Parallel Optical Architectures For Some Comparison-Based Problems

by

Mahmood Hossain

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

January, 1992

Order Number 1354052

Parallel optical architectures for some comparison-based problems

Hossain, Mahmood, M.S.

King Fahd University of Petroleum and Minerals (Saudi Arabia), 1992

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

PARALLEL OPTICAL ARCHITECTURES FOR SOME COMPARISON-BASED PROBLEMS

BY

MAHMOOD HOSSAIN

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

JANUARY 1992

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

This thesis, written by **MAHMOOD HOSSAIN** under the direction of his thesis committee, and approved by all the members, has been presented to and accepted by the Dean, College of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Thesis committee

Chairman (*Dr. SUBBARAO GHANTA*)

Member (*Dr. MOHSIN GUIZANI*)

Member (*Dr. YILMAZ AKYILDIZ*)

Member (*Dr. NASIR AL-DARWISH*)

Dr. MOHAMMED AL-TAYYEB
Department Chairman

Dr. ALA H. AL-RABEH
Dean College of Graduate Studies

Date : 19.1.92

ACKNOWLEDGEMENT

Acknowledgement is due to King Fahd University of Petroleum and Minerals, specifically the Information and Computer Science Department for providing the opportunity to carry out this research work.

I gratefully acknowledge the guidance of my thesis supervisor Dr. Subbarao Ghanta and the help of the thesis committee members Dr. M. Guizani, Dr. Y. Akyildiz and Dr. N. Al-Darwish. I also express my gratitude to all the members of the Optical Computing Group at CCSE. I must render my sincere thanks to my friend Husni Al-Muhtaseb who translated and typeset the thesis abstract in Arabic.

Lastly but not least let me mention my sincere appreciation to my friends and well-wishers home and abroad who helped and encouraged me throughout: Abdullah, Adel, Hitu, Garout, Ismail, Jaweed, Limalia, Khalid, Kornel, Noman, Moazzem, Moshiur & Rita, Monti, Mukul, Murad & Dolly, Purushothaman, Rana, Rashed, Said, Salah, Shantanu & Shangita, Dr. Shafique, Shelly, Dr. Tayyeb, Yahya, Zia; they have been a constant source of support.

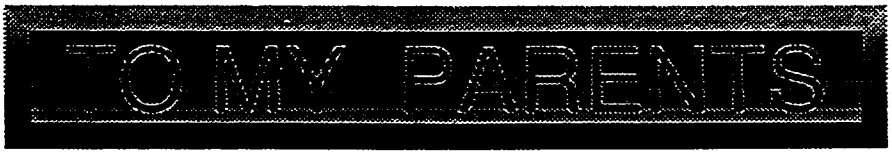

TO MY PARENTS

TABLE OF CONTENTS

Chapter *Page*

Chapter *Page*

Chapter *Page*

LIST OF FIGURES

Figure *Page*

THESIS ABSTRACT

NAME OF THE STUDENT : MAHMOOD HOSSAIN

TITLE OF THE STUDY : PARALLEL OPTICAL ARCHITECTURES FOR SOME COMPARISON-BASED PROBLEMS

MAJOR FIELD : COMPUTER SCIENCE

DATE OF DEGREE : JANUARY, 1992

Parallel computing requires architectures that are radically different from the inherently sequential computer architectures of today. In addition to processor-level parallelism of parallel computation models, where many processors collectively and concurrently work on a given task, parallelism at the basic instruction-level within each individual processor, is equally sought. While the former objective is achievable through different parallel computation models i.e., *SIMD, MIMD* models, the latter is difficult to achieve in the electronic domain. Optics provides parallelism at the basic instruction-level due to its massive inherent parallelism, large space-bandwidth product, and cross-talk-free free-space connectivity.

In this study, attempts have been made to combine the field of *Parallel Computing* and *Optical Computing* to solve some comparison-based problems. Some parallel optical architectures along with an *Optical Vector Processor* and associated algorithms are developed to solve problems that are based on comparison. Optical implementation of Clos Nonblocking-Multicast-Switching Network and an *O(1)* network control algorithm have been developed. Optical Reconfigurable Bus System has been developed to solve unary-sorting problem in *O(1)* time. The class of Binary-Tree-Computations has been simulated on optical architectures along with algorithms for *Parallel Select* and *Parallel Quicksort* problems.

MASTER OF SCIENCE DEGREE

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

JANUARY, 1992

خلاصة الرسالة

اسـم الطالـب: محمود حسين
عنوان الدراسـة: بناء ضوئي متوازي لبعض المسائل المعتمدة على المقارنة
التـخصـــص: علوم الحاسب الآلي
تاريخ الدرجـة: يناير (كانون الثاني) ١٩٩٢م

يتطلب تصميم الحسابات المتوازية هندسة بناء تختلف عن تلك المستخدمة في الحسابات المتوازية. فبالإضافة الى التوازي على مستوى المعالجات في نماذج الحسابات المتوازية حيث يعمل عدة معالجات في نفس الوقت لتنفيذ عمل معين فإن التوازي على مستوى الأوامر الأساسية مطلوب أيضا.

ومع ان التوازي على مستوى المعالجات يمكن الحصول عليها بستخدام نماذج عديدة منها SIMD و MIMD فإن التوازي على مستوى الأوامر الأساسية من الصعب الحصول عليها في مجال الالكترونيات.

أما في مجال الضوئيات فإن التوازي على مستوى الأوامر الأساسية يمكن الحصول عليه لأسباب عديدة منها خاصية التوازي في الضوء والمنتجات عريضة الموجة جدا وإمكانية الوصل العالية لعدم التأثير السلبي لتقاطع الضوء.

يقدم هذا البحث محاولة لجمع الحسابات المتوازية مع الحسابات الضوئية لمعالجة بعض المسائل المعتمدة على المقارنة. وقد تم اقتراح بناء ضوئي متوازي مع معالج ضوئي متعدد اضافة الى الخوارزميات التي تعالج بعض المسائل التي تعتمد على المقارنة. كما تم تطوير شبكة مفتاح متعدد الارسال دون توقف مع خوارزمية للتحكم بالشبكة بدرجة تعقيد مقدارها ١. وكذلك تم حل مسألة الترتيب بنفس درجة التعقيد وذلك بتطوير نظام ضوئي خاص. اضافة الى انه قد تم محاكاة الحسابات الشجرية الثنائية بخوارزميات الاختيار المتوازي والترتيب المتوازي.

درجة الماجستير في العلوم
جامعة الملك فهد للبترول والمعادن
الظهران - المملكة العربية السعودية
كانون الثاني (يناير) ١٩٩٢م

Chapter 1

Introduction

1.1 Objective of the Study

Over the past four decades of computing history, dramatic increases in computing speed were achieved. As we went from relays to vacuum tubes to transistors and from small to medium to large and then to very large scale integration, we witnessed the geometric growth of computational speed and hence increasingly greater problem-solving capability. We would be happy if this progress continues. Unfortunately, this trend has its peak due to fundamental limits. The problem is not with the computational speed (switching speed to be more precise) but the communication limitations, namely, propagation delays, interconnection constraints associated with the resistance and capacitance of electronic interconnects, inadequate space-bandwidth product and finally, undesired interactions between electrons due to miniaturization of components beyond the lowest physical limit. State-of-the-art computing machines, from stand-alone desk-top PC to the largest supercomputer, are basically similar in their computation models: their hardware is based upon processing units fabricated on VLSI chips, that operate on instructions and data kept in memory. They run sequential programs that specify exactly what to do at each step. Broadly speaking, these are the Von Neumann machines. The high speed gain in electronic switching does not cope with the ever-increasing demand for computational power because of the drag caused by the low communication capability. This is the so-called Von Neumann bottle-neck. This limitation of conventional computers has initiated research efforts in two different directions, one in the field of *Parallel* and *Distributed Computing* and the other in the newly emerged field of *Optical Computing*.

In Parallel and Distributed computation models, several uniprocessors, random access machines (RAM) work independently either with proper synchronization among themselves or without any synchronization at all. Given a problem to be solved, it is broken into a number of subproblems.

All of these subproblems are now solved simultaneously, each on a different processor. The results of all the subproblems are then combined to produce the solution to the original problem. This is a radical departure from Von Neumann-machine-model, namely, the sequential uniprocessor machine. The declining cost of VLSI chips has made it possible to assemble machines with millions of processors. The inter-processor communication is handled through a sophisticated, well-defined interconnection network. Much work has been done in developing efficient interconnection networks[SEIG 91]. Though we might get large speed-up in terms of time needed to solve a particular problem on a parallel computational model, we are still faced with the problem that inter-processor communication lags behind the computational speed of individual electronic processor.

In their search for universal parallel computer models, researchers explored a new field called *Optical Computing*. The field of *Optical Computing* is quite broad and includes different ideas and possibilities. Essentially, computation and communication using optics for information processing is termed as Optical Computing[FEIT 88]. Optics has a proven superiority to electronics when compared in terms of communication performance. While it appears feasible to improve switching speed by several order of magnitude (to perhaps *10* ps [ANDO 89]), it does not appear possible to make an electronic interconnect which can support such short pulses through a long distance but the shortest gate-to-gate paths. It is, however, possible to transmit a *10* ps optical pulse through an optical fiber or free-space with minimal distortion[ANDO 89]. Commercially available optical fibers have bandwidths in excess of *1000* Gb/s, which remains unused due to the lack of fast tapping devices. Again, the number of interconnects required by an electronic VLSI chip scales in proportion to the chip area. The perimeter grows only as the square root of the area. As a chip gets larger, the space available for each interconnect on the perimeter becomes restricted. This problem can be overcome by multiplexing signals on a small number of interconnects. This, however, incurs a time penalty. An alternative solution is to make interconnections within the whole chip area. This is not as possible in electronics as in optics.

Following is a list of points that make optics superior to electronics :

- **Direct Image Processing.**

Certain image processing functions can be carried out by specific optical systems directly on the image, with no need for sampling, quantization and such. These optical systems operate much faster and with better resolution than their electronic counterparts.

- **Massive Inherent Parallelism and Connectivity. .**

The problems restricting the ultimate speed-up of the conventional computers do not arise from the inadequate speed of the basic elements. Rather, interconnection is the limiting factor. Optical communication allows new ideas unheard of before: millions of data channels may operate in parallel, each with a bandwidth much greater than that of any electronic link.

- **Speed.**

The possibilities of special purpose systems and massive parallelism give rise to high speed in data processing. This is important in modern applications like oil exploration, satellite image processing, weather forecasting, biomedical analysis, modelling fusion reactors, cryptanalysis, solution of very large systems of differential equations arising from numerical simulations in disciplines as diverse as seismology, aerodynamics, atomic, nuclear and plasma physics. No computer exists today that can deliver the processing speeds required by these applications(in the order of 10^{13} operations per second [AKL 89]). Even the so-called supercomputers peak at a few billion operations per second.

- **Immunity to EMI.**

The electrons circulating in a VLSI chip are susceptible to electromagnetic interference(EMI). In electrically noisy environments electronic VLSI chips may malfunction. Optics is completely free from EMI.

- **Free Space Connectivity.**

Since light beams do not interact with each other, an optical computer can exploit this feature to have free space interconnections between its components.

The above list is quite impressive and calls for extensive research efforts towards *Optical Computing*, where computation and communication coexists. The marriage between the fields of *Parallel Computing* and *Optical Computing* appears to provide a desired and suitable alternative to the Von Neumann model of sequential computation. In this study, issues related to optical computation and communication on a parallel computation model to solve some basic comparison problems are addressed.

1.2 Thesis Organization

The results of this thesis are distributed into different chapters starting from Chapter 2 to Chapter 7. Chapter 2 reviews the related background work both in *Optical Computing* and *Parallel Computation Models* to solve some of the representative comparison-related problems that includes general interconnection network to sorting network.

Chapter 3 presents an Optical implementation of Clos nonblocking-broadcast-network. An $O(1)$ control algorithm on a proposed Optical Parallel architecture is proposed. The optical nonnumeric processing unit, *Optical Vector Processor(OVP)* with some primitive optical operations on binary vectors is also presented.

Chapter 4 proposes an Optical Reconfigurable Processor Matrix architecture to sort n numbers in constant time i.e., in $O(1)$ time.

In Chapter 5, Arbitrary *Binary-tree-computations* are simulated on Optical Reconfigurable Processor Array architecture.

Chapter 6 presents parallel algorithms to solve *Parallel Selection Problem* and *Parallel Quicksort* on an Optical Reconfigurable Processor Array.

Finally, in Chapter 7 conclusion and recommendations for future work are presented.

Chapter 2

Background Work

2.1. Optical Computing

Optical computers are based on phenomena related to optics and light. The ancient history of Optical computing is linked to a large extent, to that of radar systems[LEIT 77]. The need to process vast amounts of data supplied by radars used in mapping was the motivating force behind early optical signal processing research efforts. Optical Computing received a great push from the invention of the LASER in 1960. The characteristics of this light source allowed numerous new operations to be realized by optical means. These operations were analog in nature, and are best described by the term "signal processing". Unlike electronic signal processing systems, which have one (temporal) degree of freedom, typical optical processors have two (temporal and spatial) degrees of freedom; the data is a 2-D image. One of the basic operations that can be performed on images in laser light is the Fourier Transform. Numerous applications grew out of this ability.

During the time when conventional electronic computers grew tremendously in speed, power and, in miniaturization, sporadic efforts were also made to realize digital logic by optical means. These efforts did not succeed mainly for two reasons. One was the success of Von Neumann computers. They answered the world's computational needs. They are well-developed both theoretically and practically. They gave results that were accurate to any desired degree. When this is compared to the basically analog optical systems with their new and immature technology and potentially problematic operations, it is clear why research in optical computing took a back seat. The second reason was the absence of substantial research projects to find an alternative to sequential machines. Electronic computers received a great push from the invention of transistors and then the development of integrated circuits. The need for any alternative and/or radical change to the electronic computing has not been too great until recently. The so-called Von Neumann bottleneck of sequential computation resumes the interests in Optical Computing in search for true

parallel computers.

There are two distinct trends in optical computing : that of special purpose analog systems and that of general purpose digital optical computers. Special purpose analog optical computers are again divided into two classes: those that deal with image and signal processing, and those that deal with numerical processing. In image and signal processing analog systems, we are talking about optical systems that accept one beam of light as input, manipulate it, and finally produce an output beam of light. These systems exploit various physical phenomena from the field of optics, such as the ability to perform 2-D Fourier Transforms. In contrast, numerical processors do not manipulate beams of light. rather they deal with arrays of numbers, represented by multiple points of light.

General purpose digital optical computers have been a major research topic recently. They are usually based on nonlinear optical effects. The invention of first nonlinear electrical device (where electron controls electrons) back in 1907, namely *vacuum tube,* made it possible to implement logic circuits. Logic circuits are the backbone of conventional digital electronic computers. Similarly, the invention of optical nonlinearity (where light controls light) and nonlinear devices [FEIT 88] promises the opportunity to realize optical logic circuits. This led research towards realizing digital optical computers [JORD 91]. These research efforts devote to develop and realize optical devices that mimic their existing electronic counterparts: logic gates, memory elements(optical flip-flop), integrated optical circuits [FEIT 88]. The goal is to create a general purpose digital computer (either hybrid i.e., optoelectronic or all-optical) that will be comparable to electronic ones, but better in some significant way, e.g., it might be faster.

Optical computers use light to convey information. Light is an electromagnetic wave that is characterized by its frequency, wavelength, amplitude, phase, polarizatrion, and degree of coherence. Light sources used in optical computers are semiconductor LEDs and LASERs. Information is imposed on a light beam by modulation of its cross-section; this is done by spatial light modulators (SLM, a new addition to optical devices that are used in information processing) [FEIT 88]. Light beams are manipulated by various optical elements, e.g., beam-splitters,

diffraction gratings, lenses, polarizers, holograms and holographic optical elements. The intensity of a light beam is detected by photodiode detector with various threshold levels [FEIT 88]. Feitelson [FEIT 88] has done an excellent task of surveying the various optical devices, and phenomena, their capabilities and limitations and in indicating the role they can play in computing from the point of view of Computer Science. In optical computing, two types of memory are discussed. One is more or less equivalent to primary memory, and consists of one-bit-store elements. the other is mass storage, which is implemented by optical disks or by holographic storage systems. This type of memory promises very high capacity and storage density on line. Holographic memory consists of a set of carrier and position multiplexed phase transmission holograms recorded on a silver halide emulsion. It is used to store a number of patterns in the same small emulsion area. The recordings are made with plane wave object and reference beams.

2.2. Optical Architectures

Broadly speaking, the architecture of conventional electronic computers supports the Von Neumann model: a CPU manipulates the contents of the memory by executing a sequential program; for each instruction data is loaded from the memory to the CPU's internal registers, and the computed result is stored back in the memory. The technology used to implement these computers is based on planar integrated circuits. A possible approach to choosing an architecture for optical computers is to simply copy the very successful electronic precedence, replacing electronic logic elements by optical ones, and interconnection wires by optical fibers or optical waveguides [WEST 87], [JORD 91]. The goal of these research efforts is centered around the design and construction of an optical version of general purpose, stored program, digital computer. The design of Digital Optical Computer (DOC) at Boulder [JORD 91], a prototype, is bit-serial to minimize the number of active devices. This bit-serial design uses bandwidth or time domain capacity to achieve processing power. Optical fibers form all memory and interconnection. Fiber delay lines or loops are used as storage. Due to time division multiplexing and due to the use of fiber delay loops as memory, the terabits per second information capacity of fiber channel is not utilized fully. This approach is merely a transfer of conventional electronic computer model to optics without any major change in the underlying model. However, this approach requires

each optical element to be highly superior to the functionally equivalent electronic counterparts in order to justify the transition to optics. Specifically the optical elements should be faster by at least two orders of magnitude [RHOD 86]. As it is not clear whether such expectations are practical, many researchers feel that a radically different architecture should be pursued.

The architecture of a computer should be related to the capabilities of the underlying technology. In the case of optical computers, the possibility of dense interconnections coupled with interference free propagation, implies that parallel architectures might be fitting. A beam of light may contain millions of resolvable pixels, each of which may be used as a distinct data channel. Two primary factors limit the density of optical interconnects. They are the resolvable spot size for free-space interconnects and the required size of a waveguide. Single mode fibers with core diameter of *1* μm are commonly available and it is possible to fabricate optical guides on the order of *1* μm as is routinely done in integrated optics. Free-space propagation is fundamentally limited by the diffraction-limited spot-size of image systems. Even with this restriction, however, it is possible theoretically to pack optical waveguides(fibers) or free-space channels on the order of *10,000* to *50,000* per mm^2 [HAUG 86]. As the beam traverses an optical system, various operations are performed on all the points in parallel. Therefore the parallel architecture that seems most suitable to an optical computer is that of an SIMD(Single Instruction-stream Multiple Data-stream) machine. The basic data element in this architecture is not a byte or word but rather a vector or matrix of pixels or binary digits. This brings parallelism at the component level (within a single optical processor) as well as processor level (within an interconnected multiprocessors). Processors are connected through a generalized optical interconnection network (capable of broadcast facility). Intra-processor and Inter-processors interconnections by optical means provide the following lucrative advantages:

- **Large Bandwidth and Space-Bandwidth Product**

The use of light as a carrier of information is a step in use of carrier of higher and higher frequencies. The need for carriers with higher frequencies stems from the need for larger bandwidth.

If the carrier frequency is of the order of 10^{14} cps(cycles per second), a bandwidth near 10^{13} cps should be available. This implies possible data rate in access of 10^{12} bps [FEIT 88]. Such rates are *3* orders of magnitude faster than the fastest data rate of today. Communication at a data rate of 10^{12} bps means that the modulation of the carrier changes 10^{12} times every second. Electronic switching devices fall at least *2* orders of magnitude short of the objectives. In optics, large number of independent channels can propagate together in parallel, resulting in large space-bandwidth product.

- **High Speed Propagation**

Electrical signals propagate two orders of magnitude slower than light [WILK 83]. The reason is that the electronic conductor has a capacitance that has to be charged— it can be described as a pipe that has to be filled in before anything arrives at the other end. Therefore, the propagation time increases with the length of the connection and also with the fan-out. With light these problems do not arise; it always propagates at the speed of light (scaled down by the refractive index of the medium).

- **Non Interference Propagation**

Electronic interconnects must be laid some distance apart, so as to prevent cross-talk. Electrons in a metallic conductor also interact with electromagnetic radiation: conductor simply serves as an antenna. Thus electronic interconnects are susceptible to electromagnetic interference. Photons on the other hand, do not interact with each other or with other radiation. This is a major asset in optical interconnects.

- **Density and Parallelism**

As light beams do not interact, optical interconnections need not be guided at all within the vicinity of optical processor or integrated optical chip. Light beams can propagate freely in space, with

different communication channels crossing each other. Pin lay-out problem encountered in VLSI technology is reduced. The interconnection density in optics is much higher [HAUG 86].

- **Dynamic Reconfigurability**

Optical interconnection are often controlled by holograms and mask patterns. If these elements are implemented by spatial light modulators(SLMs) or dynamic grating pattern, they can be dynamically reconfigured. Thus the interconnection mapping patterns are modified in real time(within ms) [SCHU 87], [SAWC 86].

Optical Interconnection Media

- **Optical Fibers**

An optical fiber is a conductor or waveguide for light. The propagation of light in optical fibers is explained by the phenomenon of *Total Internal Reflection* [KAPA 60]. Fiber optics are already being used in telecommunication with great success. It has been suggested that they may be used at a smaller scale, for connections between boards and chips [GOOD 84]. A Typical optical fiber has information capacity in excess of *1000* GHz.

- **Free Space Interconnections**

Within the confines of an optical computer, there is actually no reason to conduct the light in optical fibers; as the distances are short. As the light beams do not interact with each other, there is no danger of cross-talk. The advantages of the so-called free-space interconnection scheme stem from the fact that the third dimension of space is utilized, instead of confining the interconnection topology to a 2-D surface. This alleviates many topological difficulties. The main issue in free-space interconnection scheme is how to direct many beams of light and focus them on the right ports. The technique that has been used is through holographic optical elements.

A *hologram* is a mechanism that can be used to modify and redirect a light beam that is incident upon it. Because of this capability, a hologram or collection of holograms can be thought of as photonic switches. A simple hologram, similar to a diffraction grating, can deflect its light in any desired direction(depending on the orientation of the grating pattern imposed on it) and focus it on a detector. In general, we have a set of sources, a set of detectors and an arbitrary mapping pattern on hologram. In order to implement this mapping, the hologram is divided into subholograms one for each source-detector mapping [SCOT 88], [FEIT 88], [KOST 87]. The interconnection pattern on the hologram can be dynamically reconfigured resulting in dynamic reconfiguration of the interconnection [SCHU 87].

For the realization of the holographic interconnection scheme it is important to know what spatial resolution can be achieved directing incoming light beams to different distinct spots. With the *Reyleigh criteric*, it can be shown [SCHU 87] that the minimum resolvable distance h between two spots, which can be addressed by a hologram of size $L \times L$ is given by

$$h = \lambda S / L$$

where λ denotes the wavelength, S is the distance of the surface from the hologram where the spot is focused. For an optical interconnect using laser source with $\lambda = 1.3\ \mu m$ and a hologram aperture $L \times L = 1 \times 1$ mm and a distance of 20 mm of the hologram from the detectors, the minimum separation h between two adjacent detectors is:

$$h = 1.3 \times 20 / 1 = 26\ \mu m.$$

If $h = 50\ \mu m$, an area of $1\ mm^2$ enables one to carry 20×20 detectors. This packing density of detectors is sufficient in most cases [SCHU 87].

2.3. Inter-Processor Interconnection

A SIMD model of multiprocessor computer model best suits the potential and the inherent capabilities provided by optics. In electronic multiprocessor/multicomputer environment, inter-processor communication is handled by different interconnection networks. They range from a common bus on one side of the spectrum to a crossbar network on the other extreme. In the middle lies the set of multi-stage interconnection networks: Clos, Shuffle-Exchange, Omega, Benes, Hypercube, Banyan and so on. Much work has been done on multistage interconnection networks [SEIG 91] with different objectives. Optical multiprocessor computer can adapt any one of these well-defined interconnection networks suited to its underlying capability and characteristics. Optical implementation of some of these interconnection networks have been proposed using optical elements; such as, Crossbar network [FRAC 90], Perfect Shuffle [LOHM 86], [HEIN 88], [BIAN 91], and Banyan network [JAHN 90].

- **Bus**

A bus is a common transmission medium, used by the communicating entities. At each moment, at most one user is transmitting a message on the bus, while all others listen. The optical implementation of a bus is simply an optical fiber link, running through all the communicating processors. Different modules access the information flowing in the fiber by means of special couplers. These couplers are curved pieces of fiber that touch the main fiber; their operation is similar to that of directional couplers [MILT 76].

- **Crossbar Switch**

The Crossbar is a general switching device that can connect any one of N inputs to any one of N outputs(1 to 1 connection). A generalized crossbar allows 1 to many inputs to outputs mapping(broadcasting). A generalized crossbar is the most desirable type of interconnection to be used in optical parallel computers because it is *rearrangeable* (any permutation of input to output

ports can be realized) and *nonblocking* (connection between input and output ports can be established independently without disturbing the other connections). In optics, a matrix of holographic mask pattern realizes a crossbar function where connections can be controlled dynamically in real time [SCHU 87], [FRAC 90], [McAU 86], [SAWC 86].

- **Clos Multistage Interconnection Network[Clos 74]**

Clos multistage interconnection network is based on a number of interconnected crossbar switches of smaller sizes as basic building blocks spread over different stages. This network has great potential for parallel optical multiprocessor architecture. Clos multistage interconnection network with broadcast capability, has been studied extensively [YANG 90].

2.4 Conclusion

The success of optical computing depends on how efficiently the inherent parallelism and enormous communication capabilities of optics is explored in new computer architectures. A simple transfer to optics from electronics without a radical change in the underlying computational model, may fail to justify the transition of technology. Optics demands parallel architectures to match with its capabilities. To be more specific, optical computer architecture is expected to be SIMD (if not MIMD) machine and the basic optical processing unit must exploit the natural parallelism provided by optics. The result is two-fold gain in parallelism — parallelism at the processor level and parallelism at the basic instruction level.

Chapter 3

Optical Realization of CLOS Nonblocking-Broadcast-Switching Network with constant time Network Control Algorithm

3.1 Introduction

A generalised and efficient switching network is an absolute necessity for the success of parallel computing and processing. Switching networks provide a set of interconnection or mapping between two sets of nodes ; the input and the output ports. For N input ports and M output ports there are N^M *well-defined* mappings from inputs to outputs. The term *well-defined* means that each output is defined in terms of one and only one input port. A network performing all N^M such mappings is called *Generalised Connection Network (GCN).* If we limit the mapping function only to the class of one-to-one, then $N!$ such mappings are well defined. This is the set of all possible permutations over the input set which does not include the broadcast assignments. A network with broadcast capability from any input port to a set of output ports is called *Broadcast Network.*

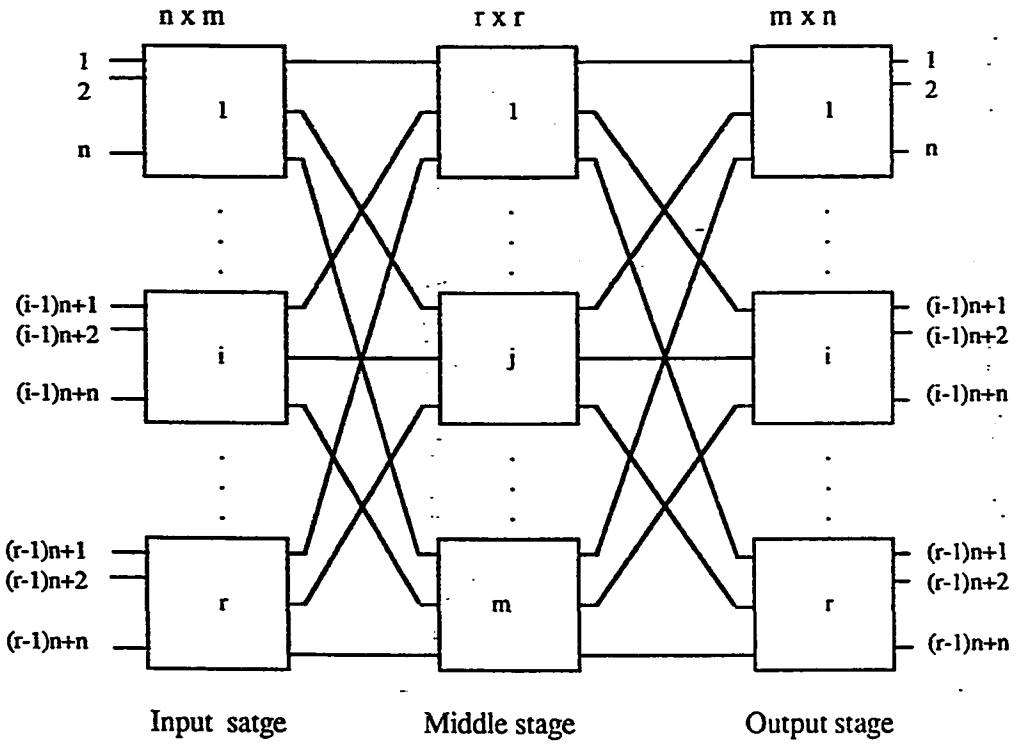
An obvious and easy way of implementing arbitrary collections of broadcast requests is with a complete crossbar network. A crossbar is the most desirable type of interconnection topology to connect processors with processors and/or with memories because it is *rearrangeable* (any one of $N!$ permutations of input ports to output ports can be realized) and *nonblocking* (connections between input and output ports can be established independently without disturbing the existing connections). Unfortunately, as the number of input ports, N grows, the number of crosspoints also grows as N^2, making it uneconomical to realize large systems.

Multistage networks provide a cheaper alternative to the complete crossbar switch. They are based on a number of interconnected crossbar switches of smaller sizes as basic building blocks, spread over different stages. A 3-stage Clos network is a good example. The trade-off here is between cost and complexity of control. Much work has been done along the line of multistage interconnection networks with different objectives [SEIG 91]. Recently, Yang and Masson [YANG90] proposed a design framework for nonblocking broadcast switching networks. Their work was based on Clos network topology. They analysed Clos network's characteristics mathematically and found out a nonblocking condition on the number of middle stage switches to realize arbitrary broadcast requests.

We adapt and extend the work of Yang and Masson to the domain of optics. Since optics provide natural parallelism, higher space-bandwidth product and immunity from mutual interference, we exploit these attributes of free space optics to implement the multistage nonblocking broadcast network . Further we propose a constant time algorithm for the network control, in contrast to the linear time algorithm proposed in [YANG 90] .

3.2 Multistage Interconnection Network with Broadcast Capability

Multistage switching networks, which consist of multiple cascaded stages of switching elements, are networks with a dynamic topology. They differ in the interconnection pattern between stages, the type and operation of individual switching elements, and the control scheme for setting up the switching elements. Some of them realize all the possible permutations of input sets whereas some do not [LAXM 90]. Example of such networks are : Clos network, Benes network, Baseline network, Omega network, Shuffle Exchange network, Banyan network [SEIG91] etc. Clos 3-stage network is based on crossbar switches of smaller sizes. Our interest lies in the Clos network topology with additional broadcast capability.



Fiqure 1: N x N Nonblocking Clos Network

In an *NxM Multistage Interconnection Network with Broadcast capability (MINB), N* input ports are mapped onto *M* output ports in a one-to-many broadcast fashion. The switching elements are grouped into smaller modules called *SWITCH MODULES (SM).* Each *SM* is of the size nxm ; $n<N,m<M$ and provides broadcast connections for n input ports to m output ports, that is, any input can request connections to any subset of available outputs. Switch modules are the basic building blocks. *SMs* are grouped into different stages. Each *SM* in stage i has dedicated links to all *SMs* of stage *(i+1).* An N_1xN_2 3-stage *MINB* has r_1 *SMs* of size n_1xm in the input stage, m *SMs* of size r_1xr_2 in the middle stage and r_2 *SMs* of size mxn_2 in the output stage where $N_1 = r_1xn_1$, $N_2 = r_2xn_2$. Without loss of generality and for notational convenience we assume $N_1=N_2$ i.e., number of input and output ports to be equal. It turns out that if we choose $n_1 = n_2$ then $r_1 = r_2$.

In a Nonblocking broadcast network any broadcast request from an input port to a set of free output ports can be realized without any disturbance to other existing connections.

3.3 Previous Results

Much work has been done on multistage interconnection networks [SEIG 91]. To realize a connection path between inputs and outputs in a multistage interconnection network a network controller is needed. A network controller executes a control algorithm that sets the required switches in various stages to set up the requested path. Until recently, the best control algorithm that could be achieved was of $O(NlogN)$. Yang and Masson[YANG90] gave a linear time $O(N)$ algorithm. They have found a lower bound on the number of middle stage switches to ensure nonblocking broadcast connection from any arbitrary input to a set of output ports. According to Clos, the nonblocking condition for permutation assignment requires that $m \geq 2n - 1$. Although it works fine for point-to-point connections, it does not necessarily satisfy all broadcast assignment. Masson [MASS71,72] first proposed a design for strictly nonblocking and rearrangeable multistage switching network for broadcast assignment. For broadcast assignment in a 3-stage network it was shown that if $m > n(r+1) - 1$, the network is strictly nonblocking, and if $m \geq nr$, the network is rearrangeable.

The inherent characteristics of Multistage Nonblocking Broadcast Networks have been studied [YANG 90]. For a 3-stage Clos broadcast network to be nonblocking it has been shown that,

$$m > \min_{x} \{ (n-1)(x + r^{1/x}) \} \quad \text{for } 1 \leq x \leq \min \{ n - 1, r \} \qquad (3.1)$$

where m = number of middle switch modules (SM)

r = number of output switch modules (SM)

= maximum fanout for each input request.

x = number of available middle stage switches for an input connection request to be satisfied for certrainity. $(x \geq 1)$

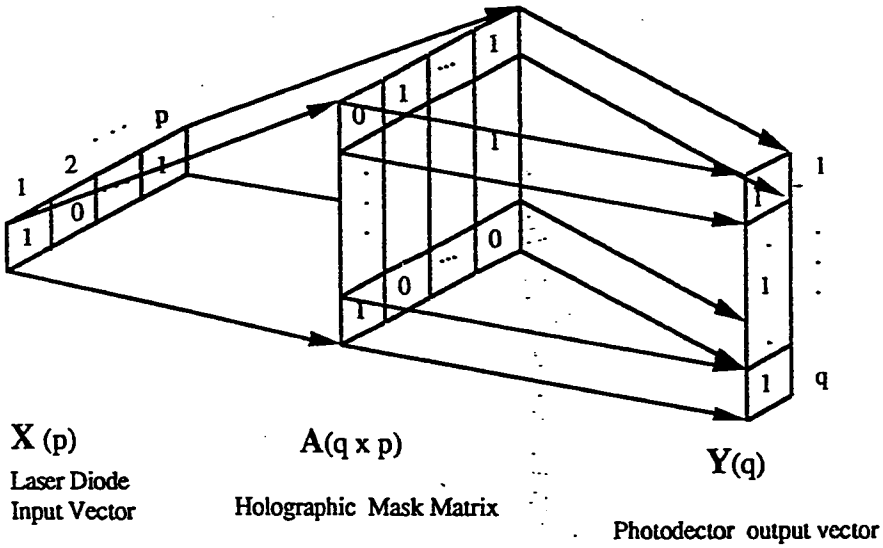
Given this nonblocking condition on m, a linear time $O(N)$ algorithm has been proposed to satisfy any input broadcast request. We will be using this nonblocking condition on m (3.1) to implement a reconfigurable Nonblocking Optical Broadcast Network. Our control algorithm given in section 5.2 takes $O(1)$ time.

3.4 Proposed Optical Multistage Nonblocking Broadcast network

In this section, we explain the design of the proposed optical nonblocking broadcast network from the basic switch modules to the dedicated inter stage links.

3.4.1 Switch Module (SM)

Each SM of size *nxm* is the basic building block. It is realized as an optical crossbar switch. Essentially it performs a vector-matrix multiplication as : **Y = AX**



Fiqure 2 : Switch Module Design

where **X** = Bit- vector of size *1xp* representing the input lines,

Y = Bit- vector of size *1xq* representing the output lines,

and **A** = Interconnection binary mask matrix of size *qxp*. Its entries are either *0'* s or *1'* s. An entry of *1* in row *i* and column *j* of A means input *j* is connected to output *i*.

Schulze [SCHU87] has proposed an optical crossbar switch where the connection can be controlled dynamically. The input vector X is a laser diode array and the output vector Y is an array of photodetectors. The interconnection mask matrix A is a matrix of phase holograms. Its interference pattern can be changed dynamically using optical control. He has shown that to accomplish such holographic switching technique, dynamically changeable holographic phase gratings are imposed onto very high resolution spatial light modulators (SLMs). The interference patterns necessary to build the holograms can be obtained by an optical interference process. For notational convenience we assume that each entry of *1* in matrix A represents a closed switch and that of a zero an open switch. In addition, each entry of *1* in vector X represent a laser diode source or optical fibre port.

To realize broadcast connection, each column of A can have more than one *1* (closed switch). The matrix A is phase grated such that

$$\text{If} \quad A(i,j)=1 \quad \text{for } 1 \le i \le n ; 1 \le j \le m$$

$$\text{then } X(j) \text{ is connected to } Y(i) \text{ for all i}, 1 \le i \le m. \quad (n \le m)$$

We denote each SM in different stage as $\alpha_{ij}(p, q)$

where i = stage number and

j = switch position within the stage

pxq = size of the switch module

The proposed network has 3 stages; input, middle and output stage (Figure 1).

Input Stage :

There are r SM's of size nxm. Each one is denoted as : $\alpha_{1k}(nxm)$; $1 \le k \le r$

We can write connection between its input and output ports as:

$$X_{1k}(1xn)\,A_{1k}(mxn) = Y_{1k}(1xm)$$

equivalently, $X_{1k}A_{1k} = Y_{1k}\ ;\ 1 \le k \le r$.

Middle Stage :

This stage consists of m SM's of size rxr. They are denoted as: $\alpha_{2k}(rxr)$; $1 \le k \le m$

We can write connection between its input and output ports as:

$$X_{2k}(1xr)\,A_{2k}(rxr) = Y_{2k}(1xr)$$

equivalently, $X_{2k}A_{2k} = Y_{2k}\ ;\ 1 \le k \le m$.

Output Stage :

This stage has again r SM's of size mxn. They are denoted as: $\alpha_{3k}(nxm)$; $1 \le k \le r$

We can write connection between its input and output ports as :

$$X_{3k}(1xm)\,A_{3k}(nxm) = Y_{3k}(1xn)$$

equivalently, $X_{3k}A_{3k} = Y_{3k}\quad 1 \le k \le r$.

All X and Y vectors are holographic memory. They store the logical contents of the laser diode input vectors and photodetector output vectors.

3.4.2 Optical Fibre Inter-stage Links :

Switch Modules (SMs) in stage 1 and 2 and those of stages 2 and 3 are statically connected by optical fibre links according to the Clos network.

Links between Stage 1 to stage 2 are denoted as :

$$Y_{1k}(j) \text{-------} > X_{2l}(i)$$

$$1 \le k \le r \qquad 1 \le l \le m$$

$$1 \le j \le m \qquad 1 \le i \le r$$

Links between Stage 2 to stage 3 are denoted as :

$$Y_{2l}(i) \text{-------} > X_{3k}(j)$$

$$1 \le l \le m \qquad 1 \le k \le r$$

$$1 \le i \le r \qquad 1 \le j \le m$$

3.5 Network Controller

To exploit the inherent parallelism of free space optics we propose a parallel architecture / algorithm for the control circuitry. The network controller is analogous to a SIMD machine model. It has r identical *Optical Vector Processors (OVP)* sharing a common memory. It has a control circuitry to execute the stored network control program to activate different OVPs. Each OVP performs some primitive nonnumeric vector operations using all optical elements. Processing is done by OVPs concurrently. Now we define these primitive hardware operations. All vectors are holographic memory elements. Each element has two phase gratings reflecting incoming light beams in two different directions. We call them *grating1* and *grating0* respectively. *Grating1* represents binary *1* and *grating0* represents binary *0*. We first realize a number of primitive vector operations using all optical elements and use them in the network control algorithm.

3.5.1 Optical Vector Processor Primitive Operations

Each OVP is capable of performing some nonnumeric processing on binary vectors using all optical elements. The following operations are defined. Even though each operation is illustrated by a simple example, its generality is well understood.

Vector Copy : ***COPY(M)***

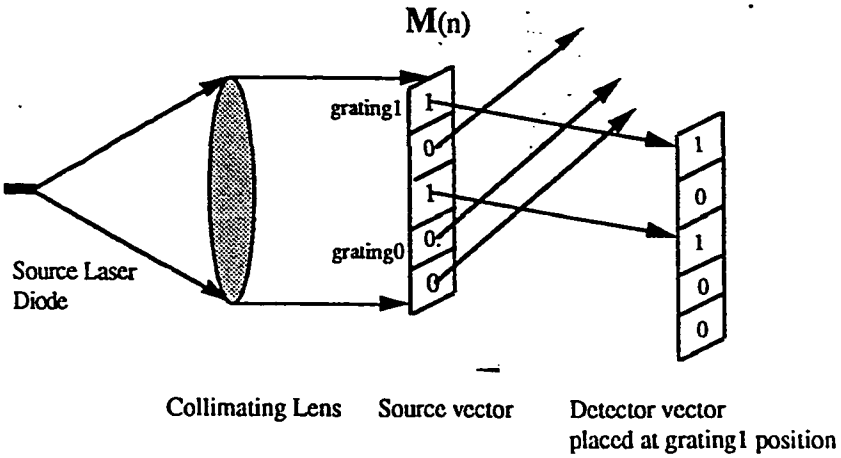


Figure 3: Copy of vector M

The COPY instruction makes a copy of a binary vector. The hardware realization is done by a laser light source, a collimating lens, the source vector, a detector vector and the destination vector. In one clock cycle the source is copied to the destination vector.The detector vector is aligned such that it collects all reflected lights from grating1 of source vector. This is illustrated in figure *3*.As an example consider a vector $M = < 1,0,0,1,0,1 >$. Since the values in positions *2,3,5* are zero they would be reflected away from those with logical value of *1*. The result vector, $M_{new} = \text{COPY}(M) = < 1,0,0,1,0,1 >$.

Vector Compliment : *INV(M)*

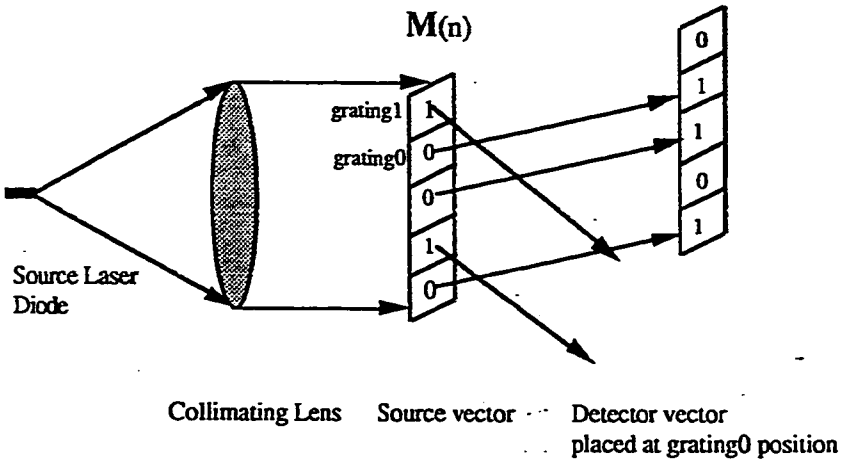


Figure 4 : Compliment of vector M

The INV instruction takes the compliment of a binary vector. Its hardware realization is the same as the COPY except that the detector vector is aligned the other way so that it collects the light beams coming from elements of source vector with grating0. Thus bitwise compliment of a vector can be achieved in one clock cycle of the network controller. Let us take same vector $M = < 1,0,0,1,0,1 >$. Here light reflected from position 2,3 and 5 are collected at the corresponding positions of the detector array and are set to logical *1*'s and the rest of the positions are set to logical *0*'s. As a result we have, $M_{new} = INV(M) = < 0,1,1,0,1,0 >$. The optical setup for this operation is depicted in Figure 4.

Intra Vector Bitwise UNION : *BITUNION(M)*

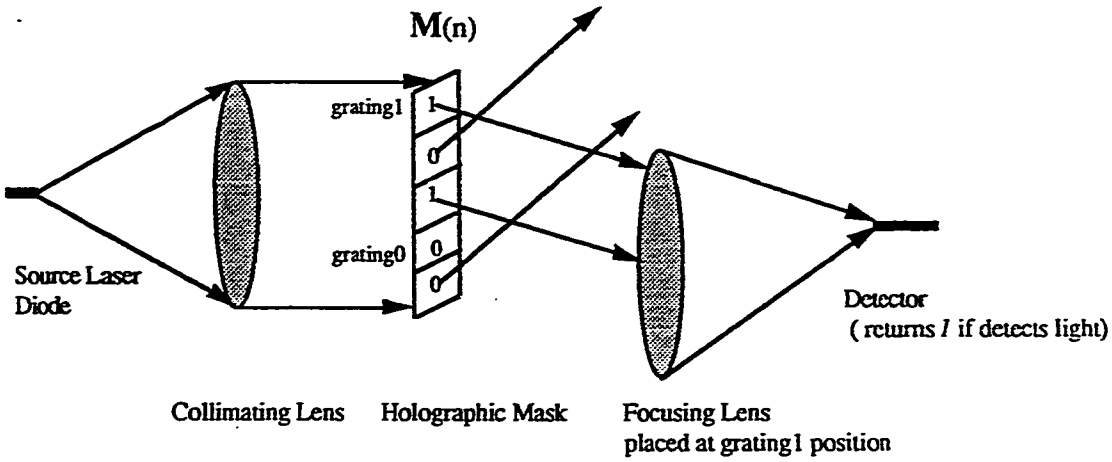


Figure 5: Bitwise Union Of M

BITUNION instruction takes a vector as input and finds the bitwise union of that vector in one clock cycle. Its hardware realization includes a laser source, two collimating lens and a photodetector. All the light beams coming from grating1 elements of source vectors are focused at the detector point. If any one of the entries of the source vector is logical *1* i.e. *grating1* then the detector will detect a logical *1*. For the vector $M = < 1,0,0,1,0,1 >$, $BITUNION(M) = 1$ because *M* has at least one element with logical value of *1*. It is shown in Figure 5.

Intra Vector Bitwise AND : *BITAND(M)*

BITAND instruction takes a vector as input and returns the bitwise ANDing of that vector in one clock cycle. This unary operator can be realized using a laser source, two collimating lens and a detector. Unlike BITUNION this time the detector is placed such that it can detect the beams coming from grating0 of the source vector. If any one entry of the source vector is *0* i.e. grating0 then the detector detects it and takes its compliment as the output of the operation.This is shown in Figure 6. Consider the vector $M = < 1,0,0,1,0,1 >$. Light reflected from positions *2,3,5* are focused at the detector and the result is set to logical value *0* , that is, $BITAND(M) = 0$. If all the

elements of *M* are logical *1*'s then all of them will be reflected away from the detector and the result is set to logical *1*, i.e., *M* = < *1,1,1,1,1,1* > then *BITAND(M) = 1*.

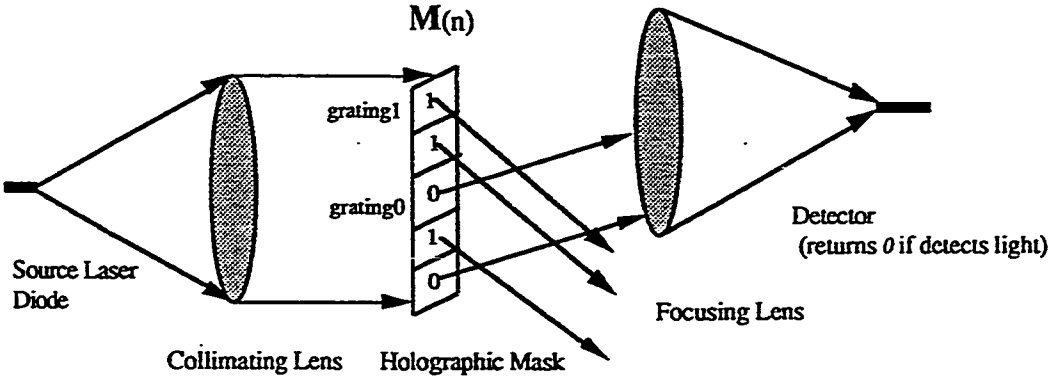


Figure 6: Bitwise ANDing of M

Vector Union : *UNION(X,Y)*

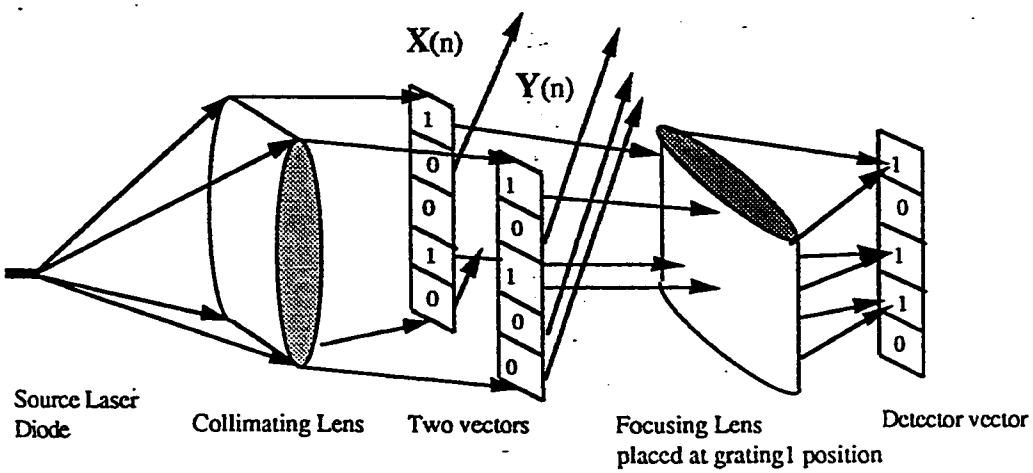


Figure 7: Union Of vectors X, Y

UNION instruction takes two/more vectors of equal length and returns their union. Its realization is the same as BITUNION except that a detector vector is used instead of a single detector and a cylindrical focusing lens or lenslet array is used instead of a single focusing lens. Figure 7 shows

the optical setup. Let us consider two vectors : $A = < 1,0,0,1,0,0 >$ and $B = < 0,1,0,1,1,0 >$. Now $C = UNION(A, B) = < 1,1,0,1,1,0 >$

Vector Intersection : *INTERSECTION(A, B)*

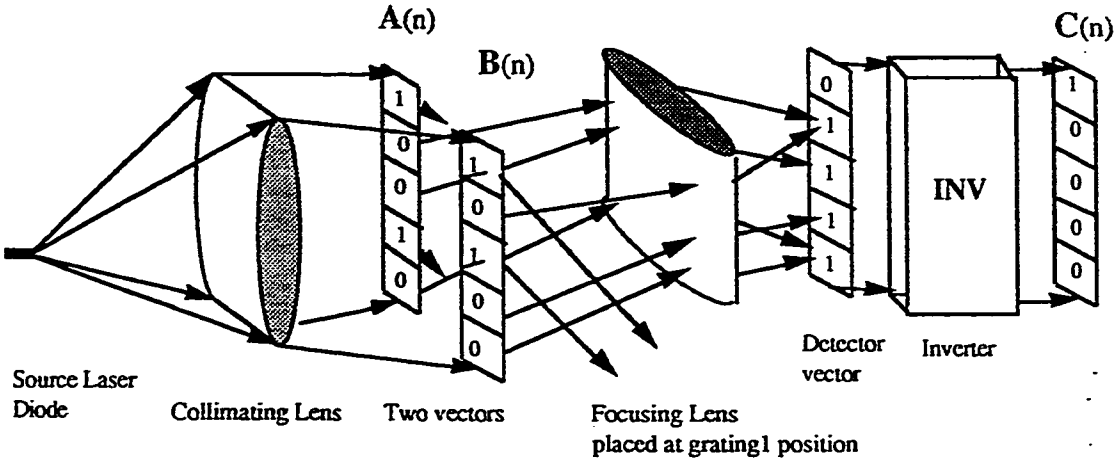


Figure 8: Intersection of vectors A, B

INTERSECTION instruction performs the binary AND operation of the corresponding bits of two vectors of equal length. Also its realization is the same as that of BITAND except that the detector is replaced by a detector vector and the focusing lens is replaced by a cylindrical lens or a lenslet array. The detector vector is then inverted by INV circuitry. So the INTERSECTION operation takes twice the time of INV. For example, $A = <1,0,0,1,0,0 >$ and $B = < 0,1,0,1,1,0 >$. The detector array detects logical value $< 1,1,1,0,1,1 >$. This vector then gets inverted. $C = INTERSECTION(A, B) = < 0,0,0,1,0,0 >$. The optical set up is given in Figure 8.

Masking Out All But The Leftmost *1* : *KEEPLEFTMOST(M)*

To mask out all but the left most *1* of a vector, we use the operation KEEPLEFTMOST. Its optical implementation involves the holographic beam steering element with reflective type of real time holograms. The operand vector is written on a holographic mask element with *0* as transparent

and *1* as reflective grating. Now a beam of light is passed through the mask plane tangentially. The beam will be deflected to its corresponding detector position of a detector vector, when it first encounters a *1* , that is, a deflecting grating. The rest of the detector vector positions are set to *0* , since there will be no light transmitted to those positions. This operation takes time proportional to the length of the operand vector. Since we assume the length of the vector is constant and the clock speed is less than the speed of light, this operation takes constant time. Figure 9 shows the optical design of the operation. As for example, KEEPLEFTMOST(< *0,0,1,0,1,0* >) = < *0,0,1,0,0,0* >.

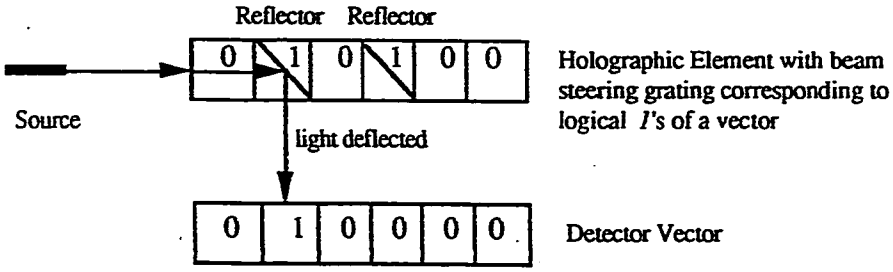


Figure 9: Masking out trailing *1*'s from a vector

3.5.2 Control Algorithm

Given the nonblocking condition on the number of middle stage switches (m), our algorithm always satisfies a valid input broadcast request. If a network input port requests broadcast connection to more than one output port on the same output Switch Module (SM) then it only needs a link to that output SM. Connections to other ports within that particular SM could be broadcasted locally. So input broadcast request is expressed as a vector of output SMs.

Preliminaries for the algorithm :

— Input broadcast request is denoted as

I_i , i $in < 1, 2, 3, ..., r >$ = Input Switch Modules Set and

$I_i = < \alpha_{3k} /_{k\ in\ \{1,2,.....,r\}} >$

= Output Switch Modules Set

We assume I_i is expressed in terms output SMs. We also assume the requested ports are grouped into r ordered vectors of size n each. For example, an input broadcast request from port j in the i th. input-stage SM to output ports $1,3,5,8$ is expressed as$<1,0,1,0,1,0,0,1,0>$ and the individual request vectors for each output SMs are: $O_1 = < 1,0, 1 >$, $O_2 = < 0,1,0 >$ and $O_3 = < 0,1,0 >$. $I_i = < 1,1,1 >$. We express the input request in terms of output SMs by I_i and in terms of output ports by vectors Os.

— Associated with each input stage SM is a vector of available middle switches to which a free link is available.

$AVAIL_i = < d_j /_{1 \le j \le m} >$ where $d_j = 1$ if $Y_{1i}(j) = 0$ or

$d_j = 0$ if $Y_{1i}(j) = 1$

clearly,

$$AVAIL_i = INV(Y_{1i})$$
$$1 \le i \le r$$

— Associated with each output switch module is a *candidate* middle switch vector to which free links are available from that output SM.

$$CANDIDATE_k = \langle a_j / 1 \le j \le m \rangle \qquad 1 \le k \le r$$

where $a_j = 0$ if $X_{3k}(j) = 1$ or $a_j = 1$ if $X_{3k}(j) = 0$

clearly,

$$CANDIDATE_k = INV(X_{3k}) \qquad 1 \le k \le r$$

— A bit vector of size (r) is required for the algorithm to check the feasibility of realizing any input broadcast request. This vector is called ***CHECK*** and initialized to all one's. If any of its entries, is found to be zero at the end of computation then that particular request could not be satisfied. The following three conditions sould be checked before any input broadcast request could be realized.

(1) All the requested output ports must be free.

(2) There sould be at least one middle stage SM available for the input request.

(3) To establish a path from input stage SM to each and every required output stage SM there should be at least one middle stage SM available.

Now we give our algorithm. Any arbitrary input port issues a broadcast request I_i. Some of the steps in our algorithm are executed in parallel by different OVPs. To indicate the processor index on which certain step is executed we use its index in paranthesis after the step. Processors having the same index and satisfying the condition on index associated with the step are allowed to execute the step in parallel.

For example :

for (k **in** $\{1,2,.....,r\}$) **do in parallel**

$A_k := COPY(B_k) ; (k)$

means that $OVP_1, OVP_2, ..., OVP_r$ each duplicates its local vector B to local vector A.

ALGORITHM

Step 1.

1.1 $AVAIL_i := INV(Y_{1i})$; (i)

1.2 { Check if there is at least one middle stage SM available }

if $BITUNION(AVAIL_i) = 0$ **then return** (unsuccessful); (i)

1.3 { Check if all the requested output ports are free }

for $(j$ **in** $\{1,2,.....,r\}$ **and** $I[j]=1)$ **do in parallel**

$CHECK[j] := BITAND(UNION(INV(O_j), INV(Y_{3j})))$; (j)

1.4 **if** $BITAND(CHECK) = 0$ **then return** (unsuccessful); (i)

Step 2. **for** $(j$ **in** $\{1,2,.....,r\}$ **and** $I[j]=1)$ **do in parallel**

$CANDIDATE_j := INV(X_{3j})$; (j)

Step 3. { Compute the sufficient number of middle stage SMs }

for $(j$ **in** $\{1,2,.....,r\}$ **and** $I[j]=1)$ **do in parallel**

$S_j := INTERSECTION(AVAIL, CANDIDATE_j)$; (j)

Step 4. { Check if there is at least one middle stage SM available for each and every required output SM }

4.1 **for** $(j$ **in** $\{1,2,.....,r\}$ **and** $I[j]=1)$ **do in parallel**

if $BITUNION(S_j) = 0$ **then** $CHECK[j] := 0$; (j)

4.2 **if** $BITAND(CHECK) = 0$ **then return** (unsuccessful); (i)

Step 5. { Compute the necessary number of middle stage SMs }

for $(j$ **in** $\{1,2,.....,r\}$ **and** $I[j]=1)$ **do in parallel**

$T_j := KEEPLEFTMOST(S_j)$; (j)

Step 6. $R := Union(T_j \ ; \ j \text{ in } \{1,2,.....,r \} \text{ and } I[j]=1) ; (i)$

{ The vector R of size m contains the set of sufficient and necessary (one and only one middle stage SM for each required output SM) number of middle stage switches through which the input broadcast request can be satisfied. So R contains at most r 1s since there are r output stage SMs. If $R(k) = 1 \ , \ 1 \leq k \leq m$ then SM α_{2k} is to be connected to input switch i. }

{ Setting the connections for input broadcast request and updating the network state }

Step 7. **for** (k **in** $\{1,2,.....,m\}$ **and** $R[k]=1$) **do in parallel**

7.1 $CSET_k := INTERSECTION (INV(Y_{2k}) , I_i); (k)$

7.2 { Copy the vector $CSET$ to the column i of mask matrix A_{2k} }

{ This will realize the connection through the kth middle stage SM }

$A_{2k}[*, i] := COPY(CSET_k); (k)$

7.3 $Y_{2k} := UNION(Y_{2k}, CSET_k); (k)$

Step 8. **for** (j **in** $\{1,2,.....,r \}$ **and** $I[j]=1$) **do in parallel**

$X_{3j} := UNION(X_{3j}, T_j) ; (j)$

Step 9.

9.1 $Y_{1i} := UNION(Y_{1i} ; R); (i)$

9.2 { Copy vector R to the pth. column of mask matrix A_{1i} of input SM, α_{1i} }

{ where p is the position of the requesting input port within that SM }

{ This will realize the connection through i th input stage SM, α_{1i} }

$A_{1i}[*, p] := UNION (A_{1i}[*, p] , R) ; (i)$

Explanation of the algorithm

Let us consider two cases of input broadcast requests for the current state of the network at any given time as shown in figure 10.

Case 1: Input port *4* issues a broadcast request to output ports *2,5,9*. In terms of SMs in the output stage this request is expressed as $I_2 = < 1,1,1 >$ and in terms of output ports it is expressed as $O_1 = < 0,1,0 >, O_2 = < 0,1,0 >, O_3 = < 0,0,1 >$. We have the following vectors for the present state of the network.

$Y_{11} = < 1,1,1,0 >, Y_{12} = < 0,1,0,0 >, Y_{13} = < 0,0,0,1 >$

$Y_{21} = < 0,1,0 >, \quad Y_{22} = < 0,1,1 >, Y_{23} = < 1,0,1 >, Y_{24} = < 0,0,1 >$

$Y_{31} = < 1,0,1 >, \quad Y_{32} = < 1,0,1 >, \quad Y_{33} = < 1,1,0, >$

$X_{31} = < 0,0,1,0 >, X_{32} = < 1,1,0,0 >, X_{33} = < 0,1,0,1 >$

And the middle stage SM Mask Matrices are (columnwise) :

$A_{21} = \{ < 0,1,0 >, < 0,0,0 >, < 0,0,0 > \}$

$A_{22} = \{ < 0,1,0 >, < 0,0,1 >, < 0,0,0 > \}$

$A_{23} = \{ < 1,0,0 >, < 0,0,0 >, < 0,0,1 > \}$

$A_{24} = \{ < 0,0,0 >, < 0,0,0 >, < 0,0,1 > \}$

Now we walk through the algorithm. After Step 1, $AVAIL_2 = < 1,0,1,1 >$, since there are available middle switches and also available output ports, we continue. Step 2 is done concurrently and we have the following vectors : $CANDIDATE_1 = < 1,1,0,1 >$, $CANDIDATE_2 = < 0,0,1,1 >$ *and* $CANDIDATE_3 = < 1,0,1,0 >$. Step 3 is also done in parallel and it produces possible middle switch sets (set S's) for the request I_2 , resulting in $S_1 = < 1,0,0,1 >$, $S_2 = < 0,0,1,1 >$, $S_3 = < 1,0,1,0 >$. Union of these sets produces sufficient number of middle switches required. We need to find the necessary condition also. If any one of them were empty then clearly, the request could not be satisfied. Step 4 takes care of this case. Some of the output SMs could be reached by more than one middle stage SMs (as indicated by more than one entries of *1* in Ss). We choose one and only one middle stage SM for each output SM. Essentially, we do this by taking out all the *1' s* but the left most one from each one of S's.

This is done concurrently by Step 5 and it results in :

$T_1 = < 1,0,0,0 >$, $T_2 = < 0,0,1,0 >$, $T_3 = < 1,0,0,0 >$.

Step 6 computes the union of all T's, resulting in $R = < 1,0,1,0 >$, which contains the necessary and sufficient number of middle stage SM's. In Step 7, *OVP(1)* and *OVP(3)* do the following in parallel : $CSET_1 = < 1,0,1 >$ and $CSET_3 = < 0,1,0 >$.

Step 7 and Step 8 do the job of setting the connection and updating the network status. The new network status is shown in Figure 11 . So after the update, the new vectors are :

$Y_{11} = < 1,1,1,0 >$, $Y_{12} = < 1,1,1,0 >$, $Y_{13} = < 0,0,0,1 >$

$Y_{21} = < 1,1,1 >$, $Y_{22} = < 0,1,1 >$, $Y_{23} = < 1,1,1 >$, $Y_{24} = < 0,0,1 >$

$Y_{31} = < 1,1,1 >$, $Y_{32} = < 1,1,1 >$, $Y_{33} = < 1,1,1, >$

$X_{31} = < 1,0,1,0 >$, $X_{32} = < 1,1,1,0 >$, $X_{33} = < 1,1,0,1 >$

Mask Matrices of the middle stage SMs are (columnwise) :

$A_{21} = \{ < 0,1,0 >, < 1,0,1 >, < 0,0,0 > \}$

$A_{22} = \{ < 0,1,0 >, < 0,0,1 >, < 0,0,0 > \}$

$A_{23} = \{ < 1,0,0 >, < 0,1,0 >, < 0,0,1 > \}$

$A_{24} = \{ < 0,0,0 >, < 0,0,0 >, < 0,0,1 > \}$

Case 2 : Consider the network state of figure 10. Now input port 3 issues a broadcast request for output ports 5 and 9. So $I_1 = < 0,1,1 >$. $AVAIL_1 = < 0,0,0,1 >$. Since there is at least one available middle stage SM and also available output ports ($CHECK$ contains no *0* 's) we can proceed. Now *OVP(2)* and *OVP(3)* do the following in parallel :

$CANDIDATE_2 = < 0,0,1,1 >$

$CANDIDATE_3 = <1,0,1,0 >$

$S_2 = < 0,0,0,1 >$

$S_3 = < 0,0,0,0 >$

$CHECK = < 1,1,0 >$.

Now in Step 3 *OVP(1)* takes the bitwise union of the vector **CHECK**, which is *0*. So the controller aborts the control algorithm with unsuccessful message to the requesting input port. The connection to SM 3 of output stage can not be satisfied as it is clear from the figure 10.

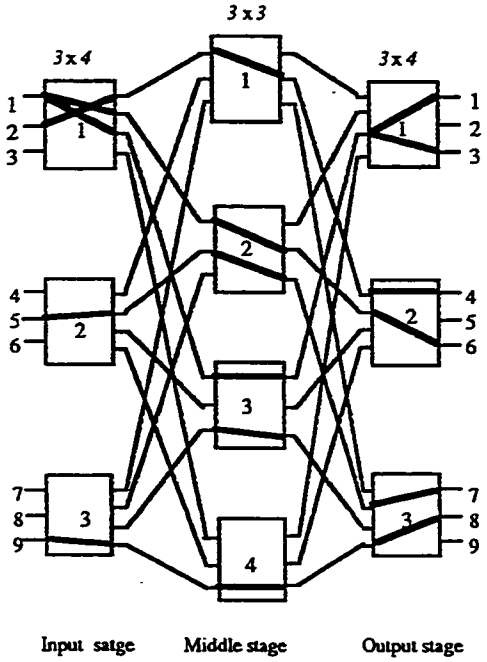


Figure 10
9 x 9 Optical Broadcast Network state befor the current input request from port 4 i.e. I_2=< *1, 1, 1* >

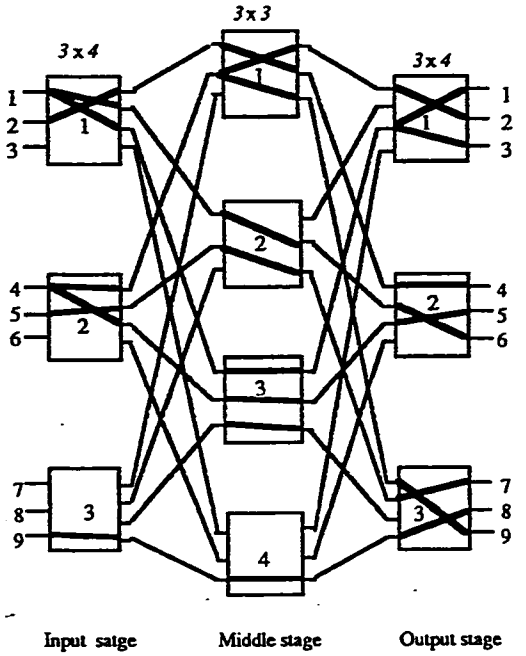


Figure 11
New state of the Network after the input request is satisfied

Complexity Analysis of the Algorithm

Each of the primitives used in the algorithm have *O(1)* time complexity as illustrated in section 5.1. Each and every step in the algorithm has *O(1)* time complexity. So the overall complexity of the network control algorithm is *O(1)*.

3.6 Conclusion

We have extended the design of Nonblocking Broadcast Network proposed by Yang and Masson using free space optics. This led us to exploit the inherent parallelism provided by optics to come up with a constant time network control algorithm on a network controller, using nonnumeric computation.

Chapter 4

Optical Realization of Constant Time Sorting Algorithm on a Processor Matrix with a Reconfigurable Inter-Processor Bus System

4.1 Introduction

Sorting problem is one of the most studied problems in computer Science. Given N numbers, the problem asks the numbers to be arranged in nondecreasing or nonincreasing order. A good number of literature on different methodologies and using different algorithms on conventional Von Neumann machine is available[AHO 74, KNUTH 73]. The time complexity of the best algorithm on such sequential computing is $O(NlogN)$. To seek for better solutions, parallel algorithms have been proposed on various parallel computation models[LAKS 84], [AKL 85], [AKL 89].

Theoretically, a constant time sorting can be achieved on an extremely powerful CRCW PRAM model (*Concurrent-read-concurrent-write parallel random access machine).* In this model concurrent access (read or write) to the same memory location is allowed and the write conflict is resolved by allowing all the processors to write the sum-of-all-writes only. Obviously this machine is too idealistic to be implemented with the state-of-the-art VLSI technology in terms of hardware complexity, cost and communication overhead required between processors.

Processor arrays interconnected by locally controllable global buses have attracted the attention of many researchers[KUMA 87, BOKH 84, AGGA 86]. Each processor within the grid structure can change the bus connection by reconfiguring the local buses dynamically. A bus connected system whose configuration can be changed by distributed control algorithm is called a *Reconfigurable Bus System (RBS).*

Using the reconfigurable bus based grid structure of RAM machines or conventional processors, there have been two designs to sort N numbers in constant time, proposed in the literature. The first one [WANG89] proposes a constant time sorting algorithm with hardware complexity of $O(N^3)$, that is, using RAM processors of the order of N^3. The sorting is done by enumeration. In the second paper [NAKA90] a new technique is proposed to reduce the enormous hardware complexity to $O(N^2 \log^2 N)$. The hardware complexity is still a concern when it comes to the question of implementation. Further more the reconfigurable bus system is more of a theoretical interest rather than a practical system as far as electronic implementation is concerned.

The capability of optics to do processing and communicating between points in parallel has a great potential for implementing a constant time sorter. In this Chapter, we propose a constant time sorting algorithm using all optical processing elements. We exploit the natural parallelism and the interference-free connectivity provided by free-space optics. Our proposed sorter has a hardware complexity of $O(N^2)$ in terms of number of optical processors to be defined later.

4.2 Proposed Optical Vector Processor Matrix Architecture

A *Reconfigurable Processor Matrix(RPM)* consists of processors arranged in a two dimensional grid with wrap around connections(Figure 1). Each processor is an *Optical Vector Processor (OVP)* (Appendix 1) with the extension that each has *4* parallel ports. These ports provide connections either to vertical or horizontal global bus. In an NxN RPM, *OVP(i, j)* stands for OVP in i^{th} row, j^{th} column, $0 \le i, j \le N-1$. These ports are denoted N,S,W,E and each port is capable of communicating $W \ge N$ (W is the word length of the processor) bits of information in parallel. The ports of the adjacent processors facing each other are connected by fixed buses, that is, *S(OVP(i, j))* and *N(OVP(i, (j+1) modulo N))* are connected by fixed optical fibres. Similarly, *E(OVP(i, j))* and *W(OVP((i+1) modulo N, j))* are also connected statically.

4.2.1 Optical Vector Processor (OVP)

An OVP perform some primitive operations on binary vectors as discussed in Chapter III. We extend the design of OVP, adding *4* W -bit parallel ports, where W is the precision of the proposed processor. The *4* ports are : N,S,W and E. To indicate the ports on certain *OVP(i, j)*, we use the index of that particular processor as argument with the port name. For example, port N on *OVP(2,3)* is to be denoted as N*(2,3)*. Each processor can connect its ports W with E and ports N with S by writing the required grating patterns on a holographic mask matrix M. The mask matrix is of size WxW, where W is the vector-width or word-length of the OVP. To connect any two opposite ports (such as, N with S and W with E), the OVP records a fixed grating pattern on all the elements of the forward diagonal of the connection mask matrix, M, to create the required connection.

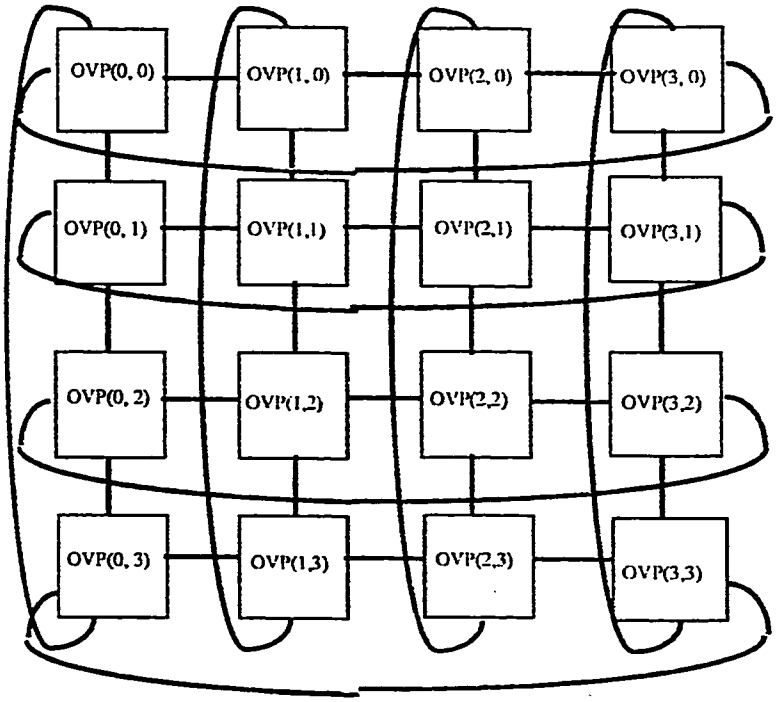


Figure 1
***4 x4* Reconfigurable Processor Matrix with wrap around bus connection**

Erasing an existing grating pattern and then writing a new one takes constant time, since there are W elements to be written and the patterns are fixed. So each OVP can change the local bus connection in constant time resulting in dynamic reconfuguration of the global bus. To establish the required local connection demanded by the global algorithm each OVP executes CONNECT instruction. This instruction is discussed later in this section.

Associated with each OVP, there is a laser source vector **C** and a detector vector **D**. Vector **C** of source is used to send information to the global bus whereas vector **D** of photodetectors is used to read the value(binary vector) on the global bus. Sending to and receiving information from the bus are carried out by simple beam steering elements like Beam Splitters.

4.2.1.1 Comparing Two Numbers on OVPs

Since each OVP performs nonnumerical processing on vectors as discussed in Chapter 3, we define some higher level relational operators on top of primitive vector operations of OVPs. These primitive vector operations are : COPY, INV, BITUNION, BITAND, UNION, INTERSECTION, KEEPLEFTMOST. These operations are the basis for the higher level relational operators . We assume that A and B are binary vectors and the following operators compare their values expressed in binary and return the result.

4.2.1.1.1 NumGT (vector1, vector2)

To find whether the numerical value of a vector (binary value) is greater than that of the other, we use the following procedure.

Procedure NumGT(A, B)
begin
Step 1 *X := UNION(INTERSECTION(A, INV(B)), INTERSECTION(INV(A), B));*
Step 2 *X := KEEPLEFTMOST(X);*
Step 3 **if** *BITUNION(X) = 0* **then return(0);**
Step 4 **if** *BITUNION(INTERSECTION(X, A)) =0* **then return(** *0* **) else return(1);**
end { NumGT }

As for example, consider A = < *0,1,0,1,1,0* > and B = < *0,0,1,1,1,1*>. Let us follow the above procedure. After Step 1, X = < *0,1,1,0,0,1*> and after Step 2 X =< *0,1,0,0,0,0*>. Since the BITUNION(X) is not equal to zero we move to Step 4. Taking INTERSECTION between X and A results in < *0,1,0,0,0,0*>. As BITUNION(<*0,1,0,0,0,0*>) is not equal to zero the procedure returns *1* , meaning A > B.

4.2.1.1.2 NumLT (vector1, vector2)

To find whether the numerical value of a vector (binary value) is less than that of the other, we use the following procedure.

```
Procedure NumLT(A, B)
begin
    C := UNION( INTERSECTION(A , INV(B )), INTERSECTION( INV(A ), B ));
    C := KEEPLEFTMOST( C );
    if BITUNION(C ) = 0 then return(0);
    if BITUNION( INTERSECTION( C , A )) =0 then return( 1) else return(0);
end{ NumLT }
```

4.2.1.1.3 NumEQ (vector1, vector2)

This procedure finds whether two vectors are equal or not.

```
Procedure NumEQ( A, B)
begin
    C := UNION( INTERSECTION(A , INV(B )), INTERSECTION( INV(A ), B ));
    C := KEEPLEFTMOST( C );
    if BITUNION(C ) = 0 then return(1) else return(0);
end { NumEQ }
```

4.2.1.1.4 NumGE (vector1, vector2)

To find whether the numerical value of a vector is greater than or equal to that of the other, we define the following procedure.

```
Procedure NumGE( A, B)
begin
        C := UNION( INTERSECTION(A, INV(B)), INTERSECTION( INV(A), B));
        C := KEEPLEFTMOST( C );
        if BITUNION(C) = 0 then return(1);
        if BITUNION( INTERSECTION( C, A)) =0 then return( 0 ) else return(1);
end { NumGE }
```

4.2.1.1.5 NumLE (vector1, vector2)

The following procedure checks whether numerical value of the first vector is less than or equal to that of the second vector.

```
Procedure NumLE( A, B)
begin
        C := UNION( INTERSECTION(A, INV(B)), INTERSECTION( INV(A), B));
        C := KEEPLEFTMOST( C );
        if BITUNION(C) = 0 then return(1);
        if BITUNION( INTERSECTION( C, A)) =0 then return( 1 ) else return(0);
end { NumLE }
```

4.2.1.1.6 NumNE (vector1, vector2)

To find the non equality between two vectors we define the following procedure.

Procedure NumNE(A, B)
begin
    C := UNION(INTERSECTION(A, INV(B)), INTERSECTION(INV(A), B));
    C := KEEPLEFTMOST(C);
    if *BITUNION(C) = 0* **then return**(*0*) **else return**(*1*);
end { NumNE }

As we have shown earlier that all the primitive vector operations take constant time, the above mentioned relational operators also take constant time. We will use these relational operators in our sorting algorithm, which will be given later.

4.2.1.2 Connecting Local Buses

As we have mentioned earlier, each Optical Vector Processor(OVP) has four ports: **N,S,W** and **E**. Each OVP can form a local bus by connecting either **NS** or **EW**. This is done by writing two directional gratings, *NorthSouth (NS)* or *EastWest (EW)*, on the forward diagonal of the interconnection mask matrix, M (Figure 2). The bus connection can be straight or shifted one position to the right (to shift each beam one position to the right). If the NS or EW gratings are copied on the forward diagonal of the mask matrix then the straight bus connection is established(Figure 3.a, 3.b). Again if the gratings are written on the cells that lies on a line parallel to the diagonal but below (as shown in Figure 3.c, 3.d), then the *Shift Right (SR)* connection is established. We parametrised the type of connection between ports by *0* meaning straight connection and by *1* meaning right shifted connection. To provide the desired local bus connection capability each OVP executes the following instructions. The first parameter indicates the direction of the grating(EW/NS) and the second parameter indicates the type of gratings(*0/1*, meaning straight or shifted).

CONNECT(EW, *0*) : Writes EW gratings on the forward diagonal of the mask matrix, **M**. This establishes the straight local bus connection between ports **E** and **W**. This is illustrated in Figure 3(a).

CONNECT(NS, *0*) : Writes NS gratings on the forward diagonal of the mask matrix, **M**. This establishes the straight local bus connection between ports **N** and **S**. This is shown in Figure 3(b).

CONNECT(EW, *1*) : Writes EW gratings of type SR on the mask matrix, **M** to establish the shifted local bus connection between ports **E** and **W**. This is illustrated in Figure 3(c).

CONNECT(NS,*1*) : Writes NS gratings of type SR on the mask matrix, **M**. This establishes the shifted local bus connection between ports **N** and **S**. This is shown in Figure 3(d). Writing a fixed mask pattern(either EW gratings or NS gratings) takes constant time. So each OVP executes the CONNECT instruction in constant time.

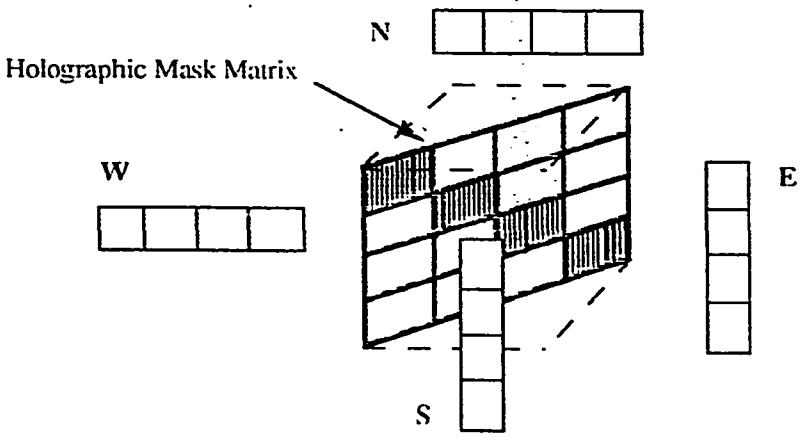


Figure 2
3-D view of the Straight local bus connection, EV within each OVP by writing on the forward diago the Mask matrix EW (NS) gratings.

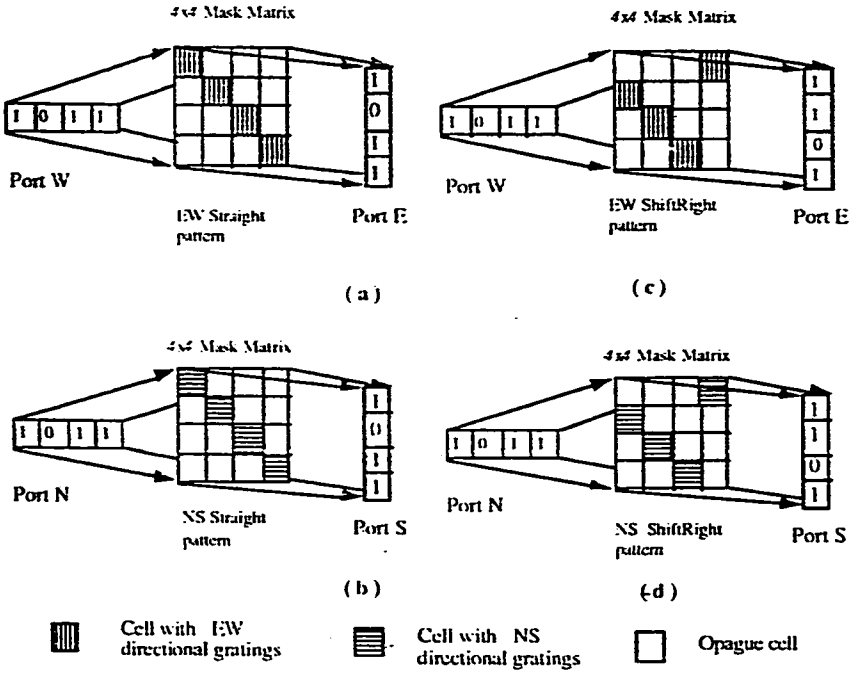


Figure 3. Establishing local bus between ports
(a) EW, Straight (b) NS , Sraight
(c) EW , ShiftRight (d) NS , ShifhtRight

4.2.1.3 Data Broadcast

Each OVP, connected with a global bus, broadcasts a data item expressed in binary, to all the other OVPs connected to the global bus through individual local bus connection. The OVP which wants to broadcast a data, loads its source vector, C with the binary representation of the data. It then lights the source vector. In one clock cycle the binary data is received/detected by all the OVPs (including itself) connected to the global bus. We assume that the clock period is greater than the travel-time of the broadcast beam, which travels at the speed of light. Since clock speed can not be achieved as high as the speed of light, our assumption is not unreasonable to make. So an OVP can broadcast a data item to all the other OVPs connected on the bus, in one clock cycle. Each OVP performs this broadcast executing the **BROADCAST(<binary sequence>)** instruction. This is shown in Figure 4.

4.3 Computing the Sum of a Binary Sequence on a Reconfigurable Processor Array in Constant Time

Consider a binary sequence of length N and an interconnected Processor Array consisting of N OVPs. Each one of the binary digits is assigned to each OVP in the same order. Each processor now establishes its local bus connection in either of two ways depending on its binary digit. If the binary digit is 0, the OVP establishes a straight local bus connection . If the binary digit is 1, the OVP erases the current local bus connection and writes the SR mask pattern on the interconnection mask matrix (as shown in Figure 4) to establish a shifted local bus connection. Any beam of light passing through this OVP is shifted one position to the right. Now the first OVP in the array broadcasts $<1,0,0,0,....0>$ on the established global bus. The last processor in the array checks its received value and finds the position(j, $0 \leq j \leq N-1$) of 1 in the vector. It then stores either k *or* $k+1$, depending on its binary digit equal to 0 or 1. Each OVP takes constant time to write the SR pattern on its interconnection mask matrix.The following algorithm finds the sum of a binary sequence of length N on a processor array of N processors.

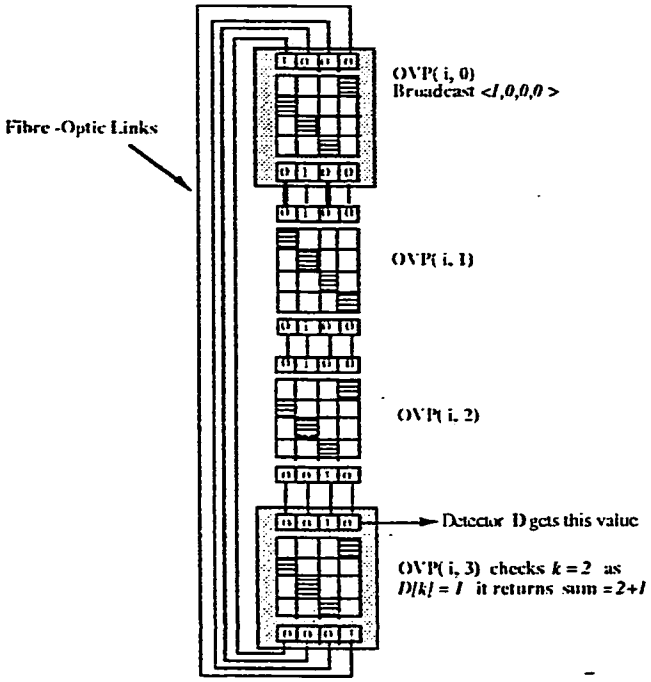


Figure 4
Individual Mask settings of each processors to find the Sum of the binary sequence <1, 0, 1, 1> on an array of 4 *OVPs* without any arithmatic computation.

ALGORITHM 1 : SumBinarySequence(i) : return $rank_i$;

{ A binary sequence $<d_0, d_1 . d_2, \ldots d_j, \ldots d_{N-1} >$ is assigned to OVP(i, j). $0 \le j \le N-1$, i.e.}
{ to all the OVPs on column k and this algorithm returns the sum of the binary sequence that }
{ are assigned to the OVPs of column i. }
{ Each d_j (0 / 1) in OVP(i, j) is expressed as a vector, R of length N. The index i}
{indicates the column number of the processor, which will be used later in algorithm 2}

Step 1 { Establish the global bus by forming straight local bus connection }

for all (j **in** { $0,1,2,\ldots,N-1$ }) **do in parallel**
CONNECT (NS,0): (i , j)

Step 2 { Establish the shift connection in local buses if the binary digit is 1 }

for all (j **in** { $0,1,2,\ldots,N-1$ }) **do in parallel**
if BITUNION(**R**) = 1 **then** CONNECT(NS,1); (i , j)

Step 3 { $OVP(i,1)$ broadcasts a vector $< 1,0,0,\ldots,0 >$ to all the other OVPs using the}
{ global bus established in Step2 }

BROADCAST($<1,0,0,\ldots,0>$); (i , 1)

Step 4 {*OVP(i, N-1)* detects the position of *1* in vector **D** it received in Step3}
{ and returns the position, k ($0 \leq k \leq N-1$) such that $D[k]=1$, as the sum }
{ of the binary sequence if its own binary value is *0* otherwise returns $k+1$ }
{ as the sum of the binary sequence }

if BITUNION(**R**) = *0* **then return(** k , such that $D[k]=1$ **);**
else return($k+1$ **);**
endif; *(i , N-1)*

end { SumBinarySequence}

It is clear that the algorithm to compute the sum of binary sequence of length N, takes constant time on a processor array of N processors. An example is illustrated in Figure 4.

Lemma 1. *N Optical Vector Processors (OVPs) connected in a Reconfigurable Bus, compute the sum of a binary sequence of length N in constant time.*

Proof. From the definitions of all the primitive instructions used in the algorithm, it is obvious that each step in the algorithm takes constant time. Hence the total time complexity of Algorithm 1 is *O(1)*.

4.4 Sorting N numbers on a Reconfigurable Processor Matrix(RPM) of NxN Optical Vector Processors(OVPs) in constant time

Consider a RPM(as illustrated in Figure 1) of size NxN. N numbers denoted as v_i , $0 \le i \le N-1$, are to be sorted in nondecreasing order. These N numbers are assigned to the respective OVPs at the top row of the RPM, that is, $OVP(i, 0)$ gets v_i ; $0 \le i \le N-1$.

The sorting algorithm is based on enumeration. First the numbers are broadcast vertically to all the OVPs columnwise. So each number is thus spread in the respective column. Then each OVP on the diagonal broadcasts its number horizontally. As a result, each OVP(i , j) has two numbers: v_i and v_j. Each OVP(i , j) compares its two numbers and stores the result, which is either 0 or 1. So after the comparison step, the sum of the binary values on each column i is the rank of data item v_i .This sum is done easily by using Algorithm 1, discussed in the last section. Once the ranking is done, the data items are arranged according to their computed ranks. Now we give the sorting algorithm.

ALGORITHM 2. Sort (N)

Step 1

{ Form N vertical buses of N OVPs in each column by making straight local bus}
{ connections on each $OVP(i, j)$: $0 \le i, j \le N-1$ }

1.1 **for all** (i, j **in** { $0,1,2,\ldots,N-1$ }) **do in parallel**
CONNECT(NS,0): (i, j)

{ Broadcast v_i to all the OVPs columnwise so that every processor $OVP(i, j)$ }
{ has v_i after this step is executed }

1.2 **for all** (i **in** { $0,1,2,\ldots,N-1$ }) **do in parallel**
BROADCAST(v_i): ($i, 0$)

Step 2

{ Form N horizontal buses of N OVPs in each row by making straight local bus}
{ connections on each $OVP(i,j)$; $0 \le i,j \le N-1$ }

2.1 **for all** (i , j **in** { $0,1,2,....,N-1$}) **do in parallel**
CONNECT(EW,0) ; (i , j)

{ The diagonal OVPs broadcast v_j to all the OVPs row-wise so that every processor}
{ $OVP(i,j)$ gets v_j after this step is executed }

2.2 **for all** (j **in** { $0,1,2,.....N-1$ }) **do in parallel**
BROADCAST (v_j); (j , j)

Step 3

{ In each OVP(i,j), R is a vector initialized to 0 i.e. $R = <0,0,0,...,0>$ }
{ R stores the result of comparison between v_i and v_j, which is either 0 or 1 }

for all (i , j **in** { $0,1,2,....N-1$ }) **do in parallel**
if NumGT(v_i , v_j)=1 **then** R := $<1,0,0,...,0>$;
else if NumEQ(v_i , v_j) =1 **and** $i>j$ **then** R := $<1,0,0,...0>$;
else R :=$<0,0,0,....0>$; **endif**

Step 4

{ **Ranking Phase.** Now we call the algorithm1 to compute the binary sequence on }
{ each column to get the rank v_i, $0 \le i \le N-1$. All N ranks are stored in OVPs that }
{ are located in the last row, i.e., $OVP(i, N-1)$, $0 \le i \le N-1$. }

for all (i **in** { $0,1,2,....N-1$ } **do in parallel**
call SumBinarySequence (i) ;

Step 5 { **Rearrangement Phase.**}

{ Form N vertical buses of N OVPs in each column by making straight local}

{bus connections on each $OVP(i,j)$: $0 \le i,j \le N-1$. }

5.1 **for all** (i,j **in** { $0,1,2,\ldots,N-1$ }) **do in parallel**

CONNECT(NS,0); (i,j)

{ The last-row OVPs broadcast $rank_i$ to all the OVPs vertically so that every }

{ processor $OVP(i,j)$ gets $rank_i$ after this step is executed. }

5.2 **for all** (i **in** { $0,1,2,\ldots,N-1$ } **do in parallel**

BROADCAST($rank_i$) (i ,$N-1$)

Step 6

{ Form N horizontal buses of N OVPs in each row by making straight local bus }

{ connections on each $OVP(i,j)$: $0 \le i,j \le N-1$. }

6.1 **for all** (i,j **in** { $0,1,2,\ldots,N$ }) **do in parallel**

CONNECT(EW,0) : (i,j)

{ $OVP(i, rank_i)$ broadcast v_i to all the OVPs horizontally. }

{ $OVP(i,j)$ receives the value in a local vector *temp*. }

{ After Step 6.2 N numbers are sorted row-wise in nondecreasing order }

6.2 **for all** (i **in** { $0,1,2,\ldots,N-1$ }) **do in parallel**

BROADCAST(v_j) ; ($i, j = rank_i$)

Step 7 { **Collecting the sorted numbers in the first-row processors.** }

{ Form N vertical buses of N OVPs in each column by making straight local bus}
{ connections on each $OVP(i, j)$: $0 \le i, j \le N-1$. }

7.1 **for all** (i, j **in** { $0,1,2,\ldots,N-1$ }) **do in parallel**
CONNECT(NS,0); (i, j)

{ Each diagonal processor, OVP(i, i) broadcast *temp*, the number it received in }
{ Setp 6.2 vertically. $OVP(i, 0)$ stores the received value in v_i' }

7.2 **for all** (i **in** { $0,1,2,\ldots,N-1$ } **do in parallel**
BROADCAST(*temp*) (i, i)

After Step 7, the top row processors contain the sorted sequence. To say that the array is loaded in the top row by the unsorted sequence initially, and after Step 7, the sorted sequence would be available in the same row. The algorithm is explained with an example in Figure 5.

Lemma 2. N numbers can be sorted on a Reconfigurable Processor Matrix (RPM) consisting of NxN Optical Vector Processors (OVPs) in constant time.

Proof. Step 4 of **Sort** algorithm is a parallel call to procedure **SumBinarySequence.** This procedure is simultaneously invoked by N column-processors to compute the ranks of N input data items. As we already proved that **SumBinarySequence** procedure has a time complexity of $O(1)$, the whole step 4 in the **Sort** algorithm takes constant time. Furthermore, all the other steps used in the above algorithm also takes constant time, as is evident from the definitions of the primitive and higher-level instructions of OVPs. This leads to the proof of the lemma.

—

4.5 Conclusion

We have extended the design of a Reconfigurable Processor Matrix to sort N numbers in constant time. We use the technique of sorting by enumeration as done by [WANG89] and [NAKA90] . Our approach significantly differs from that of the other's in that we exploit the dynamic reconfigurable characteristics of optics. The vector processing capability of optics in natural parallelism takes advantage in our design to reduce the hardware complexity down to $O(NxN)$. Future work involves further reduction of hardware, perhaps by increasing the time-complexity, marginally.

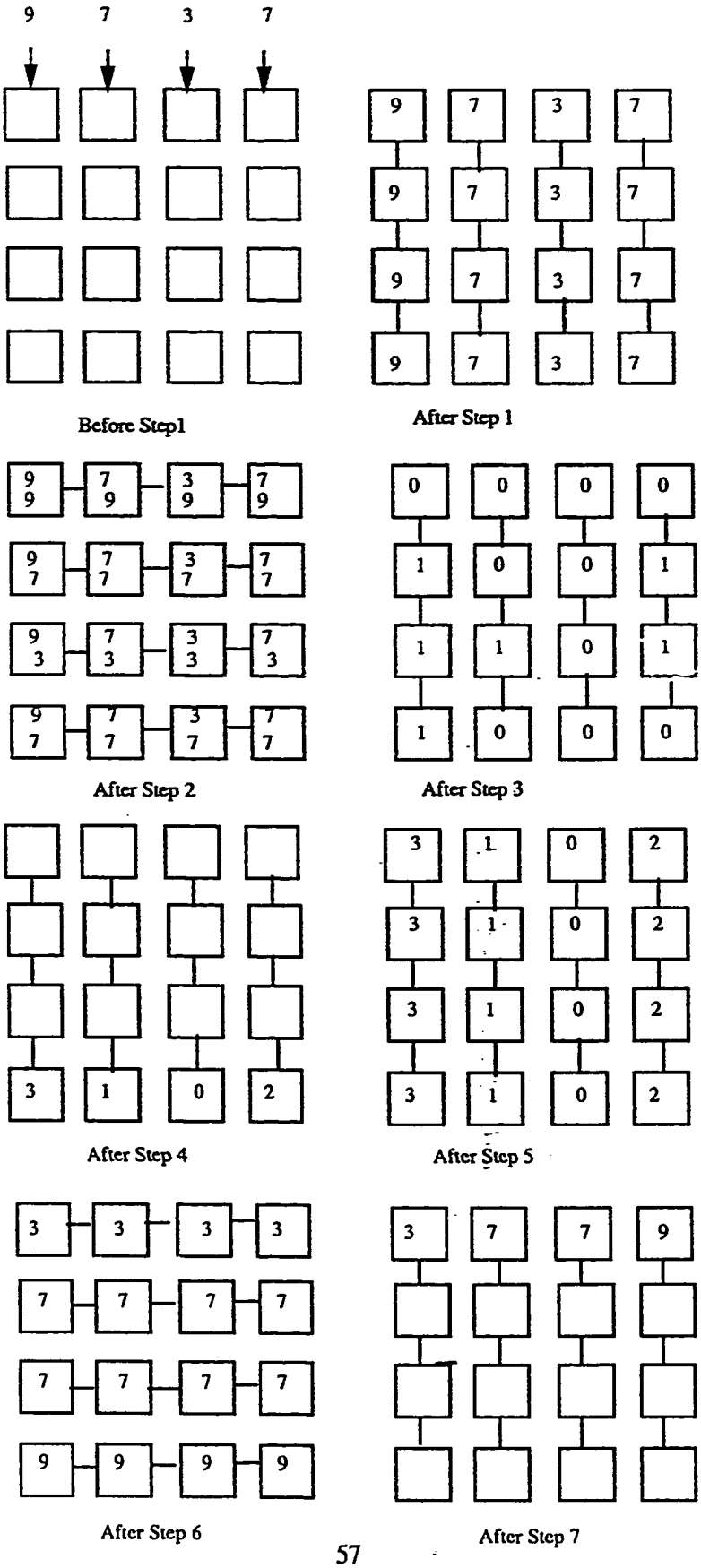


Figure 5.
Sorting numbers 9, 7, 3, 7 on an RPM of size *4* x*4*

Chapter 5

Binary-Tree-Computation on Optical Reconfigurable Vector-Processor Array

5.1 Introduction

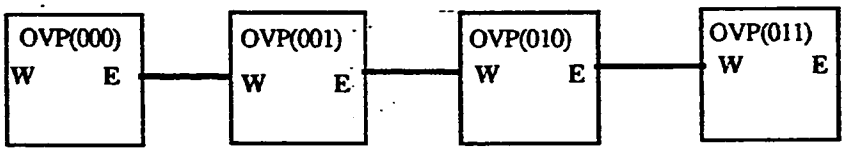
Binary tree is a basic structure in parallel computations. In efficient parallel computations, binary tree structure appears either in the form of data structure for computation or in the form of structure of processors operating in parallel. The Balanced-Binary-Tree method is a powerful paradigm in parallel algorithms[GIBB 89]. Each internal node of a tree corresponds to the computation of a subproblem with the root corresponding to the over-all problem. Any given problem is solved by bottom-up approach and the result of computation is then broadcast to each leaf node in the top-down approach. Subproblems solved by the internal nodes in a given level correspond to the parallel execution by equal number of processors. The size of the problem (n) is assumed to be a power of two to ensure the balanced binary tree structure. If this is not the case, then a minimum number of dummy elements can always be added to ensure that requirement. The depth of the balanced binary tree is thus bounded by $\lceil \log n \rceil$. Since each level of the tree is processed in parallel by employing as many processors as there are nodes in that level, complexity of the whole computation is logarithmic i.e. $O(\log n)$. The maximum number of processors needed will be $n/2$ at the leap-level and each successive level requires geometrically decreasing number of processors.

Processor array interconnected by locally controllable global bus has the potential of embedding the Binary-Tree interconnection network topology. Any computation solvable on binary tree interconnection network of processors can also be solvable on a reconfigurable processor array with broadcast capability. Each processor within the array can change its local bus connection dynamically. Thus the global bus is split into several disjoint subbuses. A bus connected

processors system with reconfiguration capability is called *Reconfigurable Bus System (RBS)*. In this Chapter we propose an optical RBS consisting of Optical Vector Processors(OVPs) and then give an algorithm to embed the Binary-Tree computations on this RBS. Essentially, we simulate a Binary-Tree interconnection on RBS.

5.2 Optical Reconfigurable Bus System (ORBS)

An ORBS consists of Optical Vector Processors (OVPs) interconnected by optical links to form a one-dimensional mesh. Each processor in an ORBS of size N has a unique index expressed in binary between 0 and $N-1$. Each OVP(i), $0 \leq i \leq N-1$, has two parallel ports; called E(East) and W(West). The ports of the adjacent processors facing each other are connected by fixed fibre optic links, forming a global bus through the processors. Each OVP can connect or disconnect this global bus running through itself. E(i), that is, port E on processor OVP(i) is connected with W($i+1$); $0 \leq i \leq N-2$. Figure 1 illustrate an ORBS of size 4.



Fiqure 1
Optical Reconfigurable Processor Array
(ORBS)

5.2.1 Optical Vector Processor

An OVP is the basic processing unit. It perform a set of primitive operations on binary vectors as discussed in Chapter 3. These operations are COPY, INV, BITUNION, BITAND, UNION, INTERSECTION, KEEPLEFTMOST. Based on these primitive operations each OVP also

performs some higher level relational operations such as, NumGT, NumLT, NumEQ, NumGE, NumLE and NumNE(Chapter 4). Each OVP has a fixed number of registers where it can do its local processing. It can send and receive a binary vector to and from its ports. To provide the global connectivity, it either connects or disconnects its ports.

5.2.1.1 Connecting Local Bus

As we have mentioned earlier, each OVP has 2 ports: **E** and **W**, it can form a local bus by connecting **E** to **W** within itself through a mask matrix **M**. This is done by writing transparent masks on the forward diagonal of the mask matrix, M(Figure 2). This mask pattern is called *Straight(ST)* mask. This will establish a straight local bus connection. Sometimes it is needed by the global algorithm to have Right Shifted or LeftShifted (by *1* position) local bus connection so that each incoming beam of light is shifted *1* position to the right or left while passing through each OVP. This is done by two mask patterns called *Shift Right(SR)* and *ShiftLeft(SL)*. Writing and erasing a fixed mask pattern on the mask matrix takes constant time. We parametrised the type of local connection between ports by *ST* meaning straight bus and by *SR* meaning right shifted bus and by *SL* meaning left shifted bus connection. To provide the desired dynamic reconfigurability of the global bus, each OVP either connects or disconnects its local bus connection by the following two instructions.

CONNECT(*ST*) / DISCONNECT(*ST*) : Writes or copies the transparent / opaque ST mask pattern on the mask matrix, **M**. This establishes/disconnects the straight local bus between its ports.

CONNECT(*SR*) / DISCONNECT(*SR*) : Writes the transparent / opaque SR mask pattern on the interconnection mask matrix, **M**. This establishes/disconnects a right shifted local bus between its ports.

CONNECT(*SL*) / DISCONNECT(*SR*) : Writes or copies the transparent / opaque SL mask pattern on the mask matrix, **M**. This establishes/disconnects the left shifted local bus between its ports.

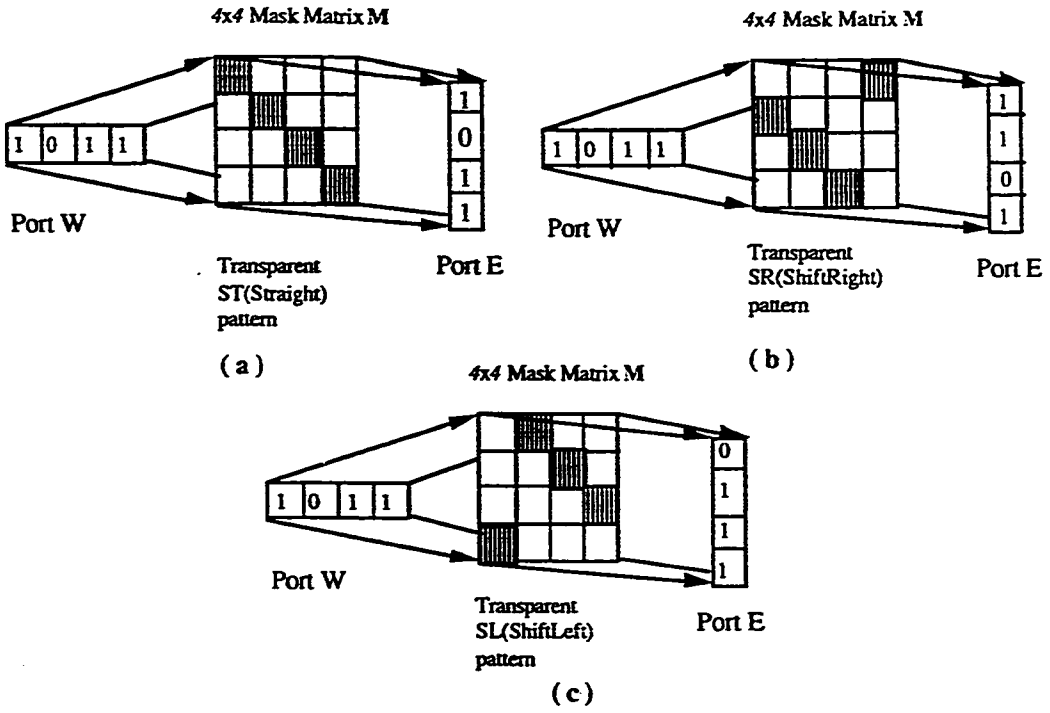


Figure 2. Establishing local bus between ports
(a) Straight (b) ShiftRight (c) ShiftLeft

5.2.1.3 Data Broadcast

Each OVP connected either on a global bus or on a split bus, can broadcast a data item expressed in binary, to all the OVPs connected on the same bus.The processor which wants to broadcast a data item, loads its source vector, **C** with the binary representation of the data. It then initiates the source vector to send the data on the bus. In one clock cycle, the transmitted vector is received by all the OVPs(including itself) connected to the bus. Here we assume that the clock period is greater than the travel-time of the broadcast vector-beam, which travels at the speed of light. Since clock speed can not be achieved as high as the speed of light, our assumption is not unreasonable to make. So each OVP can broadcast a data item to the other OVPs connected to the same bus, in one clock cycle. Each OVP does broadcasting by executing the **BROADCAST(<*binary sequence*>)** instruction.

5.3 Binary Tree Computations on ORBS

An ORBS can effectively and efficiently simulate the binary tree interconnection based parallel computations. The $O(\log n)$ time complexity of computations on balanced-binary-tree structure of processors is also maintained without increasing (rather with a reduction in the number of processors) the hardware complexity in ORBS. In addition, the $O(1)$ broadcast capability provided by ORBS increases the efficiency. In balanced-binary-tree architecture to broadcast a data to all the other processors, it takes $O(\log n)$ time. Again, the number of processors in the balanced-binary-tree interconnection network to solve a problem of size n, is $2n-1$ and the number of links required is $2n - 2$, whereas the same problem can be solved on an ORBS of n OVPs with only an $(n-1)$ -linked-segment broadcast bus linking the processors. Now we give two generic algorithms that simulates the balanced-binary-tree computations one in bottom-up fashion and the other in top-down manner on an ORBS.

Preliminaries for the Algorithms

- A sequence of N numbers v_i, $0 \leq i \leq N-1$. There is a presumption in such a computation that N is some power of 2. If this is not the case, then addition of a minimum number of dummy elements to the problem can always make the problem-size equal to some power of 2 , without affecting the original problem at all. So without any loss of generality we can say that N is some power of 2. We also assume an ORBS of N processors.

- Each OVP on the ORBS has a unique index in the range $[0, \ldots, N-1]$ and the index i expressed in binary as $i = i_{\log N-1} \ldots i_2 i_1 i_0$.

- The algorithm is distributed. Associated with each instruction in the algorithm there is an instruction mask. This instruction mask is a Boolean function used by each processor to check whether it is eligible to execute that instruction or not. This mask is written within parenthesis

right after the instruction itself. For example, in the instruction

for all (i **in** { $1, 2, ..., N-1$}) **do in parallel**

BROADCAST(<$1,0,0,0,0,0$>); ($i_0 = 1$)

($i_0 = 1$) is a mask that allows only those OVPs to execute the BROADCAST instruction whose index has bit 0 equal to 1 i.e. all the odd processors.

• Associated with each OVP, there is a Boolean value *awake*. Each processor checks this flag to determine whether it can perform certain operation in a certain parallel step.

• All the even numbered processors, including OVP(0), i.e. OVP(0), OVP(2), OVP(4) . . . etc., act as internal nodes of subsequent binary trees. A binary subtree rooted in certain even numbered processor has a right child, which is to the right of the root by a distance equal to the half of the current bus-length. In the upward direction of binary-tree-computation, buses of length $2, 4, ...N$ is formed with each iteration of the total of $\log n$ iterations. Again, in the downward direction of computation the global bus of size N is split into 2 parts with each iteration. To keep track of the previous computed results received from its right child; each internal-node-processor has an array called *prev*. Its size depends on the depth of the subtree. For example, OVP(0) acts as the root for subtrees of sizes $2, 4, 8, . . .N$. So its array *prev* has a size of $\log N$, which is the maximum.

• The algorithms assume the availability of 3 generic functions, namely f_{root} , $f_{internal}$ and f_{leaf} to be executed by the root processor; internal-node processors and leaf processors respectively. These functions are used as template parameter for our two generic algorithms.

Algorithm 1. **Procedure UpwardPass($f_{internal}$, f_{leaf})**

Step 1. { N numbers v_i , $0 \le i \le N-1$ are assigned to N OVPs each }

for all (i **in** $\{0, 1, ..., N-1\}$) **do in parallel**

begin

$s_i := f_{leaf}(v_i)$;

$awake := 1$;

end

Step 2. { $\log N$ times iteration }

for $k := 0$ **to** $logN-1$ **do**

begin

2.1 **for all** (i **in** $\{0, 1 ..., N-1\}$ **and** $awake = 1$) **do in parallel**

2.1.1 CONNECT(ST); ($i_k = 1$) / connect straight local bus /

2.1.2 BROADCAST(s_i); ($i_k = 1$)

2.1.3 $awake = 0$; ($i_k = 1$)

2.2 { Each processor on the corresponding subbus receives the value which}

{ is broadcast in the previous step in s_{rchild} }

for all (i **in** $\{0, 1, ..., N-1\}$ **and** $awake = 1$) **do in parallel**

2.2.1 $prev_i[k] := s_{rchild}$; ($i_k = 0$);

2.2.2 $f_{internal}(s_i, s_{rchild})$; ($i_k = 0$)

end

Algorithm 2. **Procedure DownwardPass(** f_{root} , $f_{internal}$, f_{leaf} **)**

Step 1. $f_{root}(s_i); (i = 0)$
$awake_i := 1; (i = 0)$

Step 2. **for all (** i **in** $\{0, 1, ..., N-1\}$ **) do in parallel**
{ Form the global bus of size N }
CONNECT(ST);

Step 3. { $\log N$ times iteration }
for $k :=$ $logN - 1$ **to** 0 **step** -1 **do**
begin
3.1 **for all (** i **in** $\{0, 1, ..., N-1\}$ **and** $awake = 1$ **) do in parallel**
$f_{internal}(s_i, prev_i[k], s_{rchild})$;

3.2 **for all (** i **in** $\{0, 1, ..., N-1\}$ **and** $awake = 1$ **) do in parallel**
3.2.1 BROADCAST(s_{rchild}); ($i_k = 0$)
{ Processors connected on the corresponding subbuses receive the}
{ value broadcast in this step in s_i }
3.2.2 { disconnect the bus into 2 parts }
DISCONNECT(ST); ($i_k = 1$)
$awake = 1$; ($i_k = 1$)
end

Step 4. **for all (** i **in** $\{0, 1, ..., N-1\}$ **) do in parallel**
$f_{leaf}(s_i)$;

Analysis

Algorithm 1

Step 1 is done in a single clock cycle. Step 2 is executed *log N* times and both steps 2.1 and 2.2 are done in parallel in constant time. So the overall time complexity is *O(log N)*.

Algorithm 2

Step 1, 2 and Step 4 are done in constant time. Step 3 is executed *log N* times and both substeps 3.1 and 3.2 are done in parallel in constant time. So the overall time complexity is again *O(log N)*.

Lemma. *An ORBS can simulate Binary-Tree Computations.*

Proof. For all computations involving realizable binary associative operations on Optical Vector Processors(OVPs), algorithm Binary_Tree_Computation is generic. Hence we can instantiate this algorithm to solve a problem which is solvable on binary-tree interconnection network of processors.

Examples of Binary-Tree Computation.

• Finding Maxima

Lets instantiate the algorithms to find the maximum (minimum) of N numbers. We assume N is some power of 2. If this not the case, we can insert some $-\infty$ ($+\infty$) to make N equal to some power of 2.

Algorithm. Find_Maxima

Step 1. Call UpwardPass(NumGE, COPY)

Step 2. { Form a global bus }
for all (i **in** $\{0, 1, ..., N-1\}$) **do in parallel**
CONNECT(ST);

Step 3. BROADCAST(s_i) ($i=0$)

• Prefix Sum Computation

Given N numbers v_i; $0 \le i \le N-1$, and an associative θ, it is required to compute $v_1 \, \theta \, v_2 \, . \, . \, . \theta \, v_i$, $0 \le i \le N-1$. This associative operator will be used to replace the $f_{internal}$ function in the template of algorithm *UpwardPass*. Let us assume θ is an *addition* operation, i.e. +. Again, let us define an operation to replace the $f_{internal}$ function in the template of *DownwardPass*. This function takes current value in s_i and split it into two parts: new s_i and s_{rchild} for its two children. For the case of θ to be *addition* (+), s_{rchild} gets the original value of s_i and the new s_i will be s_i - *prev[k]* (the value received from its right child). The following procedure does that.

Procedure ***distribute*** (s_i , *prev, k,* s_{rchild})

begin $s_{rchild} := s_i$; $s_i := s_i$ - *prev[k]* ; **end**

Now we can instantiate the algorithms *UpwardPass* and *DownwardPass* to compute the Prefix Sum computation.

Algorithm Prefix Sum(θ : associative binary operator)

begin Call UpwardPass(θ , COPY);

Call DownwardPass(COPY, *distribute,* COPY);

end

Examples of associative operators are: *addition, multiplication, maximum, mimimum* etc. For each one these operators the *distribute* function has to be written seperately as required. Each one of them can replace the operator θ in the above algorithm giving solution to a Prefix Sum problem encountered in various parallel algorithms as basic building block.

Thus by defining different realizable functions to replace the template functions in algorithms *UpwardPass* and *DownwardPass* we can really compute the class of binary-tree-computation problems efficiently in our proposed Optical Reconfigurable Bus System of Optical processors.

5.4 Conclusion

An reconfigurable bus connected optical vector processor array, ORBS, can embed in itself the Binary-Tree Interconnection topology in a dynamic way and thus can simulate Binary-Tree Computations efficiently with reduced number of processors. This architecture has a limitation that is inability to achieve pipelining of computation.

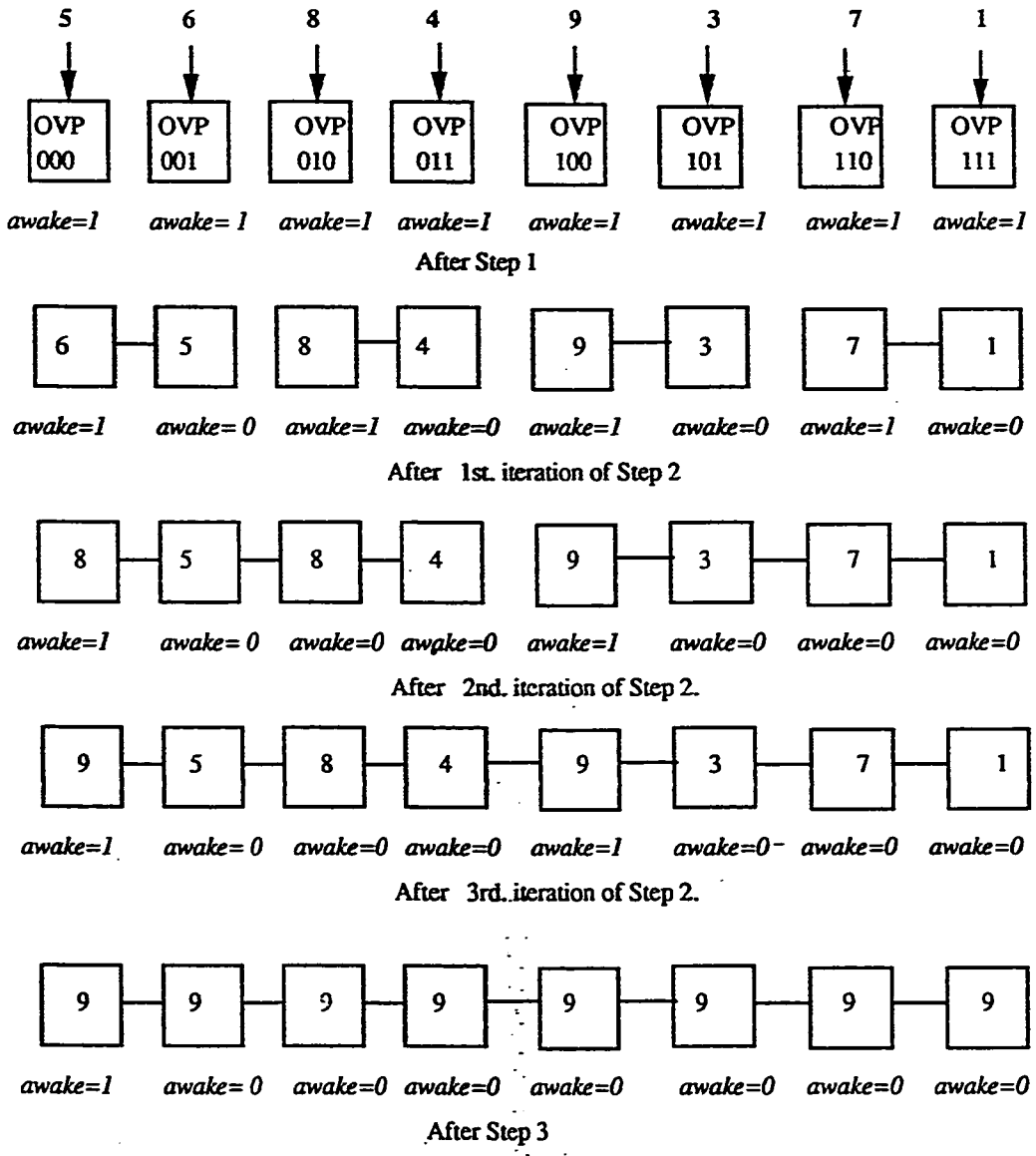


Figure 5.3 Finding the maximum on an ORBS of size 4

Chapter 6

Parallel Selection and Parallel Quicksort Algorithms on ORBS

6.1 Introduction.

In this Chapter we propose recursive parallel algorithms to be run on an Optical Reconfigurable Bus System (ORBS) to solve the *Selection Problem* and then using this, we propose a *Parallel Quicksort* algorithm. A sequence of integers $S = \{s_0, s_1, \ldots, s_{n-1}\}$ with its elements are drawn from a linearly ordered set and an integer k, where $1 \leq k \leq n$, are given. The *Selection Problem* is to determine the element with rank equals to k. Without loss of generality we can assume that S is a sequence of integers. *Selection Problem* calls for finding the k^{th} smallest element in S. This problem can be solved using a single processor with time complexity of $O(n)$ [AHO 74], [AKL89]. Based on this sequential algorithm a *Parallel Select* algorithm is proposed on an *EREW SM SIMD* computer[AKL89]. We adapt the same recursive technique and propose a different version of *Parallel Select* algorithm using all optical processing elements. Using the *Parallel Select* algorithm we later propose a *Parallel Quicksort* algorithm. In the previous Chapters we have proposed the architecture of an Optical Reconfigurable Bus System (ORBS) where Optical Vector Processors (OVP) form a reconfigurable global bus by the self-adjustment of local buses dynamically. In this Chapter we show that ORBS is a suitable parallel architecture to solve the *Selection Problem* as well as the *Parallel Quicksort*.

6.2 Preliminaries

We consider an ORBS of n OVPs connected in an array. Each OVP has a unique index i, $0 \leq i \leq n-1$. As we recall from previous chapter, an OVP is capable of performing some vector processing operation on binary vectors. We also recall that an OVP can broadcast a binary number using the BROADCAST instruction. Each OVP can also connect itself to the global bus by either straight local bus connection or by right/left shifted local bus connection. It can also disconnect its local bus connection. In this Chapter we introduce two new operations for shifting a binary sequence in either direction by a fixed number of position. These two operations will be used in *Selection* and *Quicksort* algorithms.

6.2.1 Shifting A Binary Sequence

An OVP can shift a binary sequence either to right or to left by a fixed number of positions. It is done by an optical circuit consisting of a source vector, a destination vector and a mask matrix, **Mm**, containing a fixed shift pattern. To shift in either direction, there are two types of shift patterns, namely *ShiftRight(SR)* and *ShiftLeft(SL)*. An integer *pos* specifies the amount of shift required. Then for shifting to right, say by 2, transparent masks are written on the forward diagonal which starts at cell 2 in the first column of **Mm**. Similarly, to have a leftshift of 2, transparent masks are written on the forward diagonal that starts at the second cell of first row. If no shift is required, then transparent masks are written on the forward diagonal of **Mm**. The optical setup for shifting in either directions is shown in Figure 1. So shifting in either direction is done by either of the following two instructions: **SHIFTLEFT**(X, pos) and **SHIFTRIGHT**(X, pos). The first parameter is the vector to be shifted and the second one specifies the amount of shift needed. Each one of the two instructions takes constant time.

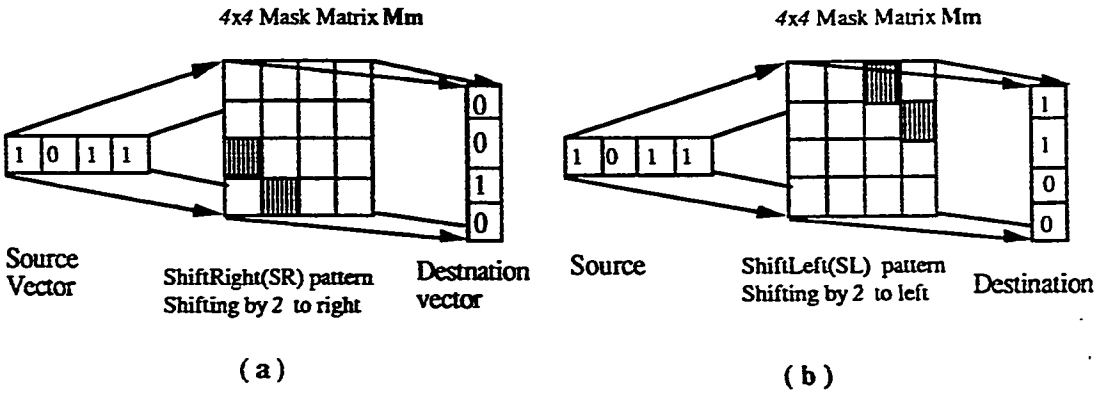


Figure 1. Shifting Binary Sequence $< 1, 0, 1, 1 >$ by 2
(a) ShiftRight($<1, 0, 1, 1 >, 2$)
(b) ShiftLeft ($<1, 0, 1, 1 >, 2$)

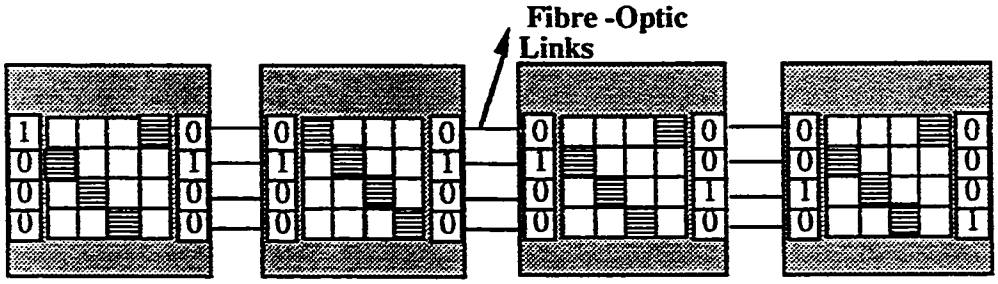
6.3 Some Useful Procedures.

In this section we give some necessary procedures that will be used in our later algorithms as modules/subroutines.

6.3.1 Computing Unary Prefix Sum of A Binary Sequence on ORBS

In the design of many parallel algorithms in general and *Parallel Select* and *Parallel Quicksort* algorithms in particular, we need to find the prefix sums of a binary sequence. On a shared memory SIMD EREW computer this takes $O(\log n)$ time . Assuming constant propagation time through the bus, in our proposed ORBS architecture *Unary Prefix Sum* can be done in constant time i.e., $O(1)$. Let us consider a binary sequence $\mathbf{B} = < b_0, b_1, \ldots, b_i, \ldots b_n >$ and an ORBS with n OVPs. Each b_i is assigned to OVP(i). Depending on its binary digit b_i equal to 0 or 1, OVP(i) connects its local bus straight or one-position shifted to the right. The first processor, OVP(1) then sends the binary sequence $< 1, 0, \ldots, 0 >$ on the global bus connecting n processors. During the broadcast cycle every processor OVP(i), $0 \leq i \leq n-1$, receives the value on the bus and detects the position of 1 and takes this integer value(position of 1) as its prefix sum i.e., sum of binary sequence up to b_1. The last processor OVP$(n-1)$ then sends its prefix sum to

all other processors to its left on the bus as the total sum of the binary sequence **B**. Figure 2 illustrates the *Unary PrefixSum* operation on an ORBS. The following algorithm performs the *UnaryPrefixSum* operation.



OVP(*0*)	OVP(*1*)	OVP(*2*)	OVP(*3*)
Broadcast <*1, 0, 0, 0*>			
ShiftRight Local bus connection	Straight Local bus connection	ShiftRight Local bus connection	ShiftRight Local bus connection
$s = 1$	$s = 1$	$s = 2$	$s = 3$ $t = 3$

Figure 2
PrefixSum(<*1, 0, 1, 1*>, *s, t*)

Algorithm UnaryPrefixSum(***R, S, T*** **)**

{ Every processor OVP(i) has its binary digit b_i in its local register vector **R**. }
{After the procedure vector **S** contains its prefix sum and vector **T** contains the total sum. }

Step 1 **for all (** i **in** {*lower...upper*} **) do in parallel**
if BITUNION($\mathbf{R}_i$) = *0* **then** CONNECT(*ST*)
else CONNECT(*SR*);

Step 2 BROADCAST(< *1, 0, 0,...,0* >, RIGHT); ($i = lower_i$)

Step 3 **for all (** i **in** {*lower ... upper*} **) do in parallel**
CONNECT(*ST*) ;

Step 4 BROADCAST($\mathbf{S}_i$, $\overline{\text{LEFT}}$) ; ($i = upper_i$)

6.3.2 Unary Subtraction on ORBS

We use the bit-by-bit shifting mechanism by processors on ORBS to find the unary subtraction of two integers. Let us assume two numbers expressed in binary sequence k_1, k_2 where $k_1 > k_2$. We can find $k_1 - k_2$ on an ORBS of size $\geq k_1$. Here is how it can be done. OVP(i), $0 \leq i \leq k_2 - 1$ connects its local bus with ShiftLeft bus connection and OVP(j), $k_2 \leq j \leq k_1 - 1$ connects its local bus straight, without any shift. Now OVP(*0*) broadcast a vector with its k_1^{th} bit equal to *1* and the rest of the bit positions as *0's*. During the broadcast cycle, every processor with left shifted local bus connection shifts the position of *1* by one position to the left. The last processor detects the position *1* in its detector vector **D** and broadcast this position, k_{new} to all the other processors as the unary subtraction of k_1 and k_2. Figure 3 is an example of this operation. Here we give a procedure.

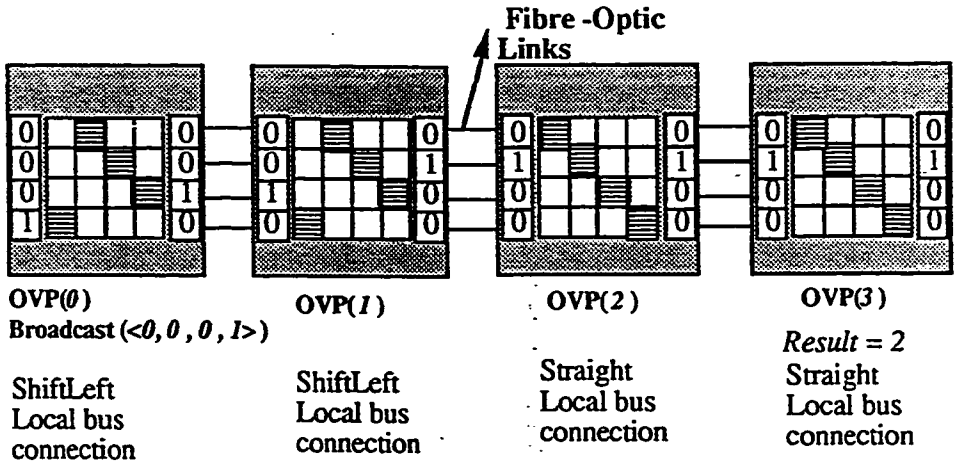


Figure 3
Unary Subtraction UnarySub(*4, 2*)

Algorithm **UnarySub**(R, S) : *return* k_{new}

{ The value in S has to be subtracted from that in R using shift mechanism.}

Step 1 **for all** (i **in** {*lower*...S }) **do in parallel**

CONNECT(SL)

Step 2 **for all** (i **in** {S...*upper*}) **do in parallel**

CONNECT(ST) ;

Step 3 BROADCAST(< 0, 0,...,$1_{|R|}$,..., 0 >, RIGHT); (i = *lower*)

{ Each processor receives this value in k_{new} }

Step 4 **for all** (i **in** {*lower*... *upper*}) **do in parallel**

CONNECT(ST) ;

Step 5 { Last processor broadcast its value in k_{new} to all the other processors }

BROADCAST(k_{new} , LEFT); (i = *upper*)

{ Each processor receives this value in k_{new} }

6.4 Parallel Select Algorithm

We are ready to present the *Parallel Select* algorithm. This algorithm is the adaption of the sequential select algorithm. In *Sequential Select* [AHO74] algorithm, n numbers are split into *3* parts L, E and G with respect to the median *(m)* of medians m_i of n/Q subgroups. Elements in L are less than m, that in E are equal to m and elements in G are greater than m. Then depending on the value of *k, Sequential select* is called recursively either on L or on G or an m is returned as the target selection. In our parallel version of this problem we extend the same technique exploiting the advantages of ORBS architecture. Before going into the details of the algorithm, let us outline some of the assumptions that our algorithm will make use of.

(a) An Optical Reconfigurable Bus System (ORBS) with n Optical Vector Processors(OVPs) is the hardware needed for the algorithm.

(b) Each OVP has a fixed number of registers and local memory. In each OVP, a variable ***lower*** contains the index of the processor with lowest index in a group of OVPs currently connected on a bus. Similarly, ***upper*** holds the highest processor's index in the same bus. The variable ***range*** holds number of active processors on the current bus. In each processor, *awake* is a flag to indicate whether that processor is active or in sleep. Each processor keeps track of the lengths of subsequences L, E, G in *3* variables ***posLT***, ***posLE*** and ***posGT***.

(c) The algorithm is a recursive one and with each recursive invocation the ORBS is split into *3* smaller parts. Associated with each invocation, a global array called SYN[***lower*** . . . ***upper***] is used for synchronization purposes. Its size is equal to the number of processor involved in that invocation.

Algorithm

Procedure Parallel Select ***(lower . . . upper, k)***

{ First parameter is the range of processor indices, $0 \le lower, upper \le n-1$. }

{ Each integer s_i , $0 \le i \le n-1$ is given to OVP(i) at the first invocation. }

{ All the processor are awake at the first invocation i.e. $awake_i = 1$. }

Step 1.

1.1 **for all** (i **in** {*lower...upper*}) **do in parallel**

Store s_i in SYN[i];

1.2 **Call** *UnaryPrefixSum(awake, r , range)*

{ All processor knows the number of processor on the bus }

1.3 { First processor checks if $range_i \le Q = 8$. If it is true then it sorts the sequence}

{in SYN by using any sequential sort and broadcast the k th smallest element }

{to all the processors and return from the procedur. }

if NumLE($range_1$, 8) = 1 **then**

begin *Sequential Sort(* SYN); BROADCAST(SYN[k]); **return end**

Step 2.

for all (i **in** {*lower . . . upper* } **and** ($i / 8 = 0$) **do in parallel**

begin Load 8 consecutive numbers starting from SYN[**ShiftLeft**(i, 8)]

to its local memory in X[0...8] ;

m_i := *median* (X); / by any sequential algorithm /

end

Step 3. { Now processors with indices from *lower* to *lower* + (*upper* - *lower*)/8 have}
{medians m_i s of groups of Q =8 elements. The *Parallel Select* is called }
{recursively on these processors to get the median (m) of medians. }
{After this step every processor knows m. }

Parallel Select(*lower* . . . *lower* + $\frac{upper - lower}{8}$, ($\frac{upper - lower}{8}$) / 2);

Step 4. { Splitting array SYN into *3* parts : Less than, Equal to and Greater than m.}

4.1 for all (i **in** {*lower...upper*}) **do in parallel**
Store m into SYN[i];

4.2 { Compute the prefix sum of binary sequence corresponding to }
{ elements in L. }

a. for all (i **in** {*lower...upper*}) **do in parallel**
if NumLT(s_i , m) =*1* **then** R_i :=*1* **else** R_i := *0* **endif;**
b. Call *UnaryPrefixSum* (R, *prefixpos*, *posLT*)
{ Every processor with element less than m now knows where to write }
{ its s_i to the global array SYN i.e. to location SYN[*prefixpos*]. }
c. for all (i **in** {*lower...upper*} **and** R_i =*1*) **do in parallel**
begin Store s_i in SYN[*prefixpos*]; R_i := *0;* **end**

4.3 { Compute the prefix sum of binary sequence corresponding to }
{ elements in **G**. }

a. for all (i **in** {*lower...upper*}) **do in parallel**
if NumGT(s_i , m) = *1* **then** R_i := *1* **else** R_i := *0* **endif;**

b. Call *UnaryPrefixSum* (**R**, *prefixpos*, *posGT*)

c.
{ Every processor with element greater than m knows where to write}
{ its s_i to the global array SYN from right i.e., to location }
{ SYN[*upper - prefixpos*] }

for all (i **in** {*lower...upper*} **and** R_i = *1*) **do in parallel**
begin Store s_i in SYN[*upper - prefixpos*]; R_i := *0;* **end**
/ This writing is done by writing from the right side of SYN /

4.4 { Compute the prefix sum of binary sequence corresponding to both }
{ elements in **L** as well as in **E**. }

a. for all (i **in** {*lower ... upper*}) **do in parallel**
if NumLE(s_i , m) = *1* **then** R_i := *1* **else** R_i := *0* **endif;**
b. Call *UnaryPrefixSum* (**R**, *prefixpos*, *posLE*);

Step 5. { Each processor now reads its new element from the rearranged array SYN }

for all (i **in** {*lower...upper*}) **do in parallel**

Load s_i from SYN[i];

Step 6. **for all** (i **in** {*lower . . . upper*}) **do in parallel**

if (NumGE(*posLT*, k) = *1*) **then**

begin

if (NumLE(i, *posLT*) =*1*) **then** $upper_i := posLT$

else $awake_i := 0$ **endif;**

Call *Parallel Select(lower . . . upper, k);*

end

else if (NumGE(*posLE*, k) = *1*) **then return** m ;

else begin *kk := UnarySub(k, posLE);* $lower_i := posLE$;

Call *Parallel Select (lower . . . upper , kk);*

end

endif

Analysis

Let us estimate the time complexity of the *Parallel Select* algorithm. We denote by $t(n)$ the time required by the *Parallel Select* for an input of size n. A recurrence relation describing $t(n)$ is now developed by analyzing each step of the procedure.

Step 1. As we have shown in before in this Chapter that procedure *UnaryPrefixSum* is done in constant time and also sequential sort on a fixed number of elements takes constant time, the whole step takes $O(1)$ time.

Step 2. Finding the median of fixed number of elements($Q = 8$) using any sequential technique takes constant time. Actually a *sequential select* algorithm finds the median in $O(\log Q)$ time and since Q is constant, time complexity is constant i.e., $O(1)$.

Step 3. Since *Parallel Select* is called with an input of size n/Q, this step takes $t(n/Q)$.

Step 4. Rearrangement of the original sequence into 3 parts: L, E and G is done in this step. Since by definition, *UnaryPrefixSum* procedure takes constant time, each one of substeps 4.1 to 4.5 takes constant time. So the whole step 4 takes time $O(1)$.

Step 5. This step also takes constant time.

Step 6. In this step, *Parallel Select* is called recursively on a sequence of reduced length than the original call. Since m is the median of n/Q elements representing the medians of Q elements in each group, $\lfloor \lceil n/Q \rceil / 2 \rfloor \times \lfloor Q/2 \rfloor$ i.e., $n/4$ elements are guaranteed to be less than or greater than m. So if $L \geq n/4$ then $G \geq 3n/4$ or vice versa. Step 6 takes time $t(3n/4)$.

The preceding analysis results in the following recurrence relation for $t(n)$:

$$t(n) = c + t(n/Q) + t(3n/4)$$

Solution to this recurrence relation yields $t(n) = O(\log n)$.

6.5 Parallel Quicksort Algorithm

We now present the *Parallel Quicksort* algorithm on an ORBS. In Quicksort algorithm the sequence of integers to be sorted is split into *3* parts: L, E and G, with respect to the median of medians of smaller subgroups of constant size*(Q)*. The median m of medians $m_i s$ is found by calling the *Parallel Select* algorithm. Then *parallel Quicksort* is called recursively on both the subsequences E and G. Thus the sequence to be sorted becomes smaller and smaller in geometric progression with each invocation of the algorithm. When the sequence becomes so small that any sequential sorting can be applied in constant time, the algorithm terminates with the whole sequence as sorted. Now we give the *Parallel Quicksort* algorithm.

Algorithm

Procedure Parallel Quicksort ***(lower ...upper)***

{ First parameter is the range of processors indices, $0 \le lower, upper \le n-1$. }

{ Each integer s_i, $0 \le i \le n-1$ is given to OVP(i) at the first invocation. }

{ All the processor are awakeinitially i.e., $awake_i = 1$. }

Step 1.

1.1 for all (i **in** {*lower... upper*}) **do in parallel**

Store s_i in SYN[i];

1.2 Call *UnaryPrefixSum(awake, r , range)*

{ All processor know the number of processors on the bus }

1.3 { First processor checks if $range_1 \le Q = 8$. If it is true then it sorts the sequence}

{in SYN by using any sequential sort and return the procedure. }

if NumLE($range_1$, 8) = 1 **then** *Sequential Sort(* SYN*);* **return**

Step 2. **Call** *parallel Select(lower ... upper, ShiftRight(upper - lower, 2))*

Step 3. { Splitting SYN into 3 parts : Less than, Equal to and Greater than m. }

3.1 for all (i **in** {*lower...upper*}) **do in parallel**

Store m into SYN[i];

3.2 { Compute the prefix sum of bit sequence corresponding to elements in L.}

a. for all (i **in** {*lower...upper* }) **do in parallel**

if NumLT(s_i , m) =1 **then** R_i:=1 **else** R_i:= 0 **endif;**

b. Call *UnaryPrefixSum* (R, *prefixpos, posLT*)

c.

{ Every processor with element less than m now knows where to write}
{ its s_i to the global array SYN i.e., to location SYN[$prefixpos_i$] }

for all (i **in** {$lower...upper$} **and** $R_i = 1$) **do in parallel**
begin Store s_i in SYN[$prefixpos_i$]; $R_i := 0$; **end**

3.3 { Compute the prefix sum of bit sequence corresponding to elements in G}

a. for all (i **in** {$lower...upper$}) **do in parallel**
if NumGT(s_i, m) $=1$ **then** $R_i := 1$ **else** $R_i := 0$ **endif;**

b. Call *UnaryPrefixSum* (R, *prefixpos*, *posGT*)

c. { Every processor with element greater than m knows where to write}
{ its s_i to the global array SYN from right i.e., to location }
{ SYN[$upper - prefixpos_i$] }

for all (i **in** {$lower...upper$} **and** $R=1$) **do in parallel**
begin Store s_i in SYN[$upper_i - prefixpos_i$]; $R_i := 0$; **end**
/ This writing is done by writing from the right side of SYN /

3.4 { Compute the prefix sum of binary sequence corresponding to }
{ elements in L as well as in E. }

a. for all (i **in** {$lower ... upper$}) **do in parallel**
if NumLE(s_i, m) $=1$ **then** $R_i := 1$ **else** $R_i := 0$ **endif;**

b. Call *UnaryPrefixSum* (R, *prefixpos*, *posLE*);

Step 4. {Each processor now reads its new element from the rearranged array SYN}
for all (i **in** {*lower...upper*}) **do in parallel**
Load s_i from SYN[i];

Step 5. **for all** (i **in** {*lower...upper*}) **do in parallel**
if (NumLT(s_i, m) = *1*) **then** $upper_i := posLT_i$;
if (NumGT(s_i, m) = *1*) **then** $lower_i := posLE_i$;
if (NumEQ(s_i, m) = *1*) **then begin** $awake_i := 0$; DISCONNECT(ST) **end**

Step 6 **for all** (i = *posLT* **or** i = *posLE* }) **do in parallel**
Call *Parallel Quicksort(lower...upper); (i)*

Analysis

To estimate *t(n)*, time required for this algorithm we can develop a similar recurrence relation as we have done for the *Parallel Select* algorithm in the last section.

Step 1. This step takes constant time.

Step 2. *Parallel Select* is called to find the median of medians, m. So the time reguired for this step is O(*logn*).

Step 3. Splitting the numbers into *3* groups takes *O(1)* time.

Step 4 and Step 5. Both take constant time.

Step 6. The *Parallel Quicksort* is called recursively on both sequences L and G concurrently. As mentioned earlier that maximum length of each one of them is $3n/4$. So this step takes time at most $t(3n/4)$.

The above analysis leads to the following recurrence relation for $t(n)$:

$$t(n) = c_1 + c_2 \log n + t(3n/4)$$

Solution to this recurrence relation gives $t(n) = \log^2 n$.

6. Conclusion

In this chapter we propose a recursive parallel algorithm to solve the *Selection Problem* and also propose a parallel version of the *Quicksort* algorithm on Optical Reconfigurable Bus System(ORBS). The time complexity of the *Parallel Select* algorithm is $O(\log n)$ and that of the *Parallel Quicksort* is $O(\log^2 n)$.

Chapter 7

Conclusion and Future Work

Optical computing provides new hope for parallel computation.We have developed optical parallel architectures and parallel algorithms to solve some comparison-based problems with better time and hardware complexities. We have exploited the full advantages of optics as the underlying technology to come up with true parallel computer architectures.

Results of the work

- An *Optical Vector Processor (OVP)* has been developed using all optical elements and primitive operations have also been developed.

- Optical Clos Nonblocking-Multicast-Switching Network has been realized and an *O(1)* control algorithm for the network controller has been developed. This allows on-the-fly dynamic reconfiguration of the network multicast connections.

- A constant time sorting algorithm has been developed to sort n numbers on an *Optical Reconfigurable Bus System (ORBS)* consisting of OVPs connected on a grid structure. Each OVP can dynamically reconfigure its local bus connection. As a result the whole global bus running through the OVPs get reconfigured dynamically.

- Binary-Tree-Computations have been simulated on an ORBS of Optical Vector Processors. Two generic algorithms have been developed to simulate the ascent and descent phases of Binary-Tree-Computations.

- *Parallel Select* and *Parallel Quicksort* algorithms have been developed on ORBS architecture.

We have touched the tip of an ice burg. Optical computing has enormous potential. Using the capabilities of our architectures solution to a number of hard problems encountered in electronic computing can be reinvestigated. A radical change in the design of parallel algorithms is needed to solve problems on parallel optical computers. Optics is very cryptic in numeric computation but its power lies in its capability of symbolic computation. Much work is needed both in hardware and software before true parallel optical computers are available side by side with their electronic counterparts.

REFERENCES

[AGGA 86] A. Aggarwal, "Optimal Bounds for Finding Maximum on Array of Processors with k Global Buses", *IEEE Transaction on Computers, C-35*,1, pp. 62-64, 1986.

[AHO 74] A. V. Aho, J. E. Hopcroft & J. D. Ullman, "The Design and Analyses of Computer Algorithms", *Addision-Wesley*, 1974.

[AKL 85] S. G. Akl, "Parallel Sorting Algorithms", *Academic Press*, 1985.

[AKL 89] S. G. Akl,"The Design and Analyses of Parallel Algorithms", *Prentice-Hall*, 1989.

[ANDO89] Ivan Andonovic & Deepak Uttamchandani, " Principles of Modern Optical Systems", *Artech House Inc.*, 1989.

[BOKH 84] S. H. Bokhari, "Finding maximum on a array processor with a global bus", *IEEE Transaction on Computers, 33* (2), 1984.

[BOWE 85] J. E. Bowers and C. A. Burrus, "Optoelectronic components and systems with bandwidths in excess of 26 GHz", *RCA Review, 46* (4), Dec 1985, pp.496.

[CLOS 53] C. Clos,"A study of non-blocking switching networks", The Bell System *Technical Journal, vol 32*, pp 406-424, 1953.

[FEIT 90] Dror G. Feitelson, "Optical Computing", *The MIT Press*, 1990.

[FRAC 90] M. Fraces, E. Bodin, J. P. Bouzinac, D. Comte, and P. Siron, " The Optical crossbar network MILORD machine : last developments and results", *SPIE special issue on Optical Interconnections and Networks, vol* 1281, pp. 66-79, 1990.

[GOOD 84] J. W. Goodmann, F. J. Leenberger, S. Y. Kung and R. A. Athale, "Optical interconnections for VLSI systems", *Proc. of IEEE, 72* (7), July 1984, pp.850-865.

[HAUG 86] P. R. Haugan, S. Rychnovsky, A. Husain and L. D. Hutcheson, "Optical interconnection for high speed computing", *Optical Engineering, vol 25*, No.1, pp. 1076-1085 October 1986,.

[JAHN 90] J. Jahns, "Optical implementation of the Banyan network", *Optics Communications, Vol. 76*, Number 5,6, May 1990, 321-324.

[JEKI 85] V. J. Jekippe and W. R. Wilson, "Single mode directional couplers", *Laser Focus 21*(5), Mar 1985, pp. 132-144.

[JORD 91] Harry F. Jordan, "Digital Optical Computers at Boulder", Centre for Optoelectronic Computing Systems, University of Colorado, Boulder.

[KOST 87] Raymond K. Kostuk, Joseph W. Goodmann and Lambertus Hesselink, "Design consideration for holographic optical interconnects", *Applied Optics, vol. 26*, No. 18, 1987, pp. 3947-3953.

[KARL 88] K. Brenner, and A. Huang,"Optical implementations of the perfect shuffle interconnection", *Applied Optics, Vol. 27*, Number 1, January 1988, 135-137.

[KUMA87] V. K. P. Kumar & C. S. Raghavendra,"Array Processors with Multiple Broadcasting", *Journal of Parallel and Distributed Computing, vol 4* (2), pp. 173-190, 1987.

[LAKS 84] S. Lakshmivarahan, S. K. Dhall & L. L. Miller, "Parallel Sorting Algorithms", in "Advances in Computers", (ed.) M. C. Yovijs;*Academic Press*; 1984.

[LAKS 90] S. Lakshmivarahan, and Sudarshan K. Dhall, "Analysis abd design of parallel algorithms: arithmetic and matrix problems", McGraw-Hill series in Supercomputing and Parallel Processing, 1990.

[LAU 85] K. Y. Lau and A. Yariv,"Ultra-high speed semiconductor lasers", *IEEE Journal on Quantum Electronics, QE-21*(2), Feb. 1985, pp. 121-138.

[LAXM 90] S. Lakshmivarahan & S. K. Dhall, "Analysis and design of parallel algorithms", *MacGraw Hill*; 1990.

[MASS 71] Gerald M. Masson and B.W. Jordan, "Realization of a class of multiple connection assignments with multistage connection networks", Proceedings of the Fifth Annual Princeton Conference on Information Sciences and Systems, pp. 316-321, 1971.

[MASS 72] Gerald M. Masson and B.W. Jordan, "Generalized multistage connection networks", *Networks, vol* 2, pp. 191-209, 1972.

[McAU 86] A. D. McAulay, "Optical crossbar interconnected digital signal processor with basic algorithms", *Optical Engineering, 25*(1), 1986, pp. 82-90.

[MILT 76] A. F. Milton and A. B. Lee,"Optical access couplers and comparison of

multiterminal fiber communication systems", *Applied Optics, 15*(1), Jan 1976, pp. 244-252.

[NAKA 90] K. Nakano, T. Masuzawa & N. Tokura,"A Fast Sorting Algorithm on a Reconfigurable Array", *Computer, 69*(15), pp. 71-78, 1990.

[RAYM 87] R. K. Kostuk, J. W. Goodman, and L. Hesselink,"Design considerations for holographic optical interconnects," *Applied Optics, Vol.* 26, Number 18, September 1987, 3947-3953.

[RHOD 87] W. T. Rhodes, "The optical margin", *Optics News, 12* (4), April 1986, pp.27-28.

[SAWC 86] Alexander A. Sawchuk and B. Keith Jenkins, "Dynamic Optical Interconnections for Parallel Processors", *SPIE, vol. 625* , Optical Computing (1986).

[SCHU87] Elmar Schulze,"Reconfigurable Optical Interconnections using dynamic optoelectronic holograms", *SPIE, vol. 862*, Optical Interconnections (1987).

[SCOT 88] H. Scott Hinton, "Architectural considerations for photonic switching networks", *IEEE Journal on selected areas in communication, vol 6*, No. 7, 1988.

[SEIG 91] A. Siegel, "Interconnection networks and parallel processing", Second edition Lexington Press; 1991.

[SHAM 87] J. Shamir, and J. Caulfield,"High-efficiency rapidly programmable optical interconnections," *Applied Optics, Vol. 26*, Number 6, March 1987, 1032-1037.

[WANG 90] B. F. Wang, G. H. Chen & F. C. Lin, "Constant Time Sorting on a Processor Array with a Reconfigurable Bus System", *Information Processing Letters, vol 34* No. 4, pp. 187-192, 1990.

[WEST 87] L. C. West,"Picosecond integrated optical logic", *Computer 20* (12), Dec 1987, pp. 34-46.

[WHER 87] B. S. Wherret, S. Desmond Smith, F. A. P. Tooley, and A. C. Walker, "Optical Components for Digital Optical Circuits", *Future Generation Computer Systems, Vol. 3,* 1987, 253-259.

[WILK 83] M. V. Wilkes, "Size, Power, and Speed", Conference Proc. 10th Annual Internationa. Symposiu. on Computer Architecture, *SIGARCH, 11*(3); Jan 1983, pp. 2-4.

[YANG 90] Yuanyuan Yang and Gerald M. Masson, "Nonblocking Broadcast Networks", Technical Report JHU 90/04 Computer Science Department, John Hopkins University, Baltimore, MD 21218.