

# Fast Sorting on a Linear Array with a Reconfigurable Pipelined Bus System\*

Amitava Datta, Robyn Owens, and Subbiah Soundaralakshmi

Department of Computer Science  
The University of Western Australia  
Perth, WA 6907  
Australia  
email:{datta, robyn, laxmi}@cs.uwa.edu.au

**Abstract.** We present a fast algorithm for sorting on a linear array with a reconfigurable pipelined bus system (LARPBS), one of the recently proposed parallel architectures based on optical buses. Our algorithm sorts  $N$  numbers in  $O(\log N \log \log N)$  worst-case time using  $N$  processors. To our knowledge, the previous best sorting algorithm on this architecture has a running time of  $O(\log^2 N)$ .

## 1 Introduction

Recent advances in optical and opto-electronic technologies indicate that optical interconnects can be used effectively in massively parallel computing systems involving electronic processors [1]. The delays in message propagation can be precisely controlled in an optical waveguide and this can be used to support high bandwidth pipelined communication. Several different opto-electronic parallel computing models have been proposed in the literature in recent years. These models have opened up new challenges in algorithm design. We refer the reader to the paper by Sahni [8] for an excellent overview of the different models and algorithm design techniques on these models.

Dynamically reconfigurable electronic buses have been studied extensively in recent years since they were introduced by Miller *et al.* [3]. There are two related opto-electronic models based on the idea of dynamically reconfigurable optical buses, namely, the *Array with Reconfigurable Optical Buses* (AROB) and the *Linear Array with Reconfigurable Pipelined Bus Systems* (LARPBS). The LARPBS model has been investigated in [2, 4–6] for designing fast algorithms from different domains. There are some similarities between these two models. For example, the buses can be dynamically reconfigured to suit computational and communication needs and the time complexities of the algorithms are analyzed in terms of the number of bus cycles needed to perform a computation, where a bus cycle  $\tau$  is the time needed for a signal to travel from end to end along a bus. However, there is one crucial difference between these two models. In the AROB model, the processors connected to a bus are able to count optical pulses within a bus cycle, whereas in the LARPBS model counting is not allowed during a bus cycle. In the LARPBS model, processors can set switches at the start of a bus cycle and take no further part during a bus cycle. In other words, the basic assumption of the

---

\* This research is partially supported by an Australian Research Council (ARC) grant.

AROB model is that the CPU cycle time is equal to the optical pulse time since the processors connected to a bus need to count the pulses. This is an unrealistic assumption in some sense since the pulse time is usually much faster than the CPU time of an electronic processor. On the other hand, the LARPBS model is more realistic since the basic assumption in this model is that the bus cycle time is equal to the CPU cycle time.

Sorting is undoubtedly one of the most fundamental problems in computer science and a fast sorting algorithm is often used as a preprocessing step in many other algorithms. The first sorting algorithm on the LARPBS model was designed by Pan *et al.* [7]. Their algorithm is based on the sequential quicksort algorithm and runs in  $O(\log N)$  time on an average and in  $O(N)$  time in the worst case on an  $N$  processor LARPBS. To our knowledge, the best sorting algorithm for this model is due to Pan [4]. His algorithm sorts  $N$  numbers in  $O(\log^2 N)$  worst-case time. We present an algorithm for sorting  $N$  numbers in  $O(\log N \log \log N)$  time on an LARPBS with  $N$  processors. Our algorithm is based on a novel deterministic sampling scheme for merging two sorted arrays of length  $N$  each in  $O(\log \log N)$  time.

## 2 Fast sorting on the LARPBS

We refer the reader to [2, 5, 6] for further details of the LARPBS model. The measure of computational complexity on an LARPBS is the number of bus cycles used for the computation and the amount of time spent by the processors for local computations. A bus cycle is the time needed for end to end message transmission over a bus and assumed to take only  $O(1)$  time. In most algorithms on the LARPBS model, a processor performs only a constant number of local computation steps between two consecutive bus cycles and hence the time complexity of an algorithm is proportional to the number of bus cycles used for communication.

We use some basic operations on the LARPBS in our algorithm. In a *one-to-one communication*, a source processor sends a message to a destination processor. In a *broadcasting operation*, a source processor sends a message to all the other  $N - 1$  processors in an LARPBS consisting of  $N$  processors. In a *multicasting operation*, a source processor sends a message to a group of destination processors. In a *multiple multicasting operation*, a group of source processors perform multicasting operations. A destination processor can only receive a single message during a bus cycle in a multiple multicasting operation. In the *binary prefix sum* computation, each processor in an LARPBS with  $N$  processors stores a binary value, with processor  $P_i$ ,  $1 \leq i \leq N$  storing the binary value  $b_i$ . The aim is to compute the  $N$  prefix sums  $S_i$ ,  $1 \leq i \leq N$ , where  $S_i = \sum_{j=1}^i b_j$ .

Suppose each processor in an  $N$  processor LARPBS is marked either as *active* or as *inactive* depending on whether the processor holds a 1 or a 0 in one of its registers  $R_i$ . Also, each processor holds a data element in another of its registers  $R_j$ . In the *ordered compression* problem, the data elements of all the active processors are brought to consecutive processors at the right end of the array, keeping their order in the original array intact.

The following lemma has been proved by Li *et al.* [2] and Pan and Li [5].

**Lemma 1.** *One-to-one communication, broadcasting, multicasting, multiple multicasting, binary prefix sum computation and ordered compression all can be done in  $O(1)$  bus cycles on the LARPBS model.*

Given a sequence of  $N$  numbers  $k_1, k_2, \dots, k_N$ , the sorting problem is to arrange these numbers in nondecreasing order. Our sorting algorithm on the LARPBS is based on the well known sequential *merge sort* algorithm. We use an algorithm for merging two sorted arrays of length  $N$  each in  $O(\log \log N)$  time on an LARPBS with  $N$  processors. We now give some definitions and properties which are necessary for designing our merging algorithm.

## 2.1 Definitions and properties

Suppose we have two arrays  $L = \{l_1, l_2, \dots, l_N\}$  and  $R = \{r_1, r_2, \dots, r_N\}$  each having  $N$  elements and each sorted according to ascending order. We assume for simplicity that all the elements in  $L \cup R$  are distinct. It is easy to modify our algorithm for the case when an element may occur multiple times. For an element  $l_i \in L$ , we denote its predecessor and successor in  $L$  by  $pred(l_i)$  and  $succ(l_i)$ . Successors and predecessors are denoted similarly for an element in  $R$ .

The *rank* of  $l_i$  in  $L$  is its index  $i$  in the array  $L$  and denoted by  $rank_L(l_i)$ . Similarly, the rank of  $r_i$  in  $R$  is its index  $i$  in the array  $R$  and denoted by  $rank_R(r_i)$ . The rank of  $l_i$  in  $R$ , denoted by  $rank_R(l_i)$ , is  $rank_R(r_j)$  of an element  $r_j \in R$  such that  $r_j < l_i$  and there is no other element  $r_k \in R$  such that  $r_j < r_k < l_i$ . Sometime we will write  $rank_R(l_i) = r_j$  by abusing the notation.

Similarly, the rank of  $r_i$  in  $L$ , denoted by  $rank_L(r_i)$ , is  $rank_L(l_j)$  of an element  $l_j$  in  $L$  such that  $l_j < r_i$  and there is no other element  $l_k \in L$  such that  $l_j < l_k < r_i$ . For an element  $l_m \in L$ , the rank of  $l_m$  in  $L \cup R$  is denoted by  $rank(l_m)$ . The following lemma is a direct consequence of definitions of these three kinds of ranks.

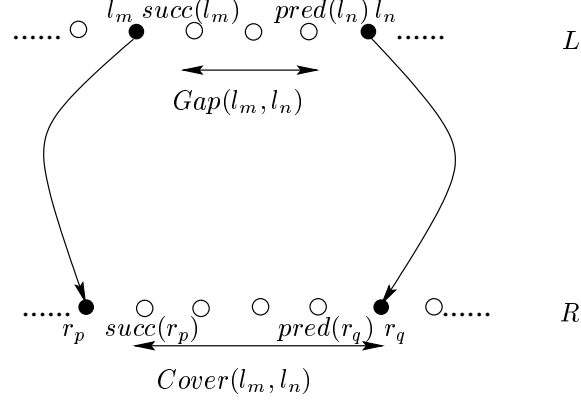
**Lemma 2.** *For an element  $l_m \in L$ ,  $1 \leq m \leq N$ ,  $rank(l_m) = rank_L(l_m) + rank_R(l_m)$ . Similarly, for an element  $r_n \in R$ ,  $1 \leq n \leq N$ ,  $rank(r_n) = rank_R(r_n) + rank_L(r_n)$ .*

It is clear from Lemma 2 that if we compute  $rank_R(l_i)$  for each element  $l_i \in L$ , we can compute  $rank(l_i)$ . Note that, we already know  $rank_L(l_i)$  since  $L$  is already sorted and  $rank_L(l_i)$  is simply the index  $i$ . Similarly, if we compute  $rank_L(r_j)$  for each element  $r_j \in R$ , we can compute  $rank(r_j)$ . We refer to these two problems as *ranking of  $L$  in  $R$*  and *ranking of  $R$  in  $L$* .

We do the ranking of  $L$  in  $R$  recursively in several stages. When every element in  $L$  is ranked in  $R$ , we say that  $L$  is *saturated*. Consider a stage when  $L$  is still unsaturated. In other words, some elements in  $L$  are already ranked in  $R$  and some are yet to be ranked.

**Definition 3** *Consider two consecutive ranked elements  $l_m$  and  $l_n$ ,  $m < n$ . All the elements between  $l_m$  and  $l_n$ , i.e.,  $succ(l_m), \dots, pred(l_n)$  are unranked and these elements are called the *gap* between  $l_m$  and  $l_n$  and denoted by  $Gap(l_m, l_n)$ .*

**Definition 4** Consider two consecutive ranked elements  $l_m$  and  $l_n$  in  $L$ . Suppose,  $\text{rank}_R(l_m) = r_p$  and  $\text{rank}_R(l_n) = r_q$ . The elements  $\text{succ}(r_p), \dots, r_q$  are collectively called the cover of  $\text{Gap}(l_m, l_n)$  and denoted as  $\text{Cover}(l_m, l_n)$ . See Figure 1 for an illustration.



**Figure 1.** Illustration for gap and cover.

**Lemma 5.** For an element  $l_i \in \text{Gap}(l_m, l_n)$ , either  $\text{rank}_R(l_i) = \text{rank}_R(l_m)$  or  $\text{rank}_R(l_i) = r_m$  such that,  $r_m \in \text{Cover}(l_m, l_n)$ .

**Definition 6** For two ranked elements  $l_m, l_n \in L$ , if  $l_n \neq \text{succ}(l_m)$ , we say that  $\text{Gap}(l_m, l_n)$  is non-empty.

**Definition 7** Consider a non-empty gap  $\text{Gap}(l_m, l_n)$  and its  $\text{Cover}(l_m, l_n)$ . We say that  $\text{Gap}(l_m, l_n)$  has an **empty cover** if  $\text{rank}_R(l_m) = \text{rank}_R(l_n)$ , i.e., if  $l_m$  and  $l_n$  are ranked at the same element in  $R$ .

The following two lemmas are crucial for our algorithm.

**Lemma 8.** If  $\text{Cover}(l_m, l_n)$  is the non-empty cover for  $\text{Gap}(l_m, l_n)$ , an element  $r_j \in \text{Cover}(l_m, l_n)$  must be ranked in  $\text{Gap}(l_m, l_n)$ .

**Lemma 9.** If  $\text{Gap}(l_m, l_n)$  and  $\text{Gap}(l_o, l_p)$  are two arbitrary and distinct non-empty gaps in  $L$ , then  $\text{Gap}(l_m, l_n) \cap \text{Gap}(l_o, l_p) = \emptyset$ . Similarly, if  $\text{Cover}(l_m, l_n)$  and  $\text{Cover}(l_o, l_p)$  are two arbitrary and distinct non-empty covers in  $R$ , then  $\text{Cover}(l_m, l_n) \cap \text{Cover}(l_o, l_p) = \emptyset$ .

We assume that the sorted sequences  $L$  and  $R$  have  $N$  and  $M$  elements respectively. First, we choose every  $\sqrt{N}$ -th element, i.e., the elements  $l_{\sqrt{N}}, l_{2\sqrt{N}}, \dots, l_{\sqrt{N}\sqrt{N}}$  from  $L$ . We denote the set  $\{l_{\sqrt{N}}, l_{2\sqrt{N}}, \dots, l_{\sqrt{N}\sqrt{N}}\}$  as  $\text{Sample}_L$ . Similarly, we choose

the elements  $r_{\sqrt{M}}, r_{2\sqrt{M}}, \dots, r_{\sqrt{M}\sqrt{M}}$  from  $R$  and denote this set of elements as  $Sample_R$ . Note that there are  $\sqrt{N}$  elements in  $Sample_L$  and  $\sqrt{M}$  elements in  $Sample_R$ .

The elements  $l_{i\sqrt{N}}$  (resp.  $r_{i\sqrt{M}}$ ),  $1 \leq i \leq \sqrt{N}$  in  $Sample_L$  (resp.  $Sample_R$ ) impose a block structure on the sequence  $L$  (resp.  $R$ ). Consider two consecutive elements  $l_{i\sqrt{N}}$  and  $l_{(i+1)\sqrt{N}}$  in  $Sample_L$ . The elements  $\{succ(l_{i\sqrt{N}}), \dots, l_{(i+1)\sqrt{N}}\}$  are called the  $i$ -th block in  $L$  imposed by  $Sample_L$  and denoted by  $Block_i^L$ . The superscript  $L$  indicates that it is a block in the sorted sequence  $L$ . The elements  $l_{i\sqrt{N}}$  and  $l_{(i+1)\sqrt{N}}$  are called the *sentinels* of  $Block_i^L$ . Similarly, we define the  $j^{th}$  block  $Block_j^R$  imposed by two consecutive elements  $r_{j\sqrt{M}}$  and  $r_{(j+1)\sqrt{M}}$  of  $Sample_R$ .

Consider the ranking of  $Sample_L$  in  $Sample_R$ . When an element  $l_{i\sqrt{N}} \in Sample_L$  is ranked in  $Sample_R$ , we denote this rank by a superscript  $S$ , i.e.,  $rank_R^S(l_{i\sqrt{N}})$ . Note that,  $rank_R^S(l_{i\sqrt{N}})$  is only an approximation of the true rank  $rank_R(l_{i\sqrt{N}})$  of  $l_{i\sqrt{N}}$  in  $R$ .

Assume that for two consecutive elements  $l_{k\sqrt{N}}$  and  $l_{(k+1)\sqrt{N}}$  in  $Sample_L$ ,  $rank_R^S(l_{k\sqrt{N}}) = r_{m\sqrt{M}}$  and  $rank_R^S(l_{(k+1)\sqrt{N}}) = r_{n\sqrt{M}}$ , where  $r_{m\sqrt{M}}$  and  $r_{n\sqrt{M}}$  are two elements in  $Sample_R$ . In the following lemma, we estimate the true ranks of the elements in  $Block_k^L$  in  $R$ .

**Lemma 10.** *If an element  $l_r \in L$  is in  $Block_k^L$ , i.e., in between the two elements  $l_{k\sqrt{N}}$  and  $l_{(k+1)\sqrt{N}}$ ,  $l_r$  must be ranked in  $Block_m^R \cup Block_{m+1}^R \cup \dots \cup Block_n^R$ , i.e., in  $Cover(l_{k\sqrt{N}}, l_{(k+1)\sqrt{N}})$ .*

## 2.2 An $O(\log \log N)$ time merging algorithm on the LARPBS

A variant of the following lemma has been proved by Pan *et al.* [7].

**Lemma 11.** *Given two sorted sequences  $A$  and  $B$  of length  $\sqrt{N}$  each, all the elements of  $A$  can be ranked in  $B$  in  $O(1)$  bus cycles on an LARPBS with  $N$  processors.*

Our algorithm is recursive and at every level of recursion, our generic task is to set up appropriate subproblems for the next level of recursion. In the following description, we explain how all the subproblems associated with  $Gap(l_m, l_n)$  and  $Cover(l_m, l_n)$  are set up for the next level of recursion. We assume that  $Gap(l_m, l_n)$  has  $N'$  elements and  $Cover(l_m, l_n)$  has  $M'$  elements.

### Step 1.

We take a sample from  $Gap(l_m, l_n)$  by choosing every  $\sqrt{N'}$ -th element from  $Gap(l_m, l_n)$ . We denote this sample by  $Sample_L(Gap(l_m, l_n))$ . Similarly, we take a sample from  $Cover(l_m, l_n)$  by choosing every  $\sqrt{M'}$ -th element from  $Cover(l_m, l_n)$  and denote it by  $Sample_R(Cover(l_m, l_n))$ . We explain how to take the sample from  $Gap(l_m, l_n)$ . The sample from  $Cover(l_m, l_n)$  is taken in a similar way.

First, each processor holding an element in  $Gap(l_m, l_n)$  writes a 1 in one of its registers. Next, a parallel prefix computation is done in one bus cycle to get  $N'$ , the total number of elements in  $Gap(l_m, l_n)$  in the processor holding  $l_n$ . This processor computes  $\sqrt{N'}$  and broadcasts  $\sqrt{N'}$  to all the processors in  $Gap(l_m, l_n)$ . We assume for simplicity that  $\sqrt{N'}$  is an integer. Each processor in  $Gap(l_m, l_n)$  determines whether its prefix sum is an integer multiple of  $\sqrt{N'}$  and marks itself as a member of

$Sample_L(Gap(l_m, l_n))$  accordingly. Note that,  $Sample_L(Gap(l_m, l_n))$  consists of the sentinels of the blocks in  $L$ .

**Step 2.**

In this step, we assume that  $\sqrt{N'} < \sqrt{M'}$  and we rank  $Sample_L(Gap(l_m, l_n))$  in  $Sample_R(Cover(l_m, l_n))$ . This ranking is done by the method in Lemma 11 in  $O(1)$  bus cycles.

**Step 3.**

After the ranking in Step 2 is over, for every sentinel  $l_{k\sqrt{N'}} \in Sample_L(Gap(l_m, l_n))$ , we know  $Block_m^R$ , the block of  $\sqrt{M'}$  elements in  $R$  in which  $l_{k\sqrt{N'}}$  should be ranked.

Next, we determine all the sentinels in  $Sample_L(Gap(l_m, l_n))$  ranked in  $Block_m^R$  in the following way. After the ranking in Step 2 is over, each processor holding a sentinel  $l_{i\sqrt{N'}}$  gets  $rank_R^S(l_{(i+1)\sqrt{N'}}$ ) from its neighbor in the sample through a one-to-one communication. After this, a group of consecutive sentinels in  $Sample_L(Gap(l_m, l_n))$  which are ranked at the same block of  $Sample_R(Cover(l_m, l_n))$  can be determined.

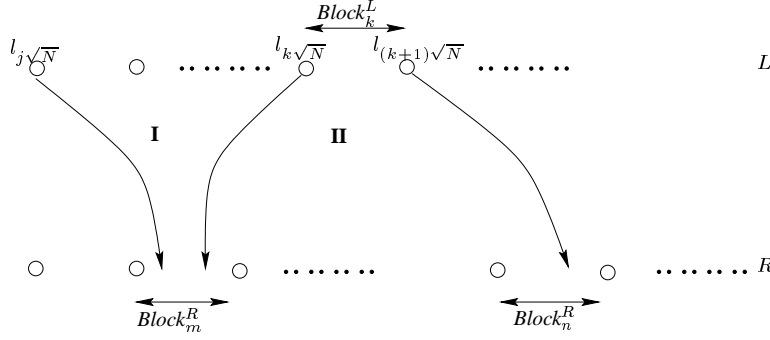
We consider two cases depending on whether a single sentinel or multiple sentinels from  $Sample_L(Gap(l_m, l_n))$  are ranked in the same block of  $Sample_R(Cover(l_m, l_n))$ .

**Case i.** In this case, only one sentinel  $l_{k\sqrt{N'}}$  in  $Sample_L(Gap(l_m, l_n))$  is ranked in  $Block_m^R$ . The processor holding  $l_{k\sqrt{N'}}$  broadcasts  $l_{k\sqrt{N'}}$  to all the processors in  $Block_m^R$  and the processors in  $Block_m^R$  determine  $rank_R(l_{k\sqrt{N'}}$ ). This takes  $O(1)$  bus cycles. We determine  $rank_R(l_{(k+1)\sqrt{N'}}$ ) in  $Block_n^R$  in a similar way. Note that, the elements  $succ(rank_R(l_{k\sqrt{N'}}), \dots, rank_R(l_{(k+1)\sqrt{N'}}))$  are the elements in  $Cover(l_{k\sqrt{N'}}, l_{(k+1)\sqrt{N'}}$ .

It follows from Lemma 5 that all the elements in  $Gap(l_{k\sqrt{N'}}, l_{(k+1)\sqrt{N'}}$ ) must be ranked either at  $rank_R(l_{k\sqrt{N'}}$ ) or among the elements in  $Cover(l_{k\sqrt{N'}}, l_{(k+1)\sqrt{N'}}$ ). Similarly, it follows from Lemma 8 that all the elements in  $Cover(l_{k\sqrt{N'}}, l_{(k+1)\sqrt{N'}}$ ) must be ranked at the elements in  $Gap(l_{k\sqrt{N'}}, l_{(k+1)\sqrt{N'}}$ ). Hence we recursively call our algorithm with elements in  $Gap(l_{k\sqrt{N'}}, l_{(k+1)\sqrt{N'}}$ ) and elements in  $Cover(l_{k\sqrt{N'}}, l_{(k+1)\sqrt{N'}}$ ). In this recursive call, all the elements from  $L$  are within a block of size  $\sqrt{N'}$ . The processors holding the elements in  $Gap(l_{k\sqrt{N'}}, l_{(k+1)\sqrt{N'}}$ ) and the elements in  $Cover(l_{k\sqrt{N'}}, l_{(k+1)\sqrt{N'}}$ ) participate in this recursive call.

**Case ii.** In this case, multiple sentinels  $l_{j\sqrt{N'}}, \dots, l_{k\sqrt{N'}}$  are ranked in  $Block_m^R$ . In two bus cycles, first  $rank_R(l_{j\sqrt{N'}}$ ) and then  $rank_R(l_{k\sqrt{N'}}$ ) are determined by broadcasting first  $l_{j\sqrt{N'}}$  and then  $l_{k\sqrt{N'}}$  to all the processors in  $Block_m^R$ . We then recursively call our algorithm with the elements in  $Gap(l_{j\sqrt{N'}}, l_{k\sqrt{N'}}$ ) and the elements in  $Cover(l_{j\sqrt{N'}}, l_{k\sqrt{N'}}$ ). Note that, all the elements from  $R$  are within a block of size  $\sqrt{M'}$  in this recursive call.

These two types of recursive calls are illustrated in Figure 2.



**Figure 2.** The two types of recursive calls are indicated by **I** and **II**. In the first type, the elements from  $R$  are within the same block of size  $\sqrt{M'}$ . In the second type, the elements from  $L$  are within the same block of size  $\sqrt{N'}$ .

Note that, the inputs to each level of recursion are disjoint subsets of processors holding elements of  $L$  and  $R$  and hence all the one-to-one communication, broadcasting and multiple multicasting operations at each level of recursion for each of the subproblems can be done simultaneously in parallel.

Once the recursive calls return, an element  $l_i \in L$  knows  $rank_R(l_i)$  and it knows  $rank_L(l_i)$  since  $L$  is already sorted. Hence the processor holding  $l_i$  can compute  $rank(l_i)$  and sends  $l_i$  to the processor with index  $rank(l_i)$  through a one-to-one communication. This can be done in one bus cycle. Similarly the overall rank of each element in  $R$  can be computed and the elements can be sent to the appropriate processors. Hence each processor  $P_i$  will hold the  $i^{th}$  element in  $L \cup R$  after the merging algorithm terminates.

This concludes the description of our merging algorithm.

**Lemma 12.** *The merging algorithm terminates in  $O(\log \log N)$  bus cycles with all the elements of  $L$  ranked in  $R$  and all the elements of  $R$  ranked in  $L$ .*

*Proof. (sketch)* Suppose in the  $i^{th}$  level of recursion, each block in  $L$  and  $R$  is of size  $\sqrt{N}$  and  $\sqrt{M}$  respectively. Suppose, the input to one of the recursive calls at the  $(i + 1)^{th}$  level of recursion are the elements in two groups of processors  $G^L$  from  $L$  and  $G^R$  from  $R$ . From the description of the algorithm, it is clear that either  $G^L$  is within a block of size  $\sqrt{N}$  or  $G^R$  is within a block of size  $\sqrt{M}$ . Hence, due to this recursive call, at the  $(i + 1)^{th}$  level of recursion, either we get new blocks of size  $N^{1/4}$  in  $L$  or we get new blocks of size  $M^{1/4}$  in  $R$ . This gives a recurrence of:  $T(N) = T(\sqrt{N}) + O(1)$  or a recurrence of:  $T(M) = T(\sqrt{M}) + O(1)$ , since each level of recursion takes  $O(1)$  bus cycles. Hence, the recursion stops after  $2 \log \log N$  levels and all the elements in  $L$  and  $R$  are ranked at that stage.

### 2.3 The sorting algorithm

**Phase 1.** Initially, each processor in an  $N$  processor LARPBS holds one element from the input. The complete LARPBS with  $N$  processors is recursively divided in this phase.

Consider a subarray with processors  $P_i, P_{i+1}, \dots, P_j$  to be divided into two equal parts. Each processor writes a 1 in one of its registers and a prefix computation is done to renumber the processors from 1 to  $j - i$ . Now, the last prefix sum is broadcast to all the processors and the processor with index  $\lfloor (j + i)/2 \rfloor$  splits the bus to divide the original subarray into two subarrays of equal size. This process is repeated for all the subarrays recursively until each subarray contains only one processor and one element which is trivially sorted. This phase can be completed in  $O(\log N)$  bus cycles.

**Phase 2.** The merging is done in this phase using the algorithm in Section 2.2. In the generic merging step, a pair of adjacent subarrays of equal size merge their elements to form a larger subarray of double the size. Each subarray participating in this pairwise merging first renumber its processors starting from 1 and then the merging algorithm is applied. At the end, processor  $P_i, 1 \leq i \leq N$  in the original array holds the element with rank  $i$  from the input set.

Since there are  $O(\log N)$  levels in the recursion and the merging at each level can be performed in  $O(\log \log N)$  bus cycles, the overall algorithm takes  $O(\log N \log \log N)$  bus cycles and hence  $O(\log N \log \log N)$  time since each bus cycle takes  $O(1)$  time.

**Theorem 1.**  $N$  elements can be sorted in  $O(\log N \log \log N)$  deterministic time on an LARPBS with  $N$  processors.

## References

1. Z. Guo, R. Melhem, R. Hall, D. Chiarulli, S. Levitan, "Pipelined communication in optically interconnected arrays", *Journal of Parallel and Distributed Computing*, **12**, (3), (1991), pp. 269-282.
2. K. Li, Y. Pan and S. Q. Zheng, "Fast and processor efficient parallel matrix multiplication algorithms on a linear array with a reconfigurable pipelined bus system", *IEEE Trans. Parallel and Distributed Systems*, **9**, (8), (1998), pp. 705-720.
3. R. Miller, V. K. Prasanna Kumar, D. Reisis and Q. F. Stout, Parallel computations on reconfigurable meshes. *IEEE Trans. Computers*, **42**, (1993), 678-692.
4. Y. Pan, "Basic data movement operations on the LARPBS model", in *Parallel Computing Using Optical Interconnections*, K. Li, Y. Pan and S. Q. Zheng, eds, Kluwer Academic Publishers, Boston, USA, 1998.
5. Y. Pan and K. Li, "Linear array with a reconfigurable pipelined bus system - concepts and applications", *Journal of Information Sciences*, **106**, (1998), pp. 237-258.
6. Y. Pan, M. Hamdi and K. Li, "Efficient and scalable quicksort on a linear array with a reconfigurable pipelined bus system", *Future Generation Computer Systems*, **13**, (1997/98), pp. 501-513.
7. Y. Pan, K. Li and S. Q. Zheng, "Fast nearest neighbor algorithms on a linear array with a reconfigurable pipelined bus system", *Journal of Parallel Algorithms and Applications*, **13**, (1998), pp. 1-25.
8. S. Sahni, "Models and algorithms for optical and optoelectronic parallel computers", *Proc. 1999 International Symposium on Parallel Architectures, Algorithms and Networks*, IEEE Computer Society, pp. 2-7.