encouragement received from Professor N. Tokura. He also wishes to thank to Associate Professors T. Araki, K. Hagihara, and Y. Tsujino. He wishes to express his thanks to the members of his laboratory, especially Assistant Professor T. Masuzawa, with whom I have worked together, for his time and kindness. Furthermore, he would like to thank T. Kasami, A. Hasimoto, and K. Taniguchi for their usefull comment on this dissertation. Finally, he wishes to thank to the members of Osaka University, especially Professors T. Kikuno, K. Torii, T. Kasiwabara, H. Miyahara, H. Nisitani, M. Sudo, M. Yachida, H. Wakita, M. Fujii, R. Mizoguchi, J. Toyoda, and T. Kitahasi.

Bibliography

[1] A. Aggarwal, *Optimal bounds for finding maximum on array of processors with k global buses*, IEEE Transactions on Computers,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-Wesley, 1974.

[3] M. Ajtai, J. Kolmos and E. Szemeredi, *An $O(n \log n)$ sorting network*, Proceedings of the 15th Annual ACM Symposium 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*, Proceedings of the 19th Annual ACM Symposium on Theory of Computing, pp.83–93, 1987.

[7] Y. Ben-Asher, D. Peleg, R. Ramaswami, and A. Schuster, *The power of reconfiguration*, Journal of Parallel and Distributed Computing, 13, pp.139–153, 1991.

[8] D. Bitton, D. J. Dewitt, D. K. Hisao and J. Menon, *A taxonomy of parallel sorting*, ACM Computing Surveys, 16, 3, pp.287–319, 1984.

[9] S. H. Bokhari, *Finding maximum on an array processor with a global bus*, IEEE Transactions on Computers, C-33, 2, pp.133-139, 1984.

[10] B.S.Chlebus, K.Diks, T.Hagerup, and T.Radzik, *Efficient simulations between concurrent-read concurrent-write PRAM models*, Proceedings of the 13th Mathematical

Foundations of Computer Science(Lecture Notes in Computer Science 324),pp.231-239, 1988

[11] R. Cole, *Parallel merge sort*, Proceedings of the 27th Annual Symposium on Foundations of Computer Science, pp.511–516, 1986.

[12] A. Gibbons and W. Rytter, *Efficient parallel algorithms*, Cambridge University Press, 1989.

[13] S. Fujita, M. Yamasita and T. Ae, *An optimal sorting algorithm for parallel processor with multiple buses*, Proceedings of Joint Symposium on Parallel Processing '90, pp.33–40, 1990, in Japanese.

[14] S. Fujita, M. Yamasita and T. Ae, *An optimal sorting algorithm for parallel processors with multiple buses*, Transactions of IPS Japan, 32, 7, pp.800–806, 1991, in Japanese.

[15] A. Gibbons and W.Rytter, *Efficient parallel algorithms*, Cambridge University Press, 1988.

[16] K. Iwama, *Feasible but still powerful PRAMs*, Technical Report COMP 86-53, Institute of Electronics, Information and Communication Engineers, 1986.

[17] K. Iwama and Y. Kambayashi, *An $O(\log n)$ parallel connectivity algorithm on the mesh of buses*, Information Processing, 11, pp.305–310, 1989.

[18] K. Iwama, E. Miyano and Y .Kambayashi, *A parallel sorting algorithm on the mesh-bus machine*, Technical Report SIGAL 18–2, Information Processing Society of Japan, 1990.

[19] J. Jang and V. K. Prasanna, *An optimal sorting algorithm on reconfigurable mesh*, Technical Report IRIS#227, Univ. of Southern California, 1991.

[20] D. E. Knuth, *The art of computer programming, vol.3 (sorting and searching)*, Addison-Wesley, 1973.

[21] V. K. P. Kumar and C. S. Raghavendra, *Array processor with multiple broadcasting*, Journal of Parallel and Distributed Computing, 4, pp.173–190,1987.

[22] T. Leighton, *Tight bounds on the complexity of parallel sorting*, Proceedings of the 16th Annual ACM Symposium on Theory of Computing, pp.71–80, 1984.

[23] R. Miller and Q. F. Stout, *Efficient parallel convex hull algorithms*, IEEE Transactions on Computers, C-37, 12, pp.1605–pp.1618, 1988.

[24] R. Miller, V. K. P. Kumar, D. I. Reisis and Q. F. Stout, *Data movement operations and applications on reconfigurable VLSI arrays*, Proceedings of International Conference on Parallel Processing, 1, pp.205–208, 1988.

[25] R. Miller, V. K. P. Kumar, D. I. Reisis and Q. F. Stout, *parallel computation on reconfigurable meshes*, to appear in IEEE Transactions on Computers.

[26] D. E. Muller and F. P. Preparata, *Bounds to complexities of networks for sorting and for switching*, Journal of ACM, 22, 2, pp.195–201, 1975.

[27] K. Nakano, T. Masuzawa, K. Hagihara, and N. Tokura, *A parallel sorting algorithm on 1-dimensional grid with buses* Transactions of IEICE(DI), J72-D-I, pp.631-641, 1989, in Japanese.

[28] K. Nakano, T. Masuzawa, K. Hagihara, and N. Tokura, *A parallel algorithm for simulating communication on 2-D Grid with buses*, Transactions of IEICE(DI),J73-D-I, 3, pp.269-279, 1990 in Japanese.

[29] K. Nakano, T. Masuzawa, K. Hagihara, and N. Tokura, *An efficient parallel algorithm for image component labeling on 2-dimensional grid with buses*, Transactions of IEICE(DI), J73-D-I, 4, pp.403-414, 1990 in Japanese.

[30] C. P. Schnorr and A. Shamir, *An optimal sorting algorithm for mesh connected computers*, Proceedings of the 18th Annual ACM Symposium on Theory of Computing, pp.255–263, 1986.

[31] Q. F. Stout, *Mesh-connected computers with multiple broadcasting*, IEEE Transactions on Computers, C-32, 9,pp.826–830,1983.

[32] Q. F. Stout, *Meshes with multiple buses*, Proceedings of the 27th Annual Symposium on Foundations of Computer Science, pp.264–272, 1986.

[33] C. D. Thompson and H. K. Kung, *Sorting on a mesh-connected parallel computer*, Proceedings of the 8th Annual ACM Symposium on Theory of Computing, pp.58–64, 1976.

[34] B. F. Wang and G. H. Chen, *Two-dimensional processor array with a reconfigurable bus system is at least as powerful as CRCW MODEL* Information Processing Letters, 36, 1, pp.31–36, 1990.

[35] B. F. Wang and G. H. Chen, *Constant time algorithms for the transitive closure and some related graph problems on processor arrays with reconfigurable bus systems*, IEEE Transactions on Parallel and Distributed Systems, 1, 4, 1990.

[36] B. F. Wang and G. H. Chen, *Constant time algorithms for sorting and computing convex hulls*, to appear in Proceedings of International Computer Symposium Tsing-Hua Univ., Taiwan, 1990.

[37] B. F. Wang and G. H. Chen, *An $O(1)$ time algorithm for generating computation tree forms*, to appear in Proceedings International Computer Symposium, Tsing-Hua Univ, Taiwan, 1990.

[38] B. F. Wang, G. H. Chen and H. Li, *Fast algorithms for some arithmetic and logic operators*, Technical Report Department of Computer Science & Information Engineering National Taiwan Univ., 1990.

[39] B. F. Wang, G. H. Chen and H. Li, *Configurational computation: a new computation method on processor arrays with reconfigurable bus systems.* to appear in Proceedings of International Conference on Parallel Processing, 1991.

[40] 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, 4, pp.187–192, 1990.