2002

Efficient parallel computation on multiprocessors with optical interconnection networks

Min He
Louisiana State University and Agricultural and Mechanical College

EFFICIENT PARALLEL COMPUTATION ON MULTIPROCESSORS WITH OPTICAL INTERCONNECTION NETWORKS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Min He
B.S., Hunan University, P.R.China, 1988
M.E., Hunan University, P.R.China, 1991
M.S., Louisiana State University, 1997
December, 2002

Acknowledgments

I am deeply grateful to my advisor, Dr. Si Qing Zheng. He has been very supportive and encouraging during my time as a graduate student. His rigorous research attitude, breadth of knowledge, and innovative ideas have been a great benefit for me to work on my Ph.D research.

I would like to give my special thanks to Dr. Donald Kraft for his support and all the helps he provided me during my time as a part-time Ph.D student. Whenever I need help, he is always there helping me.

I would like to thank Dr. Ramachandran Vaidyanathan. I learned a lot from his courses on methods and models on parallel computation. The theories and knowledge I learned from his courses laid a good foundation for my later research in parallel computation area.

I would like to thank Dr. Michael Cherry, Dr. S. Sitharama Iyengar, Dr. Jianhua Chen, Dr. Donald Kraft, and Dr. Ramachandran Vaidyanathan for serving on my Ph.D committee and their helpful comments and suggestions on my dissertation.

I would like to thank my parents, Yi Guo and Guirong He, for their endless love and support, for taking care of my daughter so that I could concentrate on my Ph.D research.

I would like to thank my husband, Qiang Liu, for his love, understanding and support.

I would like to thank my lovely daughter, Yuanyuan. She is an angel in my life and she motivates me to do my best for this dissertation.

I would like to thank my brother Kai and my sister Pei for their support and love.

Table of Contents

List of Tables

List of Figures

Abstract

This dissertation studies optical interconnection networks, their architecture, address schemes, and computation and communication capabilities. We focus on a simple but powerful optical interconnection network model – the Linear Array with Reconfigurable pipelined Bus System (LARPBS). We extend the LARPBS model to a simplified higher dimensional LAPRBS and provide a set of basic computation operations.

We then study the following two groups of parallel computation problems on both one dimensional LARPBS's as well as multi-dimensional LARPBS's: parallel comparison problems, including sorting, merging, and selection; Boolean matrix multiplication, transitive closure and their applications to connected component problems.

We implement an optimal sorting algorithm on an n-processor LARPBS. With this optimal sorting algorithm at disposal, we study the sorting problem for higher dimensional LARPBS's and obtain the following results:

- An optimal basic Columnsort algorithm on a 2D LARPBS.
- Two optimal two-way merge sort algorithms on a 2D LARPBS.
- An optimal multi-way merge sorting algorithm on a 2D LARPBS.
- An optimal generalized column sort algorithm on a 2D LARPBS.
- An optimal generalized column sort algorithm on a 3D LARPBS.
- An optimal 5-phase sorting algorithm on a 3D LARPBS.

Results for selection problems are as follows:

- A constant time maximum-finding algorithm on an LARPBS.
- An optimal maximum-finding algorithm on an LARPBS.
- An $O((\log\log n)^2)$ time parallel selection algorithm on an LARPBS.
- An $O(k(\log\log n)^2)$ time parallel multi-selection algorithm on an LARPBS.

While studying the computation and communication properties of the LARPBS model, we find Boolean matrix multiplication and its applications to the graph are another set of problem that can be solved efficiently on the LARPBS. Following is a list of results we have obtained in this area.

- A constant time Boolean matrix multiplication algorithm.
- An $O(\log n)$-time transitive closure algorithm.
- An $O(\log n)$-time connected components algorithm.
- An $O(\log n)$-time strongly connected components algorithm.

The results provided in this dissertation show the strong computation and communication power of optical interconnection networks.

Chapter 1

Introduction

1.1 Background

The performance of many existing computational systems based on micro-electronic integrated circuits is not limited by the devices themselves but by the performance of communication links between devices. Though the interconnect bottleneck occurs at all levels of electronic packaging, there are three major levels:

- *Level one*: chip-level(intra-chip) interconnections between devices on a single very-large-scale-integrated(VLSI) circuit or chip.
- *Level two*: multi-chip module (MCM) level (intra-MCM or chip-to-chip) interconnections are connections between VLSI chips within a single MCM.
- *Level three*: board-level (inter-MCM) interconnections are connections between MCMs on a single printed circuit board.

The use of optical interconnects with optical signal paths to replace particular electrical connection links can result in the following advantages: increased communication speed, reduced power consumption, reduced cross talk, increased reliability, increased fault tolerance, reduced cost, enhanced circuit testing capabilities, and provision of a path toward dynamic interconnects.

Optical waveguides have enhanced bandwidth, loosened loading constraints and large physical distribution of computing resources. What's more, the unique property of optics, namely, unidirectional propagation of signals, can be used to further increase the effective bandwidth of optical buses. Three multi-access methods are used in optical bus systems, namely, space-multiplexing, time-division multiplexing (TDM), and wavelength-division multiplexing.

- *Space Multiplexing [44]* : Space Multiplexing is a technique for pipelining messages on optical buses. The pipelining of control and data signals represents a significant

departure from the conventional exclusive access discipline which characterizes electronic bus-interconnected multiprocessors. By relaxing the exclusive access requirement, space multiplexing can support the design of large-scale, distributed, and tightly coupled multiprocessor systems.

- *Time-Division Multiplexing (TDM)* : Time-division multiplexing of messages has the same effect as message pipelining on optical waveguides. The TDM access protocol may be used to improve the temporal bandwidth of interconnection network.

- *Wavelength Division Multiplexing (WDM)* : WDM is a technology used in optical interconnection networks, where the optical spectrum is divided into many different logical channels, and each channel corresponds to a different wavelength. These channels can be carried simultaneously on a small number of physical channels, e.g., optical fibers. WDM exploits the terahertz (THz) bandwidth of optics, and enables multiple multiaccess channels to be realized on a single physical channel for large parallel computing systems.

1.2 Main Results

In this dissertation, we study both optical interconnection models and efficient algorithms for these models. We focus on a simple but powerful optical interconnection network model – A Linear Array with Reconfigurable Pipelined Bus System (LARPBS). We extend this one-dimensional optical bus model to two and three dimension to improve the scalability. We also provide several basic operations for two-dimensional LARPBS's. These basic operations are used as building blocks for designing a few efficient algorithms given in this dissertation.

Then we study parallel comparison problems, including sorting, merging and selection, the lower bounds for these problems, and the best parallel algorithms for these problems that were designed under PRAM and parallel comparison models. Sorting is a well-known problem that has been studied extensively in the literature because it has so many important applications, and it has a theoretical importance as well. There are two optimal parallel sorting algorithms: Cole's pipelined merge sort [11] and the AKS sorting network [1]. Both of them sort n elements in $O(\log n)$ time. Since the lower bound of parallel sorting problem is $\Omega(\log n)$ [76], both algorithms are actually $\Theta(\log n)$ algorithms. We chose to implement Cole's pipelined merge sort because it has a much smaller constant in running time and it does not depend on expander graphs. Li, Pan and Zheng [39] provided some methods for simulating a CRCW PRAM on an LARPBS. But as their simulation methods simulate a CRCW PRAM on an LARPBS with an $O(\log n)$ slowdown, we cannot obtain an $O(\log n)$ optimal sorting algorithm by simply applying their simulation methods. Cole also gave an EREW PRAM sorting algorithm with the same time complexity in [11]. Given that an EREW PRAM can be simulated on an LARPBS in constant time [39], another approach would be using the simulation methods given in [39] to implement this EREW PRAM sorting algorithm on an LARPBS. But the EREW PRAM sorting algorithm has a larger constant

in running time and it is much more complex. We observe that in this algorithm the owners of the data always know the address of the processors who need the data for all the required concurrent read operations. Thus, concurrent read can be done by performing a multicast operation on an LARPBS. Based on the above analysis and observation, we decide to directly implement the CREW PRAM merge sort algorithm on an LARPBS.

A lot of parallel algorithms [24, 31, 35, 37, 38, 52, 59, 60, 62, 72, 74] have been designed and implemented for the LARPBS since it is first proposed by Pan and Li [61]. Currently the best known sorting algorithms implemented on an n-processor LARPBS sort n elements in $O(\log n)$ expected time or $O(\log^2 n)$ deterministic time [59], [36]. Thus to the best of our knowledge, our $O(\log n)$-time sorting algorithm is the first optimal sorting algorithm implemented on an LARPBS. With this optimal sorting algorithm for one-dimensional LARPBS at disposal, we further extend our research on the sorting problem for higher dimensional LARPBS's.

We started our research on two dimensional sorting problems with the well-known Columnsort algorithms. There are two known versions of Columnsort algorithm: the 7-phase Columnsort [30] and the 8-phase Columnsort [29]. Since the 8-phase algorithm requires extra processors for some of its phases and it works under a more restrictive assumption ($r \geq 2(s-1)^2$, where r is the number of rows and s is the number of columns), we chose to implement the 7-phase Columnsort. We obtained our $O(\log n)$-time 7-phase Columnsort algorithm by applying our optimal sorting algorithm on one-dimensional LARPBS to sorting each column. But this 7-phase Columnsort only works under the assumption of $r \geq s^2$, where r is the number of rows and s is the number of columns. So we ask the following question: Can we loosen the assumption so that it applies to a larger range of applications? Or can we obtain an optimal sorting algorithm for an $n \times n$ 2D LARPBS? We came up with the following two solutions to these questions:

1. Generalize the idea of basic Columnsort: we can simulate an $n\sqrt{n} \times \sqrt{n}$ imaginary matrix on the $n \times n$ 2D LARPBS, and we can also simulate an $n^2 \times n$ imaginary matrix on the $n \times n \times n$ 3D LARPBS. Since the assumption $r \geq s^2$ is satisfied in both cases, we can apply the 7-phase Columnsort algorithm on the two imaginary matrices. We called the sorting algorithms obtained this way *generalized Columnsort* algorithms.

2. Multi-way merge sorted sub-matrices: apply a multi-way merge method to $\sqrt{n}$ $n \times \sqrt{n}$ shape sub-matrices to obtain a sorted $n \times n$ shape matrix. We observed that the most time consuming operations needed in the multi-way merge procedure are sorting an n-processor one-dimensional LARPBS and sorting an $n \times 2\sqrt{n}$ 2D LARPBS. We already have an optimal sorting algorithm for one-dimensional LARPBS. If we can obtain an optimal two-way merge sort algorithm, we can get an optimal algorithm for $n \times n$ shape matrix.

We designed and implemented following optimal sorting algorithms based on the above two solutions:

1. An optimal basic Columnsort algorithm on an $n \times \sqrt{n}$ two-dimensional LARPBS that sorts $n\sqrt{n}$ elements in $O(\log n)$ time.

2. An optimal two-way merge sort algorithm on an $n \times 2\sqrt{n}$ two-dimensional LARPBS that sorts $2n\sqrt{n}$ in $O(\log n)$ time.

3. An optimal two-way merge sort algorithm on an $n \times \sqrt{n}$ two-dimensional LARPBS that sorts $2n\sqrt{n}$ in $O(\log n)$ time.

4. An optimal multi-way merge sorting algorithm on an $n \times n$ two-dimensional LARPBS that sorts n^2 elements in $O(\log n)$ time.

5. An optimal generalized column sort algorithm on an $n \times n$ two-dimensional LARPBS that sorts n^2 elements in $O(\log n)$ time.

6. An optimal generalized column sort algorithm on an $n \times n \times n$ three-dimensional LARPBS that sorts n^3 elements in $O(\log n)$ time.

7. An optimal 5-phase sorting algorithm on an $n \times n \times n$ three-dimensional LARPBS that sorts n^3 elements in $O(\log n)$ time.

Selection is another comparison problem. A typical application for selection is multi-selection – a fundamental algorithmic problem arising frequently in databases. The easiest cases are extreme selection, which is finding the minimum or maximum. The hardest case is finding the median. Valiant proved the lower bound of finding the maximum of n elements to be $\Omega(\log \log n)$ in [76]. This implies that the lower bound for selection problems is $\Omega(\log \log n)$. Since sorting is a complete selection, the upper bound $O(\log n)$ for sorting provided an upper bound for selection problem. Researches for an upper bound are done on both PRAM models and Valiant's parallel comparison model [76].

Cole and Yap [16] presented a selection algorithm on parallel comparison model in year 1984. This selection algorithm improved the upper bound of selection problem to $O((\log \log n)^2)$. Two years later, a even faster selection algorithm was presented in [3] that pushes the upper bound further down to $O(\log \log n)$. The existence of such an algorithm implies that selection problem has an upper and lower bound of $\log \log n$. The best previous result for selection on the PRAM is given in [13] that uses n operations and runs in $O(\log n \log \log n)$ on a CRCW PRAM.

We did research on the two upper bound algorithms [16][3] on parallel comparison model. And we implemented the Cole-Yap's $O((\log \log n)^2)$-time selection algorithm on an LARPBS. Since the $O(\log \log n)$ algorithm is designed for parallel comparison model, time spent for calculating transitive closure is not counted as transitive closure can be obtained free of comparison cost. Since we have to count the time spent for calculating transitive closure in the LARPBS model, we cannot implement this upper bound algorithm with the same time complexity unless we can find a constant time transitive closure algorithm. It has caught our attention that Han and Pan provided an $O((\log \log n)^2/\log \log \log n)$-time selection algorithm in [24]. But their algorithm is based on Cole's EREW PRAM selection algorithm [12]. By using a fast sorting algorithm to sort a smaller set of data on the LARPBS model, Han and Pan were able to speedup Cole's algorithm and obtained the $O((\log \log n)^2/\log \log \log n)$

time selection algorithm. Our algorithm is different from Han and Pan's algorithm. Our algorithm is based on Cole's selection algorithm designed on parallel comparison model. It is conceptually simpler than the EREW PRAM selection algorithm [12].

To find an optimal maximum algorithm, we studied the accelerated cascading technology. We provided a solution for simulating doubly logarithmic tree on an LARPBS, and implemented an optimal algorithm for finding the maximum of n elements on an n-processor LARPBS. Following is a list of algorithms we implemented for selection problems.

1. A constant time algorithm that finds the maximum of n elements on an n^2-processor LARPBS.

2. A non-optimal algorithm that finds the maximum of n elements on an n-processor LARPBS in $O(\log\log n)$ time.

3. An optimal algorithm that finds the maximum of n elements on an n-processor LARPBS in $O(\log\log n)$ time.

4. An $O((\log\log n)^2)$ time parallel selection algorithm on an n-processor one-dimensional LARPBS.

5. An $O(k(\log\log n)^2)$ time parallel multi-selection algorithm on an n-processor one-dimensional LARPBS.

Apart from the comparison problems, we also studied Boolean matrix multiplication problem and its applications to graph problems. We presented a new constant time Boolean matrix algorithm on an LARPBS. Our algorithm achieves much better speed-up compared with the four Russians' algorithm proposed by Agerwala and Lint in [4] and a hypercube implementation provided by Plaxton [55]. We noticed that a parallel implementation of the Four Russians' algorithm on the LARPBS with constant running time is proposed by Keqin Li in [31]. Compared with Li's constant-time algorithm, our algorithm is not based on any existing sequential or parallel algorithm and it is much simpler and uses fewer processors. Our algorithm uses only $O(n^2)$ processors, while Li's algorithm uses $O(n^3/\log n)$ processors.

We then applied the constant time Boolean matrix multiplication algorithm to two graph problems, and obtained an efficient transitive closure algorithm and two connected component algorithms. We noticed that a constant time algorithm for the transitive closure was proposed in [77]. But their constant time algorithm is obtained at a cost of more processors. Two constant time algorithms were given in [77]. One is designed on a three-dimensional $n \times n \times n$ processor array with a reconfigurable bus system and the other is designed on a two-dimensional $n^2 \times n^2$ processor array with a reconfigurable bus system, where n is the number of vertices in the graph.

Following is a list of results we have obtained in this area.

1. A constant time algorithm on an n^2-processor one-dimensional LARPBS that multiplies two $n \times n$ Boolean matrices.

2. A constant time algorithm on an $n \times n$ two-dimensional LARPBS that multiplies two $n \times n$ Boolean matrices.

3. An $O(\log n)$-time algorithm on an n^2-processor one-dimensional LARPBS that calculates the transitive closure for a undirected graph G with n vertices.

4. An $O(\log n)$-time algorithm on an n^2-processor one-dimensional LARPBS that calculates the connected components for a undirected graph G with n vertices.

5. An $O(\log n)$-time algorithm on an n^2-processor one-dimensional LARPBS that calculates the strongly connected components for a directed graph G with n vertices.

1.3 Organization of the Dissertation

The rest of the dissertation is organized into chapters as follows. In Chapter 2, we first compare several similar optical bus models with the two models used in this dissertation. Then we give a detailed description on the two models, i.e. LARPBS and 2D LARPBS, and basic operations designed for these two models. In Chapter 3, we design a simple constant-time Boolean matrix multiplication algorithm and apply it to solving the transitive closure problem. In Chapter 4, we present an optimal sorting algorithm on LARPBS with a solution on processor addressing and reuse. In Chapter 5, we present two optimal sorting algorithms on 2D LARPBS as well as Columnsort and two-way Columnsort algorithm on 2D LARPBS. In Chapter 6, we present two optimal sorting algorithms for three-dimensional LARPBS. In Chapter 7, we presents an implementation of Cole-Yap'[16] $O([\log \log n]^2)$-time parallel selection algorithm on LARPBS and apply this algorithm to solving the parallel multi-selection problem. We conclude in Chapter 8.

We remark that results in Chapter 4 has appeared in [27].

Chapter 2

Optical Interconnection Models

2.1 Optical Bus Systems

Bus-based architectures refer to as the class of networks where arrays of processors are enhanced by various bus systems. Different from other network topologies, bus-based networks have the advantage of having direct connection between any two processors in the system. Two disadvantages of a bus-based network are: *first*, its large diameter, which causes long communication delay; *second*, small bisection width. When there is a large amount of message transfer between nodes in a network, the bus becomes a communication bottleneck because it is shared by all processors in the network.

Due to recent advances in VLSI and fiber optic technologies these two disadvantages become less significant. The study of computing on bus-based architectures becomes a new and fast growing field. Various bus-based architectures have been proposed for efficient parallel computations, including arrays of processors with global buses[7], multiple buses [10], hyperbuses [25], multi-dimensional bus architectures [32], and reconfigurable buses [47], etc. Compared with theoretical parallel computing models, such as PRAM, bus-based architectures are practical and can be implemented with current bus technologies.

In a parallel computing system, communications among processors are always the most important issue. High communication bandwidth and less communication load is the goal for designing a high performance parallel system. As the silicon and Ga-As technologies advances, processor speed has already reached the gigahertz (GHz) range. Compared with such a high processor speed, traditional metal-based communication technologies used in parallel computing systems become a bottleneck. There are two ways to solve this problem. That is, either significantly improve the performance of traditional interconnect technologies, or introduce some new interconnect technologies, such as optics, into parallel computing systems. From the technological point of view, at least three fundamental constraints bound interconnections in electronic (or metal-based) systems: limited bandwidth, capacitive loading, and cross-talk caused by mutual inductance [54].

Optical communications provide an opportunity to redesign the traditional multiprocessor solutions free of the three limitations mentioned above. Optical interconnections offer

the following attractive features: high speed, high bandwidth, low error probability, gigabit transmission capacity, increased fanout, long interconnection lengths, low power requirement, and free from capacitive bus loading, cross talk, and electro-magnetic interference. Optical waveguide can be used to implement a bus. An optical bus can connect more processors than an electrical bus. There are three main areas of application for the optical bus communication technology in parallel distributed computing: optical networks, including both local area network and wide area network; optical buses used to connect components within a single system; and inter-chip interconnections. Advances in optical and optoelectronic technologies indicate that they could also be used as interconnections in parallel computers. Actually, many commercial massively parallel computers such as the Cray T3D have already used optical technology in their communication subsystems. Thus, optical interconnections have great potential for improving massively parallel processing systems, and will improve system performance and algorithm design.

Although optical technology alleviates some of the communication bottlenecks in parallel computing systems, there are some limitations to this solution. Two obvious limitations are: first, the transmission speed on the optical channels is bounded by the speed of electronics and of the optical/electrical and electrical/optical conversion devices. Any interface between electronics and optics lowers the speed at that interface to the speed of electronics. In other words, the speed of electronics puts bounds on the transmission speed of optical buses. Secondly, the end-to-end transmission time for optical signal is not inherently shorter than those of electrical signals, especially for short distance.

An optical bus system uses optical waveguides to transfer messages among electronic processors. In addition to the high transmission speed, optical signals transmitted on an optical waveguide have two important properties that are not shared with electrical signals running on an electrical bus. That is, unidirectional propagation and predictable propagation delays per unit length. These properties enable synchronized concurrent accesses of an optical bus in a pipelined fashion [14], [33]. Compared with exclusive access for data transmission on an electrical interconnection, the ability for an optical signal to concurrently access waveguide greatly reduce the time complexity of many parallel algorithms. Pipelined optical bus systems can support a massive volume of communications simultaneously, and are particularly appropriate for applications that involve intensive communication operations such as broadcasting, n-way one-to-one communication, multicasting, global aggregation, extraction and compression, and irregular communication patterns.

The optical bus architecture has been studied by many researchers [40, 64, 21, 63, 10, 22, 65, 61, 34, 78]. Figure 2.1 shows an SIMD linear array in which electronic processors are connected with a folded optical bus. Each processor is connected to the bus with two directional couplers, one for transmitting on the upper segment (the transmitting segment) and the other for receiving from the lower segment (the receiving segment) of the bus. Optical signals propagate unidirectional from right to left on the transmitting segment and from left to right on the receiving segment. Data are represented by optical pulses in the following way: An optical pulse p can represent a binary bit 1, and the absence of a pulse represents bit 0. Messages are organized as fixed-length frames, with each frame containing

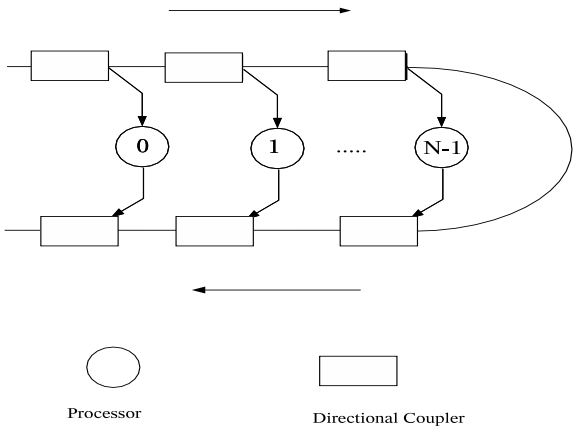


Figure 2.1: Structure of a folded optical bus

a maximum of b pulses. To be specific, let $b = 8$, the integer data 169 can be represented as a message frame $(p, -, p, -, p, -, -, p)$, where p stands for an optical pulse, and - stands for the absence of a pulse. In order to describe an optical bus system precisely, we define the most important hardware parameters of an optical bus system as follows:

- ω: the duration in seconds (temporal length) of a single optical pulse.
- c_b: the velocity (speed) of light running in waveguides.
- δ: delay (spatial length) of a single optical pulse, the relation between δ and ω is $\delta = \omega \times c_b$.
- τ: pipeline cycle (or petit cycle), i.e., the time for an optical pulse to traverse the distance in the waveguide between two consecutive processors.
- T: bus cycle, i.e., end-to-end propagation delay on the bus (the time for an optical pulse to propagate the entire bus). In the case of a folded linear optical bus, shown in Figure 2.1, the entire bus is $(P_0 \rightarrow P_1 \rightarrow ... \rightarrow P_{N-1} \rightarrow P_{N-1} \rightarrow P_{N-2} \rightarrow ... \rightarrow P_0)$. $T = (2N-1)\tau$, when there is no delay introduced for reference and select pulses; otherwise, $(2N-1)\tau \leq T \leq (2N-1)\tau + (N-1)\omega$.
- b: the length of a message frame. (We assume that a message frame is long enough to hold one basic data item such as an array element. So if processors want to send their indices, b should be at least $\lceil \log N \rceil$.)

Messages sent by different processors may overlap with each other even if they propagate in one direction on the bus. We call the overlapping of more than one message *transmission collision*. To avoid transmission collision, we must make sure that $\tau > b\omega$, so that each message frame can fit into a pipeline cycle τ. In one bus cycle T, up to N messages can be transmitted simultaneously in a pipeline fashion without collisions on the bus. And this is done by assuming that all processors are synchronized at the beginning of each bus cycle. It is a strictly non-blocking network in the sense that all processors can simultaneously send/receive messages in one bus cycle. So routing on a bus network is very simple.

2.2 Comparison of Several Optical Bus Models

In this section, we compare three similar one-dimensional optical bus models and several two-dimensional optical bus models, as they are related to the two optical models used or proposed in this dissertation: LARPBS[61] and 2D LARPBS.

As we mention in previous section, the unidirectional propagation and predictable delay of an optical bus enable synchronized concurrent access to an optical bus in a pipelined fashion. Researchers proposed several models that allow pipelining of data through an optical bus [21, 61, 34, 78, 45, 51, 57, 37]. Among them, the following three similar models are proved to be equivalent in [72]:

- LARPBS[61]–the linear arrays with a reconfigurable pipelined bus system, see Figure 2.4, Figure 2.5, Figure 2.6.
- POB[34]–the pipelined optical bus, see Figure 2.2.
- LPB[51]–the linear pipelined bus, see Figure 2.4.

The differences among the models lie in whether or not there are reconfigurable switches on the transmitting segment, the location of the conditional delay switches, and whether or not there are fixed delay loops on the receiving segment: The LPB is identical to the LARPBS with the exception that it does not have the reconfigurable switches. The POB is even simpler: it has neither the reconfigurable switches nor the conditional delay switches on the transmitting segment, but it has conditional delay switches instead of fix delay loop on the receiving segment.

The equivalence of the above three models is based on the following facts:

1. an address in a segmented bus can always be mapped to an address in the non-segment version of that bus, thus reconfigurable switches can be eliminated at the cost of extra computation on the destination address.
2. the fact that a processor always receives the first message and ignores all that follow plays an important role in simulating an operation for a segmented bus on a non-segmented bus.

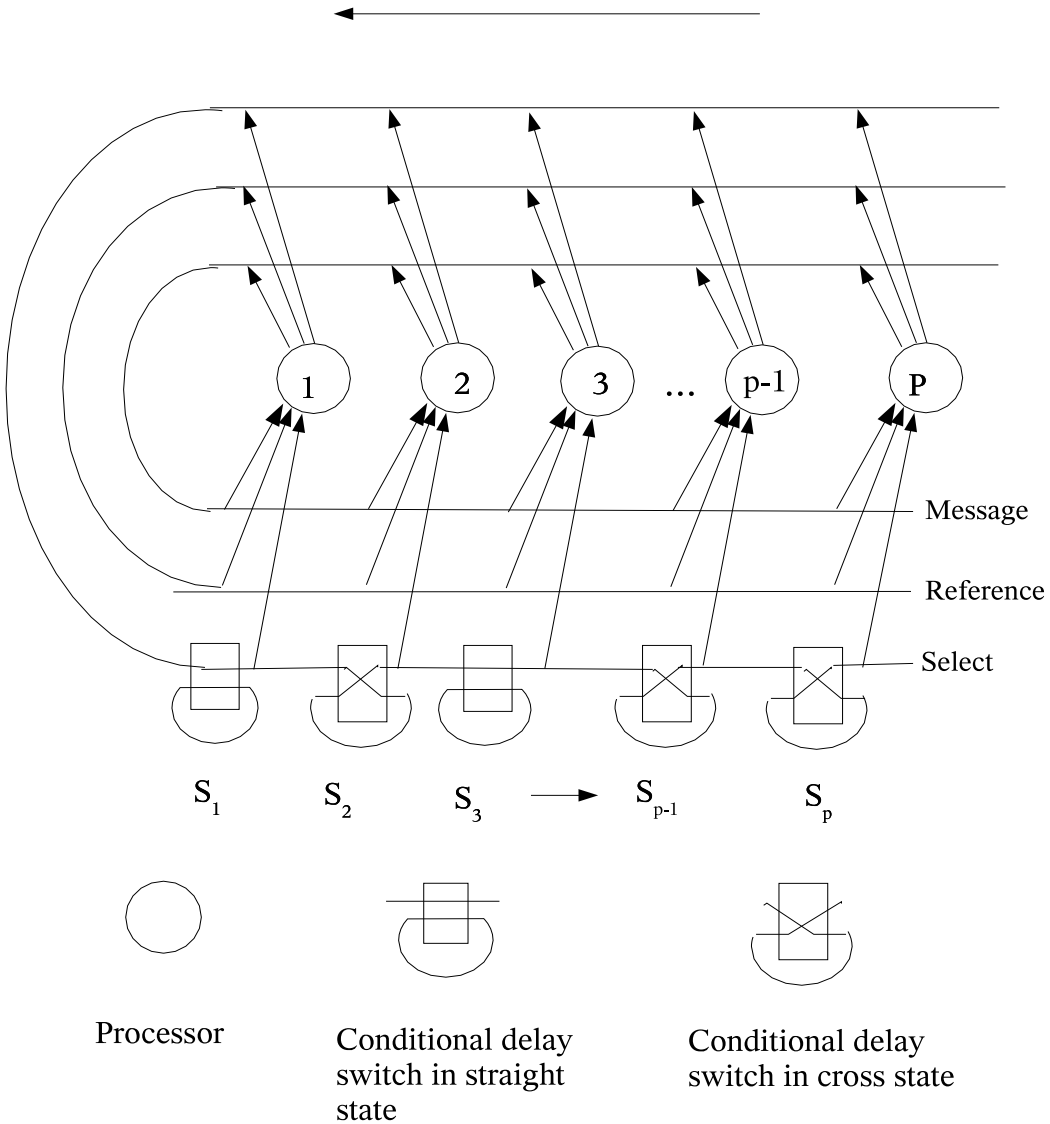


Figure 2.2: Structure of a POB

3. Basic operation *binary-prefix-sum* can be done without reconfigurable switches by using the multicasting ability instead of reconfiguration ability. This basic operation also plays an important role in the proof for equivalence.

The equivalence is proved by showing that each step in one model can be simulated by another model in constant time, where each step is defined to consist of following three phases:

1. determine parameters for the actual destinations of all messages
2. create the select frames
3. send the messages

Detailed proof can be found in [72]. In this dissertation, we are interested in LARPBS since it is the most convenient model for presenting parallel algorithms.

There are several factors that constrain the size of an optical bus system: the bus fan-out (split loss); power distribution problem; the length of unary addresses; and the increased latency when the number of processors is large. To achieve better scalability, researchers proposed the following two or higher dimensional optical array structures:

- PR-Mesh[73] – k-dimensional mesh of processors in which each processor has $2k$ ports, with each linear bus in the system being an LARPBS.
- 2-D ACPOB[63] – array connected by pipelined optical buses with conditional delays
- APPBS[21] – array processor with pipelined buses using switches
- AROB[54, 58] – array with reconfigurable optical buses
- ASOS[65] – array structure with synchronous optical switches
- SASOS[40] – symmetric ASOS
- ARSOB[64] – reconfigurable array with spanning optical buses

The common properties of the above models are:

- connect rows and columns of processors using generic optical buses
- place optical switches at the intersections of the row and column buses
- the switches are controlled in synchronization with the TDM to allow packets to be transmitted along multiple all-optical data paths without intermediate O/E and E/O conversions.

The structural differences for these models lie in the locations of the switches and the switching rules employed. Among the above five models, two-dimensional PR-Mesh, and ACPOB are the two dimensional extension of LARPBS and POB, respectively. The equivalence of PR-Mesh, APPBS, and AROB is established in [9]. The AROB model is actually an implementation of reconfigurable mesh [46] - a powerful parallel computation model that has been shown to be able to solve many problems very efficiently. All existing algorithms on a reconfigurable mesh can be ported to its corresponding AROB. But it has a major drawback: its structure, switching and operation mode are very complicated. Due to the following two facts, the ACPOB is the simplest among the above models: 1. ACPOB has only one type of switches – conditional delay switches, and one set of such switches for each dimension; 2. concurrently performing a row communication step and a column communication step is not assumed in this model. However, computationally, the ACPOB is very powerful. It is shown in [63] that ACPOB is more powerful than ASOS, SASOS, RASOB, and APPBS.

We proposed a two dimensional extension of LARPBS: we call it 2D LARPBS, see Figure 2.8. Each processor has a row reconfigurable switch pairs and a column reconfigurable switch pairs which can either be set to cross or straight. Like ACPOB, we do not assume a column communication and a row communication to be performed in the same bus cycle. The advantage of this simplification is that it simplifies the complexity of the model, and provides convenience for algorithm design. The details of a 2D LARPBS is given in section 2.4.

2.3 The LARPBS Model and Its Basic Operations

2.3.1 The Model

A Linear Array with Reconfigurable Pipelined Bus System (LARPBS) is a folded optical bus system with one set of fixed delays and two sets of flexible switches. Figure 2.4, Figure 2.5, and Figure 2.6 show the LARPBS model. To avoid confusion, we use two figures to represent the same LARPBS system. We show one set of switches in Figure 2.4 and Figure 2.5, and another set in Figure 2.6. We call the upper half of the bus transmitting segment, the lower half receiving segment. The optical bus contains three identical waveguides: *message waveguide* used to carry messages, and *reference waveguide* and *select waveguide* used together to carry address information, as shown in Figure 2.4. Messages are organized as fixed-length *message frames*. Optical signals propagate unidirectional either from left to right on the upper segment and from right to left on the lower segment of the bus, or vise versa, depending on how we fold the bus. We define a unit delay Δ to be the spatial length of a single optical pulse, that is $\Delta = \omega \times c_b$, where ω is the pulse duration in seconds and c_b is the velocity of light in the waveguides. Initially, processors are connected to those three waveguides such that the same length of fiber is used on all three waveguides between any two consecutive processors. A bus cycle for an optical bus is defined to be the end-to-end propagation delay on the bus. If we denote τ as the time taken for an optical signal to propagate between two consecutive processors, the bus cycle for the bus shown in Figure2.4 is $2N\tau$. We add

one unit delay Δ between any two processors on the receiving segments of the reference waveguides and the message waveguides. The one unit delay can be implemented by adding an extra segment of fiber and the amount of delay added can be accurately chosen based on the length of the segment. Then we add a conditional delay Δ between any two processors i and $i+1$, where $0 \leq i \leq N-2$, on the transmitting segments of the select waveguides. The switch between processor i and $i+1$ is local to processor $i+1$.

The conditional delays can be implemented by 2×2 optical switches such as the Ti:LiNbO3 switches used in an optical computer [6]. The conditional switch can be set to two different states: *cross* and *straight*. When the conditional switch is set to cross, one unit delay is added to an optical signal passing by; when it is set to straight, no delay is added to the optical signal passing by. The conditional delay switches are used to change the destination address at run time based on program needs. An address technology called *coincident pulse technique* [14, 33] is used to make it possible. This technique can be used to route information on an optical bus system in the case when the source processor knows its destination address. By making use of the two properties of optical signals, namely *unidirectional propagation* and *predictable propagation delays*, we can relate time with space and positional encode information.

Here is how it works: consider an LARPBS with n processors P_i , $i = 0, 1, ...n$. Suppose that processor P_i wants to send a message to processor P_j. Let t_{ref} denote the time when the reference pulse is ejected, and t_{sel} denote the time when a select pulse is ejected. Every time a reference pulse passes a delay loop, either a fixed delay loop or a delay loop introduced by a conditional delay switch, the reference pulse is postponed one petit time slot. By properly selecting the time difference between t_{ref} and t_{sel}, the two pulses can be made to meet at certain processor and be detected by the processor's detector. Figure 2.3(b) shows the initial address frame for sending a message to processor j, and Figure 2.3(c) shows the same address frame for the time when the message reaches its destination.

A set of flexible switches, shown in Figure 2.6, are reconfigurable switch pairs. We call such switches $RST(i)$ and $RSR(i)$, where $1 \leq i < N$. $RST(i)$ is a 1×2 optical switch on the waveguide section between P_i and P_{i+1} on the transmitting segment. $RSR(i)$ is a 2×1 optical switch on the waveguide section between P_i and P_{i+1} on the receiving segment. Both $RST(i)$ and $RSR(i)$ are controlled by processor P_i. These switch pairs are used to reconfigure the bus, either segment the bus into several independent sub-buses that can operate in parallel, or fuse two or more buses into one single bus. The reconfiguration is performed in the following way: When all RST and RSR switch pairs are set to *straight*, the bus system operates as a regular pipeline single bus system. When one pair, say $RST(i)$ and $RSR(i)$, is set to *cross*, the bus system is split into two independent sub-buses. One sub-bus consists of processor $P_0, P_1, \cdots, P_i$. The other sub-bus consists of processor $P_{i+1}, P_{i+2}, \cdots, P_{N-1}$. When $RST(i)$ is directly connected to $RSR(i)$, the total delay for a signal to pass the optical fiber between the two switches is made to be τ. Hence, in the above case, the sub-bus with processors P_0 to P_i can operate as a regular linear array with a pipelined bus system; so does the sub-bus with processors P_{i+1} to P_{N-1}. Figure 2.6 demonstrates an LARPBS model with $N = 6$ processors which is split into two segments. The first one has two processors,

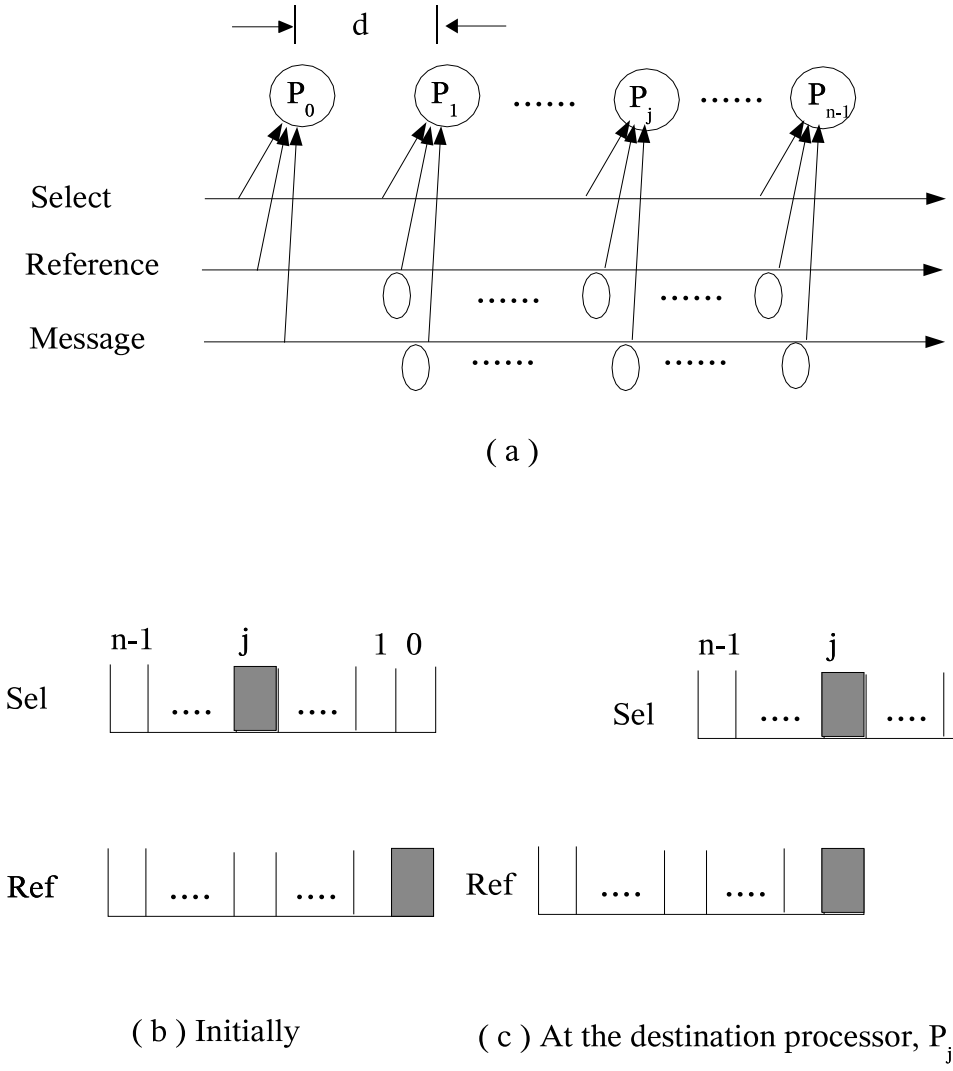


Figure 2.3: Address scheme for an LARPBS

i.e., processors P_0 and P_1; the second one has four processors, i.e., processors P_2 to P_5. In the figure, only one waveguide is shown. Another two sets of switches are omitted to avoid confusion. Notice that the time for setting up switches for one reconfiguration is a constant.

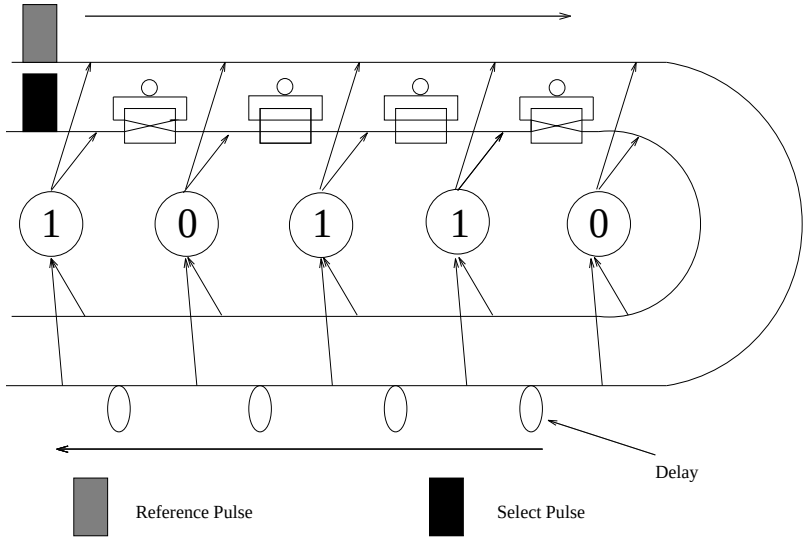


Figure 2.4: Structures of LARPBS and LPB

Figure 2.5: Conditional delay switches

Computation on an LARPBS usually consists of a sequence of alternate communication and computation steps. Communication time is measured by bus cycle. Thus, the time complexity of an algorithm implemented on LARPBS is measured in terms of total number of bus cycles and the time for parallel local computation.

The scalability of an algorithm can be defined as follows [74]: For a given algorithm and problem size, let $T_1(N)$ be the time to run the algorithm on N processors, where N is the number of processors required to run the algorithm as fast as possible. We then define $T_2(P)$ to be the time to run the algorithm adapted for P processors for the same problem size, where P is independent of N, except that $P < N$. Time $T_2(P) = T_1(N)\frac{N}{P}g(N,P)$, where $g(N,P)$ is the *scalability factor* of the algorithm. If $g(N,P)$ is a constant, then the algorithm is *completely scalable*. If $g(N,P)$ is a function of P and not of N, then the algorithm is relatively scalable.

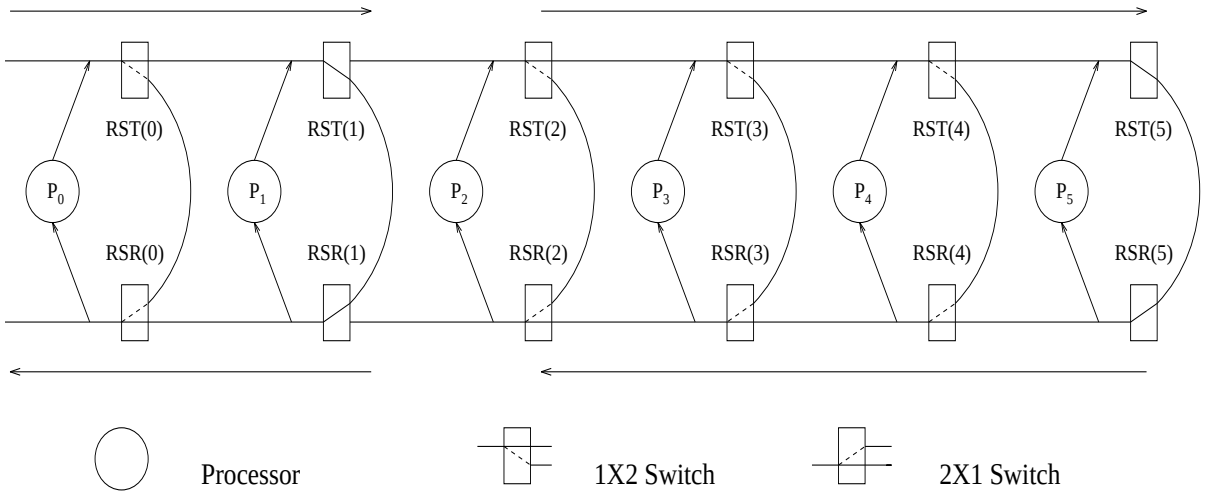


Figure 2.6: The LARPBS model of size 6 with two sub-buses

2.3.2 Power of the LARPBS Model

The LARPBS model mentioned in previous section is a powerful model for the following facts:

- An LARPBS can simulate a PRAM very efficiently. For deterministic methods, each step of a PRAM algorithm can be simulated by a p-processor LARPBS in $O(\log p + \log m)$ time [39], where m is the number of shared memory cells in the PRAM. For probabilistic methods, an $O(\log p)$ time simulation complexity can be achieved with high probability.

- If extended to higher dimension, the LARPBS model can be very versatile. Yueming Li, S. Q. Zheng, and Xiangyang Yang proved in [34] that we can embed parallel communication patterns supported by a wide variety of networks in parallel architectures based on segmented buses with small slowdown factors. Those networks include linear array, ring, complete binary tree, X-tree, mesh-of-trees, multi-dimensional mesh, torus, multi-grid and pyramid.

- For some of typical parallel algorithms, an LARPBS can achieve the same or even better time complexity than algorithms on the PRAM model. This is due to the fact that some computer tasks can be accomplished during the process of communication.

The parallel random access machine (PRAM) model is the most convenient, standard and popular abstract device for developing, and analyzing parallel algorithms. The PRAM is the synchronous version of the shared-memory model. Its power comes from the following facts [28]:

- There is a well-developed set of techniques and methods to handle many different classes of computational problems on the PRAM model.

- The PRAM model idealizes parallel computing. The use of shared memory accessible to all the processors makes interconnection communication relatively simple. Thus, one does not need to worry about algorithmic details concerning synchronization and communication, but concentrates on the structure properties of the problem and finds the best parallelism in a computation.

- The PRAM model captures several important parameters of parallel computations. A PRAM algorithm includes an explicit understanding of the operations to be performed at each time unit for traditional electrical systems, and explicit allocation of processors to jobs at each time unit.

- The PRAM gives robust design paradigms. Many practical parallel algorithms can be directly derived from PRAM algorithms.

However, PRAM is obviously far from practical. Accessing a shared memory location in constant time is impossible even in small scale shared memory systems. We cannot ignore memory access delays through a processor-memory interconnection network and the effect of memory access conflict/contention in shared memory modules. Nevertheless, many PRAM algorithms have been designed. It is clear that the time complexities of most of these algorithms are optimistic. They can be used as a measure for algorithms designed for various practical interconnection network models. Thus, the problem of implementing PRAM computations on more realistic parallel computers with minimum overheads has received considerable attention.

Keqin Li, Yi Pan, and Si-Qing Zheng presented several deterministic and probabilistic methods for simulating PRAM computation on a LARPBS model in [39]. They established the following results:

- Each step of a p-processor priority CRCW PRAM computation with $m = O(p)$ shared memory cells can be simulated by a p-processors LRAPBS in $O(\log p)$ time, where the constant in the big-O notation is small.

- Each step of a p-processor priority CRCW PRAM computation with $m = \Omega(p)$ shared memory cells can be simulated by a p-processors LARPBS in $O(\log m)$ time.

- Each step of a p-processor priority CRCW PRAM computation with m shared memory cells can be simulated on a p-processor LARPBS in $O(\log p)$ time with probability at least $1 - 1/p^c$ for all $c > 0$.

These results indicate that a number of important PRAM algorithms can be ported on the LARPBS model without loss much of speed. One good example is Cole's pipelined merge sort, which we will give a detailed implementation solution with the same time complexity on LARPBS in chapter 4. However, it is not the only way to design efficient algorithms on LARPBS by implementing or simulating the best known PRAM algorithm. Another approach is to design efficient algorithms by directly taking the advantages of the special

properties of optical interconnection. This second approach may provide unusual methods to perform some operations more efficiently than what PRAM can do. Here are two examples for which LARPBS algorithms achieve better time complexities than corresponding PRAM algorithms if we assume that a bus cycle takes $O(1)$ time: 1. n elements can be sorted in $O(1)$ time with n^2 processors in LARPBS, whereas $\Omega(\log n)$ is the lower bound for sorting on a PRAM. 2. *binary-prefix-sums* operation takes $\Omega(\log n/\log\log n)$ time on a priority CRCW PRAM with n processors [5], but it takes $O(1)$ time on an n-processor LARPBS. Based on the above results and observations, we develop our efficient parallel algorithms on LARPBS and higher dimensional LARPBS using both approaches in this dissertation.

2.3.3 Basic Operations on LARPBS

Assume there are n processors in the LARPBS: $P_0, P_1, \cdots, P_{n-1}$, and the LARPBS folded at processor P_{n-1}. All of the following operations take $O(1)$ bus cycle. No local computation is involved.

Definition 2.3.1 One-to-one[37]*: A group of processors $(P_{i_1}, P_{i_2}, \cdots, P_{i_m})$ each sends a message to one of the processors in $(P_{j_1}, P_{j_2}, \cdots, P_{j_m})$. No two processors in $(P_{i_1}, P_{i_2}, \cdots, P_{i_m})$ send a message to the same processor in $(P_{j_1}, P_{j_2}, \cdots, P_{j_m})$.*

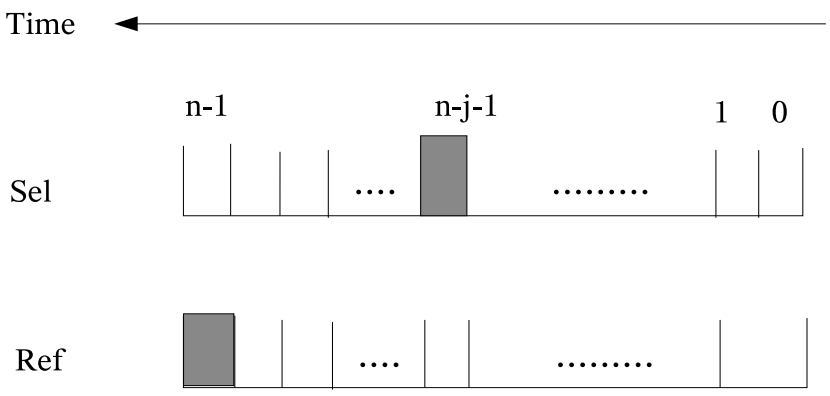
for $1 \le k \le m$ **pardo**
  $R(j_k) \leftarrow R(i_k)$
endfor

One-to-one communication operation can be implemented in one bus cycle as follows: each processor P_{i_k} in $(P_{i_1}, P_{i_2}, \cdots, P_{i_m})$ sends a reference pulse and a message frame containing $R(i_k)$ at time t_{ref}, and a select pulse at time $t_{sel}(j_k)$. See Figure 2.7(a) for the address frame. Whenever a processor P_{j_k} detects a coincidence of a reference pulse and a select pulse, it reads in the message.

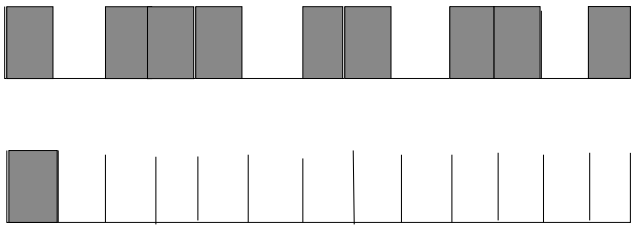
Note that a special case of the one-to-one communication operation is a permutation operation. In a permutation operation, each processor sends a message and a reference pulse at the beginning of a bus cycle. The time for a select pulse to be sent is determined by its corresponding destination processor's address. And every processor receives one and only one message at the receiving segment.

Definition 2.3.2 Broadcast[37]*: one processor P_i in an n-processor system sends a message $R(i)$ to all n processors $(P_1, P_2, \cdots, P_n)$ in the system, i.e. $R(0), R(1), R(2), ..., R(n-1) \leftarrow R(i)$.*

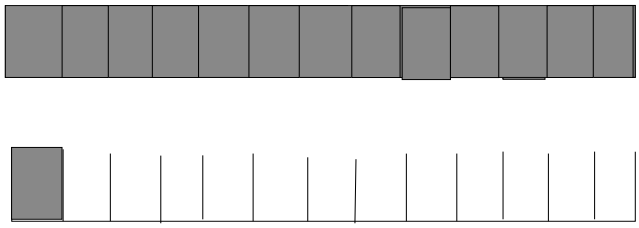
For this operation, all reconfigurable switch pairs are set to straight, and all conditional delay switches are set to straight to allow no delay to be introduced on transmitting segment of the select waveguide. The source processor P_i sends a reference pulse at the beginning of its address frame, and it sends consecutive select pulses in all the n pulse slots in its address frame on the select waveguide, as shown in Figure 2.7(c). Each processor in the system



(a) An address frame for one-to-one communication



(b) An address frame for multicast



(c) An address frames for broadcasting on a LARPBS

Figure 2.7: Address structure for basic operations

will detect a double-height pulse on the receiving segment of the bus and read the message. Clearly, one bus cycle is sufficient for broadcasting a message.

Definition 2.3.3 Multicast[62]*: one processor P_i in an n-processor system sends a message $R(i)$ to a subset $(P_{j_1}, P_{j_2}, \cdots, P_{j_m})$ of the n processors, i.e. $R(j_1), R(j_2), ..., R(j_m) \leftarrow R(i)$.*

Similar to *broadcast* operation, but processor i sends select pulses only at time $t_{sel}(j_1), t_{sel}(j_2), ..., t_{sel}(j_m)$, see Figure 2.7(b).

Definition 2.3.4 Binary-Prefix-Sums[37]*: Assume each processor P_i in the n-processor LARPBS holds a binary value $R(i)$. The problem of calculating the binary prefix sums is to calculate $S_i = R_0 + R_1 + \cdots + R_i$ for all $0 \leq i < n$.*

Step 1: Each processor P_i, where $0 \leq i < n$, sets its conditional delay switch in the following way: if $R_i = 1$, set it to cross; otherwise, set it to straight. Then P_i sends its address i to processor P_{n-1}. In other words, each processor sends a message frame, a reference pulse, and a select pulse at the beginning of the bus cycle. A processor in the system may receive multiple messages in this bus cycle, but each processor only reads the first message and ignores the rest. After this step, all processors set their conditional delay switches to straight.

Step 2: In the second bus cycle, each processor P_j, who receives address i in the first bus cycle, sends the value $s = (n - j - 1)$ to processor P_i. This is a *one-to-one* communication.

Step 3: Each processor P_i who receives a value s in the second bus cycle, sends s to processor P_{i+1}. Processor P_{i+1} then sets $S_{i+1} = s$, sets its reconfigurable switches to cross to segment the bus into $m+1$ (m is the number of 1s in the system) sub-buses, and broadcasts its S_{i+1}. After receiving the broadcasted value, each processor P_i ($1 \leq i < n$) sets its S_i to be the received value. And every processor in the system set its reconfigurable switch back to straight.

Step 4: processor P_0 checks its R_0, if $R(0) = 1$, sets $m = S_0 + 1$; otherwise, sets $m = S_0$. This is because processor P_0 does not have a conditional delay switch. Then processor P_0 broadcast m to every processor in the system. After receiving m, each processor P_i in the system modify its S_i value: $S_i = m - S_i$.

Definition 2.3.5 Ordered-Compression [62]*: Assume an array of n elements are classified into active and inactive. The ordered-compression operation is to compress all active elements to one end of the LARPBS preserving their original order.*

The ordered compression operation can be implemented with n processors in one bus cycle as follows:

Let's call each processor who holds an active element an active processor. Each processor P_i, where $0 \leq i < n$, sets its local conditional delay switch to cross if it is active; otherwise, set it to straight. Then each active processor P_i injects a reference pulse on the reference

waveguide and a select pulse on the select waveguide at the beginning of a bus cycle. In other words, every active processor tries to address processor $n-1$. A message frame containing the element it holds is also sent simultaneously with the address frames. The select pulse sent by the active processor whose index is the largest passes no cross-setting delay switch in the transmitting segment. Thus, select pulse and reference pulse sent by that processor will coincide at processor P_{n-1}, and the corresponding message is picked up by processor P_{n-1}. Similarly, the select pulse sent by the processor whose index is the second largest passes one cross-setting delay switch on the transmitting segment of the bus. Thus, the two pulses will meet at processor P_{n-2}, the corresponding message is picked up by processor P_{n-2}. In general, message sent by the active processor who is the k^{th} largest is received by processor P_{n-k}. At the end of this ordered compression, active elements are compressed to the higher index end of the bus, and the original order is preserved.

Definition 2.3.6 Split[61]*: Assume an array of n elements are classified into active and inactive. The split operation is to separate all active elements from all inactive elements.*

The split operation proposed in [61] can be performed in the following three steps:

Step 1: do an *ordered-compression* operation on active elements to move them to the higher index end of the bus. Again we call each processor who holds an active element an active processor. Then each active processor sends a dummy message to the processor on its right. The active processor P_{n-i} who does not receive a dummy in the bus cycle broadcast i to every processor in the system.

Step 2: do an *ordered-compression* operation on inactive elements to move them to the higher index end of the bus. But this time each inactive processor addresses its message to processor P_{n-1-i}. At the end of this operation, all inactive elements are moved to the left side of the bus, and all active elements are moved to the right side of the bus. And we complete the split operation.

2.4 2D LARPBS and Its Basic Operations

2.4.1 2D LARPBS – an extension to LARPBS

We propose a 2-D LARPBS model in this chapter. See Figure 2.8. A 2-D LARPBS has two sets of switches at each row and column intersection: conditional delay switches and reconfigurable switches. Figure 2.8 only draws processor connections to avoid confusion. Compared with the 2-D ACPOB, it has one extra pair of switches for the convenience of row bus reconfiguration and column bus reconfiguration. Actually, since we do not assume column communication and row communication be performed concurrently, each processor can use the same set of switches (including reconfigurable switch pairs and conditional delay switch) for its row bus and column bus if the switching between column bus and row bus can be done in a reasonable time. We classify a communication step into two types: a column communication step and a row communication step. The communication performance of an

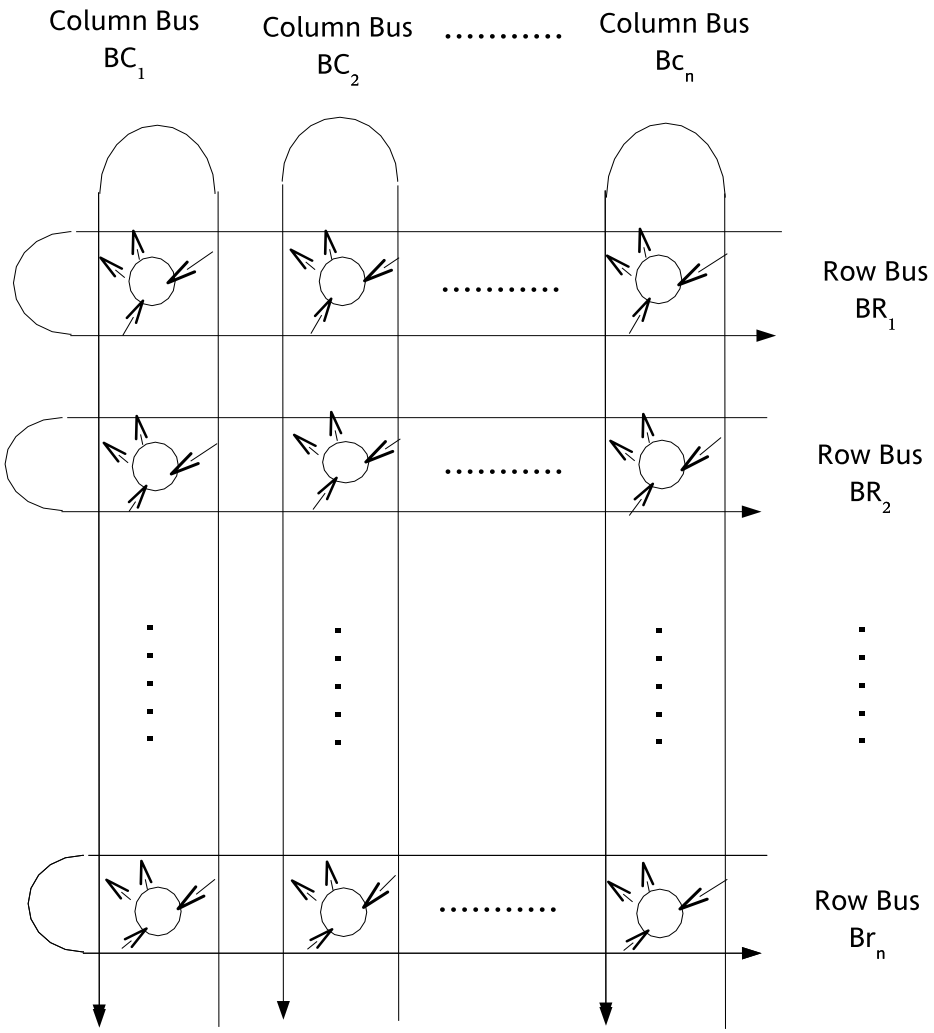


Figure 2.8: A 2D LARPBS

algorithm designed for this model is calculated by the total number of bus cycles for both row communication steps and column communication steps. In the following section, we follows the same idea in [63] to give some basic operations for the 2-D LARPBS model.

2.4.2 Basic Operations on 2D LARPBS

2-D LARPBS is more scalable than 1-D LARPBS, but communication in a 2-D LARPBS is more restrictive. One-to-one communication in LARPBS can be done in constant time, but not all permutation of the one-to-one communication can be done in 2-D LARPBS in constant time. In this section, we show the following special permutation operations can be done in $O(1)$ time: *Matrix_Transpose* and *Reverse_Matrix_Transpose*, *2D_Integer_Prefix_Sums*, *2D_Block_BPS*, *2D_BPS*, and *2D-compaction* operation. These basic permutation operations will be used as building blocks in the algorithms proposed in later chapters. Now we present the basic operations one by one.

Definition 2.4.1 Transpose Matrix*: Assume matrix A and B are both $n \times n$ matrices. Then matrix B is the transpose of matrix A, if $b_{ij} = a_{ji}$ for all $1 \leq i, j \leq n$.*

Definition 2.4.2 Matrix_Transpose*: For any matrix A, pick up the items in column-major order and set them down in row-major order.*

Definition 2.4.3 Reverse_Matrix_Transpose*: For any matrix A, pick up the items in row-major order and set them down in column-major order.*

It is easy to see, if a matrix has equal number of rows and columns, then its *Matrix_transpose* operation and *Reverse_Matrix_Transpose* operation are equivalent. Thus we will only give a description on the *Matrix_transpose* operation. We follow the same idea for the matrix transpose algorithm on ACPOB given in [63] to obtain the following algorithm on 2D LARPBS.

Algorithm 2.4.1 Matrix_Transpose
Input*: a $r \times r$ matrix A.*
Output*: matrix B, the transpose of matrix A*
Begin

1. *Eliminate collision by row communication*: processor $P_{i,j}$ sends its item $x_{i,j}$ to processor $P_{i,((j+i-1) \bmod r)}$. In other words, we imagine each row in the matrix to be a circular queue. Processor $P_{i,j}$ moves its item $i-1$ positions to the right, where $1 \leq i \leq n$. The purpose of this row shift communication is to send items on the same column in the matrix to different columns. To be precise, no two items $a_{i,j}$ and $a_{k,j}$, where $i \neq k$, are in the same column after this row communication. Thus, there will be no collision in the communication performed in the next step. Each processor will receive one and only one items in the next step.

2. *Send items to correct rows*: a multiple *one-to-one* column communication operation is performed so that all items, whose destination processors are in row i, are sent to that row. To be precise, if processor $P_{i,j}$ has item $e_{i,k}$, then $P_{i,j}$ sends $e_{i,k}$ to processor $P_{k,j}$.

3. *Send items to the correct columns*: a multiple *one-to-one* row communication operation is performed so that all items, whose destination processors are in column j, are sent to that column. To be precise, if processor $P_{i,j}$ has item $e_{h,i}$, then $P_{i,j}$ sends $e_{h,i}$ to processor $P_{i,h}$. Now all items $e_{h,i}$ are obtained by their destination processor $P_{i,h}$, where $1 \leq h, i \leq n$.

End

Example 2.4.1 Matrix_Transpose *on 2D LARPBS.*

$$\begin{bmatrix} 1,1 & 1,2 & 1,3 & 1,4 & 1,5 \\ 2,1 & 2,2 & 2,3 & 2,4 & 2,5 \\ 3,1 & 3,2 & 3,3 & 3,4 & 3,5 \\ 4,1 & 4,2 & 4,3 & 4,4 & 4,5 \\ 5,1 & 5,2 & 5,3 & 5,4 & 5,5 \end{bmatrix} \xrightarrow{remove\ collision} \begin{bmatrix} 1,1 & 1,2 & 1,3 & 1,4 & 1,5 \\ 2,5 & 2,1 & 2,2 & 2,3 & 2,4 \\ 3,4 & 3,5 & 3,1 & 3,2 & 3,3 \\ 4,3 & 4,4 & 4,5 & 4,1 & 4,2 \\ 5,2 & 5,3 & 5,4 & 5,5 & 5,1 \end{bmatrix}$$

Input *Step* 1

$$\begin{bmatrix} 1,1 & 2,1 & 3,1 & 4,1 & 5,1 \\ 5,2 & 1,2 & 2,2 & 3,2 & 4,2 \\ 4,3 & 5,3 & 1,3 & 2,3 & 3,3 \\ 3,4 & 4,4 & 5,4 & 1,4 & 2,4 \\ 2,5 & 3,5 & 4,5 & 5,5 & 1,5 \end{bmatrix} \xrightarrow{to\ correct\ column} \begin{bmatrix} 1,1 & 2,1 & 3,1 & 4,1 & 5,1 \\ 1,2 & 2,2 & 3,2 & 4,2 & 5,2 \\ 1,3 & 2,3 & 3,3 & 4,3 & 5,3 \\ 1,4 & 2,4 & 3,4 & 4,4 & 5,4 \\ 1,5 & 2,5 & 3,5 & 4,5 & 5,5 \end{bmatrix}$$

Step 2 *Step* 3

The basic idea of the 2D Binary Prefix Sums algorithm proposed in [63] is: divide the $n \times n$ matrix A into $\sqrt{n}$ sub-matrices, each has size $\sqrt{n} \times n$; compute the binary prefix sums for each sub-matrix using block binary prefix sums algorithm; then apply an integer binary prefix sums algorithm to get the prefix sums for sub-matrix sums; finally the binary prefix sums for each element can be obtained by adding its prefix sums within its sub-matrix with prefix sums for sub-matrix sums of the proceeding sub-matrix. We give the binary prefix sums algorithm by first describing the two basic algorithms: *Integer_Prefix_Sums* and *Block_BPS*, which will be used in the *2D_Binary_Prefix_Sums* for calculating binary prefix sums of the sub matrices and calculating block prefix sums. Then we give a description for the *2D_Binary_Prefix_Sums* algorithm. The 2D binary prefix sums algorithm given below uses $\sqrt{n} \times n$ processors to computer the prefix sums of $\sqrt{n}$ integers. Each integer falls into the range $[0, n^2 - 1]$. Thus the $n^{\frac{1}{2}}$-ary representation of each integer requires at most $d = 4$ digits. Assume, this $\sqrt{n}$ are first distributed to the last column of the $\sqrt{n} \times n$ 2D LARPBS. Here is the detail of the algorithm.

Table 2.1: Step 1: distributing the integers

7														7
	12													12
		16												16
			9											9

Algorithm 2.4.2 2D_Integer_Prefix_Sums
Input: *$\sqrt{n}$ integer distributed to the last column of the $\sqrt{n} \times n$ matrix A, one integer per processor. And let $d = 4$.*
Output: *integer prefix sums of the $\sqrt{n}$ integer items*
Begin

1. *Send integer to diagonal processors* : each processor $P_{i,n}$, where $1 \leq i \leq \sqrt{n}$, sends its integer I_i to processor $P_{i,i}$. This is a *one-to-one* row communication operation. Refer to Table 2.1.

2. *Calculate $n^{\frac{1}{2}}$-ary representation* : each processor $P_{i,i}$, where $1 \leq i \leq \sqrt{n}$, broadcasts I_i to every processor in column i. Refer to Table 2.2. Then, processor $P_{i,j}$, where $1 \leq i \leq d$ and $1 \leq j \leq \sqrt{n}$, locally computes the $n^{\frac{1}{2}}$-ary representation of integer I_i, and saves the j^{th} digit as its local value $b_{i,j}$. Refer to Table 2.3.

3. *Spread out each $n^{\frac{1}{2}}$-ary representation digit with a sequence of 1s* : each processor $P_{i,j}$, where $1 \leq i \leq d$ and $1 \leq j \leq \sqrt{n}$, use its $b_{i,j}$ as count for number of 1s and spread 1s to the right in the following way: if $b_{i,j} \neq 0$, processor $P_{i,j}$ multicasts 1 to processors $P_{i,(j-1)n^{\frac{1}{2}}+1}, \cdots, P_{i,(j-1)n^{\frac{1}{2}}+b_{i,j}}$. Refer to Table 2.4.

4. *Apply 1D binary prefix sums* : Apply 1D *binary-prefix-sums* operation to the first d rows to compute row binary prefix sums. Refer to Table 2.5.

5. *Collect and convert results to decimal* : processor $P_{i,k\sqrt{n}}$, where $2 \leq i \leq d$ and $1 \leq k \leq \sqrt{n}$, sends its binary prefix sums $s_{i,k\sqrt{n}}$ to processor $P_{1,k\sqrt{n}}$ one by one. As d is a constant, these multiple *one-to-one* column communication operations take constant time. Each processor $P_{1,k\sqrt{n}}$ then convert the $n^{\frac{1}{2}}$-ary representation to decimal value. Refer to Table 2.6.

6. *Move output to its corresponding input location* : processor $P_{1,k\sqrt{n}}$, where $1 \leq k \leq \sqrt{n}$, sends the decimal prefix sums value to processor $P_{k,n}$ using a *one-to-one* column communication operation followed by a *one-to-one* row communication operation. Now, the output for the integer prefix sums of an input item can be found at the same location where it is distributed at the beginning of the algorithm. Refer to Table 2.7.

End

Based on the above special case integer prefix sums algorithm, we can obtain the following block binary prefix sums algorithm.

Table 2.2: Step 2: calculating $n^{\frac{n}{2}}$-ary representation

7=013	12	16	9												
7	12=030	16	9												
7	12	16=100	9												
7	12	16	9=021												

Table 2.3: Step 2: saving corresponding $n^{\frac{n}{2}}$-ary representation digit

3	0	0	1											
1	3	0	2											
0	0	1	0											

Table 2.4: Step 3: multicasting 1s

1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	1	1	1	0	0	0	0	0	1	2	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Table 2.5: Step 4: performing row BPS

1	2	3	**3**	3	3	3	**3**	3	3	3	**3**	4	4	4	**4**
1	1	1	**1**	2	3	4	**4**	4	4	4	**4**	5	7	7	**7**
0	0	0	**0**	0	0	0	**0**	1	1	1	**1**	1	1	1	**1**

Table 2.6: Step 5: collecting and converting results to decimal values on the first row

			7=013				19=043				35=143				48=174

Table 2.7: Step 6: returning result to item's original location

														7
														19
														35
														48

Algorithm 2.4.3 2D_Block_BPS
Input: *$n \times \sqrt{n}$ Boolean values distributed in an $n \times \sqrt{n}$ 2D LARPBS A, one value per processor.*
Output: *the binary prefix sums for every item in A.*
Begin

1. *Compute binary prefix sums on each row* : do a *binary-prefix-sums* operation on each row using the basic 1D algorithm.

2. *Compute integer prefix sums for each row sum* : the binary prefix sums $s_{i,n}$, where $1 \leq i \leq \sqrt{n}$ obtained in the previous step for the last processor in each row is in the range $[0, n]$. Thus we apply algorithm 2.4.2 *2D_Integer_Prefix_Sums* to the $\sqrt{n}$ $s_{i,n}$ to obtain the row prefix sums $S_i = \sum_{j=1}^{i} s_{i,n}$.

3. *Pass row prefix sums to next row* : the last processor in each row $P_{i,n}$, where $1 \leq i < n$, sends its S_i to processor $P_{(i+1),n}$. This is a *one-to-one* column communication operation.

4. *Broadcast row prefix sums within each row* : the last processor in each row $P_{i,n}$, where $1 < i \leq n$, broadcasts the $S_{(i-1)}$ received in previous step to every processor in row i.

5. *Calculate final binary prefix sums* : every processor $P_{i,j}$, where $1 < i \leq n$ and $1 \leq j \leq n$ calculates its final prefix sums by adding the row prefix sums of the previous row with the prefix sums calculated in step 1: $s_i = S_{(i-1)} + s_i$.

End

Now we are ready for our constant time binary prefix sums algorithm on 2D LARPBS.

Algorithm 2.4.4 2D_BPS
Input: *$n \times n$ Boolean A, one element per processor.*
Output: *the binary prefix sums for every item in A.*
Begin

1. *Partition and compute Block BPS* : partition A into $\sqrt{n}$ sub-matrix A_1, A_2, $\cdots$,$A_{\sqrt{n}}$. Each of them is a $\sqrt{n} \times n$ matrix. Apply algorithm 2.4.3 *2D_Block_BPS* simultaneously on the $\sqrt{n}$ sub-matrix A_i.

2. *Compress Block BPS's to the last processor on the first $\sqrt{n}$ rows* : the last processor in each sub-matrix, processor $P_{k\sqrt{n},n}$, where $1 \leq k \leq \sqrt{n}$, sends its prefix sums $s_{k\sqrt{n},n}$ obtained in the previous step to processor $P_{k,n}$. This *one-to-one* column communication operation serves the purpose of compressing the $\sqrt{n}$ sub-matrix prefix sums to the last processor in the first $\sqrt{n}$ rows so that their prefix sums can be calculated using the first sub-matrix.

3. *Compute sub matrix prefix sums* : As $s_{k\sqrt{n},n} \leq n\sqrt{n} < n^2 - 1$, we can apply algorithm 2.4.2 *2D_Integer_Prefix_Sums* to the first sub-matrix to compute prefix sums for $s_{k\sqrt{n},n}$, where $1 \leq k \leq \sqrt{n}$. The results are stored in the last processors on the first $\sqrt{n}$ rows.

4. *Multicast sub matrix prefix sums* : processor $P_{i,n}$, where $1 \leq i < \sqrt{n}$, multicasts the prefix sums S_i obtained in the previous step to processors $P_{i\sqrt{n}+k,n}$, where $1 \leq k \leq \sqrt{n}$. This is a column *multiple multicast* communication operation. After these multicast operations, each processor $P_{j,n}$, where $1 \leq j \leq n$, broadcasts the received prefix sum $S_{\lfloor j/\sqrt{n} \rfloor}$ to every processor in row j.

5. *Calculate final binary prefix sums* : the final binary prefix sums for each processor $P_{i,j}$, where $1 \leq i, j \leq n$, can be obtained by adding its local prefix sums obtained in step 1 and the one obtained in step 4 : $s_{i,j} = s_{i,j} + S_{\lfloor i/\sqrt{n} \rfloor}$.

End

It is clear that each step in algorithm 2.4.4 takes constant time. Therefore, algorithm 2.4.4 runs in $O(1)$ time.

The following is a description for 2D compaction operation:

Algorithm 2.4.5 2D_Compaction
Input: *an $n \times n$ Boolean matrix A, where elements are distributed to n^2 processors such that each processor $P_{i,j}$ holds a Boolean value $a_{i,j}$. Each $P_{i,j}$ also holds a data item $b_{i,j}$. There are a total of x processors with $a_{i,j} = 1$.*
Output: *all items $b_{i,j}$ with $a_{i,j} = 1$ are moved to the first x processors in A in row major or column major order.*
Begin

1. *Compute binary prefix sums for each item*: do a *2D-Binary-Prefix-Sum* operation to get the binary prefix sums $s_{i,j}$ for each item $b_{i,j}$ in matrix A.

2. *Compute destination address*: each processor $P_{i,j}$ who has $a_{i,j} = 1$, compute its destination address (i', j'). For row major order, then $i' = \lfloor s_{i,j}/n \rfloor$, $j' = s_{i,j} \bmod n$; if output is in column major order, then $j' = \lfloor s_{i,j}/n \rfloor$, $i' = s_{i,j} \bmod n$.

3. *Send items to correct row or column*: each processor $P_{i,j}$ who has $a_{i,j} = 1$ does the following. For column major order, $P_{i,j}$ sends its item $b_{i,j}$ and new column address j' to processor $P_{i',j}$. As two processors on one column cannot be more than n positions apart for column major compaction, this is a *one-to-one* row communication operation. For row major order, $P_{i,j}$ sends its item $b_{i,j}$ and new row address i' to processor $P_{i,j'}$. As two processors on one row cannot be more than n positions apart for row major compaction, this is a *one-to-one* column communication operation.

4. *Send items to their final destinations*: For column major order, each processor $P_{i,j'}$ who has received item $b_{i,j}$ and row address i' in the previous step sends $b_{i,j}$ to processor $P_{i',j'}$. This is a *one-to-one* row communication operation. After this step, all items whose $a_{i,j} = 1$ are compressed to the first x processors in column major order. For row major order, each processor $P_{i',j}$ who has received item $b_{i,j}$ and column address j' in previous step sends $b_{i,j}$ to processor $P_{i',j'}$. This is a *one-to-one* column communication

operation. After this step, all items whose $a_{i,j} = 1$ are compressed to the first x processors in row major order.

End

Chapter 3

Boolean Matrix Multiplication and Its Applications

3.1 Introduction

General matrix multiplication problem on an LARPBS has been studied extensively by Keqin Li, Yi Pan, and Si Qing Zheng in [35, 37, 38]. In this chapter, we present a constant-time Boolean matrix multiplication algorithm on the LARPBS model. Our algorithm achieves better speed-up compared with the four Russians' algorithm proposed by Agerwala and Lint in [4] and a hypercube implementation provided by C. Gregory Plaxton [55]. A parallel implementation of the Four Russians' algorithm on the LARPBS with constant running time is proposed by Keqin Li in [31]. Compared with Li's constant-time algorithm, our algorithm is not based on any existing sequential or parallel algorithm and it is much simpler and uses fewer processors. our algorithm uses only $O(n^2)$ processors, while Li's algorithm uses $O(n^3/logn)$ processors. Refer to Table 3.1 for a comparison on different parallel Boolean matrix multiplication algorithms.

A closely related problem is the Boolean matrix closure (BMC) problem, i.e., the problem of calculating transitive closure for a directed graph. By using our constant time BMM algorithm, we can compute the transitive closure of a directed graph in $O(\log n)$ time on an LARPBS with $O(n^2)$ processors. If a polynomial number of processors are employed, it

Table 3.1: Comparison of different parallel Boolean matrix multiplication algorithms.

Algorithm	Time Complexity	Processor Complexity
Our new LARPBS Algorithm	$O(1)$	$O(n^2)$
Li's Parallel Four Russians' Algorithm [31]	$O(1)$	$O(n^3/logn)$
Plaxton's Hypercube Algorithm [55]	$O(\log n)$	$O(n^3/\log n^2 \log\log n)$
Agerwala and Lint's Algorithm [4]	$O(\log n)$	$O(n^3/(\log n \log\log n))$

is shown in [15] that the lower bound for parallel time complexity of transitive closure is $\Omega(\log n/\log\log n)$ on a CRCW PRAM.

3.2 Definition of the Problem

Given two Boolean matrices A and B. Let C denotes the product of matrices A and B, the entries of C are given by the following well-known equation:

$$c_{i,j} = \bigvee_{k=1}^{n} a_{i,k} \wedge b_{k,j} \tag{3.1}$$

where $\bigvee$ represents logical-or, and $\wedge$ represents logical-and.

This equation leads to a simple and well-known $O(\log n)$ time matrix multiplication algorithm running on a hypercube using n^3 processors [17].

3.3 A New Constant Time Boolean Matrix Multiplication Algorithm

In this section, we introduce a new constant time Boolean matrix multiplication algorithm for the LAPBRS model. Compared with Li's parallel implementation of four Russians' algorithm on the LARPBS, our algorithm requires much less processors and is much simpler. Instead of implementing an existing algorithm on a new model, we provide a totally new idea for solving the problem on LARPBS.

The algorithm we present here uses $O(n^2)$ processors on an LARPBS. We use a different strategy when computing matrix C: Instead of trying to send $c_{i,j}$ all the needed elements in A and B, we think the other way around: for each element $b_{i,j}$ in B, we try to find out which processor in the system should know its value in order to get the final results. Throughout this chapter, we use $P_{i,j}$ to denote the $(in+j)^{th}$ processor on a one-dimensional LARPBS. This helps simplify our algorithm description.

Assume that each processor $P_{i,j}$ has the following registers:

- R_a: a one bit register used to store matrix element $a_{i,j}$.
- R_B: an n bit register, used to store elements in row j of matrix B, $B(j) = (b_{j,1}, b_{j,2}, \cdots, b_{j,n})$.
- R_c: a one bit register, used to store matrix element $c_{i,j}$. It is initialized to 0.

At the beginning of the algorithm, elements in matrix A and B are distributed to processors in LARPBS in the following way: A is distributed to the n^2 processors one element per processor in row major order, i.e., $a_{i,j}$ is stored in processor $P_{i,j}$. B is stored in the first n processors one row per processor, i.e., $B(j) = (b_{j,1}, b_{j,2}, \cdots\cdots, b_{j,n})$ is stored in processor $P_{1,j}$, where $1 \leq j \leq n$. Results of the algorithm will be stored in the R_c register in each processor.

Algorithm 3.3.1 Boolean_Matrix_Multiplication
Input: *Two $n \times n$ boolean matrices A and B.*
Output: *boolean matrix $C = A \times B$.*
Begin

1. *Multicast* $B(j)$: each processor $P_{1,j}$, where $1 \leq j \leq n$, multicasts its $B(j)$ in R_B to processors $P_{k,j}$, where $1 < k \leq n$.

2. Each processor $P_{i,j}$ checks its R_a, and sets itself to active if $R_a = 1$.

3. *Form sub-buses*: processors $P_{i,n}$, where $1 \leq i \leq n$, set their reconfigurable switches to cross to form n sub-buses. Each sub-bus has n processors.

4. *Calculate* $c_{i,j}$: each active processor $P_{i,j}$ does the following multicast operation: uses the $B(j)$ in its R_B to set its select frame, and then multicasts a dummy message within its sub-bus. For example, if $B(j) = (1, 0, 0, 1, 0, 1)$, then the select frame is set to $(p, -, -, p, -, p)$. If processor $P_{i,j}$ receives a message in this *multiple multicast* operation, then $P_{i,j}$ sets its $R_c = 1$; otherwise it sets its $R_c = 0$. After this local operation, we obtain matrix C: processor $P_{i,j}$ holds matrix element $c_{i,j}$ in register R_c.

End

After step 1, each processor $P_{i,j}$ has bit array $B(j) = (b_{j,1}, b_{j,2}, \cdots, b_{j,n})$. So all processors have their R_B ready. This step consists of only one basic operation: *multiple multicasts* operation. Step 2 performs the following *logical-and* operation: $a_{i,j} \wedge b_{j,k}$ for $1 \leq k \leq n$. Step 3 involves no communication and computation. This step is used to reconfigure the system into n sub-buses, with each of them responsible for calculating one row in matrix C. Step 4 is the most important step that bears the main idea for this algorithm. The beauty of this step is that communication itself also accomplishes special computations. Here is how it works: The calculation of C is based on Equation 3.1. Each element $a_{i,j}$ in A will participate in the following *logical-and* calculation with all elements on the j^{th} row in matrix B: $a_{i,j} \wedge b_{j,1}$, $a_{i,j} \wedge b_{j,2}$, $\cdots$, $a_{i,j} \wedge b_{j,n}$. The results of these calculations need to be sent to processors $P_{i,1}$, $P_{i,2}$, $\cdots$, $P_{i,n}$ for computing $c_{i,1}$, $c_{i,2}$, $\cdots$, $c_{i,n}$. Here we compute the *logical-or* $\bigvee$ and send the results to their correct processors in one *multiple multicast* communication operation. Notice that if $a_{i,j} = 0$, all of the above *logical-or* operations will result in 0. As we have already initialized R_c to be 0, the *logical-or* operations are done by only allowing active processors to send dummy messages. Now we show how to send the logical-or operations results for processors whose $a_{i,j}$ equal to 1. If $a_{i,j} = 1$, then we need to let processor $P_{i,k}$ know that there is a 1 result from $a_{i,j} \wedge b_{j,k}$ only if $b_{j,k}$ is non-zero. Thus, if we use the vector $B(j)$ stored in R_B to set the select frame for the dummy message, then this message will be addressed to processor $P_{i,k}$ only if $b_{j,k}$ is non-zero. This is how $a_{i,j} \wedge b_{j,k}$ is done. Each processor in each sub-bus can receive more than one message in Step 4. For example, assume A and B are two 4×4 matrices. Then processor $P_{2,1}$ will receive a dummy message from every processor in the second sub-bus if the following matrix elements have value 1: $a_{2,1}$, $b_{1,1}$, $a_{2,2}$, $b_{2,1}$, $a_{2,3}$, $b_{3,1}$, $a_{2,4}$, and $b_{4,1}$. As each processor only cares about whether a dummy

message is sent to it or not, it reads the first message addressed to it and ignores the rest. This is how the logical-or operations of Equation 3.1 are done. Notice this *multiple multicast* operation performs all the *logical-or* calculation needed for all elements in C. The reason why we can use the binary vector $B(i)$ to set the select frame is: if the LARPBS system has n^2 processors, then the system should be able to address the n^2 processors. According to the address mechanism of the LARPBS system, a select frame should consists of at least n^2 pulses for it to address n^2 processors. For the case of $n \times n$ 2D LARPBS, the address should at least be n-pulse long. In that case, it is still long enough to hold vector $B(i)$.

It is clear that this algorithm only consists of two *multiple multicast* operations and a couple of local Boolean value comparisons. Thus, the time complexity of this algorithm is $O(1)$. We obtain the following theorem.

Theorem 1 *Given two $n \times n$ Boolean matrices A and B. Let C denote the product of matrices A and B. Matrix C can be calculated in $O(1)$ bus cycles, and in $O(1)$ computation time, using $O(n^2)$ processors on an LARPBS.*

Refer to Figure 3.1 for an example.

3.4 Extension to 2D LARPBS

Our algorithm can be easily extended to a 2D LARPBS. Here is the 2D LARPBS algorithm.

Algorithm 3.4.1 2D_Boolean_Matrix_Multiplication
Input: *Two $n \times n$ boolean matrices A and B.*
Output: *boolean matrix $C = A \times B$.*
Begin

1. *Multicast $B(j)$*: each processor $P_{1,j}$, where $1 \leq j \leq n$, multicasts its $B(j)$ in R_B to processors $P_{k,j}$ using its column bus, where $1 < k \leq n$.

2. Each processor $P_{i,j}$ checks its R_a, and sets itself to be active if $R_a = 1$.

3. *Calculate $c_{i,j}$*: active processors $P_{i,j}$ does the following multicast operation: use the $B(j)$ to set select frames, multicast a dummy message on their row buses. If processor $P_{i,j}$ receives a message in this *multiple multicast* operation, then it sets its $R_c = 1$; otherwise it sets its $R_c = 0$. After this local operation, we obtain our final result C: processor $P_{i,j}$ holds matrix element $c_{i,j}$ in register R_c.

End

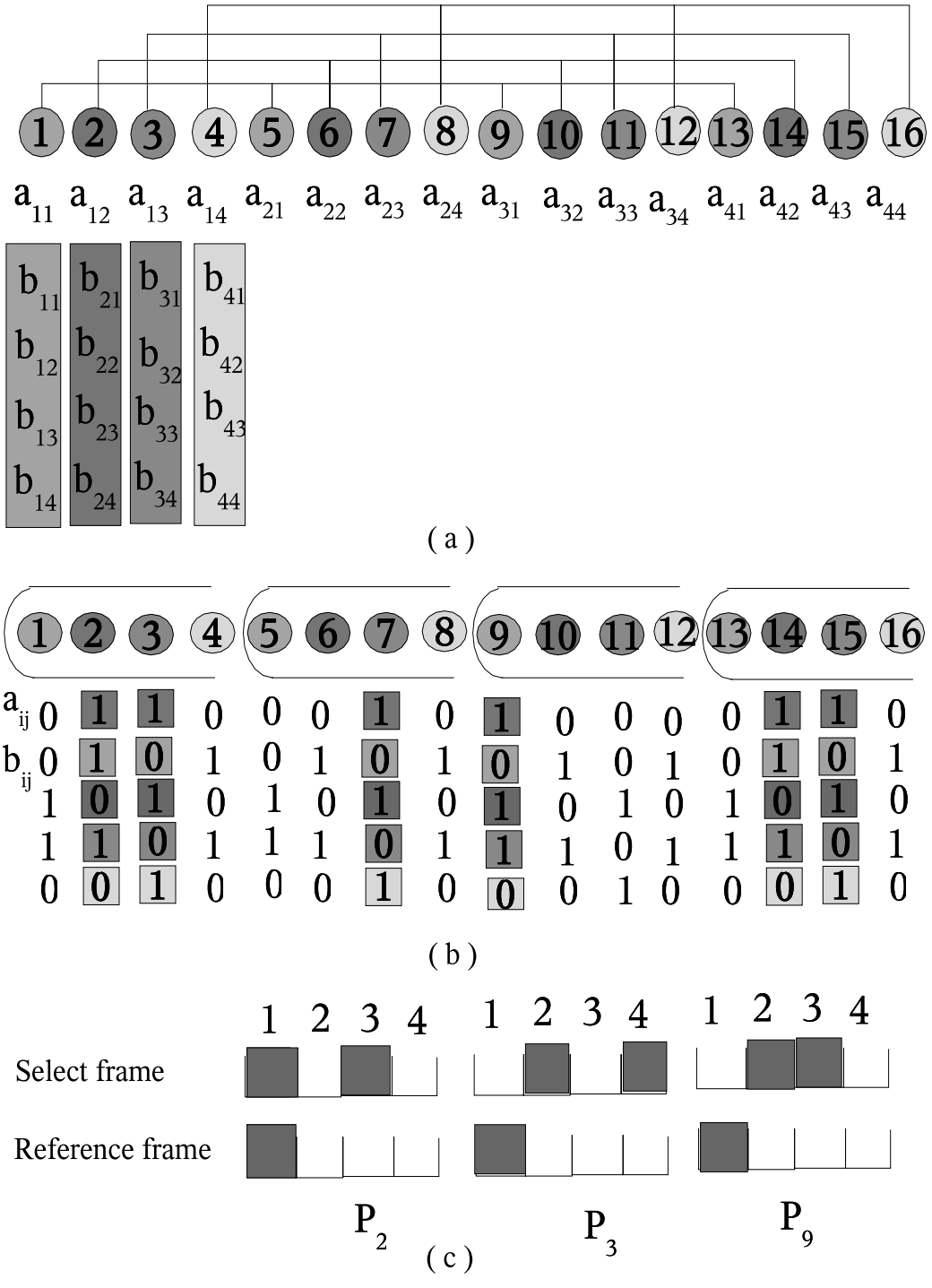


Figure 3.1: Structure of a folded optical bus: (a) step 1 (b) step 2 and step 3 (c) address frames for some of active processors

3.5 Discussion on Scalability

Suppose the number of available processors is p. If $n \leq p < n^2$, we can modify algorithm 3.3.1 in the following way to achieve a good scalability:

We still want to split the p processors into n sub-buses in Step 3. Each group is responsible for one row of matrix A, B, and C. Thus there are $m = p/n$ processors in each sub-bus. Every processor has three arrays: A, B, and C, each of which has size n^2/p. Now, let's reconsider the computation and communication time. Each of Step 1 and Step 4 takes $\frac{n^2}{p}$ bus cycles. There is no local computation in Step 1. Each of Step 4 and Step 2 takes $\frac{n^2}{p}$ computation time. Thus, the time complexity for the modified algorithm is $O(\frac{n^2}{p})$. In other words, we have $T_1(N) = O(1)$, $T_2(P) = O(\frac{n^2}{p})$, where $N = n^2$ and $P = p$. The scalability factor for our algorithm is $g(N, P) = (T_2(P)/T_1(N)) \times \frac{P}{N}$=1. Thus our Boolean matrix multiplication algorithm is *completely scalable*. We obtain the following theorem.

Theorem 2 *Given two $n \times n$ Boolean matrices A and B. Let C denote the product of matrices A and B. There is a completely scalable algorithm on the LARPBS model. The algorithm runs in $O(\frac{n^2}{p})$ time, where p is the number of processors and $1 \leq p \leq n^2$.*

3.6 Application to Transitive Closure

Li [31] gave an $O(\log n)$-time transitive closure algorithm by repeatedly applying his constant-time Boolean matrix multiplication (BMM) algorithm. Our BMM algorithm can also be used directly for calculating the transitive closure of a directed graph.

The following lemma is given in [28] with proof.

Lemma 3.6.1 *Let A be an $n \times n$ Boolean matrix representing the directed graph $G = (V, E)$. Then the incidence matrix A^* of the transitive closure of the graph G is given by $A^* = (I + A)^{2^{\lceil \log n \rceil}}$, where I is the $n \times n$ identity matrix.*

Based on Lemma 3.6.1, the incidence matrix A^* can be obtained by repeatedly running Boolean matrix multiplication algorithm for $\log n$ times to compute the following matrix:

$$(A \vee I)^{2^1}, (A \vee I)^{2^2}, \cdots, (A \vee I)^{2^{\lceil \log n \rceil}}$$

Each of these matrices is obtained by following Boolean matrix multiplication operation:

$$(A \vee I)^{2^k} = (A \vee I)^{2^{(k-1)}} \times (A \vee I)^{2^{(k-1)}}$$

where $1 \leq k \leq \log n$.

By repeatedly applying our BMM algorithm, we can obtain an $O(\log n)$-time transitive closure algorithm using only $O(n^2)$ processors.

Algorithm 3.6.1 Transitive_Closure
Input: *adjacency matrix A for graph* $G = (V, E)$.
Output: *transitive closure* A' *of graph* G.
Begin

1. *Compute* $X = A \vee I_n$: each processor P_{ii}, where $1 \leq i \leq n$, set $R_a = 1$.
2. **for** $i = 1$ **to** $\log n$ **do**
   Boolean_Matrix_Multiplication(X,X)
   endfor

End

Theorem 3 *The transitive closure of a directed graph with n vertices can be obtained in* $O(\log n)$ *time on an LARPBS using* $O(n^2)$ *processors.*

3.7 Application to Connected Component Problems

An application of transitive closure algorithm is finding the connected components for undirected graphs and finding strongly connected components for directed graphs. We show how to solve these problems on an LARPBS. Figure 3.2 shows a undirected graph G with 3 connected components.

Algorithm 3.7.1 Connected_Component
Input: *adjacency matrix A for undirected graph* $G = (V, E)$ *distributed to an* n^2*-processor LARPBS, one element per processor.*
Output: *connected components of graph G with vertices and its component ids saved in the first n processors, one vertex per processor.*

Begin

1. Apply algorithm 3.6.1 *Transitive_Closure*.
2. Each processor $P_{i,n}$, where $1 \leq i \leq n$, sets its reconfigurable switches to cross to segment the bus into n sub buses. Then each processor $P_{i,j}$ whose $a_{i,j} = 1$ sends index j to processor $P_{i,n}$. Each processor $P_{i,n}$ accepts the first message addressed to it and ignores the rest. Each processor $P_{i,n}$, where $1 \leq i \leq n$, checks the received index j: if $j = i$, sets $flag_{i,j} = 1$; otherwise, sets $flag_{i,j} = 0$. $P_{i,n}$ then broadcasts $flag_{i,n}$ on its sub bus. Each processor $P_{i,j}$ sets itself to active if the received $flag_{i,j} = 1$ and $a_{i,j} = 1$. Then all processors set their reconfigurable switches to straight to form a single bus.

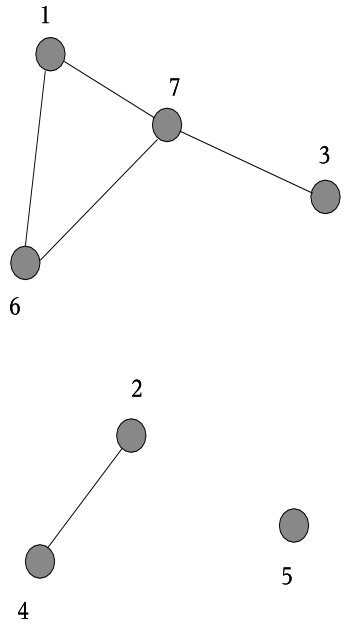


Figure 3.2: Connected components of graph G

3. *Assign index for each connected component* : each processor $P_{i,1}$, where $1 \leq i \leq n$, sets $b_{i,1} = 1$ if its $flag_{i,1} = 1$; all other processors $P_{i,j}$ set $b_{i,j} = 0$. Do a *binary-prefix-sum* operation for $b_{i,j}$. Each processor whose $flag = 1$ saves its prefix sum as its connected component id x.

4. *Collect connected components* : do an *ordered-compression* operation for all active processors $P_{i,j}$ to compress index pairs (x, j) to the first n processors of the bus. Each of the first n processors saves the received index pair (x, j): x is the connected component id, j is the vertex index.

End

Example 3.7.1 *Finding connected components for graph G in Figure 3.2.*

$$\begin{bmatrix} 1&0&0&0&0&1&1\\ 0&1&0&1&0&0&0\\ 0&0&1&0&0&0&1\\ 0&1&0&1&0&0&0\\ 0&0&0&0&1&0&0\\ 1&0&0&0&0&1&1\\ 1&0&1&0&0&1&1 \end{bmatrix} \Rightarrow \begin{bmatrix} 1&0&1&0&0&1&1\\ 0&1&0&1&0&0&0\\ 1&0&1&0&0&1&1\\ 0&\boxed{1}&0&\boxed{1}&0&0&0\\ 0&0&0&0&\boxed{1}&0&0\\ 1&0&1&0&0&1&1\\ \boxed{1}&0&\boxed{1}&0&0&\boxed{1}&\boxed{1} \end{bmatrix} \quad \begin{matrix} j \\ \begin{bmatrix} 7\\4\\7\\4\\5\\7\\7 \end{bmatrix} \end{matrix} \quad \begin{matrix} bps \\ \begin{bmatrix} 0\\0\\0\\1\\2\\2\\3 \end{bmatrix} \end{matrix} \quad \begin{matrix} index \\ \begin{bmatrix} \\ \\ \\ 1\\2\\ \\3 \end{bmatrix} \end{matrix}$$

Adjacency matrix A $\qquad$ *after step* 2

In the matrix shown above, the elements with boxes are active elements. After step 5, following data are stored in the first 7 processors:
$\{(1,2),(1,4),(2,5),(3,1),(3,3),(3,6),(3,7)\}$.
These data give us the following connected components in G: $(2,4)$, (5), and $(1,3,6,7)$.

By adding an extra step, i.e. Step 2 in the following algorithm, we can obtain the following strongly connected component algorithm.

Algorithm 3.7.2 Strongly_Connected_Component
Input: *adjacency matrix A for directed graph $G = (V, E)$ distributed to an n^2-processor LARPBS, one element per processor.*
Output: *strongly connected components of graph G with vertices and its component ids in each component saved in the first n processors, one vertex per processor.*
Begin

1. Apply algorithm 3.6.1 *Transitive_Closure*.

2. *Remove single directional connections* : each processor $P_{i,j}$ whose $a_{i,j} = 1$ sends a dummy message to processor $P_{j,i}$. Then each processor $P_{i,j}$ whose $a_{i,j} = 1$ and who does not receive a message in this *one-to-one* communication operation sets its $a_{i,j} = 0$.

3. Each processors $P_{i,n}$, where $1 \leq i \leq n$, set their reconfigurable switches to cross to segment the bus into n sub buses. Then each processor $P_{i,j}$, whose $a_{i,j} = 1$, sends its index j to processor $P_{i,n}$. Each processor $P_{i,n}$ accepts the first message addressed to it and ignores the rest. Each processor $P_{i,n}$ checks the received index j: if $j = i$, sets $flag_{i,n} = 1$; otherwise, sets $flag_{i,n} = 0$. $P_{i,n}$ then broadcasts $flag_{i,n}$ on its sub bus. Each processor $P_{i,j}$ sets itself to be active if the received $flag_{i,n} = 1$ and $a_{i,j} = 1$. Then all processors set their reconfigurable switches to straight to form a single bus.

4. *Assign index for each connected component* : each processor $P_{i,1}$, where $1 \leq i \leq n$, sets $b_{i,1} = 1$ if its $flag_{i,1} = 1$; all other processors $P_{i,j}$ set $b_{i,j} = 0$. Do a *binary-prefix-sum* operation for $b_{i,j}$. Each processor $P_{i,j}$ whose $flag_{i,j} = 1$ saves its prefix sum as its connected component id x.

5. *Collect connected components* : do an *ordered-compression* operation for all active processors $P_{i,j}$ to compress index pairs (x, j) to the first n processors of the bus. Each of the first n processors saves the received index pair (x, j): x is the connected component id, j is the vertex index.

End

Theorem 4 *The connected components of an undirected graph and the strongly connected components of a directed graph with n vertices can be obtained in $O(\log n)$ time on an LARPBS using $O(n^2)$ processors.*

Chapter 4

Parallel Sorting Algorithms on LARPBS

4.1 Introduction

Sorting is a well-known problem that has been studied extensively in the literature because it has so many important applications and it has a theoretical importance. Researches on parallel sorting algorithms have been done on different models. There are mainly two theoretical computational models that have been considered for parallel sorting algorithms – the circuit model and the PRAM model. A parallel sorting algorithm is often said to be optimal (with respect to cost) if its cost is $O(n \log n)$. The first $O(\log n)$ time sorting algorithm is the AKS sorting network on circuit model, and it is optimal as it uses $O(n)$ comparators. The second $O(\log n)$ time sorting algorithm is Richard Cole's parallel merge sort [11], which is an optimal sorting algorithm on CREW PRAM model. Compared with AKS sorting network, Richard Cole's merge sort algorithm has a much smaller constant in O notation for running time. Although the AKS-network can be combined with a variation of the odd-even network to give an optimal sorting network with $O(\log n)$ time and $O(n)$ comparators, the constant of proportionality of this algorithm is still immense.

Several researches have been done for sorting on the LARPBS model. A constant-time parallel sorting algorithm is given in [62] which uses $O(n^2)$ processors. An efficient and scalable quicksort algorithm on the LARPBS model proposed in [60] runs at $O(\log n)$ average time on an LARPBS with size n, and $O(n)$ time in the worse case. An $O(\log^2 n)$ time deterministic algorithm that sorts n elements on n processors LARPBS is given in [52]. In this chapter, we present an innovative implementation of Cole's parallel merge sort algorithm [11] on LARPBS. To the best of our knowledge, it is the first $O(\log n)$ time sorting algorithm implemented on an LARPBS of n processors. In later chapters, we will apply this algorithm to higher dimensional LARPBS to obtain optimal sorting algorithms.

This chapter is organized as follows. Section 4.2 gives the existing $O(n^2)$ processor constant time sorting algorithm on LARPBS. This algorithm is used in our selection algorithm presented in a later chapter. Section 4.3 introduces the basic idea of the Cole's CREW

PRAM merge sort algorithm. Section 4.4 describes our sorting algorithm on LARPBS model in detail. The issues of processor assignment and reuse are also addressed in this section.

4.2 A Constant-Time Sorting Algorithm on LARPBS

In this section, we give a description of the constant-time sorting algorithm, which is slightly different from the one proposed in [62]. This algorithm uses n^2 processors. Without lose of generality, we assume the n elements are distinct.

Algorithm 4.2.1 Constant_Time_Sort
Input: *a sequence X of n unsorted elements distributed to the first n processors, one per processor.*
Output: *the n elements stored in the first n processors in sorted order, one element per processor.*
Begin

1. Distribute the elements on the first n processors in the following way: processor P_i multicast its element x to processor P_{kn+i}, where $1 \leq k < n$, and $1 \leq i \leq n$.

2. Segment the bus into n sub-buses, each having n processors. Processor P_{in+i} set $y = x$ and broadcasts y to every other processor in its sub-bus.

3. Each processor compares the two elements x and y: if $x \leq y$, sets its $result = 1$; otherwise, sets its $result = 0$. Do a *binary-prefix-sums* operation on each sub-bus. Then all processors set their reconfigurable switches to straight to form a single bus.

4. Each processor P_{kn}, where $1 \leq k \leq n$, uses its binary prefix sums as the address for the destination processor, and sends its element y to that processor.

End

Since we use only basic operations in the algorithm, it is clear to see that the total time required for this algorithm is $O(1)$.

4.3 Cole's Merge Sort Algorithm for CREW PRAM

In order to obtain an optimal sorting algorithm on an LARPBS, we identify two possible approaches. The first one is to implement Cole's algorithm on an LARPBS, and the second one is to used an LARPBS to simulate an AKS sorting network.

4.3.1 Comparison of Cole's Merge Sort and the AKS Sorting Network

Both sorting algorithms are asymptotically optimal, but they are different in the following ways:

- AKS sorting is not uniform. Although for a given value of n, we can construct with a great deal of effort a sorting network with $O(n \log n)$ comparators that sorts n numbers in $O(\log n)$ time, the complexity-parameter, n, cannot be regard as one of the inputs to the sorting algorithm. Different n values require different expander graphs as input. so in a way, we have a different algorithm for each value of n.

- The constant in the complexity expression for AKS sorting is much larger than that of Cole's pipelined merge sort algorithm.

- AKS sorting uses expander graphs for computing the $\epsilon - halver$ of different element sets. Calculation of those expander graphs is not trivial and it brings in a huge constant to the running time complexity. These expander graphs are assumed to be known and treated as part of the input data of the algorithm. Thus, extra work needs to be done to make expander graphs ready before we run the AKS sorting. Cole's pipelined sorting does not make use of expander graphs or any related constructions, thus it avoids the huge constant in the running time.

- AKS sorting is a sorting network; Cole's sorting does not provide a sorting network, it is designed for PRAM.

- Both AKS and Cole's sorting algorithm use a binary tree model to solve the problem. But the way they use it is different: Cole's sorting algorithm starts from the leaves of the tree with one element assigned to each leaf, and it walks up to the root in $O(\log n)$ stages. A constant merging time is achieved at each stage by merging only samples of two sorted arrays. As the merged samples from one level of the tree provide fairly good samples at the next level of the tree, the merges at the different levels of the tree can be pipelined to achieve an overall time complexity of $O(\log n)$. The algorithm stops when it reaches the root node, with the final sorted array of elements stored in the UP array of the root node. AKS sorting starts from the root with the unsorted array of elements assigned to the root node. It tries to split the array of elements into two halves while it walks down the binary tree. But in each stage, the split is just a good approximation. The algorithm tries to correct the errors introduced in the approximation: it sends up elements with possible wrong order for error correction while it walks down the tree. And it stops when it reaches the leaf level of the tree with the elements stored in the leaves in sorted order.

4.3.2 Definitions and Notations

In this section, we give the definitions and notations that we use for describing the pipelined merge sort on an LARPBS throughout the rest of this chapter. Without loss of generality, we assume all input elements are distinct.

- *Interval* $I(e)$: Let L be a sorted array. Define $L_\infty = (-\infty, L, +\infty)$. If $e \in L$, g is the next larger element in L_∞, then we define $[e, g)$ to be the interval induced by e. We denote it as $I(e)$.

- *c-cover*: Let c be a positive integer. Given two sorted arrays L and J. L is a c-cover of J if each interval induced by an element in L contains at most c elements from J.

- *Rank*: The rank of an element x in a sorted array X is defined to be the number of elements in X that are less than or equal to x. A processor's rank refers to the rank of the element assigned to it.

- *Straddle*: Let e, f, g be three elements, with $e < g$. We say that e and g straddle f if $e \leq f$ and $f < g$. We refer to e and g as the two straddle elements for f. And if e, f, g are associated to processors P_i, P_j, and P_k respectively, we refer to P_i and P_k as the two straddle processors for processor P_j.

- $L \rightarrow J$: Let L and J be two sorted arrays, $L \rightarrow J$ denotes that L is ranked in J; i.e., for each element in L, we know its rank in J.

- $L \leftrightarrow J$: Let L and J be two sorted arrays, $L \leftrightarrow J$ denotes that L and J are cross-ranked.

- $L(u)$: a final sorted set of elements rooted at node u.

- $UP(u)$: a sorted subset of the elements in $L(u)$.

- *External node/internal node*: node u is external if it has $|UP(u)| = |L(u)|$; otherwise, it is an internal node.

- *Active node*: an active node is either an internal node whose UP array is non-empty; or it is an external node in its first three stages (external nodes do not participate in the algorithm after three stages).

- $NEWUP(u)$, $OLDUP(u)$: the corresponding UP arrays in the next, and the previous stage, respectively.

- $SUP(u)$: if u is an internal node, $SUP(u)$ is a sorted array comprising every fourth element in $UP(u)$, measured from the right end; if u is an external node, $SUP(u)$ is every fourth, every second and every element in $UP(u)$ for the first, second and third stages.

- $NEWSUP(u)$, $OLDSUP(u)$: the corresponding SUP arrays in the next, and the previous stage, respectively.

- *Calculate* $NEWUP(u)$: Let u, v and w be three nodes in the sorting tree, such that u is the parent of nodes v and w. We define $NEWUP(u) = SUP(v) \bigcup SUP(w)$, i.e., array $NEWUP(u)$ is the result of merging arrays $SUP(v)$ and $SUP(w)$.

- *find-min-representatives*: Assume that an LARPBS bus holds a sorted array of elements. Each processor in the array also holds the rank of its element. The rank can be its element's rank in the array it belongs to or the rank of its element in another sorted array. The operation is done in the following way: each processor sends its rank r to the one on its right, then compares the received rank with its own rank. If the two ranks do not match, then the processor sets itself to be a representative. The last one in the bus will not receive a rank in this one-to-one communication. It always sets itself to be a representative. This operation takes one bus cycle and one local comparison.

- *find-max-representatives*: Same as the *find-min-representatives* operation except that each processor sends its rank r to the one on its left, and the first processor always sets itself to be representative.

4.3.3 Basic Idea of the CREW PRAM Merge Sort Algorithm

In this section, we describe the basic idea in Cole's parallel merge sort on a CREW PRAM [11]. This algorithm is based on an $O(\log n)$ time binary tree merge procedure, which is performed in $3 \log n$ stages. Let u be a node in the sorting tree, v and w be the two children of node u. In each stage, the computation comprises the following two phases:

1. Form the array $SUP(u)$.

2. Compute $NEWUP(u) = SUP(v) \cup SUP(w)$, where $\cup$ denotes merging.

If a node becomes external, then it will only be active for another three stages. On each of these three external stages, phase two is not performed, and phase one is performed with sampling rate of 4, 2, and 1 respectively for the first, second, and third stage, respectively. Internal nodes use the following equation to compute $SUP(u)$ arrays:

$$r = m - 3 - 4i \tag{4.1}$$

where r refers to qualified sampling rank, m is the number of elements in $UP(u)$ array, and $0 \leq i < \lfloor m/4 \rfloor$.

As phase 1 for internal nodes is obvious, we focus on phase 2 for internal nodes. The following lemmas are provided [11], and we need them for developing our sorting algorithm on an LARPBS:

Lemma 4.3.1 *While* $0 < |UP(u)| < |L(u)|, |NEWUP(u)| = 2|UP(u)|$.

Lemma 4.3.2 *The algorithm has* $3 \log n$ *stages.*

Lemma 4.3.3 $OLDSUP(v)$ *is a* 3*-cover for* $SUP(v)$ *for each internal node.*

Lemma 4.3.4 $UP(u)$ *is a* 3*-cover of* $SUP(v)$, $UP(u)$ *is a* 3*-cover of* $SUP(w)$, *if* u *is the parent of* v *and* w.

Lemma 4.3.5 *3 stages after node* v *becomes external, its parent, node* u, *also becomes external.*

Also we assume $UP(u) \rightarrow SUP(v)$ and $UP(u) \rightarrow SUP(w)$, as it is calculated in Step 2 of Phase 2 in the previous stage.

Phase 2 for internal node can be done in two steps:

- Step 1: computing $NEWUP(u)$
- Step 2: maintaining following ranks: $NEWUP(u) \rightarrow NEWSUP(v)$ and $NEWUP(u) \rightarrow NEWSUP(w)$.

Step 1 has following three substeps:

- *Substep* 1: Compute $SUP(v) \rightarrow UP(u)$ and $SUP(w) \rightarrow UP(u)$. Let y be an element in $UP(u)$, $I(y)$ be the interval in $UP(u)$ induced by y, and r_y be y's rank in $SUP(v)$(obtained from $UP(u) \rightarrow SUP(v)$). Since $UP(u)$ is a 3-cover of $SUP(v)$, there are at most 3 elements after the ${r_y}^{th}$ element in $SUP(v)$ fall into $I(y)$. To obtain $SUP(v) \rightarrow UP(u)$, processor associated with element y can send its rank in $UP(u)$ to those elements in $SUP(v)$. $SUP(w) \rightarrow UP(u)$ is calculated in a similar way.
- *Substep* 2: Cross rank $SUP(v)$ and $SUP(w)$: $SUP(v) \leftrightarrow SUP(w)$. Using $SUP(v) \rightarrow UP(u)$, for each element e in $SUP(v)$, we can find the two elements d and f in $UP(u)$ that straddle element e. And using $UP(u) \rightarrow SUP(w)$, we can get the rank of d and f in $SUP(w)$, say r and t. So e's rank in $SUP(w)$ falls into interval $(r, t]$. As $UP(u)$ is a 3-cover for $SUP(w)$, by means of at most three comparisons we will be able to find e's rank in $SUP(w)$. Thus, we obtain $SUP(v) \rightarrow SUP(w)$. We can obtain $SUP(w) \rightarrow SUP(v)$ in a similar way.
- *Substep* 3: Compute $NEWUP(u)$: $NEWUP(u) = SUP(v) \cup SUP(w)$. Based the result obtained in Substep 2, for any element e in $SUP(v)$ or $SUP(w)$, its rank in $NEWUP(u)$ can be obtained simply by adding its rank in $SUP(v)$ and its rank in $SUP(w)$. So computing $NEWUP(u) = SUP(v) \cup SUP(w)$ can be done in constant time.

Step 2 maintains the following ranks that will be used in Step 1 of the next stage: $NEWUP(u) \rightarrow NEWSUP(v)$ and $NEWUP(u) \rightarrow NEWSUP(w)$. We describe in detail how to rank $NEWUP(u)$ in $NEWSUP(v)$. Ranking $NEWUP(u)$ in $NEWSUP(w)$ can be done in a similar way. We classify elements in $NEWUP(u)$ into two types: those came from $SUP(v)$ and those came from $SUP(w)$. As the ranking process is different for the two types of elements, we show how to rank each type in $NEWSUP(v)$ separately as follows:

Observation 1*:* Given the rank of an element from $UP(u)$ in both $SUP(v)$ and $SUP(w)$ (we obtained it in Step 2 of Phase 2 in the previous stage), and given $NEWUP(u) = SUP(v) \cup SUP(w)$, we can easily rank an element e from $UP(u)$ in $NEWUP(u)$ by adding e's rank in $SUP(v)$ and $SUP(w)$. So at the end of Step 1 of Phase 2, we have $UP(u) \rightarrow NEWUP(u)$.

- *Elements came from* $SUP(v)$: Assume that element e in $NEWUP(u)$ came from $SUP(v)$. As we know e's rank in $SUP(v)$ and $SUP(v)$ is sampled from $UP(v)$, we can obtain e's rank in $UP(v)$. Apply Observation 1 to node v, we have $UP(v) \rightarrow NEWUP(v)$. Thus, we can get e's rank in $NEWUP(v)$. As $NEWSUP(v)$ is sampled from $NEWUP(v)$, we can get e's rank in $NEWSUP(v)$. So for every element e in $NEWUP(u)$ that came from $SUP(v)$, we obtain its rank in $NEWSUP(v)$.

- *Elements came from* $SUP(w)$: Assume that element e in $NEWUP(u)$ came from $SUP(w)$. From the results obtained in Step 1, we know e's two straddle elements d and f in $SUP(v)$. Based on the results we obtained in the previous paragraph, we know the ranks r and t in $NEWSUP(v)$ for d and f, respectively. So we know the interval in $NEWSUP(v)$ that e falls into is $(r, t]$. Next we need to find out the rank for e in interval $(r, t]$. According to Lemma 4.3.3 and the 3-cover property, we can find out e's rank in $(r, t]$ by means of at most three comparisons. So for every element e in $NEWUP(u)$ that came from $SUP(w)$, we obtain its rank in $NEWSUP(v)$.

As both Step 1 and Step 2 requires constant time, and the algorithm has $3 \log n$ stages, the time complexity of this algorithm is $O(\log n)$. Each stage of the algorithm requires $O(n)$ processors. As processors can be reused from stage to stage, the algorithm needs only $O(n)$ processors.

4.4 Parallel Merge Sort on LARPBS

4.4.1 Problems Need to be Solved

PRAM is a less restrictive model. It provides an excellent framework for studying parallelism. There are no wires to worry about, and one never needs to worry about getting the right data to the right place at the right time. So the PRAM model abstractly removes most of the important details associated with implementing a parallel algorithm on a parallel machine. But in the LARPBS model, we do need to take care of inter-processor communication explicitly and efficiently. Thus to implement the pipelined merge sort algorithm on the LARPBS model, we are facing the following challenges:

- *Simulating multiple binary comparison trees on an LARPBS model*: we need to keep track of three arrays, UP, SUP, and $NEWUP$, for each active node in the sorting tree. How to assign processors for items in these arrays and how to rank them among each other are important issues need to be addressed.

- *Communications*: PRAM is a shared memory parallel computing model. So every processor has access to the shared memory. Communication among processors and exchanging information in the PRAM model is obvious and simple. But LARPBS is a distributed memory interconnection model. How to efficiently communicate among processors and exchange information among them are important issues that we have to take care of in order to implement the algorithm on LARPBS model.

- *Processor assignment and reuse*: In order to use only $O(n)$ processors, we need to find a solution to processor reuse.

4.4.2 Details of the Algorithm

This algorithm is based on the parallel sorting tree model. The algorithm "moves up" from the leaves to the root, and it runs in $3 \log n$ stages. In Cole's algorithm, each stage consists of two phrases. We convert the two phrases into 5 steps in our algorithm: 1 step for Phase 1 and 4 steps for Phase 2. In order to address the fact that external nodes only stay active for three stages, we call each 3-stage period from the time a set of nodes becomes external, a life cycle. So the algorithm runs in $log(n)$ life cycles. At the beginning of the algorithm, we only have UP arrays: each leaf has an UP array which contains one item. The LARPBS merge sort algorithm proceeds in $O(\log n)$ stages as follows:

Algorithm 4.4.1 Pipelined_Merge_Sort
Input: *a sequence of n keys S*
Output: *a sorted sequence S′ of the n keys*
Begin

for $LifeCycle = 1$ **to** $\log n$, all processors **pardo**
$DoLifeCycle(LifeCycle)$;
endfor
End

Algorithm 4.4.2 DoLifeCycle(i)
Begin

for $stage = 1$ **to** 3 **do**
Step 1 – Compute SUP **arrays**:
For internal node: each processor P_i in UP array checks its rank r in that array, if $r = m - 3 - 4j$, then P_i sends its element to the j^{th} processor in SUP array; for external nodes: sampling rate is $2^{(3-stage)}$, action for sampled processors is the same as internal nodes.
Step 2 – Compute rank $SUP(v) \rightarrow UP(u)$ **and** $SUP(w) \rightarrow UP(u)$:

1. Do a *find-min-representatives* operation in $UP(u)$ based on each element's rank in $SUP(v)$. As we have $UP(u) \rightarrow SUP(v)$ (obtained from step 5 of the previous stage and Step 1 of current stage. For stage 1, since only leaf nodes have non-empty SUP arrays, and all these SUP arrays have only one element, the ranking of $UP(u) \rightarrow SUP(v)$ is obvious), each representative processor P_i in $UP(u)$ sends its rank s in $UP(u)$ to the r^{th} processor P_j in $SUP(v)$ if P_i's rank in $SUP(v)$ is r. As different elements in $UP(u)$ can have the same rank in $SUP(v)$, we need to use the find-min-representatives operation here to guarantee the communication from $UP(u)$ to $SUP(v)$ performed in this step is *one-to-one* instead of multiple-to-one. This step consists of two basic operations: *find-min-representatives* and *one-to-one* communication. Both of them takes $O(1)$ time.

2. Each processor P_j in $SUP(v)$ who received a rank s in the previous bus cycle sets its switch to cross to form a sub-bus and broadcast rank $max(0, s-1)$ to every other processor in the sub-bus. Figure 4.1 shows two examples: example (a) illustrates a general case, and example (b) gives a boundary situation where P_i is the first processor in the $UP(u)$ array. This sub-step only consists of a multiple *broadcast* operation.

Now we have obtained $SUP(v) \rightarrow UP(u)$. $SUP(w) \rightarrow UP(u)$ is done in a similar way. We can further parallelize the two computations in the following way: do Step 1 separately for the two computations, then as elements in $SUP(w)$ array and in $SUP(v)$ array are associated with different processors, Step 2 can be performed for $SUP(w)$ and $SUP(v)$ simultaneously. The operations needed for this step is: 2 *find-min-representatives* operations, 2 *one-to-one* communication operations and 1 multiple *broadcast*. Thus it takes $O(1)$ time.

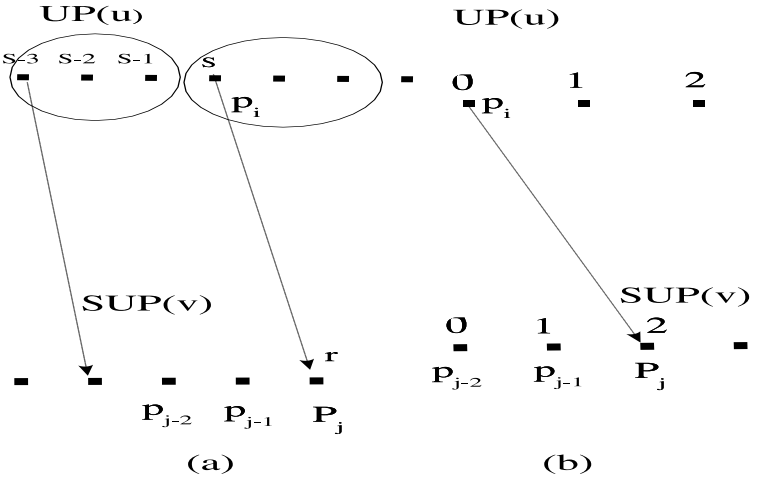


Figure 4.1: $SUP(v) \rightarrow UP(u)$: (a) P_{j-2}, P_{j-1} and P_j all have rank $s-1$ in $UP(u)$. (b) boundary condition: P_{j-2}, P_{j-1} and P_j all have rank 0 in $UP(u)$.

Step 3 – Cross-rank $SUP(v)$ and $SUP(w)$:
We give a description on the details of ranking $SUP(v)$ in $SUP(w)$. $SUP(w) \rightarrow SUP(v)$ can be done in a similar way.

1. Do a *find-max-representatives* operation in $SUP(v)$ based on elements' ranks in $UP(u)$ using $SUP(v) \rightarrow UP(u)$. The *find-max-representatives* here is necessary because two or more consecutive elements in $SUP(v)$ can have the same rank in $UP(u)$.

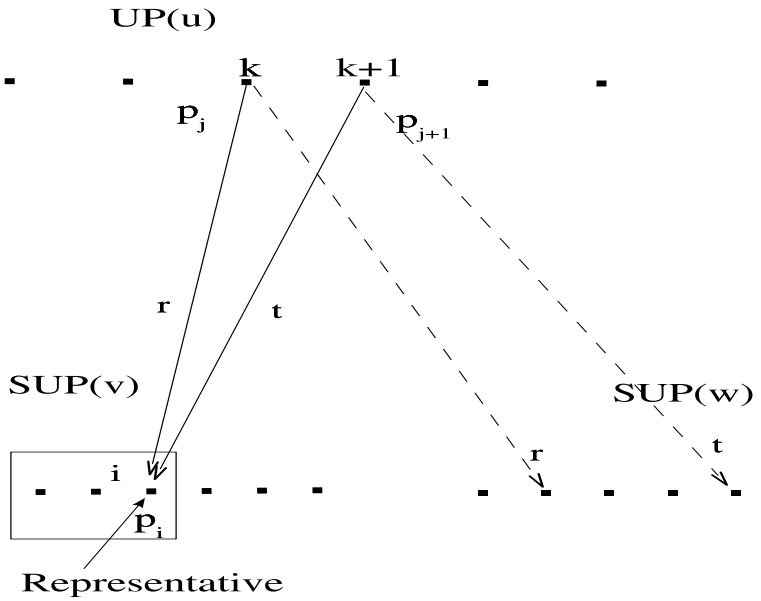


Figure 4.2: Finding the ranks in $SUP(w)$ for P_i's two straddle processors P_j and P_{j+1} in $UP(u)$. Solid lines represent data communication and dashed lines represent ranks.

2. Each representative processor P_i in $SUP(v)$ finds out the ranks in $SUP(w)$ for the two processors that straddle P_i in $UP(u)$ in the following way: using $SUP(v) \rightarrow UP(u)$, each representative P_i sends a request to its two straddle processors in $UP(u)$, P_j and P_{j+1}. This is a *multiple multicast* operation. The request specifies the order in which they should send data back. Processor P_j and P_{j+1} then send their ranks r and t in $SUP(w)$ back to P_i in the specified order. See Figure 4.2.

3. *Representative processors in $SUP(v)$ obtain elements in $(r, t]$ from $SUP(w)$*: each representative processor sends a request to processors whose ranks are in the range $(r, t]$ in $SUP(w)$. This is another *multiple-multicast* operation. The request message indicates the order in which each processor should send its element back. For example, the $(r+1)^{th}$ processor is the first one to send its elements back, $(r+2)^{th}$ is the second one, etc. Each processor in $SUP(w)$ who received a request then sends its element f back to its corresponding representative processor in $SUP(v)$ in the specified order.

As $UP(u)$ is a 3-cover of $SUP(w)$ (see Lemma 4.3.4), a maximum of 3 *one-to-one communication* operations are needed in this $SUP(w)$ to $SUP(v)$ communication.

4. *Calculate* $SUP(v) \rightarrow SUP(w)$: After receiving all elements sent by processors in $SUP(w)$, each representative processor sets its switch to cross to form a sub-bus and broadcasts these elements and value r in the sub-bus. Each processor in $SUP(v)$ now compares these elements with its own element and determines its rank in $SUP(w)$. For example, if a processor owns element x and it received elements q, y, z, and a rank r. If $q < x < y < z$, then x's rank in $SUP(w)$ is $r + 1$. Again due to the 3-cover property of $UP(u)$ stated in Lemma 4.3.4, a maximum of 3 comparisons are needed in the above local computation.

While Computing $SUP(w) \rightarrow SUP(v)$ in a similar way, the above Step 2 needs to be done separately for computing the two ranks, but Step 1, 3, and 4 can be done simultaneously for the two computation: $SUP(w) \rightarrow SUP(v)$ and $SUP(v) \rightarrow SUP(w)$.

Step 4: Compute $NEWUP(u) = SUP(v) \cup SUP(w)$:
Each processor P_i in $SUP(v)$ and $SUP(w)$ computes its rank r in $NEWUP(u)$ by adding its rank i in $SUP(v)$ and its rank j in $SUP(w)$. Each P_i then sends its element to the r^{th} processor in $NEWUP(u)$ to form $NEWUP(u)$. Note that $SUP(v) \rightarrow NEWUP(u)$ and $SUP(w) \rightarrow NEWUP(u)$ are two side products of this step that will be used in Step 5.

Step 5: Maintain ranks: $NEWUP(u) \rightarrow NEWSUP(v)$**,** $NEWUP(u) \rightarrow NEWSUP(w)$. These ranks will be used in Step 2 of the next stage. And we only need to compute them for internal nodes. We show how to compute $NEWUP(u) \rightarrow NEWSUP(v)$ in detail. Computing $NEWUP(u) \rightarrow NEWSUP(w)$ is done in a similar way. Here is the basic idea: We have $SUP(v) \rightarrow NEWUP(u)$ and $SUP(w) \rightarrow NEWUP(u)$ obtained from Step 4. If we can get $SUP(v) \rightarrow NEWSUP(v)$ and $SUP(w) \rightarrow NEWSUP(v)$, then use $NEWUP(u) = SUP(v) \bigcup SUP(w)$, we can obtain $NEWUP(u) \rightarrow NEWSUP(v)$. Now we give the detail.

1. *Compute* $UP(u) \rightarrow NEWUP(u)$: Given $UP(u) \rightarrow SUP(v)$ and $UP(u) \rightarrow SUP(w)$, each processor in $UP(u)$ computes its rank r in $NEWUP(u)$ by adding its rank i in $SUP(v)$ and its rank j in $SUP(w)$, i.e. $r = i + j$.

2. *Compute* $SUP(v) \rightarrow NEWSUP(v)$: Using $UP(v) \rightarrow NEWUP(v)$, each processor P_i in $UP(v)$ whose element is sampled into $SUP(v)$ as the k^{th} element sends its rank i in $UP(v)$ to the j^{th} processor in $NEWUP(v)$ if the rank of P_i's element in $NEWUP(v)$ is j. According to Lemma 4.4.3, this is a one-to-one communication. Each processor P_j in $NEWUP(v)$ then computes its element's rank r in $NEWSUP(v)$ and sends r back to processor P_i in $UP(v)$. After receiving the rank r, processor P_i forwards r to the k^{th} processor in $SUP(v)$. Now each processor in $SUP(v)$ has its element's rank in $NEWSUP(v)$. See Figure 4.3.

3. *Compute* $NEWUP(u) \rightarrow NEWSUP(v)$: We classify elements in $NEWUP(u)$ into two classes: those came from $SUP(v)$ and those came from $SUP(w)$.We need to compute their ranks in $NEWSUP(v)$ using different methods:

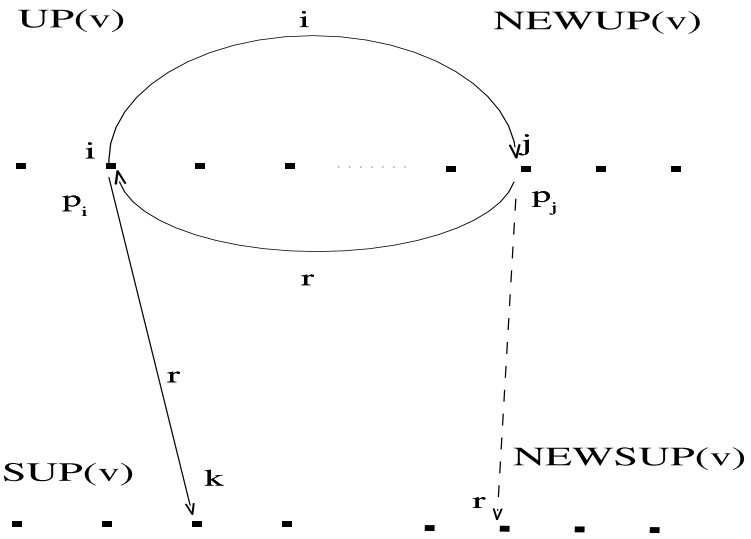


Figure 4.3: Computing $SUP(v) \rightarrow NEWSUP(v)$. Solid lines represent data communication, and dashed lines represent ranking relation.

- *Elements came from* $SUP(v)$: Using $SUP(v) \rightarrow NEWUP(u)$, each processor P_i in $SUP(v)$ sends its element's rank r in $NEWSUP(v)$ to the j^{th} processor P_j in $NEWUP(u)$ if this element's rank in $NEWUP(u)$ is j. Processor P_j in $NEWUP(u)$ set its element's rank in NEWSUP(v) to be the received rank r. After this one-to-one communication, we obtained the ranks in $NEWSUP(v)$ for all elements in $NEWUP(u)$ who came from $SUP(v)$. The correctness of this operation is based on the following two facts: (1) Processor P_i in $SUP(v)$ holds the same element as processor P_j in $NEWUP(u)$ as $NEWUP(u) = SUP(v) \cup SUP(w)$; (2) According to Lemma 4.4.4, processor P_i in $SUP(v)$ holds the same element as the r^{th} processor in $NEWSUP(v)$.
- *Elements came from* $SUP(w)$: We haven't formed $NEWSUP(v)$ arrays at this time. So what we will do is: for each element e in $NEWUP(u)$ that came from $SUP(w)$, find out the ranks r and t in $NEWSUP(v)$ for the two elements in $SUP(v)$ that straddle e, and keep them in e's owner P_i's local memory in $NEWUP(u)$ array. Then we compare e with the elements whose ranks are in the range $(r, t]$ in $NEWSUP(v)$ in Step 1 of the next stage. We use a similar method as what we do in Step 3 to obtain the two ranks r and t.
  (a) Do a *find-max-representatives* in $SUP(w)$ using the ranks $SUP(w) \rightarrow SUP(v)$. Then each representative processor P_i sends request to the two processors in $SUP(v)$ that straddle P_i for their ranks r and t in $NEWSUP(v)$. Processors in $SUP(v)$ who received the request, find out the requested rank using

$SUP(v) \rightarrow NEWSUP(v)$ and send them back to P_i.

(b) Each representative processor P_i in $SUP(w)$ then sends the two ranks r and t to the k^{th} processor P_j in $NEWUP(u)$, where k is P_i's rank in $NEWUP(u)$. Processor P_j in $NEWUP(u)$ receives and saves the two ranks, and marks itself to be a representative.

(c) In Step 1 of the next stage, after building $NEWSUP(u)$ arrays, each representative processor P_j in $NEWUP(u)$, sends a request to processors in the range $(r, t]$ in $NEWSUP(v)$. Applying a similar computation as what we do in Step 3, each processor in $NEWUP(u)$ whose elements came from $SUP(w)$ will get its rank in $NEWSUP(v)$.

Now we have $NEWUP(u) \rightarrow NEWSUP(v)$. $NEWUP(u) \rightarrow NEWSUP(w)$ can be obtained in a similar way.

endfor
End

4.4.3 Processor Assignment and Reuse

Consider the following two approaches for simulating a binary tree on an LARPBS:

1. *Depth-first search based simulation*: Associate one processor to a node in the tree based on its rank obtained from a depth-first search. See Figure 4.4(a).

2. *Breath-first search based simulation*: Associate one processor to a node in the tree based on its rank obtained from a breath-first search. See Figure 4.4(b).

The simulation for the sorting tree in our algorithm is more involved. We use a breath-first search based simulation, but we view the three groups of arrays as three binary trees – UP tree, SUP tree, and $NEWUP$ tree. We simulate the three binary trees on LARPBS by segmenting the bus into three sub-buses in that order (See Figure 4.5). We call the sub-bus allocated to one of the three trees a *tree-bus*. Within each *tree-bus*, we only assign processors to active nodes with non-zero array elements. With respect to each tree-bus, we further segment the bus into smaller sub buses, one for each node; we call it a *node-bus*. On each node-bus, elements are always aligned in sorted order, one element per processor. The final result is on the root *node-bus* of the UP *tree-bus*.

Now we determine how many processors the algorithm needs.

Lemma 4.4.1 *The number of processors needed on any stage of the parallel merge sort algorithm is upper bounded by* $4n$.

Proof. According to [11], the size of the UP arrays is upper bounded by $n + n/7$, $n + 2n/7$, and $n + 4n/7$ on the first, second and third stages respectively, in each life cycle. The sizes of $SUP/NEWUP$ arrays are upper bounded by $2n/7$, $4n/7$, and $8n/7$ on the first,

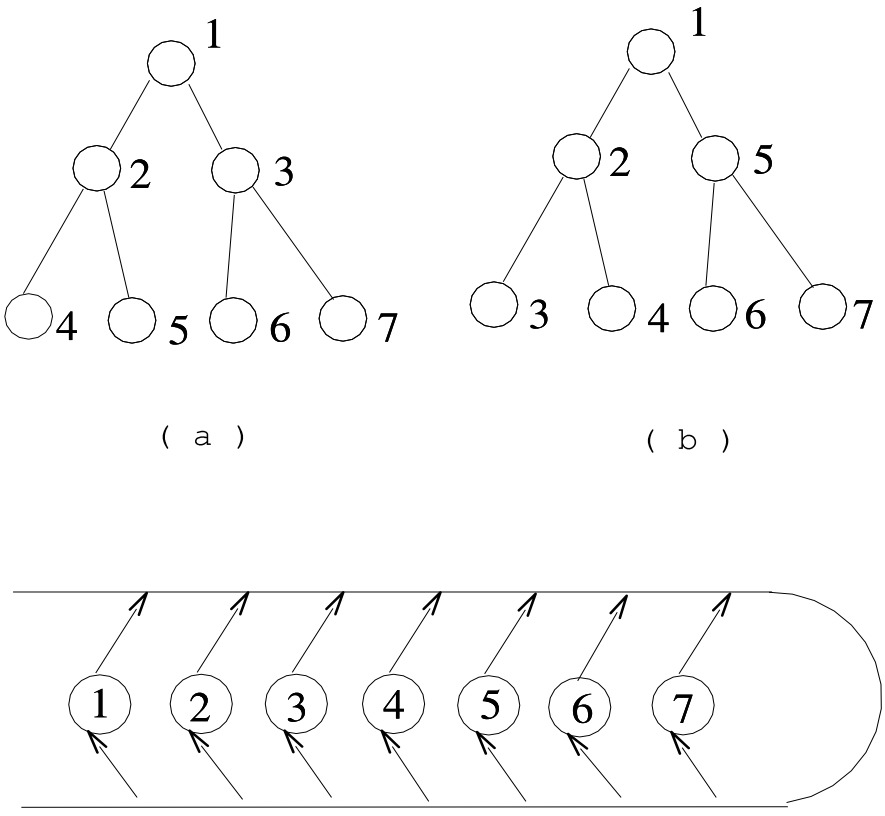


Figure 4.4: Simulating a binary tree on the LARPBS Model: (a) depth-first search ranking (b) breath-first search ranking, and (c) simulation of binary tree on LARPBS

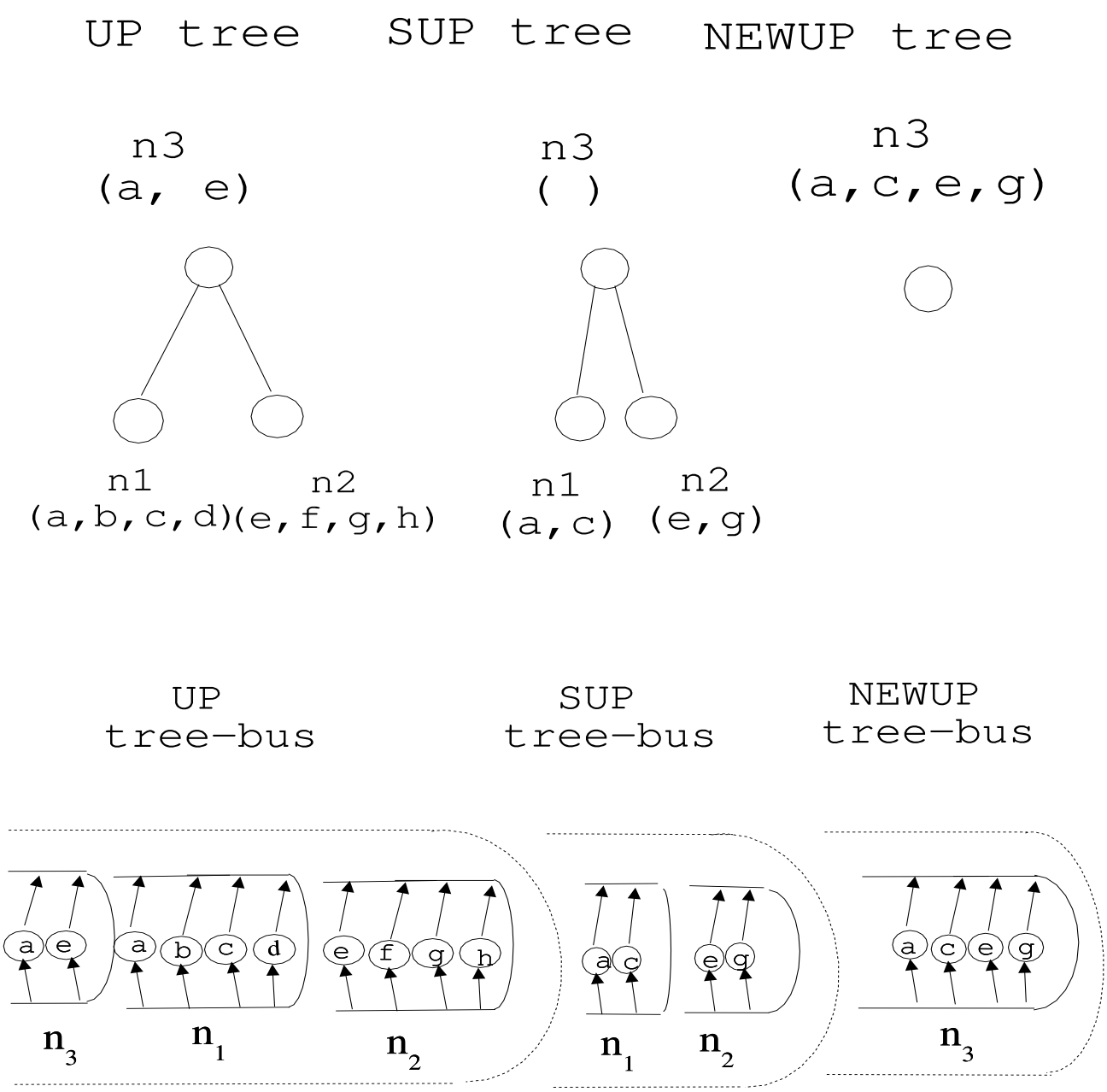


Figure 4.5: Simulating a parallel merge sort tree on the LARPBS Model

second and third stages respectively. Thus the maximum size for all arrays on any stage is upper bounded by $(3 + 6/7)n < 4n$. This proves the lemma.

Based on Lemma 4.4.1, we reserve $4n$ processors for our parallel merge sort algorithm. Notice that number of elements for each node on the three trees in any stage of a life cycle is fixed, so the address for each element in an array for any active node can be pre-calculated and available to all processors as input data. As we need one processor for each element in the UP, NEWUP, and SUP arrays, and the sizes of these arrays, for each node u, change from stage to stage, the processors must be reassigned dynamically so that we can reuse them. Processor reuse is done in the following two ways:

- At the end of each stage, we set all processors assigned to SUP to be available for reuse, and copy the $NEWUP$ tree to UP tree. Thus processors are reused from stage to stage.
- As external nodes become inactive at the end of each life-cycle, processors assigned to them can be reused by new active nodes.

4.4.4 An Example

Now we provide an example for the pipelined merge sort algorithm. Suppose the input is an array of unsorted elements, $S = 7, 3, 19, 1, 23, 74, 32, 17$. We put these elements at the leaves of a complete binary tree, one number per leaf. Let us define the level of a node to be the length of the path from that node to the root node. Then in our example, the binary sorting tree has following 4 levels from root to leaves: 0, 1, 2, 3.

Initially, elements in S are distributed to leaf nodes, one per node. And all leaf nodes are active external nodes. The UP array for each leaf node is the element stored there. All other nodes in the tree are inactive (See Figure 4.6 (a)).

life-cycle 1:

- Stage 1: According to the sampling rule for external nodes nothing happens in this stage. See Figure 4.6 (a).
- Stage 2: same as first stage, nothing happens.
- Stage 3: we have $SUP = L$ for all the leaf nodes. So UP arrays for the nodes on level 2 are given non-zero values, and become new active external nodes. The resulting tree is shown in Figure 4.6 (b).

The leaf nodes, i.e. level 3 nodes in our example, will become inactive in the next life-cycle.

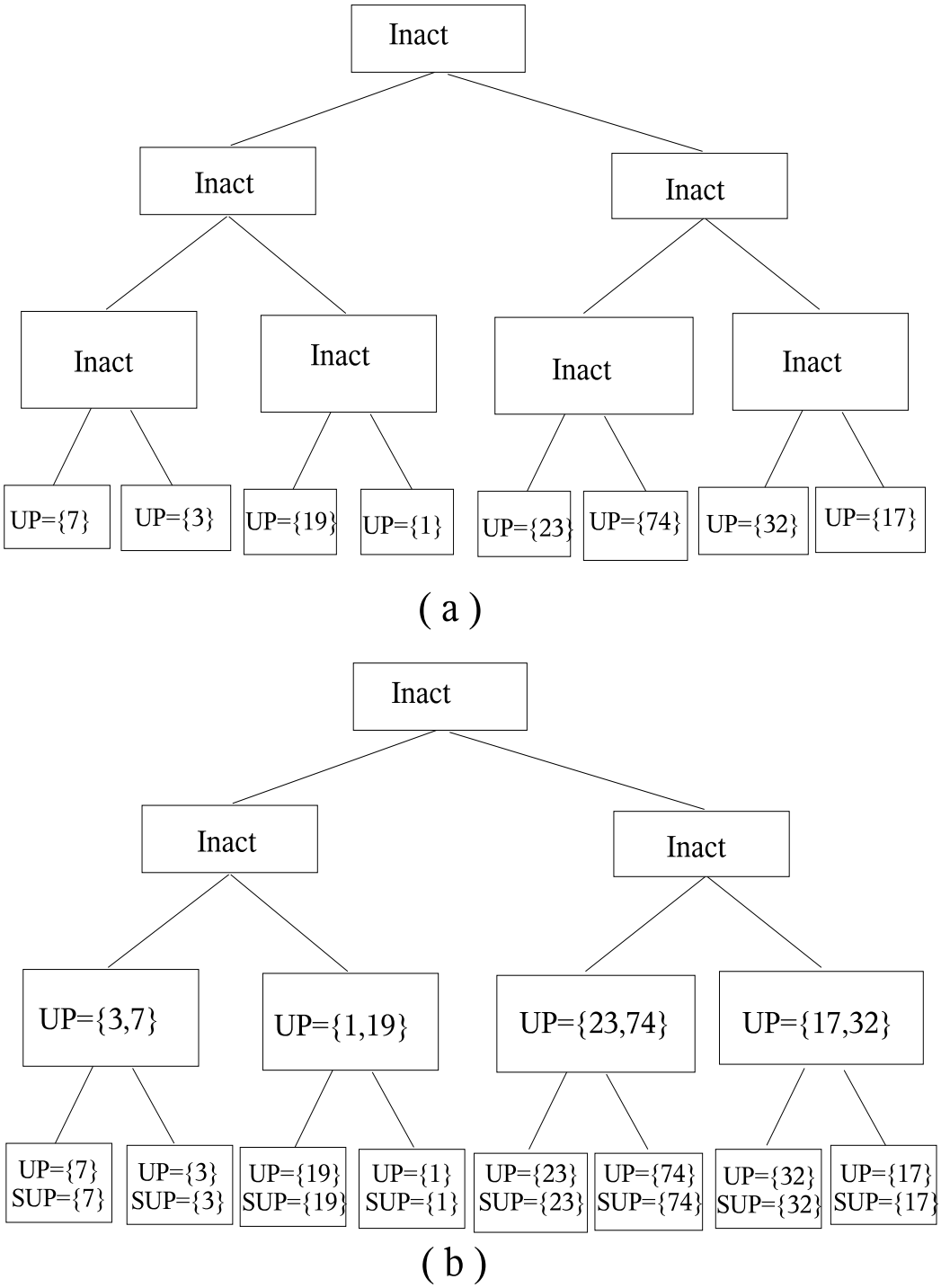


Figure 4.6: Pipelined Merge Sort: life-cycle 1 (a) input, and (b) after stage 3

life-cycle 2:

- Stage 1: as the sampling rate for this stage is 4 and there are only 2 elements in each UP array in all active nodes, nothing happens.
- Stage 2: as the sampling rate for external nodes is reduced to 2, so the first element in the UP array for level 2 node will be sampled to form its SUP array. That will provide a non-empty UP array for each node on level 1. See Figure 4.7 (a).
- Stage 3: as the sampling rate for external nodes becomes every node in UP array, we have $SUP = L$ for all nodes in level 2. But the SUP arrays for the nodes on level 1 will still be empty as number of elements in the UP for level 1 nodes are less than 4. Thus level 1 UP arrays will be updated, and level 2 nodes become external. See Figure 4.7 (b).

Level 2 nodes will become inactive in the next life-cycle.

life-cycle 3:

- Stage 1: the first element in UP array for each level 1 node will be sampled into its SUP array. Thus root node will becomes active with two elements in its UP array. See Figure 4.8 (a).
- Stage 2: the first and third elements in UP array for each level 1 node will be sampled into its SUP array. And UP array for root node will be updated to have four elements. See Figure 4.8 (b).
- Stage 3: All elements in UP array for level 1 nodes will be sampled and we will have the final sorted array of elements in the UP array of root node. See Figure 4.8 (c).

4.4.5 Correctness Proof and Complexity Analysis

The following lemmas are used to show the correctness of the algorithm:

Lemma 4.4.2 *$SUP(v)$ is a subset of $NEWSUP(v)$, i.e., $SUP(v) \subset NEWSUP(v)$.*

Proof. We prove the result by induction on the levels of the nodes in the sorting tree. The claim is true for the nodes on the lowest active level of the sorting tree in a life cycle, i.e., the external nodes. If v is an external node, then (a) in the first stage after it becomes external, $SUP(v)$ consists of every forth elements in $UP(v) = L(v)$; (b) in the second stage, $SUP(v)$ consists of every second element in $L(v)$; and (c) in the third stage, $SUP(v)$ consists of every element in $L(v)$. Thus our claim holds for all external nodes in each life cycle.

Inductive step: We seek to prove that for any node u on level k, we have $SUP(u) \subset NEWSUP(u)$, where k is a level above the external node level. Suppose our claim for

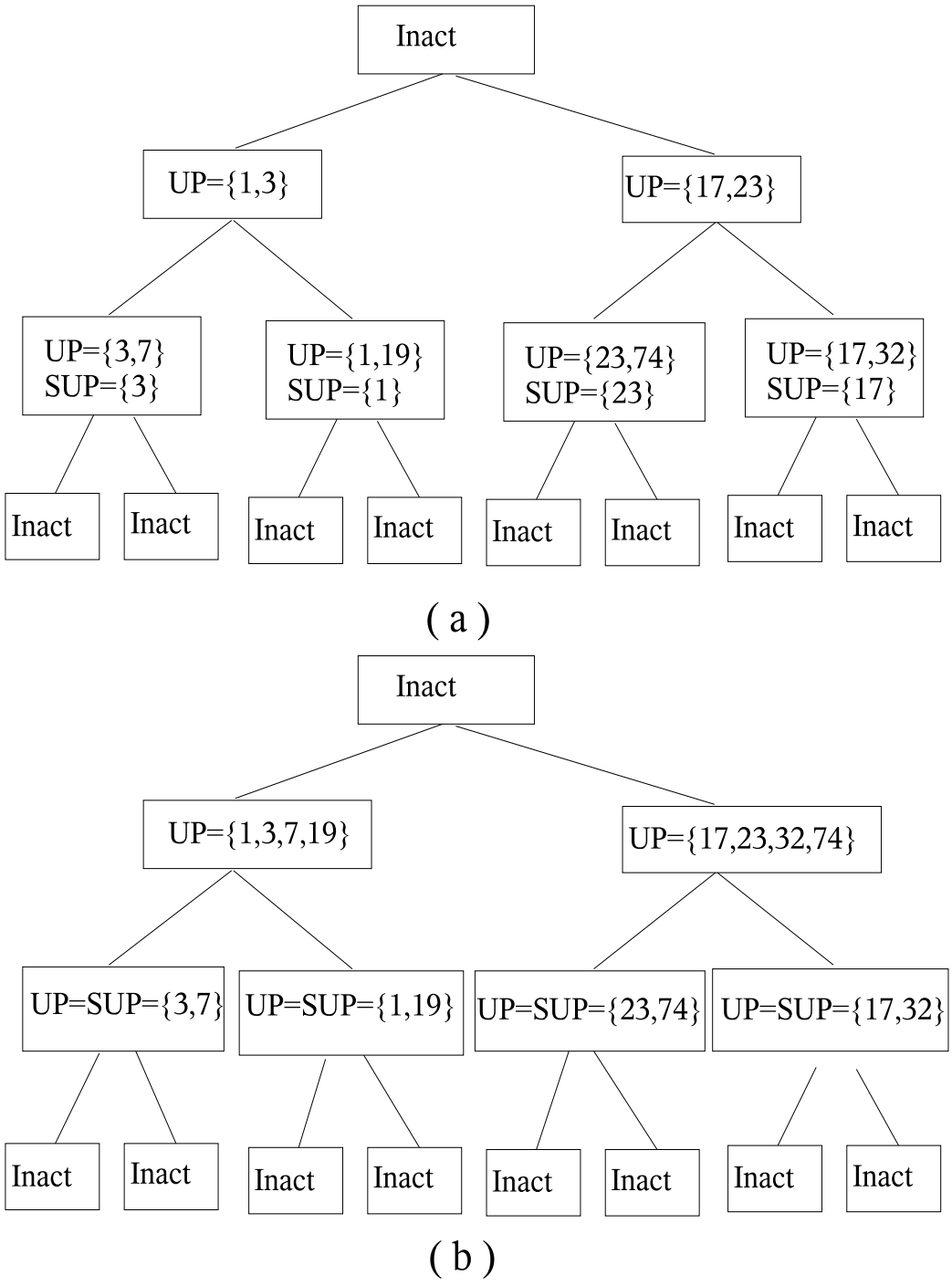


Figure 4.7: Pipelined Merge Sort: life-cycle 2 (a) after stage 2, and (b) after stage 3

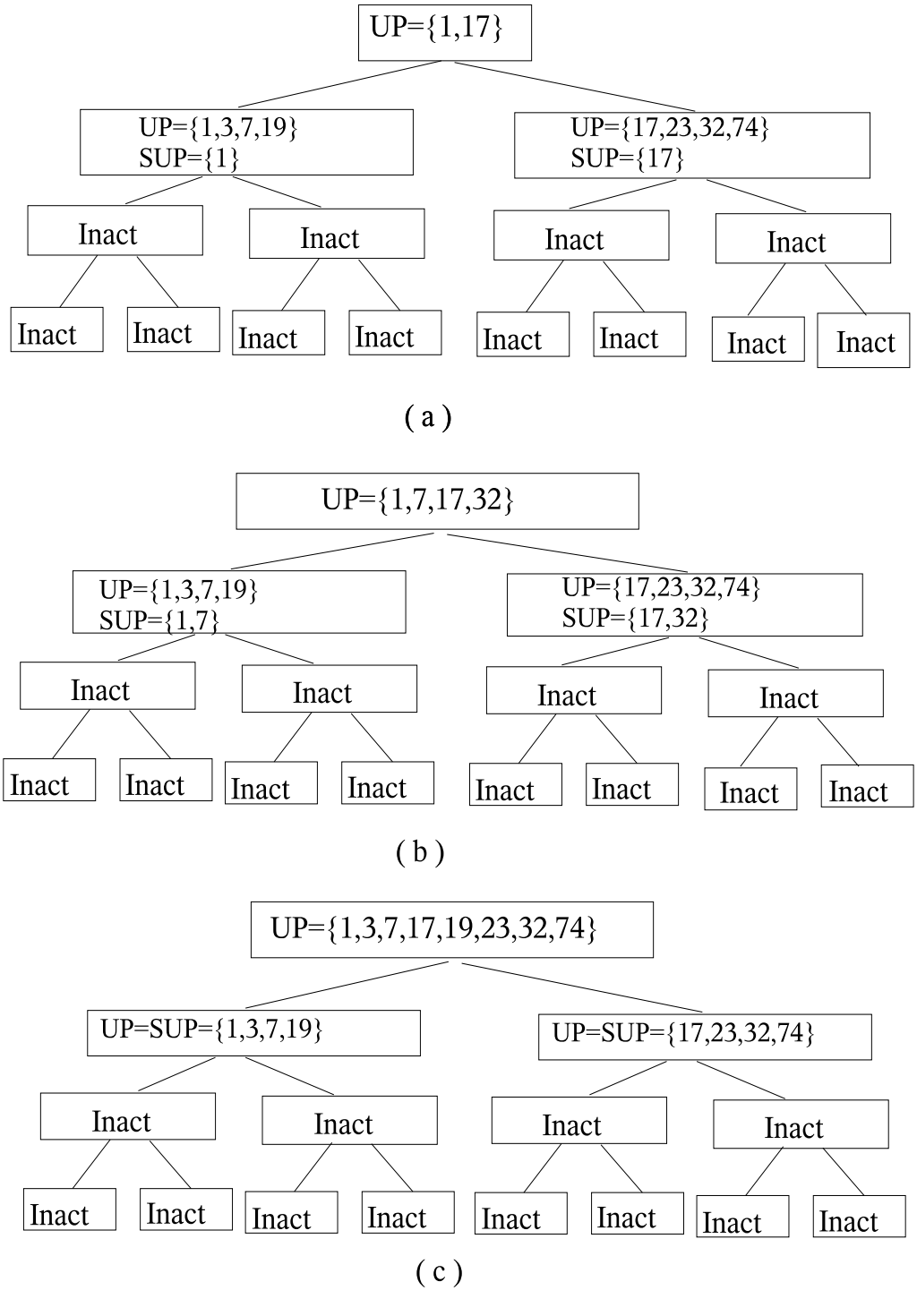


Figure 4.8: Pipelined Merge Sort: life-cycle 3 (a) after stage 1, (b) after stage 2, and (c) after stage 3

$SUP(u) \subset NEWSUP(u)$ on level k is not true. Then there must be an element $e \in SUP(u)$, but $e \notin NEWSUP(u)$. As e comes from $UP(u)$ which in term comes from the $OLDSUP$ array of one of its children, we use the same induction to trace the origin of u down to the lowest active level, which is the level that has the external nodes. Assume that e originally comes from external node f. So we have $e \in OLDSUP(f)$, but $e \notin SUP(f)$. That is conflict with what we have already proved for the initial case. Thus our claim is true for level k.

Summarizing the initial step and inductive step, we conclude that $SUP(v) \subset NEWSUP(v)$.

Lemma 4.4.3 *$UP(u)$ is a subset of $NEWUP(u)$, i.e., $UP(u) \subseteq NEWUP(u)$.*

Proof. Case 1: $|UP(u)| < L(u)$, $NEWUP(u) = SUP(v) \cup SUP(w)$, and $UP(u) = OLDSUP(v) \cup OLDSUP(w)$. Using Lemma 4.4.2, we get $UP(u) \subseteq NEWUP(u)$. Case 2, $|UP(u)| = L(u)$, we have $NEWUP(u) = UP(u)$. Combined the two cases, we obtain $UP(u) \subseteq NEWUP(u)$.

Lemma 4.4.4 *If e in $SUP(v)$ has a rank r in $NEWSUP(v)$, and the r^{th} element in $NEWSUP(v)$ is f, then we have $e = f$.*

Proof. This lemma directly follows from Lemma 4.4.2 and the assumption that all n numbers are distinct.

Lemma 4.4.5 *If e in $UP(u)$ has a rank r in $NEWUP(u)$, and the r^{th} element in $NEWUP(u)$ is f, then we have $e = f$.*

Proof. This lemma directly follows from Lemma 4.4.3 and the assumption that all n numbers are distinct.

Now we calculate the time complexity of the algorithm. Step 1 consists of a simple local comparison and a one-to-one communication, so it is clear that it takes constant time. Step 2 consists of following three basic operations: *find-min-representative*, *one-to-one communication*, and *broadcast*. Thus it runs in constant time. Step 3 consists of constant number of following basic communication operations: *find-max-representative*, *multiple-multicast*, *one-to-one communication* and *broadcast*. A maximum of 3 local comparisons is needed for finding the rank at the end of this step. Thus step 3 also runs in constant time. Step 4 consists of a simple local addition and a *one-to-one communication*, which runs in constant time. Step 5 consists of constant number of *one-to-one communication* and similar communications and computations as Step 3. Thus Step 5 runs in constant time. Now we conclude that the time complexity of each stage is $O(1)$. As the algorithm has $3 \log n$ stages, we obtain the following theorem:

Theorem 5 *n elements can be sorted in $O(\log n)$ time on an n-processor LARPBS.*

Chapter 5

Parallel Sorting Algorithms on Two Dimensional LARPBS

5.1 Introduction

By applying algorithm 4.4.1 *Pipelined_Merge_Sort* to higher dimensional LARPBS's, such as a 2-dimensional and a 3-dimensional LARPBS, we can obtain work-time optimal sorting algorithms on 2D and 3D LARPBS's.

Table 5.1 compares multi-dimensional parallel sorting algorithms implemented in different models with the algorithms that will be presented in this chapter on multi-dimensional LARPBS.

In this and the next chapter, we present the following results on efficient sorting algorithms for two-dimensional and three-dimensional LARPBS's:

1. A constant-time sorting algorithm that sorts n elements on an $n \times n$ two-dimensional LARPBS.

2. An $O(\log n)$ time basic column sort algorithm that sorts $n\sqrt{n}$ elements on a $n \times \sqrt{n}$ two dimensional LARPBS.

3. An $O(\log n)$ time two-way merge sort algorithm that sorts $2n\sqrt{n}$ elements on a $n \times 2\sqrt{n}$ two dimensional LARPBS.

4. An $O(\log n)$ time two-way merge sort algorithm that sorts $2n\sqrt{n}$ elements on a $n \times \sqrt{n}$ two dimensional LARPBS.

5. An $O(\log n)$ time multi-way merge algorithm that merges two sorted array with n elements on an $n \times n$ two dimensional LARPBS.

6. An optimal multi-way merge sort algorithm that sorts n^2 elements in $O(\log n)$ time on an $n \times n$ two dimensional LARPBS.

Table 5.1: Comparison of different parallel sorting algorithms on multi-dimensional arrays.

Algorithm	Time Complexity	Processor Array
2D Shearsort[30]	$O(n \log n)$	$O(n \times n)$
2D 8-phase Block Sort Algorithm[30]	$O(n)$	$O(n \times n)$
Sorting on AROB [69]	$O(1)$	$n^{\varepsilon} \times n$, where $\varepsilon > 0$
Algorithm 5.2.1 *2D_Constant_Time_Sort* [30]	$O(1)$	$O(n \times n)$
Algorithm 5.4.3 *Basic_Columnsort*	$O(\log n)$	$O(n \times \sqrt{n})$
Algorithm 5.5.1 *Two_Way_Merge_Columnsort*	$O(\log n)$	$O(n \times 2\sqrt{n})$
Algorithm 5.5.2 *Two_Way_Merge_Columnsort1*	$O(\log n)$	$O(n \times \sqrt{n})$
Algorithm 5.8.1 *Generalized_Columnsort*	$O(\log n)$	$O(n \times n)$
Algorithm 5.7.1 *Multiway_Merge_Sort*	$O(\log n)$	$O(n \times n)$
Algorithm 6.2.1 *5-Phase_Sort_on_3D_Array*[30]	$O(n)$	$O(n \times n \times n)$
Algorithm 6.1.1 *3D_Generalized_Columnsort*	$O(\log n)$	$O(n \times n \times n)$
Algorithm 6.2.2 *5-Phase_Sort_on_3D_LARPBS*	$O(\log n)$	$O(n \times n \times n)$

7. An optimal generalized Columnsort algorithm that sorts n^2 elements in $O(\log n)$ time on an $n \times n$ two dimensional LARPBS.

8. An optimal generalized Columnsort algorithm that sorts n^3 elements in $O(\log n)$ time on an $n \times n \times n$ three dimensional LARPBS.

9. An optimal 5-phase sorting algorithm that sorts n^3 elements in $O(\log n)$ time on an $n \times n \times n$ three dimensional LARPBS

5.2 A Constant-Time Sorting Algorithm on 2D LARPBS

We first show how to sort n elements on an n^2-processor 2D LARPBS in $O(1)$ time.

Algorithm 5.2.1 2D_Constant_Time_Sort
Input: *a set S of n elements to be sorted. Assume that the n elements are initially distributed to the n processors on the first row, one per processor. Without lose of generality, we assume that the n elements are distinct.*
Output: *a sorted array of n elements S' stored in the first row.*
Begin

1. *Broadcast elements on column buses* : processor $P_{1,j}$ broadcasts its element e_j to processor $P_{i,j}$, where $1 \leq i \leq n$, so that all processors $P_{i,j}$ in column j have a copy of the j^{th} element e_j saved as $c_{i,j}$.

2. *Diagonal processors broadcast elements on row buses* : each diagonal processor $P_{i,i}$, where $1 \leq i \leq n$, broadcasts the element $c_{i,i}$ to every processor in the i^{th} row using only row buses. After this multiple row *broadcast* communication operation, each processor $P_{i,j}$ saves the received element as $r_{i,j}$.

3. *Compare and set results* : each processor $P_{i,j}$ compares $c_{i,j}$ with $r_{i,j}$. If $c_{i,j} \leq r_{i,j}$, it sets $result = 1$; otherwise, sets $result = 0$.

4. *Sum up 1s* : perform a one-dimensional *binary-prefix-sums* operation on each row simultaneously for the value of $result$. Each processor $P_{i,j}$, where $1 \leq i, j \leq n$, stores the binary prefix sums to $b_{i,j}$.

5. *Route element to correct location* : the last processor on each row, processor $P_{i,n}$, where $1 \leq i \leq n$, sends a dummy message to processor $P_{i,k}$, where $k = b_{i,n}$. This is a multiple *one-to-one* row communication operation. After receiving the dummy message, processor $P_{i,k}$ sends its $r_{i,k}$ to processor $P_{1,k}$. After this multiple *one-to-one* column communication operation, all elements are sorted and are stored in the first row in increasing order.

End

Each step in the above algorithm runs in constant time. Thus we have the following theorem:

Theorem 6 *There is a constant time sorting algorithm on a 2D LARPBS that sorts n elements using n^2 processors.*

Example 5.2.1 *2D_Constant_Time_Sort: sort 5 elements.*

$$\begin{bmatrix} 7 & 2 & 9 & 16 & 5 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \Longrightarrow \begin{bmatrix} 7 & 2 & 9 & 16 & 5 \\ 7 & 2 & 9 & 16 & 5 \\ 7 & 2 & 9 & 16 & 5 \\ 7 & 2 & 9 & 16 & 5 \\ 7 & 2 & 9 & 16 & 5 \end{bmatrix}$$

Input $\qquad$ *step* 1

$$\begin{bmatrix} 7, \boxed{\mathbf{r{=}7}} & 2, 7 & 9, 7 & 16, 7 & 5, 7 \\ 7, 2 & 2, \boxed{\mathbf{r{=}2}} & 9, 2 & 16, 2 & 5, 2 \\ 7, 9 & 2, 9 & 9, \boxed{\mathbf{r{=}9}} & 16, 9 & 5, 9 \\ 7, 16 & 2, 16 & 9, 16 & 16, \boxed{\mathbf{r{=}16}} & 5, 16 \\ 7, 5 & 2, 5 & 9, 5 & 16, 5 & 5, \boxed{\mathbf{r{=}5}} \end{bmatrix} \Longrightarrow \begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

step 2 $\qquad$ *step* 3

$$\begin{bmatrix} 0 & 0 & 0 & 0 & b=3, r=7 \\ 0 & 0 & 0 & 0 & b=1, r=2 \\ 0 & 0 & 0 & 0 & b=4, r=9 \\ 0 & 0 & 0 & 0 & b=5, r=16 \\ 0 & 0 & 0 & 0 & b=2, r=5 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 7 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9 & 0 \\ 0 & 0 & 0 & 0 & 16 \\ 0 & 2 & 0 & 0 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 2 & 5 & 7 & 9 & 16 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

step 4 — *step* 5 *row movement* — *step* 5 : *column movement*

5.3 The Seven-Phase Columnsort Algorithm

Our $O(\log n)$ sorting algorithm on the LARPBS using $O(n)$ processors can be used to obtain an $O(\log n)$ column sort algorithm for a 2D LARPBS. There are two well known versions of Columnsort [29] [30] for sorting an $r \times s$ array in column-major order: one has eight phases and the other has seven. The 8-phase version [29] has a more restrictive assumption: it requires $r \geq 2(s-1)^2$. The 7-phase version [30] requires $r \geq s^2$. Thus the 7-phase version has a larger range of applicability. Olariu, Pinotti, and Zheng proposed an extended Columnsort algorithm in [49] that enlarges the applicability of the 8-phase Columnsort algorithm from $r \geq 2(s-1)^2$ to $r \geq s(s-1)$ by adding an additional sorting step at the end of the algorithm.

In this section, we will develop a 7-phase Columnsort algorithm that runs in $O(\log n)$ time on a 2D LARPBS by applying our $O(\log n)$ sorting algorithm presented in previous sections.

We start by presenting the original idea of the 7-phase Columnsort algorithm.

Algorithm 5.3.1 7-phase_Columnsort
Input*: rs elements distributed to an $r \times s$ processor array, where $r \geq s^2$, one element per processor.*
Output*: a stored $r \times s$ array in column major order, one element per processor.*
Begin

- **Phase 1**: sort elements within each column in ascending order.
- **Phase 2**: transpose matrix by picking up the elements in column-major order and setting them down in row major order, preserving the $r \times s$ shape.
- **Phase 3**: sort elements within each column again in ascending order.
- **Phase 4**: reverse transpose the matrix by picking up the elements in row-major order and setting them down in column major order, again preserving the $r \times s$ shape.
- **Phase 5**: sort elements within each column, but this time adjacent columns are sorted in reverse order.

- **Phase 6**: apply two steps of odd-even transposition sort to each row – in the first step, we compare elements 1 and 2, 3 and 4, etc., in each row; in the second step, we compare elements 2 and 3, 4 and 5, etc., in each row.

- **Phase 7**: apply the regular sort again in each column, i.e., sort each column in ascending order.

End

Theorem 7 *The 7-step Columnsort algorithm correctly sorts rs elements of an $r \times s$ array in column-major order.*

We need to use following well-known 0-1 sorting lemma given in [30] for the proof of the above 7-phase column sort algorithm.

Lemma 5.3.1 The 0-1 Sorting Lemma. *If an oblivious comparison-exchange algorithm sorts all input sets consisting solely of 0s and 1s, then it sorts all input sets with arbitrary values.*

Since the Columnsort algorithm is an oblivious comparison-exchange algorithm, we can use the 0-1 sorting Lemma to prove its correctness. In other words, we only need to check that it correctly sorts any set of 0s and 1s. We now give the proof by illustrating the effects of each phase on the 0-1 input in the Columnsort algorithm.

- **phase 1**: after sorting each column, 0s and 1s in each column are grouped into two separate sets (See Figure 5.1 (a)).

- **phase 2**: the operations performed in this phase can be viewed in the following way – we use a window of length s to extract the elements for each row while moving the window from left most column to right most column. Each time we move the window by s elements. Within a window we can see one of the following three possibilities: clean 0s, clean 1s, and dirty (with 0s and 1s). This is because in each column, 0s and 1s only meet at one point after phase 1. Thus we will see at most two dirty rows when we move in one column: one is the window that covers the point where 0s and 1s meet; the other is the window that covers the boundary of two columns. Thus after this phase, there are at most $2s - 1$ dirty rows. See Figure 5.1 (b).

- **phase 3**: as we sort each column in this phase. At the end of this phase, all dirty rows are moved together between clean 0 rows and clean 1 rows. For any two dirty rows, one of the following possibilities applies: more 0s, more 1s, or equal 0s and 1s. Thus after sorting each column, number of dirty rows will be reduced by one half. In other words, after this phase, number of dirty rows becomes $s - 1/2$. See Figure 5.1 (c).

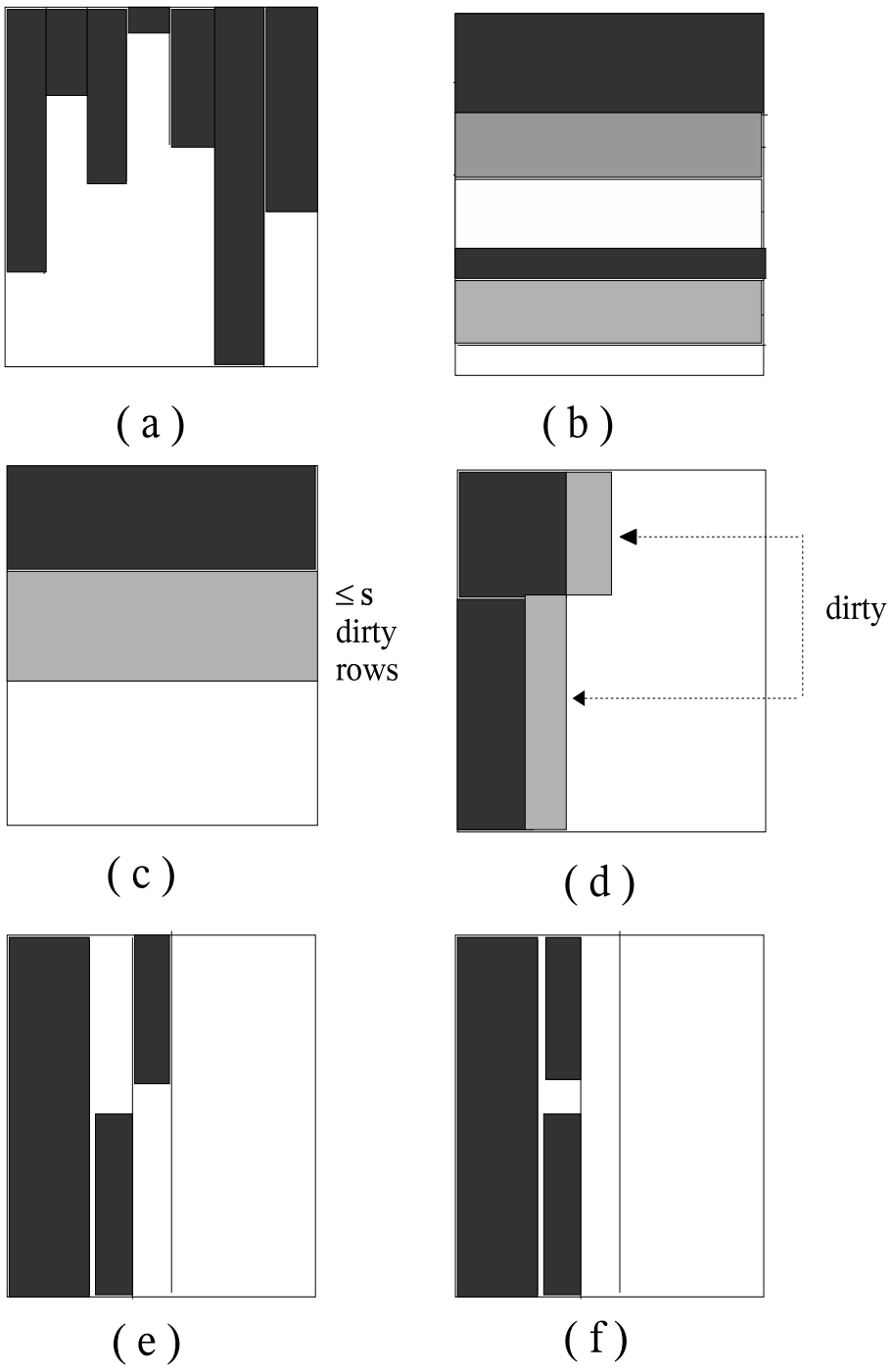


Figure 5.1: 7-step Columnsort

- **phase 4**: after the reverse transpose, all dirty rows are distributed into a number of consecutive columns. Number of dirty columns resulted from this transpose is at most $\lceil (s(s-1/2))/r \rceil \leq \lceil (s^2 - s/2)/s^2 \rceil = \lceil 1 - (1/2s) \rceil = 1$, where $r \geq s^2$ and $s \geq 1$. Due to the fact that the first dirty row may not start exactly at the beginning of a column when permuted to column, there can be at most two dirty columns if the dirty rows does not starts a column. Therefore, the $(s - 1/2)$ dirty rows gives rise to at most 2 dirty columns, with all clean 0 columns on the left and all clean 1 columns on the right. See Figure 5.1 (d).
- **phase 5**: As after phase 4, the two dirty columns are next to each other, the opposite column sorting for two adjacent columns makes sure that when we apply one step of odd-even sort on the two dirty columns in the next phase, at least on clean column will be produced. See Figure 5.1 (e)
- **phase 6**: the two steps of odd-even transposition sort are performed in this phase to guarantee the two dirty columns are compared and exchanged so that at most one dirty column is left. See Figure 5.1 (f).
- **phase 7**: the column sort invoked in this phase will make sure that the only one dirty column will be sorted. Thus after this column sort, the whole matrix will be sorted in column major order.

Example 5.3.1 *7-phase_Columnsort for a* 9×3 *matrix.*

$$\begin{bmatrix} 1 & 19 & 26 \\ 13 & 2 & 3 \\ 5 & 22 & 10 \\ 9 & 12 & 14 \\ 21 & 8 & 20 \\ 11 & 6 & 18 \\ 25 & 23 & 17 \\ 15 & 4 & 24 \\ 27 & 7 & 16 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 \\ 5 & 4 & 10 \\ 9 & 6 & 14 \\ 11 & 7 & 16 \\ 13 & 8 & 17 \\ 15 & 12 & 18 \\ 21 & 19 & 20 \\ 25 & 22 & 24 \\ 27 & 23 & 26 \end{bmatrix} \quad \begin{bmatrix} 1 & 5 & 9 \\ 11 & 13 & 15 \\ 21 & 25 & 27 \\ 2 & 4 & 6 \\ 7 & 8 & 12 \\ 19 & 22 & 23 \\ 3 & 10 & 14 \\ 16 & 17 & 18 \\ 20 & 24 & 26 \end{bmatrix} \quad \begin{bmatrix} 1 & 4 & 6 \\ 2 & 5 & 9 \\ 3 & 8 & 12 \\ 7 & 10 & 14 \\ 11 & 13 & 15 \\ 16 & 17 & 18 \\ 19 & 22 & 23 \\ 20 & 24 & 26 \\ 21 & 25 & 27 \end{bmatrix}$$

Input *step* 1 *step* 2 *step* 3

$$\begin{bmatrix} 1 & 7 & 19 \\ 4 & 10 & 22 \\ 6 & 14 & 23 \\ 2 & 11 & 20 \\ 5 & 13 & 24 \\ 9 & 15 & 26 \\ 3 & 16 & 21 \\ 8 & 17 & 25 \\ 12 & 18 & 27 \end{bmatrix} \quad \begin{bmatrix} 1 & 18 & 19 \\ 2 & 17 & 20 \\ 3 & 16 & 21 \\ 4 & 15 & 22 \\ 5 & 14 & 23 \\ 6 & 13 & 24 \\ 8 & 11 & 25 \\ 9 & 10 & 26 \\ 12 & 7 & 27 \end{bmatrix} \quad \begin{bmatrix} 1 & 18 & 19 \\ 2 & 17 & 20 \\ 3 & 16 & 21 \\ 4 & 15 & 22 \\ 5 & 14 & 23 \\ 6 & 13 & 24 \\ 8 & 11 & 25 \\ 9 & 10 & 26 \\ 7 & 12 & 27 \end{bmatrix} \quad \begin{bmatrix} 1 & 10 & 19 \\ 2 & 11 & 20 \\ 3 & 12 & 21 \\ 4 & 13 & 22 \\ 5 & 14 & 23 \\ 6 & 15 & 24 \\ 7 & 16 & 25 \\ 8 & 17 & 26 \\ 9 & 18 & 27 \end{bmatrix}$$

step 4 *step* 5 *step* 6 *step* 7

5.4 An $O(\log r)$ Column Sort Algorithm on a 2D LARPBS

Key operations needed for implementing the 7-step Columnsort:

1. Sort r elements with r processors in $O(\log r)$ parallel time.
2. $r \times s$ shape matrix transpose.
3. $r \times s$ shape reverse matrix transpose.
4. Odd-even transposition sort.

Operation 1 can be done by applying our $O(\log n)$ time LARPBS sorting algorithm. And it is obvious that operation 4 can be done in constant time. The question is: can we do operations 2 and 3 in constant time?

As in the above Columnsort, the matrix we are sorting has $r \geq s^2$. Matrix transposition cannot be done by simply applying algorithm 2.4.1 *Matrix_Transpose* for 2D LARPBS. Thus we propose the following two special matrix transpose algorithms for $r \times s$ matrices with $r = ks$, where $k > 1$. The basic idea for this extended transpose algorithm is: we split the $r \times s$ matrix into a group of k sub-matrices each of size $s \times s$, then do the matrix transpose for each sub-matrix. Although that basic matrix transpose does not give us what we want, we can get the final result by rearranging the rows. The details of the algorithm are given below.

Algorithm 5.4.1 Extended_Matrix_Transpose
Input: *an $r \times s$ matrix A, where $r = ks$, $k > 1$.*
Output: *the transpose of matrix A, preserving the $r \times s$ shape.*
Begin

- **Step 1.** *Partition into regular shape sub matrices*: processor $P_{ms,j}$, where $1 \leq m \leq k$ and $1 \leq j \leq s$, set its column reconfigurable switches to cross to partition A into k sub-matrices $A_1, A_2, \cdots, A_k$. Each of them is an $s \times s$ matrix.
- **Step 2.** *Transpose sub matrices*: apply algorithm 2.4.1 *Matrix_Transpose* to each sub-matrix A_i
- **Step 3.** *Fuse sub matrices and row shuffle*: each processor $P_{ms,j}$, where $1 \leq m \leq k$ and $1 \leq j \leq s$, sets its column reconfigurable switches back to straight to fuse the k sub-matrices into one $r \times s$ matrix. Then each processor $P_{i,j}$ in matrix A calculate its new row address i': $i' = k((i-1) \bmod s) + \lceil i/s \rceil$. A permutation communication is performed in each column bus based on the new row addresses. Each processor $P_{i,j}$ use the calculated i' as destination address to send its element to processor $P_{i',j}$. This is a *one-to-one* column communication operation and can be done in constant time.

End

Example 5.4.1 *Extended_Matrix_Transpose on an* 8×4 *matrix.*

$$\begin{bmatrix} 1,1 & 1,2 & 1,3 & 1,4 \\ 2,1 & 2,2 & 2,3 & 2,4 \\ 3,1 & 3,2 & 3,3 & 3,4 \\ 4,1 & 4,2 & 4,3 & 4,4 \end{bmatrix} \begin{bmatrix} 5,1 & 5,2 & 5,3 & 5,4 \\ 6,1 & 6,2 & 6,3 & 6,4 \\ 7,1 & 7,2 & 7,3 & 7,4 \\ 8,1 & 8,2 & 8,3 & 8,4 \end{bmatrix} \Rightarrow \begin{bmatrix} 1,1 & 2,1 & 3,1 & 4,1 \\ 1,2 & 2,2 & 3,2 & 4,2 \\ 1,3 & 2,3 & 3,3 & 4,3 \\ 1,4 & 2,4 & 3,4 & 4,4 \end{bmatrix} \begin{bmatrix} 5,1 & 6,1 & 6,1 & 7,1 \\ 5,2 & 6,2 & 6,2 & 7,2 \\ 5,3 & 6,3 & 6,3 & 7,3 \\ 5,4 & 6,4 & 6,4 & 7,4 \end{bmatrix} \Rightarrow \begin{bmatrix} 1,1 & 2,1 & 3,1 & 4,1 \\ 5,1 & 6,1 & 6,1 & 7,1 \\ 1,2 & 2,2 & 3,2 & 4,2 \\ 5,2 & 6,2 & 6,2 & 7,2 \\ 1,3 & 2,3 & 3,3 & 4,3 \\ 5,3 & 6,3 & 6,3 & 7,3 \\ 1,4 & 2,4 & 3,4 & 4,4 \\ 5,4 & 6,4 & 6,4 & 7,4 \end{bmatrix}$$

Step 1 *Step* 2 *Step* 3

Let's call each group of s elements in a column as a super element. And let A' denote the new matrix consists of the super elements. Then it is clear that matrix A' is an $s \times s$ matrix. After applying algorithm 5.4.1 *Extended_Matrix_Transpose*, we should have: the elements in the first super element on the first column are put on the first row of matrix B, the elements in the second super element on the first column are put on the second row of B, etc. In general, elements in the l^{th} super element on the k^{th} column are put on the $((k-1)s+l)^{th}$ row of B. But what we have achieved after step 5.4 is that elements in the l^{th} super element on the k^{th} column are put on the $((l-1)s+k)^{th}$ row of B. In other words, after Step 5.4, we have the correct column index for elements in each row and a wrong row index. Step 5.4 does the required row permutation to send elements to the correct row.

The idea in *Extended_Reverse_Matrix_Transpose* is similar to that in *Extended_Matrix_Transpose*. But the order for the basic operations needs to be changed. We need to do a row permutation first so that all rows are in correct position before we call the regular transpose algorithm. Here are the details.

Algorithm 5.4.2 Extended_Reverse_Matrix_Transpose
Input*: a* $r \times s$ *matrix* A*, where* $r = ks$*,* $k > 1$*.*
Output*: matrix* B*, which is the reverse transpose of matrix* A*, preserving the* $r \times s$ *shape.*
Begin

- **Step 1.** *Row shuffle*: each processor $P_{i,j}$ in matrix A calculate its new row address i': $i' = s((i-1) \bmod k) + \lceil i/k \rceil$. Then a permutation communication is performed in each column bus based on the new row addresses. Each processor $P_{i,j}$ use the calculated i' as destination address to route its element. This is a one-to-one communication operation and can be done in constant time.

- **Step 2.** *Partition into regular shape sub matrices*: processor $P_{ms,j}$, where $1 \leq m \leq k$ and $1 \leq j \leq s$, set its column reconfigurable switches to cross to divide A into k sub-matrices $A_1, A_2, \cdots, A_k$. Each of them is an $s \times s$ matrix, where $1 \leq i \leq s$.

- **Step 3.** *Reverse transpose and fuse into one matrix*: Since reverse matrix transpose is equivalent to matrix transpose for $r \times r$ matrices, apply algorithm 2.4.1 *Matrix_Transpose* to each sub-matrix A_i to reverse transpose A_i. Then processor $P_{ms,j}$, where $1 \leq m \leq k$ and $1 \leq j \leq s$, set its column reconfigurable switches back to straight to fuse the k sub-matrices into one $r \times s$ matrix.

End

Example 5.4.2 *Extended_Reverse_Matrix_Transpose on an* 8×4 *matrix.*

$$\begin{bmatrix} 1,1 & 1,2 & 1,3 & 1,4 \\ 2,1 & 2,2 & 2,3 & 2,4 \\ 3,1 & 3,2 & 3,3 & 3,4 \\ 4,1 & 4,2 & 4,3 & 4,4 \\ 5,1 & 5,2 & 5,3 & 5,4 \\ 6,1 & 6,2 & 6,3 & 6,4 \\ 7,1 & 7,2 & 7,3 & 7,4 \\ 8,1 & 8,2 & 8,3 & 8,4 \end{bmatrix} \Longrightarrow \begin{bmatrix} 1,1 & 1,2 & 1,3 & 1,4 \\ 3,1 & 3,2 & 3,3 & 3,4 \\ 5,1 & 5,2 & 5,3 & 5,4 \\ 7,1 & 7,2 & 7,3 & 7,4 \\ 2,1 & 2,2 & 2,3 & 2,4 \\ 4,1 & 4,2 & 4,3 & 4,4 \\ 6,1 & 6,2 & 6,3 & 6,4 \\ 8,1 & 8,2 & 8,3 & 8,4 \end{bmatrix} \Longrightarrow \begin{bmatrix} 1,1 & 3,1 & 5,1 & 7,1 \\ 1,2 & 3,2 & 5,2 & 7,2 \\ 1,3 & 3,3 & 5,3 & 7,3 \\ 1,4 & 3,4 & 5,4 & 7,4 \\ 2,1 & 4,1 & 6,1 & 8,1 \\ 2,2 & 4,2 & 6,2 & 8,2 \\ 2,3 & 4,3 & 6,3 & 8,3 \\ 2,4 & 4,4 & 6,4 & 8,4 \end{bmatrix}$$

input *step* 1 & 2 *step* 3

odd-even-transposition-sort: This operation is for one-dimensional LARPBS of size m. In an odd step, processor P_{2k-1}, where $1 \leq k \leq m/2$, sends its element x to the processor P_{2k}. Processor P_{2k} then compares its element y with x. If $y < x$, then processor P_{2k} set its element to be x and sends y back to processor P_{2k-1}. An even step is similar to an odd step, but this time even indexed processors send their elements to the odd index processors. In other words, processor P_{2k}, where $1 \leq k \leq m/2 - 1$, sends its element x to processor P_{2k+1}.

Now that we have all the required basic algorithms, we are ready to give our Columnsort algorithm on LARPBS.

Algorithm 5.4.3 Basic_Columnsort
Input*: an $r \times s$ matrix A, where $r = s^2$.*
Output*: matrix A sorted in column major order.*
Begin

- **Phase 1**: sort elements within each column in ascending order by calling algorithm 4.4.1 *Pipelined_Merge_Sort* for each column bus. Only column buses are used in this step.
- **Phase 2**: apply algorithm 5.4.1 *Extended_Matrix_Transpose* on A.
- **Phase 3**: for each column bus, run algorithm 4.4.1 *Pipelined_Merge_Sort* to sort elements again in ascending order. Only column buses are used in this step.
- **Phase 4**: apply algorithm 5.4.2 *Extended_Reverse_Matrix_Transpose* on A.

- **Phase 5**: for each odd column, run algorithm 4.4.1 *Pipelined_Merge_Sort* to sort elements in ascending order; and for each even column, run algorithm 4.4.1 *Pipelined_Merge_Sort* to sort elements in descending order. Only column buses are used in this step.
- **Phase 6**: use only row buses to perform the two steps of odd-even transposition sort on each row. Do *odd-even-transposition-sort* twice: one odd step followed by an even step.
- **Phase 7**: apply algorithm 4.4.1 *Pipelined_Merge_Sort* again to finally sort A in column major order.

End

The operations used in the above basic columnsort algorithm are: *Pipelined_Merge_Sort*, *Extended_Matrix_Transpose*, *Extended_Reverse_Matrix_Transpose*, and two steps of *odd-even-transposition-sort*. All those operations take constant time except *Pipelined_Merge_Sort* which takes $O(\log n)$ time. Thus algorithm 5.4.3 *Basic_Columnsort* runs in $O(\log n)$ time.

Theorem 8 *There is a Columnsort algorithm that sorts $O(n\sqrt{n})$ elements in $O(\log n)$ time on an $n \times \sqrt{n}$-processor 2D LARPBS.*

5.5 The Two-Way Merge Sort Algorithm on a 2D LARPBS

One application of the basic Columnsort algorithm is a two-way merge Columnsort algorithm. The general two-way merge problem is defined as follows:

Definition 5.5.1 *Given two sorted arrays $A = (a_1, a_2, \cdots, a_n)$ and $B = (b_1, b_2, \cdots, b_n)$, the two-way merge problem is to obtain the sorted array $C = (c_1, c_2, \cdots, c_{2n})$ by merging A and B.*

Our two-way merge Columnsort algorithm relies on the following lemma given in [49] with proof.

Lemma 5.5.1 *Assume that $A = (a_1, a_2, \cdots, a_n)$ and $B = (b_1, b_2, \cdots, b_n)$ are two sorted sequences, and $a_{\lceil n/2 \rceil} \leq b_{\lceil n/2 \rceil + 1}$. Let $D = (d_1, d_2, \cdots, d_n)$ be the sorted sequence obtained by merging $b_1, b_2 \cdots, b_{\lceil n/2 \rceil}$ and $a_{\lceil n/2 \rceil + 1}, a_{\lceil n/2 \rceil + 2}, \cdots, a_n$. Then, no element in the sequence $E = (a_1, a_2, \cdots, a_{\lceil n/2 \rceil}, d_1, d_2, \cdots, d_{\lfloor n/2 \rfloor})$ is strictly larger than any element in the sequence $F = (b_{\lceil n/2 \rceil + 1}, b_{\lceil n/2 \rceil + 2}, \cdots, b_n, d_{\lfloor n/2 \rfloor + 1}, d_{\lfloor n/2 \rfloor + 1}, \cdots, d_n)$.*

Proof. We prove this lemma by proving the following two facts:

1. no a_i, where $1 \leq i \leq \lceil \frac{n}{2} \rceil$, in E is strictly larger than any element in F.
2. no d_i, where $1 \leq i \leq \lfloor \frac{n}{2} \rfloor$, in E is strictly larger than any element in F.

Fact 1: Given the assumption that A and B are both sorted and $a_{\lceil n/2 \rceil} \leq b_{\lceil n/2 \rceil + 1}$, it is clear that no a_i, where $1 \leq i \leq \lceil \frac{n}{2} \rceil$, in E is strictly larger than any b_j in F, where $\lfloor \frac{n}{2} \rfloor + 1 \leq j \leq n$. Thus we only needs to prove that a_i is strictly larger than any d_j in F, where $\lfloor \frac{n}{2} \rfloor + 1 \leq j \leq n$. We claim that there is an (a_i in $a_1, a_2, \cdots, a_{\lceil \frac{n}{2} \rceil}$) larger than some element in F. Again the assumption $a_{\lceil n/2 \rceil} \leq b_{\lceil n/2 \rceil + 1}$ guarantees that this element can only be some d_k, where $k \geq \lfloor n/2 \rfloor + 1$, in sorted array D. Observe that all the $\lfloor n/2 \rfloor$ elements in D that comes from A are greater than or equal to a_i, where $1 \leq i \leq \lceil \frac{n}{2} \rceil$, therefore strictly larger than d_k, which implies $k \leq \lfloor n/2 \rfloor$, a contradiction. Thus Fact 1 is proved.

Fact 2: Suppose that there is a d_i in E strictly larger than some element in F. Since D is sorted, this element must be some b_k, where $k \geq \lfloor n/2 \rfloor + 1$, i.e., $c_i > b_k$. Since all elements in D that come from B are less than or equal to b_k, therefore strictly smaller than d_i. It follows that $i \geq \lfloor n/2 \rfloor + 1$. This contradicts with the fact that d_i is in E. Thus our claim cannot be true. This proves Fact 2.

To summarize Fact 1 and Fact 2, no elements in E is strictly larger than any element in F. The lemma is proved.

The following basic operation will be used as a building block in our two-way merge algorithm:

2D_broadcast: processor $P_{i,j}$ who holds the key to be broadcast performs a basic bus broadcast operation on the i^{th} row. Then every processor in the i^{th} row perform a column broadcast operation simultaneously. After this multiple column broadcast operation, every processor in the 2D LARPBS gets the key.

Now we are ready to give the details of our two-way merge columnsort algorithm on an $r \times 2s$ 2D LARPBS.

Algorithm 5.5.1 Two_Way_Merge_Columnsort
Input: *an $r \times 2s$ matrix C, where $r = s^2$.*
Output: *matrix C sorted in column major order.*
Begin

1. *Segment and sort* : Each processor $P_{i,s}$, where $1 \leq i \leq r$, in C set its row reconfigurable switches to cross to segment C into two $r \times s$ matrices: A and B. Apply algorithm 5.4.3 *Basic_Columnsort* on A and B simultaneously.

2. *Compare and exchange* : Processor $P_{1,1}$ calculate the address (α, β) for the median element in matrix A: $\alpha = \lceil sr/2 \rceil \bmod r$. If $\alpha = 0$, then set $\alpha = r$, $\beta = \lceil sr/2 \rceil / r$, and $flag = 0$; otherwise, $\beta = \lceil sr/2 \rceil / r + 1$, $flag = 1$. Then each processor $P_{i,s}$, where $1 \leq i \leq r$, sets its row reconfigurable switches back to straight to fuse A and B into one matrix C. Processor $P_{1,1}$ do a *2D_broadcast* communication operation to broadcast addresses (α, β) and $flag$ to every processor in matrix C. Now processor $P_{\alpha,\beta}$ in matrix A sends its element $a_{\alpha,\beta} = l$ to the median processor of matrix B, i.e., processor $P_{\alpha,\beta+s}$. Processor $P_{\alpha,\beta+s}$ compares the received element l with its own element $b_{\alpha,\beta} = r$. If $l > r$, processor $P_{\alpha,\beta+s}$ broadcasts a signal message to every processor in C using a *2D_broadcast* communication operation. Upon receiving the signal message, processors

in matrices A and B exchange the corresponding elements using a row one-to-one communication operation on the r rows simultaneously. To be precise, processor $P_{i,j}$ exchange element with processor $P_{i,j+s}$, where $1 \leq i \leq r$.

3. *Construct middle matrices* D : If $flag = 1$, we need to move elements around in matrices A and B to form an $r \times s$ shape middle matrix D. Thus processors $P_{i,\beta}$ exchange its element with the elements of $P_{i,\beta+s}$, where $\alpha \leq i \leq r$. This is to move all elements in $[a_{\alpha+1,\beta}, a_{r,s}]$ and $[b_{1,1}, b_{\alpha,\beta}]$ into the middle matrix D. Then processors $P_{i,\beta}$ and $P_{i,\beta+s}$, where $1 \leq i \leq r$, set their row reconfigurable switches to cross to form matrix D. See in Figure 5.2(a).

4. *Sort matrix* D : Apply algorithm 5.4.3 *Basic_Columnsort* to matrix D.

5. *Reconstruct* A *and* B : If $flag = 1$, we need to move the elements around in matrices A and B to recover the elements that belong to matrix B and were moved to matrix A for the convenience of sorting matrix D in the previous step. This can be done in the following two sub-steps:

   (a) *Circular movement on column buses in* D : Processors in matrix D move their elements up $x = r - \beta$ positions, i.e. processor $P_{i,j}$ sends its element to processor $P_{k,j}$, where $k = i - x$, if $i - x > 0$; otherwise, $k = x + i - 1$. This is a multiple *one-to-one* column communication operation. See Figure 5.2 (b).

   (b) *Partial circular movement on row buses in* A *and* B : Processors $P_{i,\beta}$ and $P_{i,\beta+s}$, where $1 \leq i \leq r$, set their row reconfigurable switches to straight to form a single matrix. Then processor $P_{i,j}$ sends its element to processor $P_{i,k}$, where $\alpha < i \leq r$, $\beta \leq j \leq \beta + s$, $k = j - 1$ for $\beta + 1 \leq j \leq \beta + s$ and $k = j + s$ for $j = \beta$. See Figure 5.2 (c).

6. *Partition and Sort* : Each processor $P_{i,s}$, where $1 \leq i \leq r$, set its row reconfigurable switches to cross to segment C again into A and B. Apply algorithm 5.4.3 *Basic_Columnsort* to matrices A and B. Then each processor $P_{i,s}$, where $1 \leq i \leq r$, sets its row reconfigurable switches back to straight to fuse A and B into one matrix C. Now matrix C is sorted in column major order.

End

Since the most time consuming operation in the above algorithm is algorithm 5.4.3 *Basic_Columnsort* which runs in $O(\log n)$ time, we conclude that algorithm 5.5.1 runs in $O(\log n)$ time.

Theorem 9 *The two-way merge sort algorithm on an* $n \times 2\sqrt{n}$ *2D LARPBS sorts* $2n\sqrt{n}$ *elements in* $O(\log n)$ *time.*

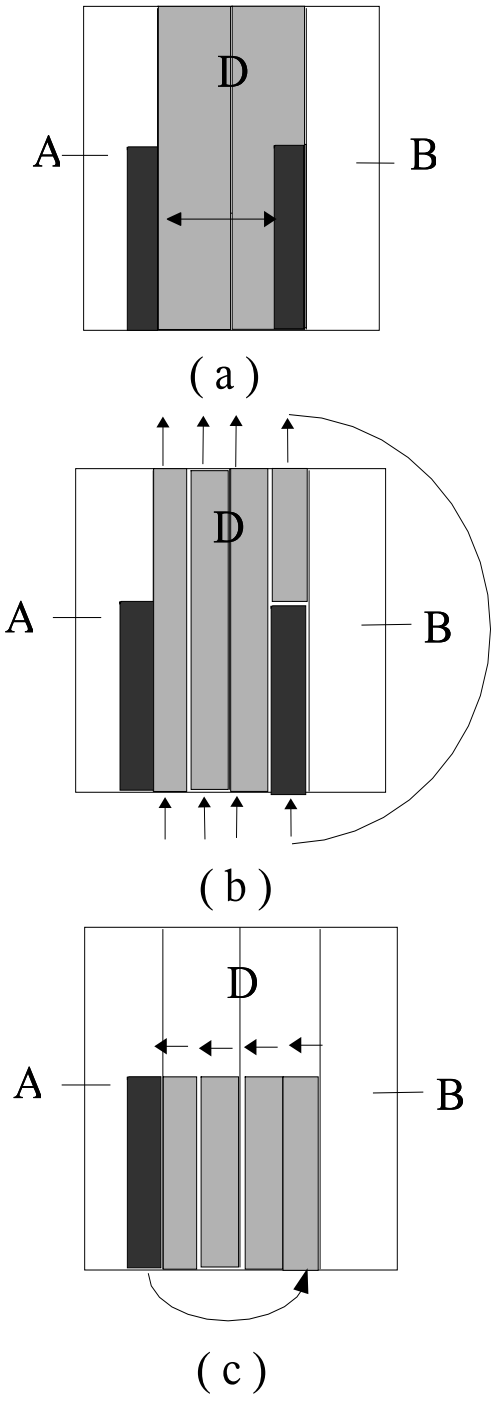


Figure 5.2: (a) Constructing middle matrix D, (b) Circular movements on column buses in matrix D, and (c) Partial circular movements on row buses in A and B

Example 5.5.1 *Two_Way_Merge_Columnsort for* 9×6 *matrix.*

$$\begin{bmatrix} 13 & 23 & 42 & 40 & 51 & 11 \\ 44 & 53 & 14 & 54 & 19 & 7 \\ 5 & 36 & 6 & 35 & 22 & 9 \\ 49 & 20 & 30 & 21 & 37 & 28 \\ 15 & 46 & 52 & 12 & 18 & 1 \\ 32 & 38 & 43 & 3 & 26 & 41 \\ 17 & 45 & 34 & 50 & 29 & 16 \\ 33 & 27 & 47 & 31 & 10 & 8 \\ 25 & 24 & 4 & 2 & 48 & 39 \end{bmatrix}$$

Input

$$\Longrightarrow \quad \begin{bmatrix} 4 & 24 & 42 \\ 5 & 25 & 43 \\ 6 & 27 & 44 \\ 13 & 30 & 45 \\ 14 & \boxed{\mathbf{32}} & 46 \\ 15 & 33 & 47 \\ 17 & 34 & 49 \\ 20 & 36 & 52 \\ 23 & 38 & 53 \end{bmatrix} \begin{bmatrix} 1 & 16 & 35 \\ 2 & 18 & 37 \\ 3 & 19 & 39 \\ 7 & 21 & 40 \\ 8 & \boxed{\mathbf{22}} & 41 \\ 9 & 26 & 48 \\ 10 & 28 & 50 \\ 11 & 29 & 51 \\ 12 & 31 & 54 \end{bmatrix}$$

Step 1

$$\begin{bmatrix} 1 & 16 & 35 \\ 2 & 18 & 37 \\ 3 & 19 & 39 \\ 7 & 21 & 40 \\ 8 & \boxed{\mathbf{22}} & 41 \\ 9 & 26 & 48 \\ 10 & 28 & 50 \\ 11 & 29 & 51 \\ 12 & 31 & 54 \end{bmatrix} \begin{bmatrix} 4 & 24 & 42 \\ 5 & 25 & 43 \\ 6 & 27 & 44 \\ 13 & 30 & 45 \\ 14 & \boxed{\mathbf{32}} & 46 \\ 15 & 33 & 47 \\ 17 & 34 & 49 \\ 20 & 36 & 52 \\ 23 & 38 & 53 \end{bmatrix}$$

Step 2

$$\Longrightarrow \quad \begin{bmatrix} 1 & 16 \\ 2 & 18 \\ 3 & 19 \\ 7 & 21 \\ 8 & 22 \\ 9 & \boxed{\mathbf{33}} \\ 10 & \boxed{\mathbf{34}} \\ 11 & \boxed{\mathbf{36}} \\ 12 & \boxed{\mathbf{38}} \end{bmatrix} \begin{bmatrix} 35 & 4 & 24 \\ 37 & 5 & 25 \\ 39 & 6 & 27 \\ 40 & 13 & 30 \\ 41 & 14 & 32 \\ 48 & 15 & \boxed{\mathbf{26}} \\ 50 & 17 & \boxed{\mathbf{28}} \\ 51 & 20 & \boxed{\mathbf{29}} \\ 54 & 23 & \boxed{\mathbf{31}} \end{bmatrix} \begin{bmatrix} 42 \\ 43 \\ 44 \\ 45 \\ 46 \\ 47 \\ 49 \\ 52 \\ 53 \end{bmatrix}$$

Step 3

$$\begin{bmatrix} 1 & 16 \\ 2 & 18 \\ 3 & 19 \\ 7 & 21 \\ 8 & 22 \\ 9 & \boxed{\mathbf{33}} \\ 10 & \boxed{\mathbf{34}} \\ 11 & \boxed{\mathbf{36}} \\ 12 & \boxed{\mathbf{38}} \end{bmatrix} \begin{bmatrix} \boxed{\mathbf{4}} & \boxed{\mathbf{24}} & \boxed{\mathbf{35}} \\ \boxed{\mathbf{5}} & \boxed{\mathbf{25}} & \boxed{\mathbf{37}} \\ \boxed{\mathbf{6}} & \boxed{\mathbf{26}} & \boxed{\mathbf{39}} \\ \boxed{\mathbf{13}} & \boxed{\mathbf{27}} & \boxed{\mathbf{40}} \\ \boxed{\mathbf{14}} & \boxed{\mathbf{28}} & \boxed{\mathbf{41}} \\ \boxed{\mathbf{15}} & \boxed{\mathbf{29}} & \boxed{\mathbf{48}} \\ \boxed{\mathbf{17}} & \boxed{\mathbf{30}} & \boxed{\mathbf{50}} \\ \boxed{\mathbf{20}} & \boxed{\mathbf{31}} & \boxed{\mathbf{51}} \\ \boxed{\mathbf{23}} & \boxed{\mathbf{32}} & \boxed{\mathbf{54}} \end{bmatrix} \begin{bmatrix} 42 \\ 43 \\ 44 \\ 45 \\ 46 \\ 47 \\ 49 \\ 52 \\ 53 \end{bmatrix}$$

Step 4

$$\begin{bmatrix} 1 & 16 \\ 2 & 18 \\ 3 & 19 \\ 7 & 21 \\ 8 & 22 \\ 9 & \boxed{\mathbf{33}} \\ 10 & \boxed{\mathbf{34}} \\ 11 & \boxed{\mathbf{36}} \\ 12 & \boxed{\mathbf{38}} \end{bmatrix} \begin{bmatrix} 14 & 28 & 41 \\ 15 & 29 & 48 \\ 17 & 30 & 50 \\ 20 & 31 & 51 \\ 23 & 32 & 54 \\ \boxed{\mathbf{4}} & \boxed{\mathbf{24}} & \boxed{\mathbf{35}} \\ \boxed{\mathbf{5}} & \boxed{\mathbf{25}} & \boxed{\mathbf{37}} \\ \boxed{\mathbf{6}} & \boxed{\mathbf{26}} & \boxed{\mathbf{39}} \\ \boxed{\mathbf{13}} & \boxed{\mathbf{27}} & \boxed{\mathbf{40}} \end{bmatrix} \begin{bmatrix} 42 \\ 43 \\ 44 \\ 45 \\ 46 \\ 47 \\ 49 \\ 52 \\ 53 \end{bmatrix}$$

Step 5 : *substep*1

$$\begin{bmatrix} 1 & 16 & 14 & 28 & 41 & 42 \\ 2 & 18 & 15 & 29 & 48 & 43 \\ 3 & 19 & 17 & 30 & 50 & 44 \\ 7 & 21 & 20 & 31 & 51 & 45 \\ 8 & 22 & 23 & 32 & 54 & 46 \\ 9 & \boxed{\mathbf{4}} & \boxed{\mathbf{24}} & \boxed{\mathbf{35}} & \boxed{\mathbf{33}} & 47 \\ 10 & \boxed{\mathbf{5}} & \boxed{\mathbf{25}} & \boxed{\mathbf{37}} & \boxed{\mathbf{34}} & 49 \\ 11 & \boxed{\mathbf{6}} & \boxed{\mathbf{26}} & \boxed{\mathbf{39}} & \boxed{\mathbf{36}} & 52 \\ 12 & \boxed{\mathbf{13}} & \boxed{\mathbf{27}} & \boxed{\mathbf{40}} & \boxed{\mathbf{38}} & 53 \end{bmatrix} \Longrightarrow \begin{bmatrix} 1 & 10 & 19 \\ 2 & 11 & 20 \\ 3 & 12 & 21 \\ 4 & 13 & 22 \\ 5 & 14 & 23 \\ 6 & 15 & 24 \\ 7 & 16 & 25 \\ 8 & 17 & 26 \\ 9 & 18 & 27 \end{bmatrix} \begin{bmatrix} 28 & 37 & 46 \\ 29 & 38 & 47 \\ 30 & 39 & 48 \\ 31 & 40 & 49 \\ 32 & 41 & 50 \\ 33 & 42 & 51 \\ 34 & 43 & 52 \\ 35 & 44 & 53 \\ 36 & 45 & 54 \end{bmatrix}$$

Step 5 : *substep*2 *Step* 6

If we take a second thought on the above two-way merge algorithm, we will find that we don't need $2n\sqrt{n}$ processors. After some modification to the algorithm, we can use only $n\sqrt{n}$ processors to sort $2n\sqrt{n}$ elements. Here is a modified version of our two-way merge sort algorithm.

Algorithm 5.5.2 Two_Way_Merge_Columnsort1
Input*: two $r \times s$ matrices A and B, where $r = s^2$. Elements for both matrices are distributed to an $r \times s$ LARPBS, one element per processor for each matrix, i.e. processor $P_{i,j}$ holds element $a_{i,j}$ from A and element $b_{i,j}$ from B.*
Output*: $2n\sqrt{n}$ sorted elements stored in A and B in following column major order: $(a_{1,1}, a_{2,1}, \cdots, a_{n,1}, \cdots, a_{1,\sqrt{n}}, a_{2,\sqrt{n}}, \cdots, a_{n,\sqrt{n}}, b_{1,1}, b_{2,1}, \cdots, b_{n,1}, \cdots, b_{1,\sqrt{n}}, b_{2,\sqrt{n}}, \cdots, b_{n,\sqrt{n}})$.*
Begin

1. *Double sort* : Apply algorithm 5.4.3 *Basic_Columnsort* twice – once for sorting matrix A and once for sorting matrix B.

2. *Compare and set* : Let (α, β) be the address for the median processor, where $\alpha = \lceil sr/2 \rceil \bmod r$. If $\alpha = 0$, then set $\alpha = r$, $\beta = \lceil sr/2 \rceil / r$; otherwise, $\beta = \lceil sr/2 \rceil / r + 1$. Then processor $P_{\alpha,\beta}$ compares its $a_{\alpha,\beta}$ and $b_{\alpha,\beta}$. If $a_{\alpha,\beta} > b_{\alpha,\beta}$, each processor $P_{i,j}$ exchanges its $a_{i,j}$ value with its $b_{i,j}$ value, where $1 \leq i \leq n$, and $1 \leq j \leq \sqrt{n}$.

3. *Construct matrix of active elements* : Processors $P_{i,j}$ in the first half of the matrix, where $1 \leq i \leq \alpha$ and $j = \beta$, or $1 \leq i \leq n$ and $j < \beta$, set $b_{i,j}$ as active element. Processors $P_{k,l}$ in the second half of the matrix, where $\alpha < k \leq r$ and $l = \beta$, or $1 \leq k \leq n$ and $l > \beta$, set $a_{k,l}$ as active element.

4. *Sort active elements* : Apply algorithm 5.4.3 *Basic_Columnsort* to A sort all active elements.

5. *Reconstruct A and B* : Processor $P_{i,j}$ exchanges its $b_{i,j}$ value with processor $P_{i,j'}$'s $a_{i,j'}$ value, where $1 \leq i \leq n$, $1 \leq j \leq \beta$ if $\alpha = 0$, otherwise $1 \leq j < \beta$, and $j' = s - j + 1$. This can be done with a *one-to-one* row communication operation. If $\alpha \neq 0$, do the following *one-to-one* column communication operation on column β: processor $P_{i,\beta}$ exchanges its $b_{i,\beta}$ value with processor $P_{i',\beta}$'s $a_{i',\beta}$ value, where $1 \leq i \leq \lfloor sr/2 \rfloor \bmod r$, $i' = r - i + 1$.

6. *Sort new A and B* : Apply algorithm 5.4.3 *Basic_Columnsort* twice again to sort matrix A and B. At the end of this step, all A-elements are sorted in column major order, all B-elements are sorted in column major order. And all B-elements are strictly larger than all A-elements. Thus the $2n\sqrt{n}$ elements are sorted.

End

Same as algorithm 5.5.1 *Two_Way_Merge_Columnsort*, the most time consuming operation in the above algorithm is algorithm 5.4.3 *Basic_Columnsort* which runs in $O(\log n)$ time, thus we have the following theorem.

Theorem 10 *There is a two-way merge sort algorithm on an $n \times \sqrt{n}$ 2D LARPBS that sorts $2n\sqrt{n}$ elements in $O(\log n)$ time.*

Example 5.5.2 *Two_Way_Merge_Columnsort1 for a 9×3 matrix. Elements in each matrix location (i, j) is always shown as $(a_{i,j}, b_{i,j})$ pair.*

$$\begin{bmatrix} 13,40 & 23,51 & 42,11 \\ 44,54 & 53,19 & 14,7 \\ 5,35 & 36,22 & 6,9 \\ 49,21 & 20,37 & 30,28 \\ 15,12 & 46,18 & 52,1 \\ 32,3 & 38,26 & 43,41 \\ 17,50 & 45,29 & 34,16 \\ 33,31 & 27,10 & 47,8 \\ 25,2 & 24,48 & 4,39 \end{bmatrix} \Longrightarrow \begin{bmatrix} 4,1 & 24,16 & 42,35 \\ 5,2 & 25,18 & 43,37 \\ 6,3 & 27,19 & 44,39 \\ 13,7 & 30,21 & 45,40 \\ 14,8 & \boxed{\mathbf{32}},\boxed{\mathbf{22}} & 46,41 \\ 15,9 & 33,26 & 47,48 \\ 17,10 & 34,28 & 49,50 \\ 20,11 & 36,29 & 52,51 \\ 23,12 & 38,31 & 53,54 \end{bmatrix}$$

Input *Step* 1

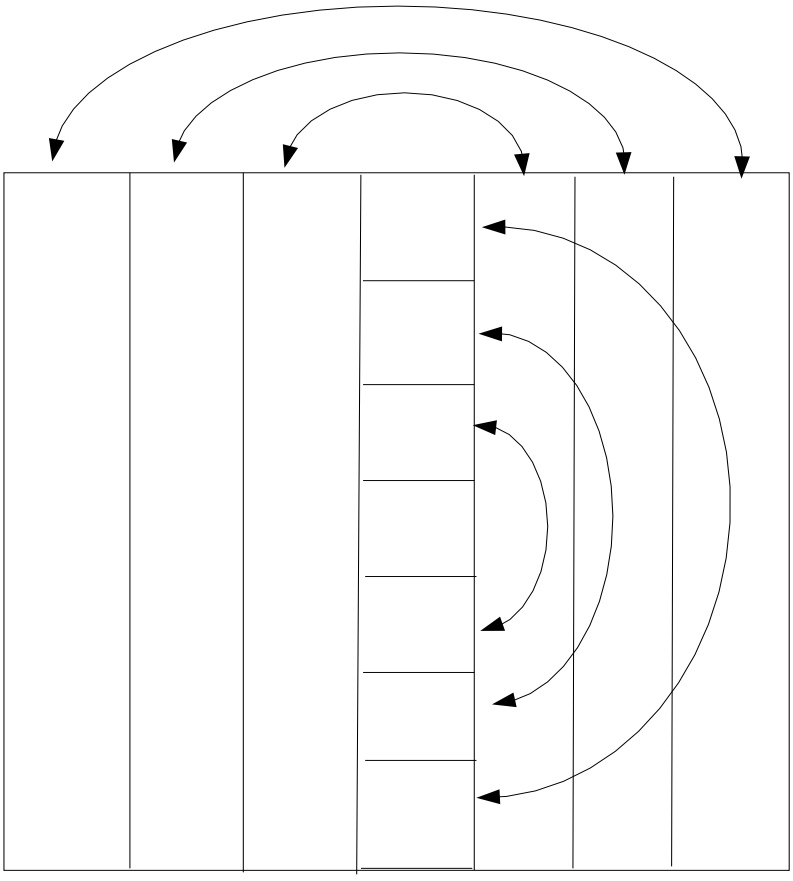


Figure 5.3: Two-way columnsort: reconstructing A and B

$$\begin{bmatrix} 1,4 & 16,24 & 35,42 \\ 2,5 & 18,25 & 37,43 \\ 3,6 & 19,27 & 39,44 \\ 7,13 & 21,30 & 40,45 \\ 8,14 & \boxed{\mathbf{22}},\boxed{\mathbf{32}} & 41,46 \\ 9,15 & 26,33 & 48,47 \\ 10,17 & 28,34 & 50,49 \\ 11,20 & 29,36 & 51,52 \\ 12,23 & 31,38 & 54,53 \end{bmatrix}$$

Step 2

$$\Longrightarrow \begin{bmatrix} 1,\boxed{\mathbf{4}} & 16,\boxed{\mathbf{24}} & \boxed{\mathbf{35}},42 \\ 2,\boxed{\mathbf{5}} & 18,\boxed{\mathbf{25}} & \boxed{\mathbf{37}},43 \\ 3,\boxed{\mathbf{6}} & 19,\boxed{\mathbf{27}} & \boxed{\mathbf{39}},44 \\ 7,\boxed{\mathbf{13}} & 21,\boxed{\mathbf{30}} & \boxed{\mathbf{40}},45 \\ 8,\boxed{\mathbf{14}} & 22,\boxed{\mathbf{32}} & \boxed{\mathbf{41}},46 \\ 9,\boxed{\mathbf{15}} & \boxed{\mathbf{26}},33 & \boxed{\mathbf{48}},47 \\ 10,\boxed{\mathbf{17}} & \boxed{\mathbf{28}},34 & \boxed{\mathbf{50}},49 \\ 11,\boxed{\mathbf{20}} & \boxed{\mathbf{29}},36 & \boxed{\mathbf{51}},52 \\ 12,\boxed{\mathbf{23}} & \boxed{\mathbf{31}},38 & \boxed{\mathbf{54}},53 \end{bmatrix}$$

Step 3

$$\begin{bmatrix} 1,\boxed{\mathbf{4}} & 16,\boxed{\mathbf{24}} & \boxed{\mathbf{35}},42 \\ 2,\boxed{\mathbf{5}} & 18,\boxed{\mathbf{25}} & \boxed{\mathbf{37}},43 \\ 3,\boxed{\mathbf{6}} & 19,\boxed{\mathbf{26}} & \boxed{\mathbf{39}},44 \\ 7,\boxed{\mathbf{13}} & 21,\boxed{\mathbf{27}} & \boxed{\mathbf{40}},45 \\ 8,\boxed{\mathbf{14}} & 22,\boxed{\mathbf{28}} & \boxed{\mathbf{41}},46 \\ 9,\boxed{\mathbf{15}} & \boxed{\mathbf{29}},33 & \boxed{\mathbf{48}},47 \\ 10,\boxed{\mathbf{17}} & \boxed{\mathbf{30}},34 & \boxed{\mathbf{50}},49 \\ 11,\boxed{\mathbf{20}} & \boxed{\mathbf{31}},36 & \boxed{\mathbf{51}},52 \\ 12,\boxed{\mathbf{23}} & \boxed{\mathbf{32}},38 & \boxed{\mathbf{54}},53 \end{bmatrix}$$

Step 4

$$\Longrightarrow \begin{bmatrix} 1,\boxed{\mathbf{35}} & 16,\boxed{\mathbf{32}} & \boxed{\mathbf{4}},42 \\ 2,\boxed{\mathbf{37}} & 18,\boxed{\mathbf{31}} & \boxed{\mathbf{5}},43 \\ 3,\boxed{\mathbf{39}} & 19,\boxed{\mathbf{30}} & \boxed{\mathbf{6}},44 \\ 7,\boxed{\mathbf{40}} & 21,\boxed{\mathbf{29}} & \boxed{\mathbf{13}},45 \\ 8,\boxed{\mathbf{41}} & 22,\boxed{\mathbf{28}} & \boxed{\mathbf{14}},46 \\ 9,\boxed{\mathbf{48}} & \boxed{\mathbf{27}},33 & \boxed{\mathbf{15}},47 \\ 10,\boxed{\mathbf{50}} & \boxed{\mathbf{26}},34 & \boxed{\mathbf{17}},49 \\ 11,\boxed{\mathbf{51}} & \boxed{\mathbf{25}},36 & \boxed{\mathbf{20}},52 \\ 12,\boxed{\mathbf{54}} & \boxed{\mathbf{24}},38 & \boxed{\mathbf{23}},53 \end{bmatrix}$$

Step 5

$$\begin{bmatrix} 1,\boxed{\mathbf{28}} & 10,\boxed{\mathbf{37}} & 19,\boxed{\mathbf{46}} \\ 2,\boxed{\mathbf{29}} & 11,\boxed{\mathbf{38}} & 20,\boxed{\mathbf{47}} \\ 3,\boxed{\mathbf{30}} & 12,\boxed{\mathbf{39}} & 21,\boxed{\mathbf{48}} \\ 4,\boxed{\mathbf{31}} & 13,\boxed{\mathbf{40}} & 22,\boxed{\mathbf{49}} \\ 5,\boxed{\mathbf{32}} & 14,\boxed{\mathbf{41}} & 23,\boxed{\mathbf{50}} \\ 6,\boxed{\mathbf{33}} & 15,\boxed{\mathbf{42}} & 24,\boxed{\mathbf{51}} \\ 7,\boxed{\mathbf{34}} & 16,\boxed{\mathbf{43}} & 25,\boxed{\mathbf{52}} \\ 8,\boxed{\mathbf{35}} & 17,\boxed{\mathbf{44}} & 26,\boxed{\mathbf{53}} \\ 9,\boxed{\mathbf{36}} & 18,\boxed{\mathbf{45}} & 27,\boxed{\mathbf{54}} \end{bmatrix}$$

Step 6

This version of two-way merge sort algorithm uses less processors and simpler. What is more important is that it greatly simplifies the algorithm design for the multi-way merge algorithm to be presented in the next section.

5.6 The Multi-Way Merge Algorithm on a 2D LARPBS

The multi-way merge problem we would like to solve in this section is defined as follows:

Definition 5.6.1 Multi-way Merge Problem*:*
Given a collection of sorted arrays $A = \{A_1, A_2, \cdots, A_m\}$*, each of them has size* $n \times \sqrt{n}$*, the multi-way merge problem is to obtain a sequence of sorted elements that contains all the elements in* A*.*

In this section, we are specifically interested in the *Multi-way Merge Problem* where $m = \sqrt{n}$, since for this case we can use algorithms developed in the previous sections to obtain an efficient multi-way merge algorithm. We follow the following basic idea proposed in [49] to develop our algorithm on a 2D LARPBS: evenly take n elements as samples from sorted arrays $A = \{A_1, A_2, \cdots, A_{\sqrt{n}}\}$, where A is an $n \times \sqrt{n}$ array; sort those samples; then partition A into $\sqrt{n}$ arrays $B_1, B_2, \cdots, B_{\sqrt{n}}$, each of them has size at most $2n \times \sqrt{n}$. To be specific, this partition is done by assigning each elements in A to one of the arrays B_i based on a partition scheme stated below in Definition 5.6.7. This partition scheme does not guarantee that elements in each B_i are in order, but it guarantees that no element in B_i is strictly greater than any element in B_{i+1}. Thus based on this partition, what we need to do to get the final results is to sort each B_i simultaneously. In summary, the multi-way sort can be done in following 4 steps:

1. Form sample set $S = \{s_1, s_2, \cdots, s_n\}$ by selecting every m^{th} element in each sorted sequence A_i, where $1 \leq i \leq \sqrt{n}$.
2. Sort sample set S.
3. Partition A into $\sqrt{n}$ arrays: $B_1, B_2, \cdots, B_{\sqrt{n}}$. Each B_i is obtained by calculating regular matrix X_i and special matrix Y_i, and let $B_i = X_i \bigcup Y_i$.
4. Sort each B_i using algorithm 5.5.2 *Two_Way_Merge_Columnsort1*.
5. Coalesce the sorted B_is into an $n \times n$ column major sorted array.

We need the following definitions and lemmas before we can describe our algorithm in details.

Definition 5.6.2 Regular Column
Let $A = \{A_1, A_2, \cdots, A_{\sqrt{n}}\}$ *be an* $n \times n$ *matrix, each* A_i *is an* $n \times \sqrt{n}$ *matrix sorted in column major order. Let* $B = \{B_1, B_2, \cdots, B_{\sqrt{n}}\}$ *be a partition of matrix* A*. Let element* b *be the leader of column* k*, and* r_b *be the rank of* b *in sorted sample set* S*. A leader of a column is defined to be the last element on that column. Column* k *is said to be regular with respect to* B_i *if* $(i-1)\sqrt{n} < r_k \leq i\sqrt{n}$*. All regular columns with respect to* B_i *are assigned to regular matrix* X_i*.*

Definition 5.6.3 Regular Matrix
Regular matrices are defined to be a set of $n \times \sqrt{n}$ *matrices,* $X = \{X_1, X_2, \cdots, X_{\sqrt{n}}\}$. *Regular matrix* X_i *consists of all regular columns with respect to* B_i.

Definition 5.6.4 Special Column
Let $A = \{A_1, A_2, \cdots, A_{\sqrt{n}}\}$ *be an* $n \times n$ *matrix. Assume* $B = \{B_1, B_2, \cdots, B_{\sqrt{n}}\}$ *is a partition of* A *obtained by the partition scheme defined in Definition 5.6.7. Assume each* $n \times \sqrt{n}$ *matrix* A_i, *where* $1 \leq i \leq \sqrt{n}$, *is sorted in column major order. Let* r_j *be the rank of the leader of column* j *in sorted sample set* S. *Then column* j *in matrix* A *is said to be special with respect to* B_t *if* $r_{(j-1)} \leq t\sqrt{n} \leq r_j$. *All special columns with respect to* B_i *are assigned to special matrix* Y_i.

Definition 5.6.5 Special Matrix
Special matrices are defined to be a set of $n \times \sqrt{n}$ *matrices,* $Y = \{Y_1, Y_2, \cdots, Y_{\sqrt{n}}\}$. *A column in each special matrix* Y_i, *is either a special column with respect to* B_i, *or a column with all elements set to* ∞.

Definition 5.6.6 Qualification Rule
Given an element e *and the sorted sample* $S = \{s_1 < s_2 < \cdots < s_n\}$. *Let* $m = \sqrt{n}$. *Then* e *is said to be qualified with respect to* B_i *if one of the following conditions is satisfied:*

1. $s_{(i-1)m} < e \leq s_{im}$ *whenever* $s_{(i-1)m} < s_{im}$
2. $s_{(i-1)m} \leq e \leq s_{im}$ *whenever* $s_{(i-1)m} = s_{im}$

Definition 5.6.7 Partition Scheme
Given $\sqrt{n}$ *matrices* A_i, *where* $1 \leq i \leq \sqrt{n}$, *the partition of* A *into* $\sqrt{n}$ *subsets* B_i, *where* $1 \leq i \leq \sqrt{n}$, *is done in three phrases:*

1. *Associate qualified regular elements on regular columns with their regular matrices: using Qualification Rule defined in Definition 5.6.6 to filter out qualified regular elements and associate them with their regular matrix* X_i.
2. *Associate leftover elements on special columns with their special matrices: for non-qualified regular elements, find each element* e*'s special matrices,* $Y_{j_1}, Y_{j_2}, \cdots, Y_{j_k}$, *where* $j_1 < j_2 < \cdots < j_k$. *Then associate* e *with special matrix* Y_{j_i}, *where* j_i *is the smallest index in* $(j_1 < j_2 < \cdots < j_k)$ *obtained by Qualification Rule defined in Definition 5.6.6.*
3. *Obtain* B_i: *let* $B_i = X_i \bigcup Y_i$

The following lemmas are implicitly given in [49] without proof. Since they are important facts that guarantee the correctness of our multi-way merge algorithm, we now give the description of the lemmas followed by a detailed proof for each of them.

Lemma 5.6.1 *Each sorted sequence* A_i *contains at most one special column with respect to* B_j, *where* $1 \leq i, j \leq \sqrt{n}$.

Proof. Assume that a sorted sequence A_i contains two special columns with respect to B_j, i.e., the g^{th} column A_{i_g} and the k^{th} column A_{i_k} in A_i. We denote the rank for the leader of those two columns as r_g and r_k, respectively. Assume $r_g < r_k$. Since column A_{i_k} is a special column with respect to B_j, we have $r_{(k-1)} \leq j < r_k$. As $r_g < r_k$, we have $r_g \leq r_{(k-1)}$. Since A_i is sorted and $r_{(k-1)} \leq j$, we must have $r_g \leq j$. But according to our assumption, column A_{i_g} is also a special column with respect to B_j. Based on the definition of a special column, we have $r_{(g-1)} \leq j < r_g$; in other words, $r_g > j$. This contradict with $r_g \leq j$. The lemma is proved.

Lemma 5.6.2 *The partition scheme defined in Definition 5.6.7 guarantees every element in A will be associated to exactly one B_i.*

Proof. Based on Definition 5.6.7, once an element is assigned to a regular matrix X_i or a special matrix Y_i, it will not be assigned to any other regular or special matrix. Thus any element will be assigned to at most one B_i.

Assume that an element e is in a regular column with respect to B_i and it does not meet the qualification rule for B_i, then we must have $e \leq s_{(i-1)\sqrt{n}}$. Since each column is sorted, we always have $s_{i\sqrt{n}} \geq e$. Assume $e \in A_k$, s_c is the leader of the column that e belongs to, s_a is the leader of the proceeding column in A_k. Since A_k is sorted in column major order, we have $e \geq s_a$. From $s_{(i-1)\sqrt{n}} \geq e$ and $e \geq s_a$, we obtain $s_{(i-1)\sqrt{n}} \geq s_a$. As $e \in B_i$, we have $s_{(i-1)\sqrt{n}} \leq s_c \leq s_{i\sqrt{n}}$. Thus we obtain $s_a \leq s_{(i-1)\sqrt{n}} \leq s_c$, which means e is in a special column with respect to at least one special matrix $Y_{(i-1)}$. We apply a similar argument if e disqualifies for special matrix $Y_{(i-1)}$ until we find a special matrix for which e qualifies. If e disqualifies for any special matrix until we have $e < s_{\sqrt{n}}$, then e qualifies for special matrix Y_1 as e is always greater than or equal to the smallest element in A. Thus we proved there exists at least one special matrix Y_i for any disqualified regular element such that this element qualifies for the special matrix Y_i. In other words, any element qualifies for at lease one B_i, thus it will be assigned to at least one B_i.

Summarizing the above argument, every element in A will be associated to one and only one B_i. The lemma is proved.

Lemma 5.6.3 *Each X_i and Y_i will be assigned at most $n\sqrt{n}$ qualified elements.*

Proof. Since exactly $\sqrt{n}$ columns are assigned to each X_i, it is clear that X_i will be assigned at most $n\sqrt{n}$ qualified elements. According to Lemma 5.6.1, there will be at most one special column from each A_j assigned to a B_i. Since $1 \leq j \leq \sqrt{n}$, there will be at most $\sqrt{n}$ special columns assigned to a special matrix Y_i. Thus Y_i will be assigned at most $n\sqrt{n}$ qualified elements. The lemma is proved.

The following lemma is given in [49] with proof. We reiterate the proof here for the completeness of this discussion.

Lemma 5.6.4 *Assume A is an $n \times n$ matrix. Matrix A_i, where $1 \leq i \leq \sqrt{n}$, is obtained by partition A into $\sqrt{n}$ sub-matrices, each of size $n \times \sqrt{n}$; and B_i, where $1 \leq i \leq \sqrt{n}$, is*

obtained by collecting qualified regular elements and qualified special elements with respect to B_i. Then all B_i, where $1 \leq i \leq \sqrt{n}$, satisfy following condition: for every i and j, where $1 \leq i < j \leq \sqrt{n}$, no element in B_i is strictly larger than any element in B_j.

Proof.
Based on the qualification rule defined in Definition 5.6.6, all elements that are assigned to B_i are in interval $[s_{(i-1)m}, s_{im}]$; all elements that are assigned to B_{i+1} are in interval $[s_{im}, s_{(i+1)m}]$; etc. In general, all elements that are assigned to B_k are in interval $[s_{(k-1)m}, s_{km}]$ for $1 \leq k \leq \sqrt{n}$. Since we have $s_1 < s_2 < \cdots < s_{\sqrt{n}}$, it is clear that for every i and j, where $1 \leq i < j \leq s$, no element in B_i is strictly larger than any element in B_j.

We develop following multi-way merge algorithm for a 2D LARPBS based on the basic idea of [49], but we use different techniques that apply to the LARPBS model.

Algorithm 5.6.1 Multiway_Merge
Input*: An $n \times n$ matrix A, split into $\sqrt{n}$ sorted sub-matrices $A = \{A_1, A_2, \cdots, A_{\sqrt{n}}\}$, each of them is an $n \times \sqrt{n}$ matrix. Elements in each sub-matrix A_i, where $1 \leq i \leq \sqrt{n}$, are sorted in column major order. But A is not sorted.*
Output*: matrix A sorted in column major order*
Begin

- **Step 1**: Let the last processor of each column in A be the leader of that column. So leader processors are $P_{n,j}$ for $1 \leq j \leq n$. The n^{th} row bus consists of all the leader processors. We call this bus the sample bus, and we use S to denote the sample set, which contains all leaders.

- **Step 2**: Run algorithm 4.4.1 *Pipepined_Merge_Sort* on the sample bus to sort the elements in the leader processors. When an element is being moved in the sample bus while being sorted in the sorting operation, the original column index of an element is also moved with that element. After sorting operation is done, each leader processor sends its element and its current column index r back to that element's original location using the original column index that moves with the element. After receiving its original element and index r, each leader processor sets its element to be equal to the original one, set its column rank to be r, and broadcast r to every processor in the same column. This is a multiple column *broadcast* communication operation.

- **Step 3**: *Construct regular matrices X_i.*

  1. *Move each column to its regular matrix* : each processor $P_{i,j}$ sends its element $a_{i,j}$ to processor $P_{i,r}$ if its column rank in S is r. After this *one-to-one* row communication operation, each processor $P_{i,j}$ sets its $x_{i,j}$ to be the element received.
  2. *Calculate bounding interval I* : each processor $P_{n,j\sqrt{n}}$, where $1 \leq j < \sqrt{n}$, sends its element $x_{n,j\sqrt{n}}$ to processor $P_{n,j\sqrt{n}+1}$. Then each processor $P_{i,j\sqrt{n}}$, where $1 \leq i \leq n$ and $1 \leq j < \sqrt{n}$, sets its reconfigurable switches to cross to form $\sqrt{n}$ sub matrices. Within each $n \times \sqrt{n}$ sub-matrix, leader of the first column broadcast the element a

just received and leader of the last column broadcasts its element b. *2D_Broadcast* operation is used in both of those broadcasting. After the two broadcasting operations, each processor in the system sets its bounding interval $I = (a, b]$, and all processors set their reconfigurable switches back to straight to form a single 2D LARPBS.

3. *Determine* $x_{i,j}$: each processor $P_{i,j}$ compares the received element with its bounding interval. If the element falls out of the bounding interval, it sets its $x_{i,j} = \infty$. Then each processor $P_{i,j}$ whose $x_{i,j} \neq \infty$ sends a signal message to processor $P_{i,k}$ if its $x_{i,j}$ comes from $P_{i,k}$. Processor $P_{i,k}$, who received a signal message in this *one-to-one* row bus communication operation, sets its $a_{i,j} = \infty$.

- **Step 4**: *Construct special matrices* Y_i.

1. *Identify special columns and their special matrix* Y_i: each leader processor sends its column rank r to the leader processor on its right. This is a *one-to-one* row communication operation. Upon receiving the rank from the processor on its left, each leader processor find the B_i for its column using Definition 5.6.4, and broadcasts the indices i of all B_is found in its column bus. This is a multiple column *broadcast* communication operation.
2. *Send special elements to their special matrices*: If processor $P_{i,j}$ received special matrix indices $(l, l+1, \cdots, h)$ in the previous step and $a_{i,j} \neq \infty$, then $P_{i,j}$ multicasts its element $a_{i,j}$ to the k^{th} column of matrices $A_l, A_{(l+1)}, \cdots, A_h$, where $k = \lceil j/\sqrt{n} \rceil$. According to Lemma 5.6.1, each processor in the system will receive at most one element in this multiple row multicast communication operation.
3. *Determine qualified special elements*: all processors do initialization $y_{i,j} = \infty$. Each processor $P_{i,j}$ who received an element in previous step, compares the received element y with its bounding interval. If y falls in the bounding interval, then processor $P_{i,j}$ does two things: (1) set $y_{i,j}$ equals to the element received. (2) multicast a dummy message to processors $P_{i,j'}$, where $j' = j + m\sqrt{n}$, $j' < n$ and $m \geq 1$ (i.e. the processors on the same row on k^{th} column in every matrix A_r that is on the right of processor $P_{i,j}$, where $k = \lceil j/\sqrt{n} \rceil$). Each processor $P_{i,j'}$ who received a dummy message in the multicast communication operation sets $y_{i,j'} = \infty$.

- **Step 5**: Apply alogirthm 5.4.3 *Basic_Columnsort* twice on each $n \times \sqrt{n}$ sub-matrix A_i, once for sorting X_i, and once for sorting Y_i.

- **Step 6**: Apply algorithm 5.5.2 *Two_Way_Merge_Columnsort1* on each $n \times \sqrt{n}$ sub-matrix A_i to merge elements in each X_i and Y_i and sort them in column-major order.

- **Step 7**: *Calculate final address for each element in* X_i *and* Y_i.

1. *Calculate number of elements in each* X_i *and* Y_i: each processor $P_{i,j}$ checks its $y_{i,j}$ value. If $y_{i,j} = \infty$, it sends a signal message to $P_{((i-1) \bmod n),j}$. If processor $P_{n,j}$

whose $y_{n,j} = \infty$, received a message in this *one-to-one* column communication operation, it sends a signal message again to processor $P_{n,j-1}$, where $j > 1$. Only one processor whose $y_{i,j} \neq \infty$ should get a message in the two *one-to-one* column and row communication operations. And this processor holds the largest element in its Y_i matrix. This processor then sends its address to the header processor of the sub-matrix it belongs to. Let $B_i = X_i \bigcup Y_i$, and $|B_i|$, $|X_i|$, and $|Y_i|$ represent number of non-∞ elements in B_i, X_i, and Y_i, respectively. Since each Y_i are sorted, the header processor who receives an address can calculate the rank r from the received address and obtain the number of elements in B_i: $|B_i| = |X_i| + |Y_i| = r + n\sqrt{n}$. If a header processor does not receive an address, then it broadcast a signal to every processor in its sub-matrix to calculate the number of elements in X_i using the same method as we used for calculating the number of elements in Y_i, and set $|B_i| = |X_i|$. For both cases, $|B_i|$ is stored in the header processor.

2. *Calculate integer prefix sums for each sub-matrix*: each header processor $P_{1,k\sqrt{n}+1}$, where $0 \leq k < n$, sends its $|B_{k+1}|$ to processor $P_{1+k,k\sqrt{n}+1}$. Processor $P_{1+k,k\sqrt{n}+1}$ then sends the received $|B_{k+1}|$ to processor $P_{1+k,n}$. Now run algorithm 2.4.2 *2D_Integer_Prefix_Sums* on the first $\sqrt{n} \times n$ sub-matrix to calculate the integer prefix sums for $|B_i|$. Let PS_i be the prefix sums of $|B_i|$. Then at the end of algorithm 2.4.2 *2D_Integer_Prefix_Sums*, PS_i will be stored in processor $P_{i,n}$.
3. *Calculate final destination address for each element*: processor $P_{i,n}$ sends PS_i to processor $P_{i,(i+1)\sqrt{n}}$, where $1 \leq i < \sqrt{n}$. Processor $P_{i,(i+1)\sqrt{n}}$ then broadcast PS_i in the i^{th} $n \times \sqrt{n}$ matrix using *2D_Broadcast* operation. Each processor $P_{i,j}$ calculates the final address for the element it holds by adding its rank in its sub-matrix with the prefix sums it received and convert it to a two-dimensional array address.

- **Step 8**: do the following operations for matrix X_i and Y_i separately: send elements to their final destinations using a *one-to-one* column communication operation followed by a *one-to-one* row communication operation – use the column communication to send an element to the correct row, and then use the row communication to send an element to the correct column.

End

The non-constant time operations used in the above algorithm are: algorithm 5.5.2 *Two_Way_Merge_Columnsort1*, alogirthm 5.4.3 *Basic_Columnsort*, and algorithm 4.4.1 *Pipepined_Merge_Sort*. All of those three algorithms run in $O(\log n)$ time, thus we obtain the following theorem:

Theorem 11 *The multi-way merge algorithm on an $n \times n$ 2D LARPBS merges $\sqrt{n}$ sorted list each of size $n\sqrt{n}$ into one sorted list in $O(\log n)$ time.*

5.7 An $O(\log n)$-Time Merge-Based Sorting Algorithm on a 2D LARPBS

With algorithm 5.4.3 *Basic_Columnsort* and algorithm 5.6.1 *Multiway_Merge* at disposal, we are ready to present our $O(\log n)$ sorting algorithm on 2D LARPBS. This algorithm sorts n^2 elements in $O(\log n)$ time using an $n \times n$ 2D LARPBS.

Algorithm 5.7.1 Multiway_Merge_Sort
Input: *a sequence of n^2 unsorted elements distributed to an $n \times n$ 2D LARPBS, one element per processor.*
Output: *a sequence of n^2 sorted elements stored in the $n \times n$ 2D LARPBS in column major order, one element per processor*
Begin

1. *Partition and sort*: each processor $P_{i,j\sqrt{n}}$ set its row reconfigurable switches to cross to partition the $n \times n$ matrix into $\sqrt{n}$ sub-matrices, where $1 \leq i \leq n$, $1 \leq j \leq \sqrt{n}$. Each sub-matrix runs the algorithm 5.4.3 *Basic_Columnsort* simultaneously.

2. *Multi-way merge*: Apply *Multiway_Merge* algorithm to the $\sqrt{n}$ sub matrices to obtain the sorted sequence in column major order.

End

Since both algorithm 5.4.3 *Basic_Columnsort* and algorithm 5.6.1 *Multiway_Merge* run in $O(\log n)$ time, it is clear that algorithm 5.7.1 *Multiway_Merge_Sort* algorithm runs in $O(\log n)$ time.

Theorem 12 *Algorithm 5.7.1 Multiway_Merge_Sort sorts n^2 elements on an $n \times n$ 2D LARPBS in $O(\log n)$ time.*

Example 5.7.1 *Multiway_Merge_Sort for a 9×9 matrix.*

$$\begin{bmatrix} 41 & 48 & 23 & 33 & 40 & 57 & 64 & 56 & 54 \\ 5 & 67 & 44 & 59 & 69 & 74 & 2 & 35 & 58 \\ 61 & 15 & 25 & 80 & 24 & 31 & 66 & 37 & 52 \\ 36 & 42 & 62 & 7 & 13 & 12 & 71 & 49 & 16 \\ 17 & 19 & 65 & 21 & 60 & 27 & 68 & 70 & 45 \\ 10 & 77 & 30 & 38 & 28 & 51 & 11 & 20 & 81 \\ 32 & 55 & 14 & 3 & 50 & 76 & 29 & 53 & 47 \\ 79 & 8 & 63 & 75 & 4 & 34 & 46 & 26 & 39 \\ 6 & 22 & 9 & 1 & 72 & 18 & 78 & 73 & 43 \end{bmatrix}$$

Input

$$\begin{bmatrix} 41 & 48 & 23 \\ 5 & 67 & 44 \\ 61 & 15 & 25 \\ 36 & 42 & 62 \\ 17 & 19 & 65 \\ 10 & 77 & 30 \\ 32 & 55 & 14 \\ 79 & 8 & 63 \\ 6 & 22 & 9 \end{bmatrix} \quad \begin{bmatrix} 33 & 40 & 57 \\ 59 & 69 & 74 \\ 80 & 24 & 31 \\ 7 & 13 & 12 \\ 21 & 60 & 27 \\ 38 & 28 & 51 \\ 3 & 50 & 76 \\ 75 & 4 & 34 \\ 1 & 72 & 18 \end{bmatrix} \quad \begin{bmatrix} 64 & 56 & 54 \\ 2 & 35 & 58 \\ 66 & 37 & 52 \\ 71 & 49 & 16 \\ 68 & 70 & 45 \\ 11 & 20 & 81 \\ 29 & 53 & 47 \\ 46 & 26 & 39 \\ 78 & 73 & 43 \end{bmatrix}$$

after split

	$\begin{bmatrix} 5 & 22 & 48 \\ 6 & 23 & 55 \\ 8 & 25 & 61 \\ 9 & 30 & 62 \\ 10 & 32 & 63 \\ 14 & 36 & 65 \\ 15 & 41 & 67 \\ 17 & 42 & 77 \\ 19 & 44 & 79 \end{bmatrix}$	$\begin{bmatrix} 1 & 27 & 57 \\ 3 & 28 & 59 \\ 4 & 31 & 60 \\ 7 & 33 & 69 \\ 12 & 34 & 72 \\ 13 & 38 & 74 \\ 18 & 40 & 75 \\ 21 & 50 & 76 \\ 24 & 51 & 80 \end{bmatrix}$	$\begin{bmatrix} 2 & 43 & 58 \\ 11 & 45 & 64 \\ 16 & 46 & 66 \\ 20 & 47 & 68 \\ 26 & 49 & 70 \\ 29 & 52 & 71 \\ 35 & 53 & 73 \\ 37 & 54 & 78 \\ 39 & 56 & 81 \end{bmatrix}$
Leader's rank	$\begin{bmatrix} 1 & 4 & 7 \end{bmatrix}$	$\begin{bmatrix} 2 & 5 & 8 \end{bmatrix}$	$\begin{bmatrix} 3 & 6 & 9 \end{bmatrix}$
Special matrix index	$\begin{bmatrix} na & 1 & 2 \end{bmatrix}$	$\begin{bmatrix} na & 1 & 2 \end{bmatrix}$	$\begin{bmatrix} na & 1 & 2 \end{bmatrix}$

sorted sub − matrices

$$\begin{bmatrix} 5 & 1 & 2 \\ 6 & 3 & 11 \\ 8 & 4 & 16 \\ 9 & 7 & 20 \\ 10 & 12 & 26 \\ 14 & 13 & 29 \\ 15 & 18 & 35 \\ 17 & 21 & 37 \\ 19 & 24 & 39 \end{bmatrix} \quad \begin{bmatrix} \infty & \infty & 43 \\ \infty & \infty & 45 \\ \infty & \infty & 46 \\ \infty & \infty & 47 \\ \infty & \infty & 49 \\ \infty & \infty & 52 \\ 41 & 40 & 53 \\ 42 & 50 & 54 \\ 44 & 51 & 56 \end{bmatrix} \quad \begin{bmatrix} \infty & 57 & 58 \\ \infty & 59 & 64 \\ 61 & 60 & 66 \\ 62 & 69 & 68 \\ 63 & 72 & 70 \\ 65 & 74 & 71 \\ 67 & 75 & 73 \\ 77 & 76 & 78 \\ 79 & 80 & 81 \end{bmatrix}$$

regular matrixes

$$\begin{bmatrix} 22 & 27 & \infty \\ 23 & 28 & \infty \\ 25 & 31 & \infty \\ 30 & 33 & \infty \\ 32 & 34 & \infty \\ 36 & 38 & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \end{bmatrix} \quad \begin{bmatrix} 48 & \infty & \infty \\ 55 & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \end{bmatrix} \quad \begin{bmatrix} \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \end{bmatrix}$$

special matrices

$$\begin{bmatrix} 1 & 10 & 19 \\ 2 & 11 & 20 \\ 3 & 12 & 21 \\ 4 & 13 & 24 \\ 5 & 14 & 26 \\ 6 & 15 & 29 \\ 7 & 16 & 35 \\ 8 & 17 & 37 \\ 9 & 18 & 39 \end{bmatrix} \quad \begin{bmatrix} 40 & 50 & \infty \\ 41 & 51 & \infty \\ 42 & 52 & \infty \\ 43 & 53 & \infty \\ 44 & 54 & \infty \\ 45 & 56 & \infty \\ 46 & \infty & \infty \\ 47 & \infty & \infty \\ 49 & \infty & \infty \end{bmatrix} \quad \begin{bmatrix} 57 & 66 & 75 \\ 58 & 67 & 76 \\ 59 & 68 & 77 \\ 60 & 69 & 78 \\ 61 & 70 & 79 \\ 62 & 71 & 80 \\ 63 & 72 & 81 \\ 64 & 73 & \infty \\ 65 & 74 & \infty \end{bmatrix}$$

sorted regular matrixes

$$\begin{bmatrix} 22 & 34 & \infty \\ 23 & 36 & \infty \\ 25 & 38 & \infty \\ 27 & \infty & \infty \\ 28 & \infty & \infty \\ 30 & \infty & \infty \\ 31 & \infty & \infty \\ 32 & \infty & \infty \\ 33 & \infty & \infty \end{bmatrix} \quad \begin{bmatrix} 48 & \infty & \infty \\ 55 & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \end{bmatrix} \quad \begin{bmatrix} \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \end{bmatrix}$$

sorted special matrices

$$\begin{bmatrix} 1 & 10 & 19 \\ 2 & 11 & 20 \\ 3 & 12 & 21 \\ 4 & 13 & 22 \\ 5 & 14 & 23 \\ 6 & 15 & 24 \\ 7 & 16 & 25 \\ 8 & 17 & 26 \\ 9 & 18 & 27 \end{bmatrix} \begin{bmatrix} 40 & 49 & \infty \\ 41 & 50 & \infty \\ 42 & 51 & \infty \\ 43 & 52 & \infty \\ 44 & 53 & \infty \\ 45 & 54 & \infty \\ 46 & 55 & \infty \\ 47 & 56 & \infty \\ 48 & \infty & \infty \end{bmatrix} \begin{bmatrix} 57 & 66 & 75 \\ 58 & 67 & 76 \\ 59 & 68 & 77 \\ 60 & 69 & 78 \\ 61 & 70 & 79 \\ 62 & 71 & 80 \\ 63 & 72 & 81 \\ 64 & 73 & \infty \\ 65 & 74 & \infty \end{bmatrix}$$
$$\begin{bmatrix} 28 & 37 & \infty \\ 29 & 38 & \infty \\ 30 & 39 & \infty \\ 31 & \infty & \infty \\ 32 & \infty & \infty \\ 33 & \infty & \infty \\ 34 & \infty & \infty \\ 35 & \infty & \infty \\ 36 & \infty & \infty \end{bmatrix} \begin{bmatrix} \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \end{bmatrix} \begin{bmatrix} \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \end{bmatrix}$$

after merging sorted special matrices and regular matrices

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 39 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 17 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 25 \end{bmatrix}$$

number of elements in each B_i

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 39 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 56 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 81 \end{bmatrix}$$

prefix sum on the number of elements in each B_i

$$\begin{bmatrix} 1,28 & 10,37 & 19 \\ 2,29 & 11,38 & 20 \\ 3,30 & 12,39 & 21 \\ 4,31 & 13 & 22 \\ 5,32 & 14 & 23 \\ 6,33 & 15 & 24 \\ 7,34 & 16 & 25 \\ 8,35 & 17 & 26 \\ 9,36 & 18 & 27 \end{bmatrix} \begin{bmatrix} \boxed{46} & \boxed{55} & \infty \\ \boxed{47} & \boxed{56} & \infty \\ \boxed{48} & \infty & \infty \\ 40 & 49 & \infty \\ 41 & 50 & \infty \\ 42 & 51 & \infty \\ 43 & 52 & \infty \\ 44 & 53 & \infty \\ 45 & 54 & \infty \end{bmatrix} \begin{bmatrix} \boxed{64} & \boxed{73} & \infty \\ \boxed{65} & \boxed{74} & \infty \\ 57 & 66 & 75 \\ 58 & 67 & 76 \\ 59 & 68 & 77 \\ 60 & 69 & 78 \\ 61 & 70 & 79 \\ 62 & 71 & 80 \\ 63 & 72 & 81 \end{bmatrix}$$

column communication : send elements to correct rows

$$\begin{bmatrix} 1 & 10 & 19 \\ 2 & 11 & 20 \\ 3 & 12 & 21 \\ 4 & 13 & 22 \\ 5 & 14 & 23 \\ 6 & 15 & 24 \\ 7 & 16 & 25 \\ 8 & 17 & 26 \\ 9 & 18 & 27 \end{bmatrix} \quad \begin{bmatrix} 28 & 37 & 46 \\ 29 & 38 & 47 \\ 30 & 39 & 48 \\ 31 & 40 & 49 \\ 32 & 41 & 50 \\ 33 & 42 & 51 \\ 34 & 43 & 52 \\ 35 & 44 & 53 \\ 36 & 45 & 54 \end{bmatrix} \quad \begin{bmatrix} 55 & 64 & 73 \\ 56 & 65 & 74 \\ 57 & 66 & 75 \\ 58 & 67 & 76 \\ 59 & 68 & 77 \\ 60 & 69 & 78 \\ 61 & 70 & 79 \\ 62 & 71 & 80 \\ 63 & 72 & 81 \end{bmatrix}$$

row communication : send elements to correct columns

5.8 An $O(\log n)$-Time Generalized Columnsort Algorithm on a 2D LARPBS

In this section, we give a generalized Columnsort algorithm to sort n^2 elements on an $n \times n$ 2D LARPBS in $O(\log n)$ time. The basic idea is: partition matrix A into $\sqrt{n}$ sub-matrices: $A_1, A_2, \cdots, A_{\sqrt{n}}$, each being an $n \times \sqrt{n}$ matrix. We then construct a virtual matrix B of size $n\sqrt{n} \times \sqrt{n}$ by stretching all elements in each sub-matrix to a so called "super-column" of $n\sqrt{n}$ elements, and we call it a super column. We call B a virtual matrix because it is a logical matrix created for the convenience of explaining the algorithm. Since the virtual matrix satisfies the restriction for column sort, i.e. number of rows greater than the square of the number of columns, we can use Algorithm 5.4.3 *Basic_Columnsort* to sort the virtual matrix B. Thus by introducing the concept of super column and virtual matrix, we convert the problem of sorting an $n \times n$ matrix into sorting an $n\sqrt{n} \times \sqrt{n}$ matrix. And we will show that sorting an $n\sqrt{n} \times \sqrt{n}$ matrix with $n \times n$ processors can be done on 2D LARPBS in $O(\log n)$ time.

Before we present our generalized Columnsort algorithm, we need to provide following basic operations:

- *$\sqrt{n}$-way-column-shuffle*: Each processor $P_{i,j}$, where $1 \leq i, j \leq n$, sends its element to processor $P_{i,k}$, where $k = (j \bmod \sqrt{n}) \times \sqrt{n} + \lceil j/\sqrt{n} \rceil$. This is a *one-to-one* row communication.

- *form-super-columns*: Processors $P_{i,j\sqrt{n}}$, where $1 \leq i \leq \sqrt{n}$, set their row reconfigurable switches to cross to partition matrix A into $\sqrt{n}$ sub-matrices: $A_1, A_2, \cdots, A_{\sqrt{n}}$. Each of those sub-matrices A_i is a $n \times \sqrt{n}$ matrix and we call it super-column i.

- *fuse-super-columns*: Processors $P_{i,j\sqrt{n}}$, where $1 \leq i \leq \sqrt{n}$, set their row reconfigurable switches to straight to fuse $\sqrt{n}$ super columns into one $n \times n$ matrix A.

Algorithm 5.8.1 Generalized_Columnsort
Input*: a sequence of n^2 unsorted elements distributed to an $n \times n$ 2D LARPBS, one element per processor.*

Output: *a sequence of n^2 sorted elements stored in the $n \times n$ 2D LARPBS in column major order, one element per processor*
Begin

1. *Partition and sort each super column*: Do a *form-super-columns* operation to partition matrix A into $\sqrt{n}$ sub-matrices. Each sub-matrix runs the algorithm 5.4.3 *Basic_Columnsort* simultaneously to sort its elements in column major order. This step serves the purpose of sorting items in each column of the virtual matrix in column major order. After sorting is done, do a *fuse-super-columns* operation to form on 2D LARPBS.

2. *Shuffle and transpose*: Each sub-matrix A_i does an *Extended_Matrix_Transpose* operation. (This operation actually equivalent to $\sqrt{n}$*-way-column-shuffle* plus transpose the $n\sqrt{n} \times \sqrt{n}$ virtual matrix)

3. *Shuffle and sort*: Do a $\sqrt{n}$*-way-column-shuffle* to form the new super columns. Then run the algorithm 5.4.3 *Basic_Columnsort* again simultaneously in each super column to sort its elements in columns major order.

4. *Shuffle and reverse transpose*: Do a $\sqrt{n}$*-way-column-shuffle* operation to form the new super columns. Then do a *Extended_Reverse_Matrix_Transpose* operation in each sub-matrix A_i. (Actually we should do a $\sqrt{n}$*-way-column-shuffle* operation after the *Extended_Reverse_Matrix_Transpose* to move the items to their correct locations for correctly reversely transpose the $n\sqrt{n} \times \sqrt{n}$ virtual matrix. But as we need another $\sqrt{n}$*-way-column-shuffle* operation after this one to shuffle elements in a super column into a sub-matrix so that we can *alternate sort* each super column, two shuffles cancel each other. So we skip the shuffle here and the one in the next step as well)

5. *Alternate sort*: Run algorithm 5.4.3 *Basic_Columnsort* again simultaneously on each sub-matrix(super column). This time, odd indexed sub-matrices sort their elements in increasing order; even indexed sub-matrices sort their elements in decreasing order.

6. *Shuffle and 2 steps of odd-even sort*: Do an $\sqrt{n}$*-way-column-shuffle* operation to put elements in their correct virtual matrix locations. Then do a *form-super-columns* operation and an odd step followed by an even step of *odd-even-transposition-sort* operation on row buses to partially sort each super column.

7. *Shuffle and final column sort*: Do a $\sqrt{n}$*-way-column-shuffle* operation to form super columns for the final column sort. Then run algorithm 5.4.3 *Basic_Columnsort* within each super column to sort elements for each super column in column major order. After this column sort step, all elements in matrix A are sorted in column major order from super column 1 to $\sqrt{n}$.

End

The correctness of this algorithm is based on the 7-phase Columnsort algorithm mentioned in section 5.3. Since the virtual matrix on which we apply the 7-phase Columnsort algorithm satisfies $r \geq s^2$, Algorithm 5.8.1 *Generalized_Columnsort* correctly sorts the n^2 elements in column-major order. As the only non-constant operation performed by Algorithm 5.8.1 is Algorithm 5.4.3 *Basic_Columnsort* which runs in $O(\log n)$ time, we obtain the following theorem:

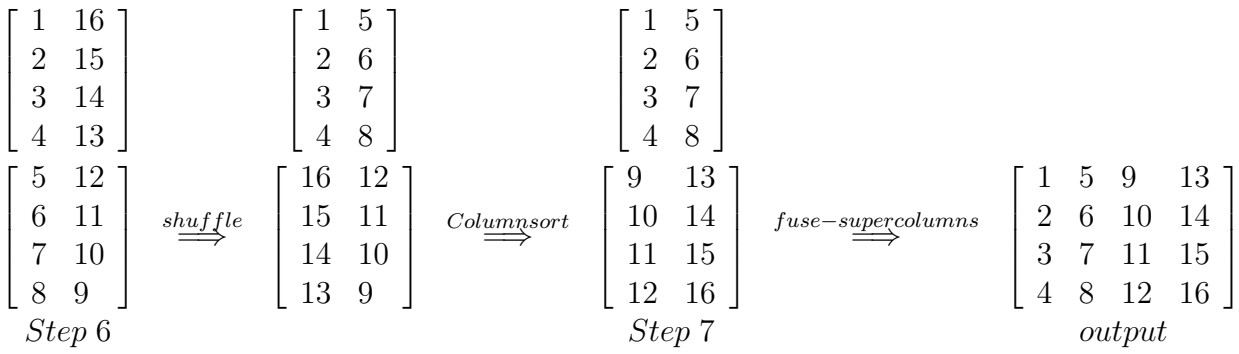
Theorem 13 *The Generalized_Columnsort algorithm sorts n^2 elements on an $n \times n$ 2D LARPBS in $O(\log n)$ time.*

Example 5.8.1 *Generalized_Columnsort for 4×4 matrix.*

$$\underset{input}{\begin{bmatrix} 4 & 13 & 9 & 16 \\ 3 & 14 & 5 & 12 \\ 1 & 8 & 6 & 15 \\ 7 & 11 & 2 & 10 \end{bmatrix}} \xrightarrow{partition} \underset{virtual\ view}{\begin{bmatrix} 4 & 13 \\ 3 & 14 \\ 1 & 8 \\ 7 & 11 \end{bmatrix} \begin{bmatrix} 9 & 16 \\ 5 & 12 \\ 6 & 15 \\ 2 & 10 \end{bmatrix}} \xrightarrow{Columnsort} \underset{Step\ 1}{\begin{bmatrix} 1 & 8 \\ 3 & 11 \\ 4 & 13 \\ 7 & 14 \end{bmatrix} \begin{bmatrix} 2 & 10 \\ 5 & 12 \\ 6 & 15 \\ 9 & 16 \end{bmatrix}} \xrightarrow{transpose}$$

$$\underset{Step\ 2}{\begin{bmatrix} 1 & 3 \\ 4 & 7 \\ 8 & 11 \\ 13 & 14 \end{bmatrix} \begin{bmatrix} 2 & 5 \\ 6 & 9 \\ 10 & 12 \\ 15 & 16 \end{bmatrix}} \xrightarrow{shuffle} \begin{bmatrix} 1 & 2 \\ 4 & 6 \\ 8 & 10 \\ 13 & 15 \end{bmatrix} \begin{bmatrix} 3 & 5 \\ 7 & 9 \\ 11 & 12 \\ 14 & 16 \end{bmatrix} \xrightarrow{Columnsort} \underset{Step\ 3}{\begin{bmatrix} 1 & 8 \\ 2 & 10 \\ 4 & 13 \\ 6 & 15 \end{bmatrix} \begin{bmatrix} 3 & 11 \\ 5 & 12 \\ 7 & 14 \\ 9 & 16 \end{bmatrix}} \xrightarrow{shuffle} \begin{bmatrix} 1 & 3 \\ 2 & 5 \\ 4 & 7 \\ 6 & 9 \end{bmatrix} \begin{bmatrix} 8 & 11 \\ 10 & 12 \\ 13 & 14 \\ 15 & 16 \end{bmatrix}$$

$$\xrightarrow{reverse\ transpose} \underset{Step4}{\begin{bmatrix} 1 & 4 \\ 3 & 7 \\ 2 & 6 \\ 5 & 9 \end{bmatrix} \begin{bmatrix} 8 & 13 \\ 11 & 14 \\ 10 & 15 \\ 12 & 16 \end{bmatrix}} \xrightarrow{Columnsort} \underset{Step\ 5}{\begin{bmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 9 \end{bmatrix} \begin{bmatrix} 16 & 12 \\ 15 & 11 \\ 14 & 10 \\ 13 & 8 \end{bmatrix}} \xrightarrow{shuffle} \begin{bmatrix} 1 & 16 \\ 2 & 15 \\ 3 & 14 \\ 4 & 13 \end{bmatrix} \begin{bmatrix} 5 & 12 \\ 6 & 11 \\ 7 & 10 \\ 9 & 8 \end{bmatrix} \xrightarrow{odd-even-sort}$$


1 16
2 15
3 14
4 13
5 12
6 11
7 10
8 9
shuffle
1 5
2 6
3 7
4 8
16 12
15 11
14 10
13 9
Columnsort
1 5
2 6
3 7
4 8
9 13
10 14
11 15
12 16
fuse−supercolumns
1 5 9 13
2 6 10 14
3 7 11 15
4 8 12 16
Step 6
Step 7
output

Chapter 6

Optimal Sorting Algorithms on Higher Dimensional LARPBS

6.1 An $O(\log n)$-Time Generalized Columnsort Algorithm on a 3D LARPBS

The basic idea of our 3D generalized Columnsort algorithm is: partition the n^3 elements into n group, each of which is an $n \times n$ matrix. We call each of these $n \times n$ matrices a super column. So the n^3 elements can be visualized as distributed into an $n^2 \times n$ virtual matrix, where the number of rows $r = n^2$, and the number of columns $s = n$. As $r = s^2$, we can apply basic Columnsort algorithm to this virtual matrix. For the convenience of description, we denote the three different groups of buses as: x buses, y buses, and z buses for dimension 1, 2, and 3, respectively. We denote each xy array an xy-plane, each xz array an xz-plane, and each yz array a yz-plane. Our sorting algorithm sorts the n^3 elements distributed to the three-dimensional LARPBS into zyx-order. For example, a yx-order for a two dimensional array refers to column-major order, and an xy-order for a two dimensional array refers to row-major order.

The following basic operation will be used in our algorithm:

n-way-column-shuffle: each processor $P_{i,j,k}$, where $1 \leq i, j, k \leq n$, sends its element to processor $P_{i,j',k'}$, where $j' = k, k' = j$. This can be done by simultaneously running algorithm 2.4.1 *Matrix_Transpose* on all xz-planes($1 \leq i \leq n$) (See Figure 6.1 for an example).

Algorithm 6.1.1 3D_Generalized_Columnsort
Input: *a sequence of n^3 unsorted elements distributed to an $n \times n \times n$ 3D LARPBS, one element per processor.*
Output: *a sequence of n^3 sorted elements stored in the $n \times n \times n$ 3D LARPBS in zyx-order, one element per processor*
Begin

1. *Sort each xy-plane*: run algorithm 5.8.1 *Generalized_Columnsort* simultaneously for

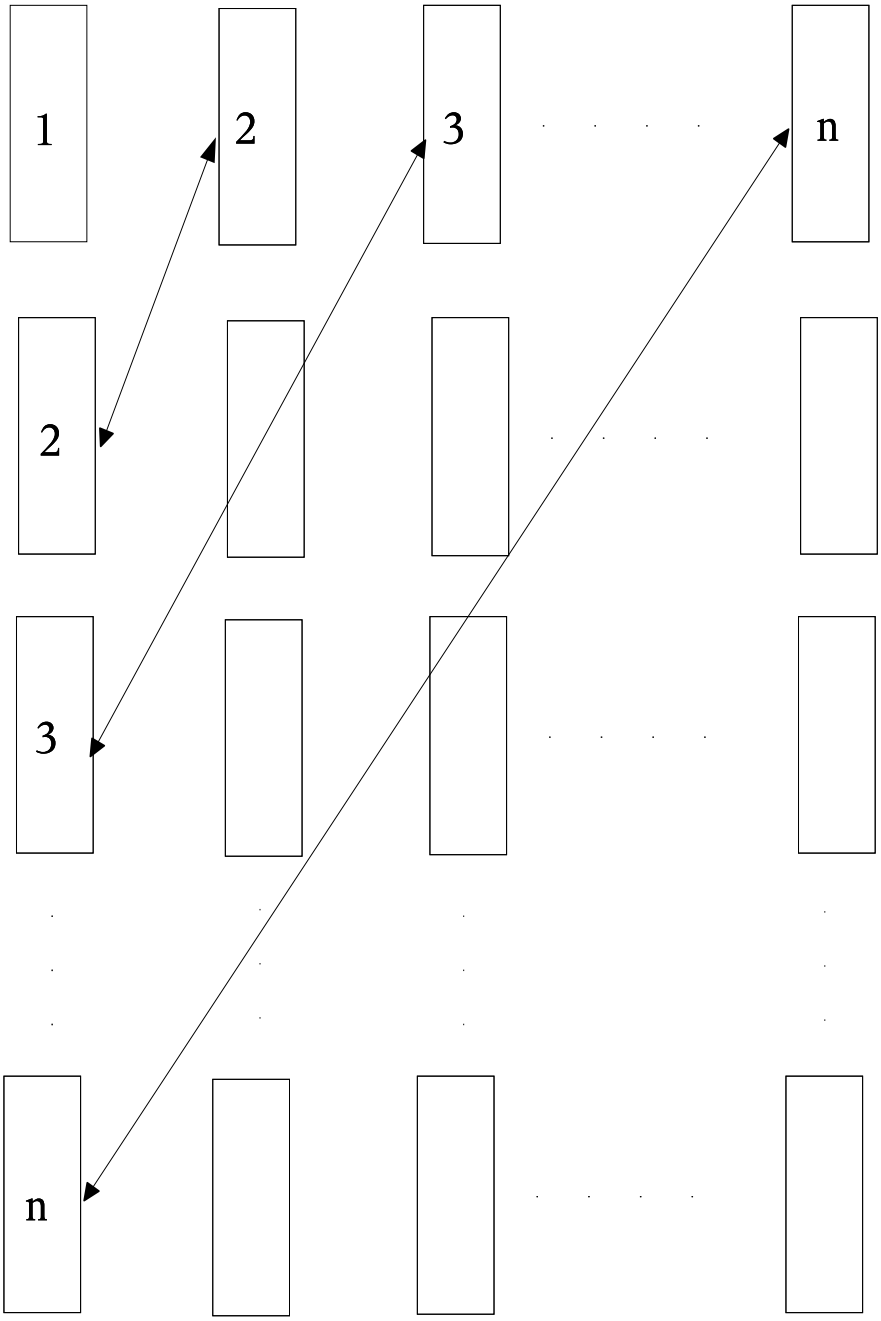


Figure 6.1: n-way-column-shuffle

each xy-plane using only x buses and y buses. This will sort the n xy-planes in yx-order.

2. *Transpose and shuffle*: run algorithm 2.4.1 *Matrix_Transpose* on each xy-plane. Then do an *n-way-column-shuffle* operation on the 3D LARPBS. (The *Matrix_Transpose* operation performed here is equivalent to an *n-way-column-shuffle* operation plus a transposition operation on an $n^2 \times n$ matrix.)

3. *Sort super column*: run algorithm 5.8.1 *Generalized_Columnsort* again on each xy-plane to sort each of n $n \times n$ matrix in yx-order.

4. *Shuffle and reverse transpose*: do an *n-way-column-shuffle* operation, then run algorithm 2.4.1 *Matrix_Transpose* on xy-planes (the *Matrix_Transpose* operation performed in this step is equivalent to a reverse transpose on $n^2 \times n$ matrix followed by an *n-way-column-shuffle* operation).

5. *Alternate sort*: run algorithm 5.8.1 *Generalized_Columnsort* on each xy-plane so that adjacent xy-planes are sorted in reverse yx-order.

6. *shuffle and odd-even transposition sort*: do an *n-way-column-shuffle* operation, followed by an odd step and an even step of *odd-even-transposition-sort* on y buses.

7. *shuffle and sort*: do an *n-way-column-shuffle* operation, then run algorithm 5.8.1 *Generalized_Columnsort* to finally sort the n^3 elements in zyx-order.

End

Since Algorithm 6.1.1 *3D_Generalized_Columnsort* applies the 7-phase Columnsort algorithm to an $r \times s$ matrix, where $r = n^2$, $s = n$, and $r = s^2$, it correctly sorts the n^3 elements. This $r \times s$ matrix is simulated by an $n \times n \times n$ 3D LARPBS in Algorithm 6.1.1 *3D_Generalized_Columnsort*. As the only non-constant time operation performed in the above algorithm is algorithm 5.8.1 *Generalized_Columnsort*, we obtain the following theorem:

Theorem 14 *Algorithm 6.1.1 3D_Generalized_Columnsort sorts n^3 elements on an $n \times n \times n$ 3D LARPBS in $O(\log n)$ time.*

Example 6.1.1 *3D_Generalized_Columnsort on a $4 \times 4 \times 4$ array. We show xy-planes for all the steps.*

$$\begin{bmatrix} 22 & 7 & 39 & 50 \\ 47 & 53 & 10 & 27 \\ 41 & 25 & 15 & 63 \\ 13 & 31 & 38 & 45 \end{bmatrix} \begin{bmatrix} 28 & 54 & 42 & 8 \\ 5 & 34 & 21 & 57 \\ 44 & 60 & 3 & 26 \\ 61 & 24 & 12 & 4 \end{bmatrix} \begin{bmatrix} 32 & 46 & 56 & 43 \\ 11 & 29 & 17 & 9 \\ 52 & 62 & 20 & 14 \\ 1 & 23 & 48 & 30 \end{bmatrix} \begin{bmatrix} 49 & 16 & 6 & 40 \\ 2 & 19 & 58 & 18 \\ 36 & 59 & 35 & 51 \\ 55 & 33 & 64 & 37 \end{bmatrix}$$

Input

$\Longrightarrow$

$$\begin{bmatrix} 7 & 22 & 38 & 47 \\ 10 & 25 & 39 & 50 \\ 13 & 27 & 41 & 53 \\ 15 & 31 & 45 & 63 \end{bmatrix} \begin{bmatrix} 3 & 12 & 28 & 54 \\ 4 & 21 & 34 & 57 \\ 5 & 24 & 42 & 60 \\ 8 & 26 & 44 & 61 \end{bmatrix} \begin{bmatrix} 1 & 17 & 30 & 48 \\ 9 & 20 & 32 & 52 \\ 11 & 23 & 43 & 56 \\ 14 & 29 & 46 & 62 \end{bmatrix} \begin{bmatrix} 2 & 19 & 37 & 55 \\ 6 & 33 & 40 & 58 \\ 16 & 35 & 49 & 59 \\ 18 & 36 & 51 & 64 \end{bmatrix}$$

Step 1
$yx - ordered$

$\Longrightarrow$

$$\begin{bmatrix} 7 & 10 & 13 & 15 \\ 22 & 25 & 27 & 31 \\ 38 & 39 & 41 & 45 \\ 47 & 50 & 53 & 63 \end{bmatrix} \begin{bmatrix} 3 & 4 & 5 & 8 \\ 12 & 21 & 24 & 26 \\ 28 & 34 & 42 & 44 \\ 54 & 57 & 60 & 61 \end{bmatrix} \begin{bmatrix} 1 & 9 & 11 & 14 \\ 17 & 20 & 23 & 29 \\ 30 & 32 & 43 & 46 \\ 48 & 52 & 56 & 62 \end{bmatrix} \begin{bmatrix} 2 & 6 & 16 & 18 \\ 19 & 33 & 35 & 36 \\ 37 & 40 & 49 & 51 \\ 55 & 58 & 59 & 64 \end{bmatrix}$$

Step 2
$xy - transpose$

$$\begin{bmatrix} 7 & 3 & 1 & 2 \\ 22 & 12 & 17 & 19 \\ 38 & 28 & 30 & 37 \\ 47 & 54 & 48 & 55 \end{bmatrix} \begin{bmatrix} 10 & 4 & 9 & 6 \\ 25 & 21 & 20 & 33 \\ 39 & 34 & 32 & 40 \\ 50 & 57 & 52 & 58 \end{bmatrix} \begin{bmatrix} 13 & 5 & 11 & 16 \\ 27 & 24 & 23 & 25 \\ 41 & 42 & 43 & 49 \\ 53 & 60 & 56 & 59 \end{bmatrix} \begin{bmatrix} 15 & 8 & 14 & 18 \\ 31 & 26 & 29 & 36 \\ 45 & 44 & 46 & 51 \\ 63 & 61 & 62 & 64 \end{bmatrix}$$

Step 2
$column - shuffle$

$\Longrightarrow$

$$\begin{bmatrix} 1 & 12 & 28 & 47 \\ 2 & 17 & 30 & 48 \\ 3 & 19 & 37 & 54 \\ 7 & 22 & 38 & 55 \end{bmatrix} \begin{bmatrix} 4 & 20 & 33 & 50 \\ 6 & 21 & 34 & 52 \\ 9 & 25 & 39 & 57 \\ 10 & 32 & 40 & 58 \end{bmatrix} \begin{bmatrix} 5 & 23 & 41 & 53 \\ 11 & 24 & 42 & 56 \\ 13 & 27 & 43 & 59 \\ 16 & 35 & 49 & 60 \end{bmatrix} \begin{bmatrix} 8 & 26 & 44 & 61 \\ 14 & 29 & 45 & 62 \\ 15 & 31 & 46 & 63 \\ 18 & 36 & 51 & 64 \end{bmatrix}$$

Step 3
$yx - sorted$

$\Longrightarrow$

$$\begin{bmatrix} 1 & 4 & 5 & 8 \\ 2 & 6 & 11 & 14 \\ 3 & 9 & 13 & 15 \\ 7 & 10 & 16 & 18 \end{bmatrix} \begin{bmatrix} 12 & 20 & 23 & 26 \\ 17 & 21 & 24 & 29 \\ 19 & 25 & 27 & 31 \\ 22 & 32 & 35 & 36 \end{bmatrix} \begin{bmatrix} 28 & 33 & 41 & 44 \\ 30 & 34 & 42 & 45 \\ 37 & 39 & 43 & 46 \\ 38 & 40 & 49 & 51 \end{bmatrix} \begin{bmatrix} 47 & 50 & 53 & 61 \\ 48 & 52 & 56 & 62 \\ 54 & 57 & 59 & 63 \\ 55 & 58 & 60 & 64 \end{bmatrix}$$

Step 4
$column - shuffle$

$$\begin{bmatrix} 1 & 2 & 3 & 7 \\ 4 & 6 & 9 & 10 \\ 5 & 11 & 13 & 16 \\ 8 & 14 & 15 & 18 \end{bmatrix} \begin{bmatrix} 12 & 17 & 19 & 22 \\ 20 & 21 & 25 & 32 \\ 23 & 24 & 27 & 35 \\ 26 & 29 & 31 & 36 \end{bmatrix} \begin{bmatrix} 28 & 30 & 37 & 38 \\ 33 & 34 & 39 & 40 \\ 41 & 42 & 43 & 49 \\ 44 & 45 & 46 & 51 \end{bmatrix} \begin{bmatrix} 47 & 48 & 54 & 55 \\ 50 & 52 & 57 & 58 \\ 53 & 56 & 59 & 60 \\ 61 & 62 & 63 & 64 \end{bmatrix}$$

Step 4

$xy - transpose$

$$\Longrightarrow$$

$$\begin{bmatrix} 1 & 5 & 9 & 14 \\ 2 & 6 & 10 & 15 \\ 3 & 7 & 11 & 16 \\ 4 & 8 & 13 & 18 \end{bmatrix} \begin{bmatrix} 36 & 29 & 24 & 20 \\ 35 & 27 & 23 & 19 \\ 32 & 26 & 22 & 17 \\ 31 & 25 & 21 & 12 \end{bmatrix} \begin{bmatrix} 28 & 37 & 41 & 45 \\ 30 & 38 & 42 & 46 \\ 33 & 39 & 43 & 49 \\ 34 & 40 & 44 & 51 \end{bmatrix} \begin{bmatrix} 64 & 60 & 56 & 52 \\ 63 & 59 & 55 & 50 \\ 62 & 58 & 54 & 48 \\ 61 & 57 & 53 & 47 \end{bmatrix}$$

Step 5

$$\Longrightarrow$$

$$\begin{bmatrix} 1 & 36 & 28 & 64 \\ 2 & 35 & 30 & 63 \\ 3 & 32 & 33 & 62 \\ 4 & 31 & 34 & 61 \end{bmatrix} \begin{bmatrix} 5 & 29 & 37 & 60 \\ 6 & 27 & 38 & 59 \\ 7 & 26 & 39 & 58 \\ 8 & 25 & 40 & 57 \end{bmatrix} \begin{bmatrix} 9 & 24 & 41 & 56 \\ 10 & 23 & 42 & 55 \\ 11 & 22 & 43 & 54 \\ 13 & 21 & 44 & 53 \end{bmatrix} \begin{bmatrix} 14 & 20 & 45 & 52 \\ 15 & 19 & 46 & 50 \\ 16 & 17 & 49 & 48 \\ 18 & 12 & 51 & 47 \end{bmatrix}$$

Step 6

$column - shuffle$

$$\begin{bmatrix} 1 & 28 & 36 & 64 \\ 2 & 30 & 35 & 63 \\ 3 & 32 & 33 & 62 \\ 4 & 31 & 34 & 61 \end{bmatrix} \begin{bmatrix} 5 & 29 & 37 & 60 \\ 6 & 27 & 38 & 59 \\ 7 & 26 & 39 & 58 \\ 8 & 25 & 40 & 57 \end{bmatrix} \begin{bmatrix} 9 & 24 & 41 & 56 \\ 10 & 23 & 42 & 55 \\ 11 & 22 & 43 & 54 \\ 13 & 21 & 44 & 53 \end{bmatrix} \begin{bmatrix} 14 & 20 & 45 & 52 \\ 15 & 19 & 46 & 50 \\ 16 & 17 & 48 & 49 \\ 12 & 18 & 47 & 51 \end{bmatrix}$$

Step 6

$o/e\ sorted$

$$\Longrightarrow$$

$$\begin{bmatrix} 1 & 5 & 9 & 14 \\ 2 & 6 & 10 & 15 \\ 3 & 7 & 11 & 16 \\ 4 & 8 & 13 & 12 \end{bmatrix} \begin{bmatrix} 28 & 29 & 24 & 20 \\ 30 & 27 & 23 & 19 \\ 32 & 26 & 22 & 17 \\ 31 & 25 & 21 & 18 \end{bmatrix} \begin{bmatrix} 36 & 37 & 41 & 45 \\ 35 & 38 & 42 & 46 \\ 33 & 39 & 43 & 48 \\ 34 & 40 & 44 & 47 \end{bmatrix} \begin{bmatrix} 64 & 60 & 56 & 52 \\ 63 & 59 & 55 & 50 \\ 62 & 58 & 54 & 49 \\ 61 & 57 & 53 & 51 \end{bmatrix}$$

Step 7

$column - shuffle$

$$\Longrightarrow$$

$$\begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix} \begin{bmatrix} 17 & 21 & 25 & 29 \\ 18 & 22 & 26 & 30 \\ 19 & 23 & 27 & 31 \\ 20 & 24 & 28 & 32 \end{bmatrix} \begin{bmatrix} 33 & 37 & 41 & 45 \\ 34 & 38 & 42 & 46 \\ 35 & 39 & 43 & 47 \\ 36 & 40 & 44 & 48 \end{bmatrix} \begin{bmatrix} 49 & 53 & 57 & 61 \\ 50 & 54 & 58 & 62 \\ 51 & 55 & 59 & 63 \\ 52 & 56 & 60 & 64 \end{bmatrix}$$

Step 7

$zyx - sorted$

6.2 A 5-Phase Sorting Algorithm on a 3D LARPBS

Leighton has described an $O(n^{1/3})$-time algorithm that sorts n elements on an $n^{1/3} \times n^{1/3} \times n^{1/3}$ array [30]. In other words, this algorithm sorts n^3 elements in $O(n)$ time on a n^3 array. By applying either algorithm 5.8.1 *Generalized_Columnsort* or 5.7.1 *Multiway_Merge_Sort* to an $n \times n \times n$ 3D LARPBS, the time complexity of this algorithm can be reduced to $O(\log n)$.

This $O(n)$ time 3D sorting algorithm proceeds in 5 simple phases. Here are the details of the algorithm.

Algorithm 6.2.1 5-Phase_Sort_on_3D_Array
Input: *a sequence of n^3 unsorted elements distributed to an $n \times n \times n$ array, one element per processor.*
Output: *a sequence of n^3 elements stored into zyx-order, one element per processor.*
Begin

1. **Phase 1**. sort the elements within each xz-plane into zx-order.
2. **Phase 2**. sort the elements within each yz-plane into zy-order.
3. **Phase 3**. sort the elements within each xy-plane into yx-order, but reverse the order on every other plane.
4. **Phase 4**. do two steps of odd-even transposition sort to each z-line.
5. **Phase 5**. sort the elements within each xy-plane into yx-order.

End

We reiterate the correctness proof here for the completeness of this algorithm. Since the algorithm is an oblivious comparison-exchange algorithm, it is sufficient to prove the correctness using Lemma 5.3.1 *The 0-1 Sorting Lemma*. A dirty line is defined to be a line with both 0 and 1. The argument goes as follows:

At the end of Phase 1, since each xz-plane is sorted in zx-order, there is at most one dirty x-line in each xz-plane. Hence, the number of 1s for any two z-lines in a xz-plane differs at most 1. Since each yz-plane contains n z-lines, the number of 1s for any two yz-plane can differ at most n after Phase 1.

After Phase 2, each yz-plane is sorted into zy-order. Since the number of 1s for any two yz-plane differs at most n, the number of all-1 and all-0 y lines between any two yz-plane can differ at most 1. Since there is at most one dirty y-line in each yz-plane and this dirty line is between all-0 lines and all-1 lines, we can conclude that there are at most two dirty xy-planes at the end of phase 2, and the two dirty planes are consecutive.

Phase 3 sorts the two consecutive dirty xy-planes in reverse orders which makes it possible for Phase 4 to merge those two dirty xy-planes using odd-even transposition sort so that only one dirty xy-plane is left.

Phase 5 is used to sort the one dirty xy-plane into yx-order. Since Phase 1 has already sort the z-lines, Phase 5 completes the sorting, i.e., the n^3 elements are sorted in zyx-order.

By using an $O(n)$-time sorting algorithm for two dimensional arrays, the time complexity of 6.2.1 *5-Phase_Sort_on_3D_Array* is $O(n)$.

Observe that only the following two key operations are involved in this algorithm:

1. a two-dimensional sorting operation
2. two steps of one dimensional odd-even transposition operation

The two dimensional sorting operation can be done on a 2D LARPBS in $O(\log n)$ time using either algorithm 5.8.1 *Generalized_Columnsort* or algorithm 5.7.1 *Multiway_Merge_Sort*, and two steps of the one dimensional odd-even transposition sort can be done in constant time which we have already shown in previous section. Thus we have a 5-phrase $O(\log n)$ sorting algorithm that sorts n^3 elements on an $n \times n \times n$ LARPBS. Here is the algorithm.

Algorithm 6.2.2 5-Phase_Sort_on_3D_LARPBS
Input: *a sequence of n^3 unsorted elements distributed to an $n \times n \times n$ 3D LARPBS, one element per processor.*
Output: *a sequence of n^3 elements stored in zyx-order, one element per processor.*
Begin

1. **Phase 1**. run algorithm 5.8.1 *Generalized_Columnsort* in each xz-plane to sort the elements into zx-order.
2. **Phase 2**. run algorithm 5.8.1 *Generalized_Columnsort* in each yz-plane to sort the elements into zy-order.
3. **Phase 3**. run algorithm 5.8.1 *Generalized_Columnsort* in each xy-plane to sort the elements into yx-order, but reverse the order in every other plane.
4. **Phase 4**. do two steps of *odd-even-transposition-sort* operation on each z-bus.
5. **Phase 5**. run algorithm 5.8.1 *Generalized_Columnsort* in each xy-plane to sort the elements into yx-order.

End

Since the only non-constant time operation performed in the above algorithm is algorithm 5.8.1 *Generalized_Columnsort*, we obtain the following theorem:

Theorem 15 *Algorithm 6.2.2* **5-Phase_Sort_on_3D_LARPBS** *sorts n^3 elements on an $n \times n \times n$ 3D LARPBS in $O(\log n)$ time.*

Example 6.2.1 *5-Phase_Sort_on_3D_LARPBS on a $4 \times 4 \times 4$ LARPBS.*

$$\begin{bmatrix} 22 & 28 & 32 & 49 \\ 47 & 5 & 11 & 2 \\ 41 & 44 & 52 & 36 \\ 13 & 61 & 1 & 55 \end{bmatrix} \begin{bmatrix} 7 & 54 & 46 & 16 \\ 53 & 34 & 29 & 19 \\ 25 & 60 & 62 & 59 \\ 31 & 24 & 23 & 33 \end{bmatrix} \begin{bmatrix} 39 & 42 & 56 & 6 \\ 10 & 21 & 17 & 58 \\ 15 & 3 & 20 & 35 \\ 38 & 12 & 48 & 64 \end{bmatrix} \begin{bmatrix} 50 & 8 & 43 & 40 \\ 27 & 57 & 9 & 18 \\ 63 & 26 & 14 & 51 \\ 45 & 4 & 30 & 37 \end{bmatrix}$$

Input
$xz-planes$

$$\begin{bmatrix} 1 & 13 & 36 & 49 \\ 2 & 22 & 41 & 52 \\ 5 & 28 & 44 & 55 \\ 11 & 32 & 47 & 61 \end{bmatrix} \begin{bmatrix} 7 & 24 & 33 & 54 \\ 16 & 25 & 34 & 59 \\ 19 & 29 & 46 & 60 \\ 23 & 31 & 53 & 62 \end{bmatrix} \begin{bmatrix} 3 & 15 & 35 & 48 \\ 6 & 17 & 38 & 56 \\ 10 & 20 & 39 & 58 \\ 12 & 21 & 42 & 64 \end{bmatrix} \begin{bmatrix} 4 & 18 & 37 & 50 \\ 8 & 26 & 40 & 51 \\ 9 & 27 & 43 & 57 \\ 14 & 30 & 45 & 63 \end{bmatrix}$$

Step 1
$zx-ordered$
$xz-planes$

$$\begin{bmatrix} 1 & 13 & 36 & 49 \\ 7 & 24 & 33 & 54 \\ 3 & 15 & 35 & 48 \\ 4 & 18 & 37 & 50 \end{bmatrix} \begin{bmatrix} 2 & 22 & 41 & 52 \\ 16 & 25 & 34 & 59 \\ 6 & 17 & 38 & 56 \\ 8 & 26 & 40 & 51 \end{bmatrix} \begin{bmatrix} 5 & 28 & 44 & 55 \\ 19 & 29 & 46 & 60 \\ 10 & 20 & 39 & 58 \\ 9 & 27 & 43 & 57 \end{bmatrix} \begin{bmatrix} 11 & 32 & 47 & 61 \\ 23 & 31 & 53 & 62 \\ 12 & 21 & 42 & 64 \\ 14 & 30 & 45 & 63 \end{bmatrix}$$

Step 2
$yz-planes$

$$\begin{bmatrix} 1 & 13 & 33 & 48 \\ 3 & 15 & 35 & 49 \\ 4 & 18 & 36 & 50 \\ 7 & 24 & 37 & 54 \end{bmatrix} \begin{bmatrix} 2 & 17 & 34 & 51 \\ 6 & 22 & 38 & 52 \\ 8 & 25 & 40 & 56 \\ 16 & 26 & 41 & 59 \end{bmatrix} \begin{bmatrix} 5 & 20 & 39 & 55 \\ 9 & 27 & 43 & 57 \\ 10 & 28 & 44 & 58 \\ 19 & 29 & 46 & 60 \end{bmatrix} \begin{bmatrix} 11 & 21 & 42 & 61 \\ 12 & 30 & 45 & 62 \\ 14 & 31 & 47 & 63 \\ 23 & 32 & 53 & 64 \end{bmatrix}$$

Step 2
$zy-ordered$
$yz-planes$

$$\begin{bmatrix} 1 & 3 & 4 & 7 \\ 2 & 6 & 8 & 16 \\ 3 & 7 & 11 & 19 \\ 4 & 8 & 12 & 23 \end{bmatrix} \begin{bmatrix} 13 & 15 & 18 & 24 \\ 17 & 22 & 25 & 26 \\ 20 & 27 & 28 & 29 \\ 21 & 30 & 31 & 32 \end{bmatrix} \begin{bmatrix} 33 & 35 & 36 & 37 \\ 34 & 38 & 40 & 41 \\ 39 & 43 & 44 & 46 \\ 42 & 45 & 47 & 53 \end{bmatrix} \begin{bmatrix} 48 & 49 & 50 & 54 \\ 51 & 52 & 56 & 59 \\ 55 & 57 & 58 & 60 \\ 61 & 62 & 63 & 64 \end{bmatrix}$$

Step 3
$xy-planes$

$$\begin{bmatrix} 1 & 5 & 9 & 14 \\ 2 & 6 & 10 & 16 \\ 3 & 7 & 11 & 19 \\ 4 & 8 & 12 & 23 \end{bmatrix} \begin{bmatrix} 32 & 28 & 24 & 18 \\ 31 & 27 & 22 & 17 \\ 30 & 26 & 21 & 15 \\ 29 & 25 & 20 & 13 \end{bmatrix} \begin{bmatrix} 33 & 37 & 41 & 45 \\ 34 & 38 & 42 & 46 \\ 35 & 39 & 43 & 47 \\ 36 & 40 & 44 & 53 \end{bmatrix} \begin{bmatrix} 64 & 60 & 56 & 51 \\ 63 & 59 & 55 & 50 \\ 62 & 58 & 54 & 49 \\ 61 & 57 & 52 & 48 \end{bmatrix}$$

Step 3
$yx-ordered$
$xy-planes$

$$\begin{bmatrix} 1 & 5 & 9 & 14 \\ 2 & 6 & 10 & 16 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 13 \end{bmatrix} \begin{bmatrix} 32 & 28 & 24 & 18 \\ 31 & 27 & 22 & 17 \\ 30 & 26 & 21 & 19 \\ 29 & 25 & 20 & 23 \end{bmatrix} \begin{bmatrix} 33 & 37 & 41 & 45 \\ 34 & 38 & 42 & 46 \\ 35 & 39 & 43 & 47 \\ 36 & 40 & 44 & 48 \end{bmatrix} \begin{bmatrix} 64 & 60 & 56 & 51 \\ 63 & 59 & 55 & 50 \\ 62 & 58 & 54 & 49 \\ 61 & 57 & 52 & 53 \end{bmatrix}$$

Step 4
$odd-even-sorted$
$xy-planes$

$$\begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix} \begin{bmatrix} 17 & 21 & 25 & 29 \\ 18 & 22 & 26 & 30 \\ 19 & 23 & 27 & 31 \\ 20 & 24 & 28 & 32 \end{bmatrix} \begin{bmatrix} 33 & 37 & 41 & 45 \\ 34 & 38 & 42 & 46 \\ 35 & 39 & 43 & 47 \\ 36 & 40 & 44 & 48 \end{bmatrix} \begin{bmatrix} 49 & 53 & 57 & 61 \\ 50 & 54 & 58 & 62 \\ 51 & 55 & 59 & 63 \\ 52 & 56 & 60 & 64 \end{bmatrix}$$

Step 5
$zyx-ordered$
$xy-planes$

6.3 Comparison of The Two 3D Sorting Algorithms

We have presented two three-dimensional sorting which achieve the same time complexity. Now let's compare the two algorithms:

In Algorithm 6.1.1 *3D_Generalized_Columnsort*, all sorting are done in the xy-plane. Since only the sorting operations require reconfigurable switches, we can eliminate reconfigurable switches on z-buses. But this algorithm runs in seven phases. The trade-off for reducing one set of reconfigurable switches is to introduce several transpose operations which can be done in constant time.

In Algorithm 6.2.2 *5-Phase_Sort_on_3D_LARPBS*, we need to sort elements in the xy-plane, the yz-plane, and the xz-plane. Thus reconfigurable switches are needed for all three buses. But this algorithm is simpler in terms of the number of phases involved. With only 5 phases, this algorithm completes the sorting.

To Summarize, algorithm 6.1.1 *3D_Generalized_Columnsort* can run on a simpler 3D LARPBS model, but requires several extra constant time operations; algorithm 6.2.2 *5-Phase_Sort_on_3D_LARPBS* runs in fewer phases but requires a more complicated 3D LARPBS.

Chapter 7

Selection Problems

7.1 Introduction

The selection problem is to determine the kth largest element in a given set of n elements. Parallel sorting trivially solves the selection problem, but sorting is known to be computationally harder than selection. The easy cases are extreme selection where $k = 1$ or $k = n$, and the hard case is the median, where $k = \lceil n/2 \rceil$. Valliant proved the lower bound of finding the maximum to be $\Omega(\log \log n)$ in [76]. This implies that the lower bound of selection problem is $\Omega(\log \log n)$. Cole and Yap [16] present a selection algorithm on parallel comparison model. This selection algorithm brings the upper bound of selection problem from $O(\log n)$ down to $O((\log \log n)^2)$. Later, a even tighter upper bound selection algorithm [3] is presented that pushes the upper bound further down to $O(\log \log n)$. The existence of such an algorithm implies that selection problem has an upper and lower bound of $\log \log n$. The best previous result for selection on the PRAM is given in [13] that uses n operations and runs in $O(logn \log \log n)$ on a CRCW PRAM.

Previous research results for parallel selection problem on LARPBS model are the following: Pan gave a randomized algorithm which runs in $\tilde{O}(n \log n/p)$ expected time using p processors [51]. Li et al. presented a deterministic selection algorithm that runs in $O(\log n)$ time on an n-processor LARPBS. Han and Pan described an $O((\log \log n)^2 / \log \log \log n)$ deterministic selection algorithm on an n-processor LARPBS [24]. Research results for selection problems on other parallel computing models are listed in Table 7.1.

The rest of this chapter is organized as follows. In section 7.2, we give a definition on parallel comparison model. In section 7.3, we study the extreme selection problem, i.e. finding the maximum. In section 7.4, we do research on two upper bound algorithms and implement the Cole-Yap's algorithm on the LARPBS model. In section 7.5, we provide an application of our selection algorithm.

Table 7.1: Comparison of different parallel selection algorithms.

Algorithm	**Model**	**Time Complexity**	**Processor**
Cole-Yap's[16]	PCT	$O((\log\log n)^2)$	n
An optimal algorithm[3]	PCT	$O(\log\log n)$	n
Li et al	LARPBS	$O(\log n)$	n
Our LARPBS algorithm	LARPBS	$O((\log\log n)^2)$	n
Han and Pan [24]	LARPBS	$O((\log\log n)^2/\log\log\log n)$	n
Cole [12]	EREW PRAM	$O(\log n \log^* n)$	n
Chaudhuri et al.[13]	CRCW PRAM	$O(\log n/\log\log n)$	n
Pan [51]	LARPBS	$\tilde{O}(n\log n/p)$	p
Rajasekaran & Sahni [68]	AROB	$\tilde{O}(1)$	n^2

7.2 Parallel Comparison Model

Valliant first introduced parallel comparison model in [76] for the purpose of studying comparison dominated parallel problems. Since then, a lot of researchers used this model as a tool to study the parallelism in comparison problems. On the parallel comparison model, we only count the time taken to carry out comparisons, and ignore time spent in communications, in determining which comparisons to do next, and any other side computations. Thus any lower bound on the time complexity of solving a problem on this model will necessarily also be a lower bound for any other model of parallelism that has binary comparisons as the basic operations. And the best constructive upper bound will correspond to the fastest algorithm for comparison dominated problems. The common properties of such problems are: for each problem, the input consists of a set of elements on which there is a linear ordering. By performing a comparison operation on any pair of elements, we can find the ordering relationship between these two elements. The following are some problems that can be studied using the parallel comparison model: finding the maximum, finding the medium, sorting, and merging two sorted lists. The following definition for parallel comparison model comes from [76].

Definition 7.2.1 Parallel comparison model
There are k processors available in the model, so k comparisons can be performed simultaneously. The processors are synchronized so that each of them completes a comparison within each time interval. At the end of the interval the algorithm decides which k (not necessarily disjoint) pairs of elements are to be compared during the next interval and have processors assigned to them. The decision mentioned above is made by inspecting the ordering relationships that have already been established in previous comparisons. The computation terminates when the relationships that have been discovered are sufficient to specify the solution to the given problem. The time complexity of each problem will be expressed as a function of the number of processors, and of the size of the input set.

7.3 Finding Maximum

7.3.1 A Constant-Time Algorithm

This section gives a constant time algorithm for finding the maximum of n elements using n^2 processors on LARPBS. Assume that the n elements are initially distributed to the first n processors.

Algorithm 7.3.1 Basic_Maximum
Input: *An unsorted array A of n distinct elements.*
Output: *the maximum element x stored in the first processor.*
Begin

1. Distribute the elements on the first n processors in the following way: processor P_i multicast its element x to processor P_{kn+i}, where $1 \leq k < n$, and $1 \leq i \leq n$.

2. Segment the bus into n sub-buses so that each has n processors. Processor P_{in+i}, where $1 \leq i \leq n$, sets $y = x$ and broadcasts y to every other processor in its sub-bus.

3. Each processor in the system compares its x and y. If $x > y$, then sends a dummy message to the first processor in its sub-bus. Then all processors set their reconfigurable switches to straight to fuse sub-buses into one bus.

4. The first processor of a sub-bus who does not receive anything in previous bus cycle holds the maximum element y. So it sends its y to the first processor in the system.

End

The above algorithm runs in constant time with n^2 processors. Although the number of processors is not appealing to us, the constant time complexity is very appealing. By using a doubly logarithmic tree structure, the algorithm can be improved so that the number of processors is reduced to $O(n)$ and running time reaches lower bound $\Omega(\log \log n)$ [28].

7.3.2 A Fast Non-Optimal Algorithm

A fast algorithm [28] that reaches the lower bounds $\Omega(\log \log n)$ of the maximum finding problem can be obtained on a CRCW PRAM based on the idea of the doubly logarithmic-depth tree [28].

Definition 7.3.1 *A doubly logarithmic tree is a balanced tree with the following property the number of children of an internal node u is equal to $\lceil \sqrt{n_u} \rceil$, where n_u is the number of leaves in the subtree rooted at u.*

We implement the same idea on an LARPBS as one of the comparison algorithms. Two key issues need to be considered for developing this fast maximum algorithm:

1. how to simulate the doubly logarithmic tree on an LARPBS model; and

2. how to perform the computation needed and how to move intermediate results around for the computation performed in the next stage.

Our solutions are:

1. *Simulating the doubly logarithmic tree on an LARPBS* : At the beginning of the algorithm, elements are distributed to the n processors, one for each processor. As the algorithm goes from stage to stage, we segment the bus into sub-buses in the following way: on the first stage, each sub-bus has size 2; on the second stage, each sub-bus has size $2^2 = 4$; on the third stage, each sub-bus has size $2^4 = 16$. In general, on the i^{th} stage, where $0 \leq i \leq \log\log n$, each sub-bus has size 2^{2^i}. So when the algorithm reaches the $\log\log n$ stage, the system consists only one bus which has a size of $2^{\log n} = n$.

2. *Computation and communication* : The algorithm proceeds in $O(\log\log n)$ stages. On the first stage, a binary tree comparison is used to reduce number of candidates by a factor of 2. On each stage that follows, the system is segmented into multiple sub-buses based on the above segmentation rule. After segmentation is done, do a *compression* operation to compress results from the previous stage to the left most processors of the sub-bus. Then each sub-bus calls Algorithm 7.3.1 *Basic_Maximum* which runs in constant time. The reason why Algorithm 7.3.1 *Basic_Maximum* can be used starting from the second stage is as follows: in the second stage, the number of candidates on each sub-bus is $m' = 2$, and the number of processors on each sub-bus is $n' = 4$; in the third stage, the number of candidates on each sub-bus is $m' = 4$, and the number of processors on each sub-bus is $n' = 16$; etc. So in general, in the $(i+1)^{th}$ stage, the number of candidates on each sub-bus is $m' = 2^{2^i}$, and the number of processors on each sub-bus is $n' = 2^{2^{(i+1)}}$. Thus we have

$$n' = 2^{2^{(i+1)}} = 2^{2\times 2^i} = (2^{2^i})^2 = m'^2$$

Assume that n elements are initially distributed to n processors, one element per processor. Here is the detail of the algorithm:

Algorithm 7.3.2 Fast_Maximum
Input*: An array A of n distinct elements.*
Output*: the maximum element x stored in the first processor.*
Begin

1. Each processor P_{2i}, where $0 < i \leq n/2$, sends its element to processor P_{2i-1}. Processor P_{2i-1} sets itself to active, compare the element it received with its own, and keeps the larger one.

2. **for** $i = 1$ **to** $\log\log n$ **pardo**

(a) Segment the array into $n/2^{2^i}$ sub-buses: each processor P_j sets its reconfigurable switches to cross if $j = k2^{2^i}$, where $1 \leq k \leq n/2^{2^i}$. All other processors set their reconfigurable switches to straight. Each sub-bus does a *ordered-compression* operation to compress all elements held by the active processors to the left most processors of the sub-bus.

(b) Each sub-bus calls Algorithm7.3.1 *Basic_Maximum* and sets the processor who holds the maximum element for a sub-bus to be active.

End

This algorithm is very fast, it reaches the lower bound of maximum finding problem. But it is not optimal as the operations required is $O(n \log\log n)$. By using the accelerated cascading technique, we can obtain a fast optimal algorithm[28].

7.3.3 An Optimal-Algorithm

Definition 7.3.2 *The Accelerated Cascading Strategy is an algorithm design strategy that starts with the optimal algorithm until the size of the problem is reduced to a certain threshold value, then shifts to the fast but non-optimal algorithm.*

The accelerated cascading strategy is usually used to make a fast non-optimal algorithm optimal. Here is how it is used to make our fast maximum finding algorithm optimal:

Algorithm 7.3.3 Optimal_Maximum
Input*: An array A of n distinct elements.*
Output*: the maximum element x stored in the first processor.*
Begin

1. Perform $\log\log\log n$ steps of the binary tree maximum algorithm.

2. Run algorithm 7.3.2 *Fast_Maximum* for the remaining candidates.

End

The algorithm called in the first step is optimal but not fast, the running time is $O(\log\log\log n)$. The number of operations required in Step 1 is $O(n)$, and the number of operations required in Step 2 is $O((n/\log\log n)\log\log(n/\log\log n)) = O(n)$. Thus the total number of operations required is reduce to $O(n)$. Obviously Step 1 requires $O(\log\log\log n)$ computation time. As at the end of Step 1, the size of the array is reduced to $n/\log\log n$, Step 2 only requires $O(\log\log(n/\log\log n)) = O(\log\log n)$ time. Thus algorithm 7.3.3 *Optimal_Maximum* is both very fast and optimal.

7.4 Selection

7.4.1 The Cole-Yap $O((\log\log n)^2)$ Time Parallel Selection Algorithm

Cole and Yap [16] present a well-known $O((\log\log n)^2)$ time parallel selection algorithm for parallel comparison model. This algorithm uses $O(n)$ processors. The Cole-Yap algorithm proceeds in $O(\log\log n)$ stages. In each stage, it finds two approximations to the k^{th} smallest item e: one is guaranteed to be no larger than e, and the other is guaranteed to be no smaller than e. Thus, candidate elements can be reduced by eliminating all elements that fall outside the interval defined by these two approximations. The approximation calculation performed in each stage of the selection algorithm proceeds by sampling the surviving set of elements and recursively find the approximation for the k^{th} element in the sample set. By smartly sampling the remaining elements, the size of the surviving set shrinks in a doubly logarithmic rate. Thus each call to the approximate calculation takes $O(\log\log n)$ time, which brings the total time required for selection algorithm to $O((\log\log n)^2)$.

Now we need to prove why the approximation strategy mentioned above correctly selects the samples. A brief proof for lemma 7.4.1 is given in [16]. We give a more detailed proof in the following paragraphs.

Lemma 7.4.1 *For $j \leq 2m/r$, the $(\lfloor j/r \rfloor - s)^{th}$ smallest element in S_{i+1} is no larger than the j^{th} smallest element in S_i, and is at least the $(\lfloor j/r \rfloor r - sr)^{th}$ smallest element in S_i.*

Proof. Assume that the number of processors is n, and the number of elements in S_i is m, i.e., $|S_i| = m$. Let $r = \lceil \sqrt{\frac{n}{m}} \rceil$. In stage i of the selection algorithm, the S_i elements is divided into $s = m/r^2$ short sets, each has r^2 elements. As S_{i+1} consists of the $(rk)^{th}$ elements, for $k = 1, 2, \cdots, r$, for each of the sorted short sets, the number of elements sampled into S_{i+1} in each of short set is r.

First we prove that after sorting those short sets, and forming the surviving set S_{i+1}, we have the following upper and lower approximation for the k^{th} element in S_{i+1}:

- *upper approximation*: the k^{th} smallest element in S_{i+1} is at most the $(kr + s(r-1))^{th}$ smallest element in S_i.
- *lower approximation*: the k^{th} smallest element in S_{i+1} is at least the kr^{th} smallest element in S_i.

Suppose x and y are the $(l-1)r^{th}$ and the lr^{th} element in short set V, where $1 < l \leq r$. We associate with element y those elements in V lying between x and y, plus y, and denote it as a sorted array $L(y)$. So each element z in S_{i+1} is associated with r elements $L(z)$ in S_i, and z is the largest of the r elements in $L(z)$. There are rs such sorted arrays. Assume the k^{th} element is e. If $x \leq e$, then all the elements in $L(x)$ are not greater than e. Therefore there are at least kr elements in S_i that are less than or equal to e. Thus the lower approximation of the k^{th} smallest element in S_{i+1} is the kr^{th} element in S_i.

Now we calculate the upper approximation: since we have already taken care of all $L(x)$'s for $0 < x \leq e$, let's consider all $L(y)$'s, where $e < y \leq r$. Notice that each short set is sorted. So no matter how many samples in a short set are greater than e, at most $r-1$ of all the elements associated with the samples can be smaller than e. To be specific, only the elements in $L(y)$ (not including y), where y is the smallest sample in a short set that is greater than e, can be smaller than e. Thus a maximum of $r-1$ elements associated with samples greater than e in one short set can be smaller than e. Since there are s short sets, the maximum number of elements who are associated with samples greater than e and who themselves are smaller than e is $s(r-1)$. So the total maximum number of elements in S_i that is smaller than e is: $kr + s(r-1)$.

To prove the two claims stated in the lemma, we do the following:

For $j \geq 2m/r$, let $k = \lfloor \frac{j}{r} \rfloor - s$, the upper approximation for the rank of k^{th} element e in S_i is

$$kr + s(r-1) = (\lfloor \frac{j}{r} \rfloor - s)r + s(r-1) < (\lfloor \frac{j}{r} \rfloor - s)r + sr < j$$

and the lower approximation for the rank of k^{th} element e in S_i is

$$kr = (\lfloor \frac{j}{r} \rfloor - s)r = \lfloor \frac{j}{r} \rfloor r - sr$$

Thus the lemma is proved.

The reason why sorting each short set in each stage of the approximate algorithm takes constant time is that the number of processors assigned to each short set is

$$n_p = \frac{n}{m} \times r^2 = \frac{n}{m} \times \frac{n}{m} = (\frac{n}{m})^2$$

and number of elements in each short set is

$$n_i = \frac{n}{m}$$

So we have $n_p = {n_i}^2$, and sorting can be done in constant time using algorithm 4.2.1 *Constant_Time_Sort*.

7.4.2 An Optimal Parallel Selection Algorithm

An $O(\log \log n)$ parallel selection algorithm was presented in [3], which proves the upper bound of parallel selection problem to be $O(\log \log n)$, and establishes that all comparison tasks (selection, merging, sorting) have upper and lower bounds of the same order in both random and deterministic models. This algorithm is designed on Valian's PCT model, i.e., the parallel comparison tree model.

There are two basic ideas underlying the algorithm:

1. Use the transitivity of the relation R, obtained free of comparison cost, to achieve a constant time approximation in each stage of the selection algorithm. R refers to the relation obtained after each comparison.

2. Using a weak $(A, \alpha) - expander$ graph to make sure that if $m \leq n/(\log n)^3$, the upper and lower approximations l and r are quite accurate for most elements of V and l and r can be determined in constant time.

Similar to Cole-Yap's selection algorithm, this optimal selection algorithm uses a procedure called *SELECT* which, given $V = v_1, \cdots, v_m$, will return the k^{th} smallest element v_k. *SELECT* proceeds in $O(\log \log n)$ stages. On each stage, an upper approximation and a lower approximation for the rank of each elements in the surviving set is calculated in the following way: the n processors will perform n comparisons according to an undirected graph G in the natural way: G has vertices $v_1, \cdots, v_m$ and n of the possible $\binom{m}{2}$ edges. Elements x and y are compared only if there is an edge between them. The n comparisons induce a relation R on $V \times V$. Let $\leq_G$ represent the partial order given by the transitive closure of the relation R. Then the algorithm calculates the upper and lower approximations ρ^+ and ρ^- for the rank of each element based on the transitive closure $\leq_G$ of relation R where ρ^+ and ρ^- are defined to be:

$$\rho^- = |j : v_j \leq_G v_i|$$

and

$$\rho^+ = (m+1) - |j : v_i \leq_G v_j|$$

Lower approximation l and upper approximation r for k is determined using the following two inequalities:

$$j - D \leq {\rho_l}^- \leq {\rho_l}^+ \leq j \tag{7.1}$$

$$j \leq {\rho_r}^- \leq {\rho_r}^+ \leq j + D \tag{7.2}$$

where $D = (20m \log m)/A$.

Then form a new surviving set $V' = \{u \in V : l \leq u \leq r\}$. It can be done in two parallel steps: the m processors compare its estimated rank simultaneously with l and r to decide whether the element associated with it is a survival element or not. The graph G that the *SELECT* procedure uses is called a weak (A, α)-expander with m vertices and n edges, where $m \leq n/(\log n)^3$. The property of the weak (A, α)-expander and the transitive closure of the n comparison results guarantee the existence of such l and r whose upper and lower approximate ranks given by the transitive closure of R satisfy both inequality 7.2 and inequality 7.2. By choosing $A = n/(m \log n)$, we have

$$D = (20m \log m)/A = (20m \log m)/(n/(m \log n)) = (20m^2 \log m \log n)/n \leq (20(m \log n)^2)/n$$

Thus

$$|V'| \leq 40|V|(\log n)^2/(n/|V|)$$

Therefore, if

$$|V| \leq n/(\log n)^{1+\delta}$$

then

$$|V'| \leq 40|V|/(\log n)^{2\delta}$$

At this rate of shrinking, $O(\log\log n)$ recursive calls to *SELECT* will force the surviving set V' to reach a size of at most $\sqrt{n}$. Now as $m < \sqrt{n}$, the surviving set V' can be sorted in constant time. But in order to have $|V| \leq n/(\log n)^{1+\delta}$, a preprocessing is needed to shrink the original set size to the desired one. Now AKS sorting algorithm comes into play.

The following property of the AKS sorting algorithm is implicit in the argument in [61]: After running $O(\log\log n)$ stages, $\frac{9}{10}$ of the elements being sorted will be within m of $k+m$. So the preprocessing consists of the following function call sequence:

1. $O(\log\log n)$ steps of AKS sorting algorithm
2. $O(\log\log n)$ recursive calls to *SELECT*
3. another $O(\log\log n)$ steps of AKS sorting algorithm

After the above $O(\log\log n)$ preprocessing, the size of the surviving set can be reduced to $n/(\log n)^3$ or less, without increasing the overall time complexity. After preprocessing, $O(\log\log n)$ recursive calls to *SELECT* will reduce the size for surviving set to $\sqrt{n}$ or less, and then we find out the j^{th} element by sorting the surviving set. Thus the overall time complexity of this algorithm is $O(\log\log n)$. Notice that only the time spent for comparison is counted in parallel comparison model, thus the following key computation is considered to be available free of time cost: computing the transitive closure of relation R.

Also this algorithm does not count the time for computing the expander graph. It assumes the expander graphs used by the algorithm are known and just feed the algorithm with the required expander graphs. This algorithm is not uniform in a sense because different expander graphs are required for different n.

We did some work implementing this optimal parallel selection algorithm on the LARPBS model. We found that all the operations in this algorithm can be implemented on the LARPBS model with the same time complexity except the calculation for transitive closure. Our best transitive closure algorithm runs in $O(\log n)$ time. If we use this transitive closure algorithm, the overall time complexity of the selection algorithm would be $O(\log n \log\log n)$, which is slower than the Cole-Yap's selection algorithm. So we conclude that this optimal parallel selection algorithm can be implemented on the LARPBS with same time complexity only if there is a constant time transitive closure algorithm available for this model.

7.4.3 An Implementation of Cole-Yap's Algorithm on an LARPBS

The basic operations needed for implementing the Cole-Yap's $O((\log\log n)^2)$ selection algorithm are:

1. Sort multiple sub-sequences of length $O(m)$ in constant time using $O(m^2)$ processors.
2. Sample a set to form a subset for further processing.
3. Find out all elements that are greater or smaller than a given element.

4. Compress selected elements to one end of the bus.

Operation 1 can be done using algorithm 4.2.1 *Constant_Time_Sort* designed for LARPBS. Operation 2 and 3 can be done in constant time. And operation 4 can be done by the basic operation *order-compression*. Now we present the implementation details of the $O((\log\log n)^2)$ selection algorithm. Assume we have $n = 4096m$ processors, and the n elements in A are initially distributed to the first m processors. Here are some data that will be used in the algorithm:

- *number of processors*: n
- *number of short sets*: $s = \lceil \frac{m}{r^2} \rceil = \frac{m^2}{n}$
- *number of elements in each short set*: $r^2 = \lceil \frac{n}{m} \rceil$
- *number of processors assigned to each short set*: $\frac{n}{s} = \frac{n^2}{m^2}$

Algorithm 7.4.1 Find_Lower_Approximation(n,m,k)
Input: *A sequence L of m distinct elements distributed to the first m processors in an n-processor system, one per processor.*
Output: *lower approximation l for the k^{th} element. l is stored in the first processor, which is called the header processor.*
Begin

if $n \geq m^2$, apply algorithm 4.2.1 *Constant_Time_Sort* to find the k^{th} element directly, and processor P_k sends its element to the header processor.
else

1. *Calculate s and r^2*: The header processor performs the following calculation:
   - *Calculate number of short sets*: $s = \lceil \frac{m}{r^2} \rceil = \frac{m^2}{n}$
   - *Calculate number of elements in each short set*: $r^2 = \lceil \frac{n}{m} \rceil$
   - *number of processors assigned to each short set*: $p = \frac{n}{s} = \frac{n^2}{m^2}$
2. *Partition List L*: each processor P_i who initially holds an element, calculates the destination processor address j using $j = i \times r + i \bmod r$. Then P_i sends its element to processor P_j. After this *one-to-one* communication operation, processors P_{lp}, where $1 \leq l \leq s$, set their switches to cross to form s sub-buses.
3. *Sort short sets*: apply algorithm 4.2.1 *Constant_Time_Sort* on each sub-bus to sort each short set in increasing order.
4. *Identify sample elements*: In each sub-bus, processor P_i sets itself to active if its rank obtained in the previous step is jr, where $0 < j \leq r$.

5. *Form surviving set*: All processors set their switches to straight to form a single bus. Do a *order-compression* operation to move all surviving elements to the beginning of the bus.

6. *Update k*: if $k \leq 8m/r$, then set $k = 0$; otherwise, set $k = \lfloor k/r \rfloor - s$.

7. *Update m*: set $m = m/r$

8. *recursively find l*: do *Find_Lower_Approximation(n,m,k)*

endif
End

The value of j in step 2 comes from the following calculation:

$$j = \frac{i}{r^2} \times \frac{n}{s} + i \bmod r^2 = \frac{i}{n/m} \times (n^2/m^2) + i \bmod (n/m) = i \times r + i \bmod r$$

Finding the upper approximation is a symmetric operation, the only difference is in the calculation of the new rank k, see details below:

Algorithm 7.4.2 Find_Upper_Approximation(n,m,k)
Input: *An sequence L of m distinct elements. Assume the m elements are distributed to the first m processors in the n-processor system, one per processor.*
Output: *upper approximation u for the k^{th} element. u is stored in the first processor, which is called the header processor.*
Begin

if $n \geq m^2$, apply algorithm 4.2.1 *Constant_Time_Sort* to find the k^{th} element directly, and processor P_k sends its element to the header processor.
else

1. *Calculate s and r^2*: The header processor performs the following calculation:

   - *Calculate number of short sets*: $s = \lceil \frac{m}{r^2} \rceil = \frac{m^2}{n}$
   - *Calculate number of elements in each short set*: $r^2 = \lceil \frac{n}{m} \rceil$
   - *number of processors assigned to each short set*: $p = \frac{n}{s} = \frac{n^2}{m^2}$

2. *Partition array A*: each processor P_i who initially holds an element, calculates the destination processor address j using $j = i \times r + i \bmod r$. Then P_i sends its element to processor P_j. After this *one-to-one* communication operation, processors P_{lp}, where $1 \leq l \leq s$, set their switch to cross to form s sub-buses.

3. *Sort short sets*: apply algorithm 4.2.1 *Constant_Time_Sort* on each sub-bus to sort each short set in increasing order.

4. *Identify sample elements*: In each sub-bus, processor P_i sets itself to active if its rank obtained in the previous step is jr, where $0 < j \leq r$.

5. *Form sample sets*: All processors set their switches to straight to form a single bus. Do a *order-compression* operation to move all sampled elements to the beginning of the bus.

6. *Update k*: if $k \geq m - 8m/r$, then set $k = m$; otherwise, set $k = k/r$.

7. *Update m*: set $m = m/r$

8. *recursively find u*: do *Find_Upper_Approximation(n,m,k)*

endif
End

Algorithm 7.4.3 Select(n,m,k)
Input*: An array A of m distinct elements. Assume the m elements are distributed to the first m processors in the system, one per processor.*
Output*: the k^{th} element x stored in the first processor, which is called the header processor.*
Begin

if $n \geq m^2$, apply algorithm 4.2.1 *Constant_Time_Sort* to find the k^{th} element directly, and processor P_k send its element to the header processor.
else

1. *Calculate the lower and upper approximation l and u*:

   - Do $Find\_Lower\_Approximate(n, m, k)$ to find lower approximation l.
   - Do $Find\_Upper\_Approximate(n, m, k)$ to find upper approximation u.

   After the above two operations, header processor P_0 has both the upper approximation u and the lower approximation l. Then P_0 broadcasts u and l to the first m processors in the system.

2. *Find the rank r_l and r_u for the lower and upper approximations*: each P_i, where $0 \leq i < m$, compares its own element x with l it received in the previous step, and set $flag = 1$ if $x < l$; otherwise, set $flag = 0$. Processor P_{m-1} set its switch to cross to form a sub-bus that contains the first m processors. Do a *binary-prefix-sums* operation in this sub-bus. Processor P_{m-1} then sends its prefix sums to the header processor. This prefix sums is the rank for l. The header processor sets r_l to be equal to the prefix sums received in previous bus cycle. r_u is obtained in the same way.

3. *Identify surviving elements*: each processor P_i who received the two approximations in previous step, compares its element x with the two approximations, and sets itself to active if $l \leq x \leq u$. Otherwise, set it to inactive.

4. *Form surviving set*: Do a *compression* operation to move all surviving elements to the beginning of the bus.

5. *Update k and m*: set $k = k - r_l$, $m = r_u - r_l + 1$.

6. *Recursively find k*: Apply algorithm 7.4.3 $Select(n, m, k)$.

endif
End

Step 1 takes $O(\log\log n)$ time, all other steps take constant time. The recursive procedure terminates after $\log\log n$ stages. So the total execution time is $O((\log\log n)^2)$.

Theorem 16 *Cole-Yap's selection algorithm can be implemented on an LARPBS in $O((\log\log n)^2)$ time using n processors.*

7.5 Application to Parallel Multi-selection

An immediate application of algorithm 7.4.3 *Select(n,m,k)* is the parallel multi-selection problem. Multi-selection is a fundamental algorithmic problem arising frequently in databases. It is defined as follows: Given an unordered set S of n records and a sequence of m integers $1 \leq q_1 < q_2 < \ldots < q_m \leq n$, we are interested in answering queries of the type "find the q_i^{th} smallest elements in S". The problem is to answer all these queries as fast as possible. In database applications, this problem translates as answering a corresponding set of queries on a record set S. A sequential solution runs in $O(min\{mn, n\log n\})$ time. S. Olariu and Z. Wen proposed an EREW-PRAM solution running in $O(n/p \log i \log 2m)$ time using p processors with $1 \leq p \leq \lceil n/\log n \log^* n \rceil$ [50]. By applying algorithm 7.4.3 *Select(n,m,k)* k times we can obtain an $O(k[\log\log n]^2)$ multi-selection algorithm on an LARPBS.

Chapter 8

Conclusion

In this chapter, we would like to summarize the contributions of this dissertation and point out several topics that we would like to work on in the future.

8.1 Contributions

In this dissertation, we studied both optical interconnection models and efficient algorithms for these models. We focus on a simple but powerful optical interconnection network model – A Linear Array with Reconfigurable Pipelined Bus System (LARPBS). We extended this one-dimensional optical bus model to two and three dimension to improve the scalability. We also provided several basic operations for two-dimensional LARPBS. These basic operations are used as building blocks for designing a few efficient algorithms given in this dissertation.

Then we studied parallel comparison problems, including sorting, merging and selection, the lower bounds for these problems, and the best parallel algorithms for these problems that were designed under PRAM and parallel comparison models. Sorting is a well-known problem that has been studied extensively in the literature because it has so many important applications and it has a theoretical importance. We implement an optimal CREW PRAM merge sort algorithm on an LARPBS. To the best of our knowledge, our $O(\log n)$-time sorting algorithm is the first optimal sorting algorithm implemented on an LARPBS. With this optimal sorting algorithm for one-dimensional LARPBS at disposal, we further extended our research on the sorting problem for higher dimensional LARPBS's.

We obtained our $O(\log n)$-time 7-phase Columnsort algorithm by applying our optimal sorting algorithm on one-dimensional LARPBS to sorting each column. We further generalized our two-dimensional sorting to a larger range of two-dimensional LARPBS's, i.e. $n \times n$ shape two-dimensional LARPBS's. The generalization was done based on the following two ideas:

1. Generalize the idea of basic Columnsort: we can simulate an $n\sqrt{n} \times \sqrt{n}$ imaginary matrix on the $n \times n$ 2D LARPBS, and we can also simulate an $n^2 \times n$ imaginary matrix on the $n \times n \times n$ 3D LARPBS. Since the assumption $r \geq s^2$ is satisfied in both

cases, we can apply the 7-phase Columnsort algorithm on the two imaginary matrices. We called the sorting algorithms obtained this way *generalized Columnsort* algorithms.

2. Multi-way merge sorted sub-matrices: apply a multi-way merge method to $\sqrt{n}$ $n \times \sqrt{n}$ shape sub-matrices to obtain a sorted $n \times n$ shape matrix. We observed that the most time consuming operations needed in the multi-way merge procedure are sorting an n-processor one-dimensional LARPBS and sorting an $n \times 2\sqrt{n}$ 2D LARPBS. We already have an optimal sorting algorithm for one-dimensional LARPBS. If we can obtain an optimal two-way merge sort algorithm, we can get an optimal algorithm for $n \times n$ shape matrix.

The algorithms we designed and implemented for two-dimensional sorting problem are listed below:

1. An optimal basic Columnsort algorithm on an $n \times \sqrt{n}$ two-dimensional LARPBS that sorts $n\sqrt{n}$ elements in $O(\log n)$ time.

2. An optimal two-way merge sort algorithm on an $n \times 2\sqrt{n}$ two-dimensional LARPBS that sorts $2n\sqrt{n}$ in $O(\log n)$ time.

3. An optimal two-way merge sort algorithm on an $n \times \sqrt{n}$ two-dimensional LARPBS that sorts $2n\sqrt{n}$ in $O(\log n)$ time.

4. An optimal multi-way merge sorting algorithm on an $n \times n$ two-dimensional LARPBS that sorts n^2 elements in $O(\log n)$ time.

5. An optimal generalized column sort algorithm on an $n \times n$ two-dimensional LARPBS that sorts n^2 elements in $O(\log n)$ time.

6. An optimal generalized column sort algorithm on an $n \times n \times n$ three-dimensional LARPBS that sorts n^3 elements in $O(\log n)$ time.

7. An optimal 5-phase sorting algorithm on an $n \times n \times n$ three-dimensional LARPBS that sorts n^3 elements in $O(\log n)$ time.

We also did some research on selection problem and its application to multi-selection – a fundamental algorithmic problem arising frequently in databases. The easiest cases are extreme selection, which is finding the minimum or maximum. The hardest case is finding the median. We studied the upper bound and lower bound of selection problem. To find an optimal maximum algorithm, we studied the accelerated cascading technology. We provided a solution for simulating doubly logarithmic tree on an LARPBS, and implemented an optimal algorithm for finding the maximum of n elements on an n-processor LARPBS. We implemented the Cole-Yap's $O((\log \log n)^2)$-time selection algorithm on an LARPBS. Although it is a little bit slower than Han and Pan's selection algorithm [24], it is simpler. Following is a list of algorithms we implemented for selection problems.

1. A constant time algorithm that finds the maximum of n elements on an n^2-processor LARPBS.

2. A non-optimal algorithm that finds the maximum of n elements on an n-processor LARPBS in $O(\log\log n)$ time.

3. An optimal algorithm that finds the maximum of n elements on an n-processor LARPBS in $O(\log\log n)$ time.

4. An $O((\log\log n)^2)$ time parallel selection algorithm on an n-processor one-dimensional LARPBS.

5. An $O(k(\log\log n)^2)$ time parallel multi-selection algorithm on an n-processor one-dimensional LARPBS.

Apart from the comparison problems, we also studied Boolean matrix multiplication problem and its applications to graph problems. We presented a new constant time Boolean matrix algorithm on an LARPBS. Our algorithm achieves much better speed-up compared with the four Russians' algorithm proposed by Agerwala and Lint in [4] and a hypercube implementation provided by C. Gregory Plaxton [55]. We noticed that a parallel implementation of the Four Russians' algorithm on the LARPBS with constant running time is proposed by Keqin Li in [31]. Compared with Li's constant-time algorithm, our algorithm is not based on any existing sequential or parallel algorithm and it is much simpler and uses fewer processors. Our algorithm uses only $O(n^2)$ processors, while Li's algorithm uses $O(n^3/\log n)$ processors.

We then applied the constant time Boolean matrix multiplication algorithm to two graph problems, and obtained an efficient transitive closure algorithm and two connected component algorithms. We noticed that a constant time algorithm for the transitive closure was proposed in [77]. But their constant time algorithm is obtained at a cost of much more processors. Two constant time algorithms were given in [77]. One is designed on a three-dimensional $n \times n \times n$ processor array with a reconfigurable bus system and the other is designed on a two-dimensional $n^2 \times n^2$ processor array with a reconfigurable bus system, where n is the number of vertices in the graph.

Following is a list of results we have obtained in this area.

1. A constant time algorithm on an n^2-processor one-dimensional LARPBS that multiplies two $n \times n$ Boolean matrices.

2. A constant time algorithm on an $n \times n$ two-dimensional LARPBS that multiplies two $n \times n$ Boolean matrices.

3. An $O(\log n)$-time algorithm on an n^2-processor one-dimensional LARPBS that calculates the transitive closure for a undirected graph G with n vertices.

4. An $O(\log n)$-time algorithm on an n^2-processor one-dimensional LARPBS that calculates the connected components for a undirected graph G with n vertices.

5. An $O(\log n)$-time algorithm on an n^2-processor one-dimensional LARPBS that calculates the strongly connected components for a directed graph G with n vertices.

8.2 Future Works

We identified that the following topics are interesting and can be further studied in the future:

1. Extend the generalized Columnsort algorithms to d-dimension: we expect that the same idea of generalized Columnsort algorithms can be applied to a d-dimensional LARPBS. We define a d-dimensional LARPBS to be an $\overbrace{n \times n \times \cdots \times n}^{d}$ LARPBS. Again we assume that communications on different dimension cannot be done in the same bus cycle. This assumption helps us simplify the architecture of the d-diemnsional LARPBS. With this assumption, we can use only one set of switches for all d dimensions. Only the buses that are being used in a bus cycle will be connected to the switches.

2. Extend the multi-way merge algorithm to d-dimension: we expect that the multi-way merge algorithm can be extended to d-dimension.

3. Extend the 5-phase sorting algorithm to d-dimension: we expect that the 5-phase merge sort algorithm can be extended to d-dimension with a combination of wither multi-way merge algorithm or generalized Columnsort algorithm.

4. Improve the selection algorithm: we expect that our $O((\log \log n)^2)$-time selection algorithm on an n-processor LARPBS can be further improved. We have two ideas for improving this algorithm: First, we can use a fast sorting algorithm to help speedup the selection process. Second, we can use the current method for some iteration to reduce the size of the surviving set. At some point, we may have enough processors to perform a constant time transitive closure algorithm provided in [77]. Then we can switch to the optimal selection algorithm given in [3]. The challenges for this improvement will be:

   (a) This constant time transitive closure algorithm provided in [3] is designed on a different model. So it needs to be converted to an LARPBS first.

   (b) Even if we have a constant time transitive closure algorithm, there are still a lot of details needs to be worked out for implementing the $O(\log \log n)$ optimal selection algorithm on an LARPBS.

The results obtained in this dissertation show the strong communication and computation power of optical interconnection networks. The special properties of an optical interconnection network give us a chance to design more efficient algorithms for a lot of problems

in a new and different way, and to implement efficient parallel algorithms that were designed under theoretical parallel models, such as PRAM, parallel comparison model, etc.

Bibliography

[1] M. Ajtai, J. Komlos, and E. Szemeredi, "An $O(n \log n)$ Sorting Network," *Proc. 15th Ann. ACM Symp. On Theory of Computing*, pp.1-9, August 1983.

[2] M. Ajtai, J. Komlos, and E. Szemeredi, "Sorting in $C \log N$ Parallel Steps," *Combinatorica*, vol 3. pp.1-19, 1983.

[3] Miklos Ajtai, Janos Komlos, W. L. Steiger, and Endre Szemeredi, "Optimal Parallel Selection Has Complexity $O(\log \log n)$," *Journal of Computer And System Sciences*, vol.38, pp.125-133, 1989.

[4] T. Agerwala and B. Lint, "Communication in Parallel Algorithm for Boolean Matrix Multiplication," *Proceedings of the 1978 IEEE International Conference on Parallel Processing*, pp.146-153, 1978.

[5] P. Beame and J. Hastad "Optimal Bounds for Decision Problems on the CRCW PRAM," *Journal of the ACM*, vol.36, pp. 643-670, 1989.

[6] A. F. Benner, H. F. Jordan, and V. P. Heuring, "Digital Optical Computing with Optically Switched Directional Couplers," *Optical Engineering*, 30, 12, pp.1936-1941 (1991).

[7] S. H. Bolhari, "Finding Maximum on an Array Processor with a Global Bus," *IEEE Transactions on Computers*, vol.32, pp.133-139, 1984.

[8] Y. Ben-Asher, D. Peleg, R. Ramaswami, A. Shuster, "The Power of Reconfiguration," *Journal of Parallel and distributed Computing*, 13, 1991, pp. 139-153.

[9] A. G. Bourgeois and J. L. Trahan "Relating Two-Dimensional Reconfigurable Meshes with Optically Pipelined Buses," *International Journal of Foundations of Computer Science*, vol.11, no.4, pp. 553-571, 2000.

[10] K. L. Chung, "Generalized Mesh-Connected Computers With Multiple Buses," *Proceedings of International Conference on Parallel and Distributed systems*, pp.622-626, December 1993.

[11] Richard Cole, "Parallel Merge Sort," *SIAM J. Computing*, vol.17, No. 4, pp.1431-1442, August 1988.

[12] R.J. Cole, "An Optimally Efficient Selection Algorithm," *Information Processing Letters* vol.26, pp.295-299, 1987/1988.

[13] S. Chaudhuri, T. Hagerup, and R. Raman, *Computer Science*, Springer-Verlag, pp.352-361, 1993.

[14] D. Chiarulli, R. Melhem, and S. Levitan, "Using Coincident Optical Pulses for Parallel Memory Addressing," *IEEE Computer*, vol.30, pp.48-57, 1987.

[15] A. Chandra, L. Stockmeyer, and U. Vishkin. "Constant Depth Reducibility," *SIAM Journal on Computing*, 13, pp.423-439, 1984.

[16] Richard Cole and Chee K. Yap, "A Parallel Median Algorithm," *Information Processing Letters*, vol.20, pp.137-139, 1985.

[17] E. Dekel, D. Nassimi, and S. Sahni, "Parallel Matrix and Graph Algorithms," *SIAM Journal on Computing*, 10, pp. 657-673, 1981.

[18] J. A. Fernandez-Zepeda, R. Vaidyanathan, and J. L. Trahan "Using Bus Linearization to Scale the Reconfigurable Mesh," *Journal on Parallel and Distributed Computing*, vol.62, no.4, pp. 495-516, 2002.

[19] Paul E. Green, IBM T. J. Watson Research Center, "The Future of Fiber Optic Computer Networks," *IEEE Computer*, September 1991, pp. 78-87.

[20] Z. Guo, "Sorting on Array Processors with Pipelined Buses," Proceedings 1992 International Conference on Parallel Processing, 1992, III, pp. 289-292.

[21] Z. Guo, "Optically Interconnected Processor Arrays With Switching Capability," *Journal of Parallel and Distributed Computing* **23**, pp.314-329, 1994.

[22] Z. Guo, R. Melhem, R. Hall, D. Chiarulli, and S. Levitan, "Array Processors with Pipelined Optical Buses," *Journal of Parallel and Distributed Computing*, vol.12, no. 3, pp. 269-282, 1991.

[23] D, H. Hartman, "Use of Guided Wave Optics for Board Level and Mainframe Level Interconnects," *Proceedings of the 41st Electronic Components and Technology Conference*, June 1991, pp. 463-474.

[24] Yijie Han, Yi Pan, "Sublogarithmic Deterministic Selection on Arrays with a Reconfigurable Optical Bus," *IEEE Transaction on Computers* vol.51, No.6, June 2002.

[25] S. J. Horng, "Prefix Computation and Some Related Applications on Mesh Connected Computers with Hyperbus Broadcasting," *Proceedings of International Conference on Computing and Information*, pp.366-388, July 1995.

[26] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. New York, NY: McGraw-Hill, 1993.

[27] Min He, S.Q.Zheng, "An Optimal Sorting Algorithm on a Linear Array with Reconfigurable Pipelined Bus System," *Proceedings of the ISCA 15th International Conference on Parallel and Distributed Computing Systems*, 2002.

[28] Joseph JaJa, *An Introduction to Parallel Algorithms*, Addison-Wesley Publishing company, 1992.

[29] F.T. Leighton, "Tight Bounds on the Complexity of Parallel Sorting," *IEEE trans. Computers*, vol.34, pp.344-354, 1985.

[30] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. San Mateo, Calif.:Morgan Kaufmann, 1992.

[31] KeQin Li, "Constant Time Boolean Matrix Multiplication on a Linear Array with a Reconfigurable Pipelined Bus System," *The Journal of Supercomputing*, vol.11, pp.391-403, 1997.

[32] A. Louri, "Three-Dimensional Optical Architecture and Data-Parallel Algorithms for Massively Parallel Computing," *IEEE Micro*, pp.24-81,1991.

[33] S. Levitan, D. Chiarulli, and R. Melhem, "Coincident Pulse Techniques For Multiprocessor Interconnection Structures," *Applied Optics*, 29(14), pp.2024-2039,1990.

[34] Yueming Li, Yi Pan and S.Q.Zheng, "Pipelined Time-Division Multiplexing Optical Bus with Conditional Delays," *Optical Engineering*, vol.36, no. 9, pp.2417-2424, September 1997.

[35] K. Q. Li, Y. Pan, S. Q. Zheng, "Fast and Processor Efficient Parallel Matrix Multiplication Algorithms on a Linear Array with a Reconfigurable Pipelined Bus System," *IEEE transactions on Parallel and Distributed systems*, Vol.9, No.8, pp.705-720, August 1998.

[36] K.Li, Y.Pan, and S.Q. Zheng, eds., *Parallel Computing Using Optical Interconnections*, Kluwer Academic Publishers, Boston, Massachusetts, 300 pp., 1998.

[37] K. Li, Y. Pan, S.Q. Zheng, "Fast and Efficient Parallel Matrix Computations on a Linear Array with a Reconfigurable Pipelined Bus System," *High Performance Computing Systems and Applications*, J. Schaeffer ed., pp. 363-380, Kluwer Academic Publishers, Boston, Massachusetts, 1998.

[38] K. Li, Y. Pan, and S.Q. Zheng, "Fast Matrix Multiplication and Related Operations Using Reconfigurable Optical Buses," *Parallel Computing Using Optical Interconnections*, pp. 249-273, Kluwer Academic Publishers, 1998.

[39] Keqin Li, Yi Pan, Si-Qing Zheng, "Efficient Deterministic and Probabilistic Simulations of PRAMs on Linear Arrays with Reconfigurable Pipelined Bus Systems," *Journal of Supercomputing*, pp. 163-181, 2000.

[40] Y. Li, J. Tao and S.Q.Zheng, "A Symmetric Processor Array With Synchronous Optical Buses And Switches," *Parallel Processing Letters*, Vol. 8, No. 3, 1998.

[41] Y. Li, and S. Q. Zheng, "Parallel Selection on a Pipelined TDM Optical Bus," *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, pp.69-73, 1996.

[42] Y. Li, and S. Q. Zheng, "Processor Arrays with Asynchronous TDM Optical Buses,"it Proceedings of SPIE Photonics West '97, pp. 291-302, 1997.

[43] Yueming Li, S. Q. Zheng, Xiangyang Yang, "Versatile Processor Arrays Based on Segmented Optical Buses," *Proceedings of SPIE Photonics West '97*, pp. 280-290, 1997.

[44] R. Melhem, D. Chiarulli, and S. Levitan, "Space Multiplexing of Waveguides in Optically Interconnected Multiprocessor Systems," *The Computer Journal*, vol. 32, no. 4, pp. 362-369, 1989.

[45] M. Middendorf and H. ElGindy, "Matrix Multiplication on Processors with Optical Buses" *Informatica*, in press.

[46] R. Miller, V. K. Prasanna-Kumar, D. I. Reisis, and Q. F. Stout, "Meshes with Reconfigurable Buses," *Proc. MIT Conf. on Advanced Research in VLSI*, pp.163-178, 1988.

[47] R. Miller, V. K. Prasanna-Kumar, D. I. Reisis, and Q. F. Stout, "Parallel Computations on Reconfigurable Meshes," *IEEE Transactions on Computers*, vol.42, pp.678-692, 1993.

[48] R. A. Nordin, A. F. J. Levi, R. N. Nottenburg, J. O'Gorman, T. Tanbun-Ek, and R. A. Logan, "A Systems Perspective on Digital Interconnection Technology," *J. Lightwave Tech.*, Vol. 10, No.6, pp.811-827, June 1992.

[49] Stephan Olariu, Cristina Pinotti, and Si Qing Zheng, "An Optimal Hardware-Algorithm for Sorting Using a Fixed-Size Parallel Sorting Device," *IEEE Transaction on Computers*, Vol.49, No.12, pp.1310-1324, December, 2000.

[50] S. Olariu and Z. Wen, "An Efficient Parallel Algorithm for Multiselection," *Parallel Computing* 17, 1991, pp. 689-693.

[51] Y. Pan, "Order Statistics on Optically Interconnected Multiprocessor Systems, *Opti. Laser Tech.*, vol.26, pp.281-287, 1994.

[52] Y. Pan, "Basic Data Movement Operations on the LARPBS Model, "in *Parallel Computing using Optical Interconnections*, ed. By S,Q,Zheng, Kluwer Academic Publishers, Boston, USA, 1998.

[53] M. S. Paterson, "Improved Sorting Networks with $O(\log N)$ Depth," *Algorithmica*, vol.5, pp.75-92, 1990.

[54] Sandy D Pavel, *Computation and Communication Aspects of Arrays with Optical Pipelined Buses*, A thesis submitted to the Department of Computing and Information Science in conformity with the requirements for the degree of Doctor of Philosophy, Queen's University, 1996

[55] C. Gregory Plaxton, *Efficient Computation on Sparse Interconnection Networks*, Thesis, Department of Computer Science, Stanford University, September, 1989.

[56] S. Pavel and S. G. Akl, "Matrix Operations Using Arrays with Reconfigurable Optical Buses," *Parallel Algorithms and Applications*, vol. 11 , pp. 223-242, 1996.

[57] S. Pavel and S. G. Akl, "Integer Sorting and Routing in Arrays with Reconfigurable Optical Buses," *Proc. Intl. Conf. Par. Processing*, pp.III-90-III-94, 1996.

[58] S. Pavel and S. G. Akl, "On The Power of Arrays with Optical Pipelined Buses," *Proceedings of 1996 International Conference on Parallel and Distributed Processing Techniques and Applications*, pp.1443-1454, Sunnyvale, California, August 9-11, 1996.

[59] Y. Pan and M. Hamdi, "Quicksort on a Linear Array With a Reconfigurable Pipelined Bus System," *Proc. of IEEE International Symposium on Parallel Architectures, Algorithms, and Networks*, pp. 313-319, June 12-14, 1996.

[60] Yi Pan, Mounir Hamdi, and Keqin 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.

[61] Y. Pan and K. Li, "Linear Array with Reconfigurable Pipelined Bus System - Concepts and Applications," *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, vol.III, pp.1431-1442, August 1996.

[62] Y. Pan, K. Q. Li, S. Q. Zheng, "Fast Nearest Neighbor Algorithms on a Linear Array with a Reconfigurable Pipelined Bus System," *Parallel Algorithms and Applications*, Vol.13, pp.1-25, 1998.

[63] M. C. Pinotti and S. Q. Zheng, "Efficient Parallel Computation on a Processor Array with Pipelined TDM Optical Buses," *Proceedings of 12th ISCA International Conference on Parallel and Distributed Computing Systems*, pp.114-120, 1999.

[64] C. Qiao, "On Designing Communication-Intensive Algorithms For A Spanning Optical Bus-Based Array," *Parallel Processing Letters*, **5**(3), pp.499-511, 1995.

[65] C. Qiao, R. Melhem, "Time-Division Optical Communications in Multiprocessor Arrays," *IEEE Trans. On Computers*, 42(5), pp.577-590, 1993.

[66] C. Qiao, R. Melhem, D. Chiarulli, and S. Levitan, "Optical Multicasting in Linear Arrays," *International Journal of Optical Computing*, vol.2, no. 1, pp. 31-48, 1991.

[67] M. Ryckebusch, "A High Performance Electrical and Optical Interconnection Technology," *Proceedings of the 41st Electronic Components and Technology Conference*, June 1990, Vol.2, pp.974-979.

[68] S. Rajasekaran and S. Sahni, "Sorting, Selection and Routing on the Arrays with Reconfigurable Optical Buses," *IEEE Transaction on Parallel and Distributed Systems* vol.8, no.11, pp.1123-1132, November, 1997.

[69] S. Rajasekaran and S. Shni, "Sorting, Selection and Routing on The Arrays with Reconfigurable Optical Buses," *IEEE transactions on Parallel and Distributed Systems*, vol. 8, no. 11, pp. 1123-1132, Nov. 1997.

[70] S. Sahni, "Models and Algorithms for Optical and Optoelectronic Parallel Computers", *Proceedings of the Fourth IEEE International Symposium on Parallel Architectures, Algorithms, and networks*, pp. 2-7, Fremantle, Australia, June 23-25, 1999.

[71] H. Shen, "Improved Universal k-Selections in Hypercubes," *Parallel Computing* vol.18, no.2, pp.177-184, 1992.

[72] Jerry L. Trahan, Anu G. Bourgeois, Yi Pan, and Ramachandran Vaidyanathan, "Optimally Scaling Permutation Routing on Reconfigurable Linear Arrays with Optical Buses,", *Journal of Parallel and Distributed Computing*, vol.60, pp.1125-1136, 2000.

[73] J. L. Trahan, A. G. Bourgeois and R. Vaidyanathan, "Tighter and Broader Complexity Results for Reconfigurable Models," *Parallel Processing Letters*, special issue on Bus-based Architectures, vol.8, no.3, pp. 271-282, 1998.

[74] J. L. Trahan, Y. Pan, R. Vaidyanathan and A. Bourgeois, "Scalable Basic Algorithms on a Linear Array with a Reconfigurable Pipelined Bus System, " *Proc. 10th ISCA International Conference on Parallel and Distributed Computing Systems*, pp. 564-569, New Orleans, LA, October 1997.

[75] J. L. Trahan, R. Vaidyanathan and C. P. Subbaraman, "Constant Time Graph Algorithms on the Reconfigurable Multiple Bus Machine," *Journal of Parallel and Distributed Computing*, vol.46, pp.1-14, 1997.

[76] L.Valiant, "Parallelism in Comparison Problems,"SIAM J. Comput. vol.4, pp.348-355, 1975.

[77] Biing-Feng Wang, and Gen-Huey 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* vol.1, No. 4, pp.500-507, October 1990.

[78] S. Q. Zheng and Y. Li, "Pipelined Asynchronous Time-division Multiplexing Optical Bus," *Optical Engineering* vol.36, pp.3392-3400, 1997.

Vita

Min He got her Bachelor's and Master's degrees at Hunan University, Changsha, Hunan, P.R.China in year 1988 and 1991, respectively. She worked in Zhuhai Telecommunication Office for several years before she came to US and got her Master's degree in Systems Science at Louisiana State University in year 1997. She started working for Qualcomm, Inc in year 1998. And she is now a senior engineer in Qualcomm Consumer Product, Inc.