Parallel Computer Vision on a Reconfigurable Multiprocessor Network

Suchendra M. Bhandarkar, *Member*, *IEEE Computer Society*, and Hamid R. Arabnia

Abstract—A novel reconfigurable architecture based on a multiring multiprocessor network is described. The reconfigurability of the architecture is shown to result in a low network diameter and also a low degree of connectivity for each node in the network. The mathematical properties of the network topology and the hardware for the reconfiguration switch are described. Primitive parallel operations on the network topology are described and analyzed. The architecture is shown to contain 2D mesh topologies of varying sizes and also a single one-factor of the Boolean hypercube in any given configuration. A large class of algorithms for the 2D mesh and the Boolean *n-cube* are shown to map efficiently on the proposed architecture without loss of performance. The architecture is shown to be well suited for a number of problems in low- and intermediate-level computer vision such as the FFT, edge detection, template matching, and the Hough transform. Timing results for typical low- and intermediate-level vision algorithms on a transputer-based prototype are presented.

Index Terms—Reconfigurable multiring network, reconfigurable architectures, scalable architectures, parallel processing, distributed processing, computer vision, image processing, parallel algorithms, distributed algorithms.

1 Introduction

Problems in computer vision are known to be computationally intensive. Inherent limitations on the computational power of uniprocessor architectures, especially under real-time constraints, have led to the development of multiprocessor architectures for computer vision problems. From a theoretical point of view, a multiprocessor architecture should facilitate the design and implementation of efficient parallel algorithms for computer vision problems. These algorithms should optimally exploit the capabilities of the architecture. From an architectural point of view, the multiprocessor architecture should have low hardware complexity and, preferably, be composed of components that can be easily replicated thus making it suitable for VLSI implementation. Additionally, the architecture should exhibit good scalability of computational performance, hardware complexity, and cost with increasing number of processors.

Many multiprocessor architectures have been proposed for computer vision such as, the *hypercube* [29], *butterfly* [7], *systolic array* [9], *2D mesh* [10, [34], and the *pyramid* [35]. Several researchers have attempted to map computer vision problems on these architectures. However, the fixed topology of the interconnection networks describing these architectures leads to an inevitable tradeoff between the need for low network diameter and the need to limit the number of interprocessor communication links. Moreover, vision problems are especially difficult since they place different requirements on the underlying interconnection network topology and the mode or granularity of parallelism (i.e., SIMD, SPMD, or MIMD) depending on whether the problem can be classified as one of low-, intermediate-, or high-level vision. No fixed topology interconnection network with a given mode or granularity of parallelism has proven effective at tackling computer vision problems at all levels of abstraction, i.e., low-, intermediate-, and high-level vision.

Reconfigurable networks attempt to address the aforementioned tradeoff between the need for low network diameter and the need to limit the number of interprocessor communication links. In a reconfigurable network each node has a bounded degree of connectivity but the network diameter is restricted by allowing the network to reconfigure itself into different configurations. Examples of reconfigurable multiprocessor systems include the Polymorphic Torus [23], [24], Gated-Connection Network (GCN) [18], [36], CLIP7 [13], PAPIA2 [1], Reconfigurable Bus Architecture (RBA) [26], and the Reconfigurable Mesh Architecture (RMA) [27]. A considerable amount of research in recent times has been devoted in attempts to show that these reconfigurable systems are well-suited for computer vision problems.

Broadly speaking, a reconfigurable system needs to satisfy the following properties in order to be considered practically feasible:

1) In each configuration the nodes in the network should have a reasonable degree of connectivity with respect to the number of processors in the network (i.e., network size). This is to ensure that the number of interprocessor communication links does not grow very rapidly with network size. It is desirable that the number of interprocessor communication links scale subquadratically (preferably linearly) with respect to the network size.
2) The network diameter should be kept low via the reconfiguration mechanism. The network diameter

• *The authors are with the Department of Computer Science, 415 Boyd Graduate Studies Research Center, the University of Georgia, Athens, GA 30602-7404. E-mail: suchi@cs.uga.edu.*

Manuscript received Nov. 18, 1994.
For information on obtaining reprints of this article, please send e-mail to: transpds@computer.org, and reference IEEECS Log Number D95243.

should scale sublinearly (preferably logarithmically) with respect to the network size.
3) The hardware for the reconfiguration mechanism (i.e., reconfiguration switch) should be of reasonable complexity.
4) The algorithmic complexity of the reconfiguration operation should be low.

In this paper, we describe a novel reconfigurable architecture which we term as the Reconfigurable MultiRing Network (RMRN) [2], [5], [6]. The RMRN is shown to be highly scalable and amenable to VLSI implementation. The RMRN results in a physically compact multiprocessor system, which is an especially important criterion for computer vision on mobile and autonomous robotic platforms. We prove some important properties of the RMRN topology. As a result, we show that a broad class of algorithms for the *2D mesh* and the *n-cube* can be mapped to the RMRN in a simple and elegant manner. We design and analyze a class of procedural primitives for the RMRN and show how these primitives can be used as building blocks for more complex parallel operations. We show that the RMRN can support major parallelization strategies (and their various combinations) such as functional parallelism (i.e., pipelining), data parallelism, and control parallelism and is effective in both the SIMD (Single Instruction Multiple Data) and the SPMD (Single Program Multiple Data) modes of parallelism. We demonstrate the usefulness of the RMRN for problems in low- and intermediate-level computer vision by considering typical operations such as the Fast Fourier Transform (FFT), convolution, template matching, and the Hough Transform. Timing results for these operations on a transputer-based prototype are also presented.

The organization of the remainder of this paper is as follows: In Section 2, we describe the basic topology of the RMRN. We also state and prove some important properties of the RMRN. In Section 3, we design some basic procedural primitives on the RMRN which could be used as building blocks for more complex parallel algorithms. In Section 4, we present algorithms on the RMRN for typical low- and intermediate-level vision operations such as the FFT, convolution, template matching, and the Hough Transform. In Section 5, we describe a transputer-based prototype RMRN and the switching hardware used. We also present a performance evaluation of the prototype. In Section 6, we conclude the paper and outline future directions.

2 The Basic Topology and Properties of the RMRN

In this section, we define the RMRN topology in precise mathematical terms and also give a functional description of the hardware needed for enabling reconfigurability and providing the input/output connections.

2.1 Basic Definitions

Let $RMRN_n$ denote an RMRN with $N = 2^n$ processors. The processors are numbered 0, 1, 2, …, $(N - 1)$. Each processor p in the RMRN is uniquely specified using an n bit address $(p_0, p_1, \ldots p_{n-1})$. The $RMRN_n$ has $n + 1$ different configurations where each configuration is denoted by $config(RMRN_n, i)$, $0 \le i \le n$. Let $r = 2^i$, then $config(RMRN_n, i)$ consists of $r = 2^i$ rings $R_0, R_1, \ldots, R_{r-1}$ stacked in a pipelined fashion such that each ring has $k = 2^{n-i}$ processors. For $0 < j < r - 1$ every processor in R_j is connected to a processor in R_{j+1} and R_{j-1}. The input of each processor in R_0 is multiplexed between an external input channel and the output of a processor in R_{r-1}. Analogously, the output of each processor in R_{r-1} is demultiplexed between an external output channel and the input of a processor in R_0.

Given that the $RMRN_n$ is in configuration $config(RMRN_n, i)$, a processor p is in ring R_j iff $p \bmod r = j$. Also, processor p is connected to processors $((p + r) \bmod N)$ and $((p - r) \bmod N)$ in ring R_j via bidirectional links. Furthermore, we say that processor p is in position q in the ring R_j iff $p \operatorname{div} r = q$. If $0 < j < r - 1$ then there are bidirectional links between processors p and $p + 1$ and between processors p and $p - 1$. If $j = r - 1$ then there is a demultiplexed link between processor p and an external output channel. Conversely, processor p is connected to an external input channel via a multiplexed link if $j = 0$. For example, Figs. 1a and 1b show an $RMRN_4$ system of 16 processors in configurations $config(RMRN_4, 1)$ and $config(RMRN_4, 2)$, respectively. In $config(RMRN_4, 1)$ we have two rings R_0 and R_1 where R_0 consists of processors {0, 2, 4, 6, 8, 12, 14} and R_1 consists of processors {1, 3, 5, 7, 9, 11, 13, 15} (Fig. 1a). In $config(RMRN_4, 2)$ we have four rings R_0, R_1, R_2, and R_3 where R_0 consists of processors {0, 4, 8, 12}, R_1 consists of processors {1, 5, 9, 13}, R_2 consists of processors {2, 6, 10, 14}, and R_3 consists of processors {3, 7, 11, 15} (Fig. 1b).

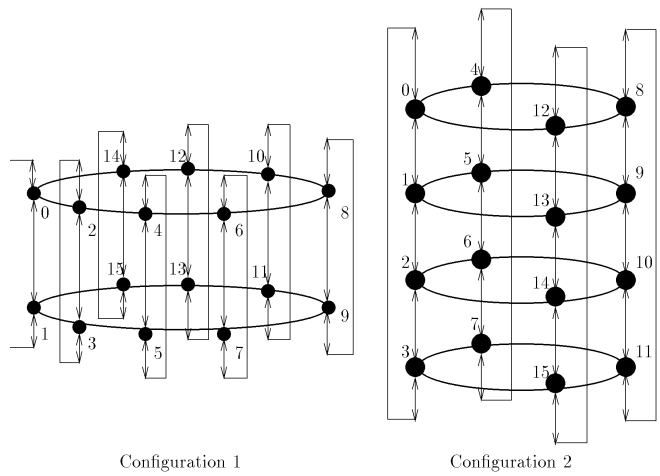


Fig. 1. $RMRN_4$ in configurations 1 and 2.

2.2 Connections to the External Input/Output Channels

The input and output control switches that provide connections to the external input and output channels respectively are straightforward to design. These switches are designed to check whether the last i bits of the processor address p are all 0 or 1, respectively. In $config(RMRN_n, i)$ with $r = 2^i$, there is a link between the external input channel and processor p iff $p \bmod r = p \bmod 2^i = 0$ (i.e., p is in R_0)

and the input control signal *IN* is high else p is connected to the output of processor $(p - 1) \bmod N$. Conversely, iff $p \bmod r = p \bmod 2^i = r - 1$ and the output control signal *OUT* is high, then processor p is connected to the external output channel, else processor p is connected to the input of processor $(p + 1) \bmod N$. Calculating mod 2^i is equivalent to examining the last i bits of the n bit address of processor p, i.e., $p \bmod 2^i = 0$ or $p \bmod 2^i = r - 1$ iff the last i bits of p are all 0 or all 1, respectively. Thus, the input control switch needs to check whether or not the least significant i bits of the processor address are all 0 whereas the output control switch needs to check whether or not the least significant i bits of the processor address are all 1. The hardware for the input/output switches is described in Section 5.1.

2.3 The Reconfiguration Switching Network

A multistage switching network enables the RMRN to reconfigure itself from *config*($RMRN_n$, i) to *config*($RMRN_n$, j) where $0 \le i, j \le n$ and $i \ne j$. In *config*($RMRN_n$, i), processor p is connected to processors $(p + 2^i) \bmod N$ and $(p - 2^i) \bmod N$. The function that the switching network needs to perform is related to that of a perfect shuffle network [37] and the barrel shifting network [25], but it is not exactly identical to either of them. The switching network is built up in the same way a shuffle network is, and, in fact, does contain some shuffle and unshuffle constituents. However, since it does not perform at all like a shuffle, it is unrelated functionally to the shuffle network. The relation to the barrel shifter is functional. The switching network serves to configure connections in a manner similar to how a barrel shifter would configure the bits of an operand. However, it is not constructed at all like a barrel shifter, so it is unrelated structurally to the barrel shifter. Section 5.2 details the hardware implementation of the reconfiguration switching network.

The switching network bears structural and functional resemblance to the other hypercubic multistage interconnection networks such as the butterfly network [4], omega network [15], [19], [21], [30], delta network [28], baseline network [39], and the banyan network [14]. It can be shown in a straightforward manner that the interconnection links in a single stage of the butterfly network are contained within a specific configuration of the RMRN topology. This property can be proved in a manner analogous to the proof of Property 4 presented in the following subsection. Leighton [22] has shown the other commonly encountered multistage interconnection networks, i.e., omega, baseline, delta, and banyan, to be variants of the butterfly network. In [22], Leighton has presented a class of graph similarity transformations for systematically transforming the other networks into an equivalent representation in terms of the butterfly network. This implies that the various stages of the omega, baseline, delta, and banyan networks can also be shown to be contained within specific configurations of the RMRN topology.

Since the reconfiguration switching network for the RMRN is a hierarchical multistage network like the butterfly network, the number of elementary switching elements needed to implement a reconfiguration switching network for N processors scales as $O(N \log_2 N)$. Although the asymptotic hardware complexity of $O(N \log_2 N)$ is not attractive for very large values of N, it is manageable for moderate values of N. The $O(N \log_2 N)$ hardware complexity is offset by the fact that the switching network is modular and hierarchically expandable and the expansion can be done using a single logical component as a building block. For example, one could use the switching network for $RMRN_4$ as a building block for larger systems. One of the most important properties of the switching network is that each processor in $RMRN_n$ needs four bidirectional channels irrespective of the value of $N = 2^n$. Since the connectivity of each node in $RMRN_n$ is fixed, the number of interprocessor communication links grows *linearly* with the system size $N = 2^n$. The RMRN thus satisfies an important criterion for a general purpose reconfigurable multiprocessor. In the following sections we explore some other important properties of the RMRN which underscore its viability for computer vision problems.

2.4 Basic Properties of the RMRN Topology

We state and prove some important properties of the *RMRN* topology.

PROPERTY 1. *All 2D mesh topologies of size $2^i \times 2^j$ are subsets of the $RMRN_n$ where $i + j = n$.*

PROOF. Consider a grid with 2^i rows and 2^j columns such that $i + j = n$. Consider a position (k, l) in the grid. Under a row-major mapping, the processor address at position (k, l) is given by

$$p = 2^j k + l, \quad 0 \le k \le 2^i - 1, \quad 0 \le l \le 2^j - 1$$

The processor p is connected to its four nearest neighbors at locations $(k - 1, l)$, $(k + 1, l)$, $(k, l - 1)$, and $(k, l + 1)$ with the corresponding processors at these locations denoted by p_n, p_s, p_e, and p_w, respectively. Under the row-major mapping the processor addresses p_n, p_s, p_e, and p_w are given by:

$$p_n = 2^j(k - 1) + l = p - 2^j$$
$$p_s = 2^j(k + 1) + l = p + 2^j$$
$$p_e = 2^j k + l + 1 = p + 1$$
$$p_w = 2^j k + l - 1 = p - 1$$

Consider the $RMRN_n$ in *config*($RMRN_n$, j) with 2^j rings each ring containing $2^{n-j} = 2^i$ processors. Under the row-major mapping, the processor p at grid location (k, l) is mapped onto the kth position in the lth ring. From the structural property of the $RMRN_n$ in *config*($RMRN_n$, j) processor p is connected to processors $p + 2^j$ and $p - 2^j$ in the lth ring and processors $p + 1$ and $p - 1$ in the $(l + 1)$th and $(l - 1)$th ring, respectively. Thus, it is clear that the the $RMRN_n$ in *config*($RMRN_n$, j) embeds the connectivity pattern of a $2^i \times 2^j$ mesh. This proves the property that all 2D mesh topologies of size $2^i \times 2^j$ are subsets of the $RMRN_n$ where $i + j = n$. □

PROPERTY 2. *The n-cube (i.e., a hypercube with $N = 2^n$ processors) can embed all possible configurations of the $RMRN_n$ (though not simultaneously).*

PROOF. See Appendix A.

PROPERTY 3. *The configuration config($RMRN_m$, j) is a subconfiguration of config($RMRN_n$, i) whenever $m < n$, $j < i$, and $n - m = i - j$. Thus config($RMRN_m$, j) is a subset of config($RMRN_n$, i) with the connectivity pattern preserved.*

PROOF. The processor address p in *config*($RMRN_n$, i) can thus be decomposed as $p = p_m p_{n-m}$ where the lower-order $n - m$ bits specify a particular subconfiguration of the type *config*($RMRN_m$, j) and the higher-order m bits specify the address p' within the subconfiguration *config*($RMRN_m$, j). Consider a processor p in *config*($RMRN_n$, i). From the definition of *config*($RMRN_n$, i), p is adjacent to processors $p_r = (p + 2^i) \bmod N$ and $p_l = (p - 2^i) \bmod N$. In the configuration *config*($RMRN_m$, j), the address of p is given by $p' = p \operatorname{div} 2^{n-m}$. The address of p_r in *config*($RMRN_m$, j) is given by

$$p'_r = p_r \operatorname{div} 2^{n-m} = \left(p + 2^i\right) \operatorname{div} 2^{n-m}$$
$$= \left(p \operatorname{div} 2^{n-m}\right) + \left(2^i \operatorname{div} 2^{n-m}\right).$$

Since $p \operatorname{div} 2^{n-m} = p'$ and $2^i \operatorname{div} 2^{n-m} = 2^i \operatorname{div} 2^{i-j} = 2^j$, we get the result $p'_r = p' + 2^j$. Similarly, it can be shown that $p'_l = p' - 2^j$. Thus *config*($RMRN_m$, j) is a subset of *config*($RMRN_n$, i) with the connectivity pattern preserved. □

PROPERTY 4. *Any algorithm which uses the communication links along a single dimension of the n-*cube *at any given point in time can be mapped to the $RMRN_n$.*

PROOF. Let $p(i)$ represent the processor whose n bit address differs from that of processor p only in the ith bit. Let $p[i]$ denote the ith bit in the processor address p. Let F_i^n be the collection of edges in the *n-cube* in the ith dimension, i.e., $F_i^n = \{(p, p(i))\}$. Then $F_0^n, \ldots, F_{n-1}^n$ is a *one-factor* of the *n-cube*. It can be easily proved that $F_i^n \subset config(RMRN_n, i)$. Without loss of generality, we may assume that $p[i] = 0$ and $p(i)[i] = 1$. Therefore $p(i) = p + 2^i$ which means that p and $p(i)$ are connected in *config*($RMRN_n$, i). Let $A(p)$ denote the A register in processor p. Let the statement *config*(n, i) reconfigure $RMRN_n$ in *config*($RMRN_n$, i). Also, let *left(p)*, *right(p)*, *next(p)*, and *previous(p)* denote the processors $(p - 2^i) \bmod N$, $(p + 2^i) \bmod N$, $(p + 1) \bmod N$, and $(p - 1) \bmod N$, respectively, in *config*($RMRN_n$, i). The statements $A(p) \rightarrow A(p(i))$ and $A(p) \leftarrow A(p(i))$ in an algorithm for the *n-cube* which move the contents of the A register of p to the A register of $p(i)$ and vice versa along the ith dimension edge $(p, p(i))$ are, respectively, equivalent to the following statements on the $RMRN_n$:

```
config(n,i);
IF (p[i] = 0) THEN A(p) -> A(right(p))
ELSE A(p) -> A(left(p));
```

and

```
config(n,i);
IF (p[i] = 0) THEN A(p) <- A(right(p))
ELSE A(p) <- A(left(p));
```

This completes the proof of Property 4. □

In summary, **Property 1** states that the $RMRN_n$ can be configured into a variety of mesh topologies. In fact, we have proved that the $RMRN_n$ in *config*(n, i) contains as its subset a $2^i \times 2^{n-i}$ processor 2D wrap-around (i.e., toroidal) mesh. **Property 2** shows how the *n-cube* can be used to simulate the behavior and function of the $RMRN_n$. In fact, any given configuration of the $RMRN_n$ is a subset of the *n-cube*. **Property 3** shows that the $RMRN_n$ possesses an elegant recursive property with regard to its structure in a manner similar to the *n-cube*. In general, *config*($RMRN_n$, i) would contain 2^{n-m} subconfigurations of the type *config*($RMRN_m$, j) such that $m < n$, $j < i$, and $n - m = i - j$. The condition $n - m = i - j$ ensures that the number of processors in each ring of *config*($RMRN_n$, i) and *config*($RMRN_m$, j) is the same. If the RMRN is represented as a graph with the processors as nodes and the interprocessor links as arcs between nodes, then each of the subconfigurations *config*($RMRN_m$, j) would constitute a subgraph of the *config*($RMRN_n$, i). This property is important because it enables recursive decomposition of a procedure into independent subprocedures which could then be executed concurrently on individual subconfigurations. The usefulness of this property will be brought out in Sections 3 and 4 wherein we describe certain imaging operations that are to be performed in parallel on subimages or windows within the entire image. A subimage or window will be shown to essentially define a subconfiguration within the RMRN.

Property 4 shows that the edges along a given dimension of the *n-cube* are contained within a specific configuration of the $RMRN_n$. This result is of special significance since it allows a wide class of algorithms designed for the *n-cube* to be mapped onto the $RMRN_n$ without loss of performance, assuming that the overhead in reconfiguration is not excessive. Any algorithm which uses the communication links along a single dimension of the *n-cube* at any given point in time can be mapped to the $RMRN_n$ in $O(1)$ time.

In summary, the $RMRN_n$ in any single configuration is a proper subset of the *n-cube*, whereas the edges along a specific dimension of the *n-cube* are a proper subset of the $RMRN_n$. This implies that the $RMRN_n$ in any given configuration is more restrictive and hence less powerful than the *n-cube*. However, it should be noted that a vast majority of algorithms that use the *n-cube* in the SIMD or SPMD (and sometimes even in the MIMD) modes of parallelism, use the edges of the *n-cube* along one specific dimension at a given time. One can therefore conclude that the RMRN topology, on account of its reconfigurability, provides the same generality *in practice* as the *n-cube* for a large class of problems. In fact, the RMRN can be seen to offer a cost-effective alternative to the *n-cube* architecture without substantial loss in performance.

3 SOME BASIC OPERATIONS AND ALGORITHMS ON THE RMRN

The RMRN that has been designed and simulated to date can be operated in the SIMD and the SPMD modes of parallelism. We shall prove that a wide class of basic opera-

tions and algorithms can be easily implemented on $RMRN_n$. Each processor in the RMRN has four bidirectional channels which are denoted as *left*, *right*, *next*, and *previous*. The processors in the RMRN multiprocessor can operate in either the SIMD or the SPMD mode of parallelism. Each processor has its own local memory for data storage and, in the case of the SPMD mode, also program storage. In the SIMD mode, a single control unit broadcasts a common instruction stream to all the processors. Each processor can either execute the current instruction or ignore it altogether depending on the state of the variables in its local memory. The control unit also issues the command(s) to the reconfiguration switch to reconfigure the RMRN multiprocessor in a given configuration.

In the SIMD mode, all the processors and the reconfiguration switch are constrained to operate in lock-step synchronism. In the SPMD mode of parallelism, each processor runs its local program asynchronously on its local data. However, all the processors are synchronized at each *reconfigure* command using *barrier synchronization*. A failure to do so would cause an inconsistency if different processors assume different states of the interconnection network and also switch-level contention if the reconfiguration switch is forced to route messages in two different reconfiguration states at the same time. We denote intraprocessor assignments by := whereas $\rightarrow$ is used to denote interprocessor assignments. Interprocessor assignments utilize the interprocessor links in the RMRN. The term *unit hop* is used to denote communication between processors in the RMRN that are directly connected. The asymptotic complexity of any algorithm for the RMRN is decided by the number of *unit hops* in the algorithm if each unit hop involves $O(1)$ (i.e., constant) amount of data transfer [22].

3.1 General Message Passing Operations

3.1.1 Broadcast Operation

Assume that the data in register X of processor 0 needs to be broadcast to all the other processors. Reconfigurability permits the broadcast operation to be performed in $O(n)$, that is $O(\log_2 N)$ unit hops where each unit hop entails $O(1)$ data transfer. In fact, the diameter of the $RMRN_n$ network with reconfigurability can be shown to be $n = \log_2 N$. The broadcast algorithm is shown in Fig. 2. This algorithm describes a *simple* broadcast wherein a data item in a processor 0 is broadcast to all the other processors in the RMRN. The function $MSONE(p)$ denotes the position of the most significant 1 in the binary processor address p. For example, $MSONE(5) = MSONE(0101) = 2$, and $MSONE(9) = MSONE(1001) = 3$. We define $MSONE(0) = -1$.

```
Broadcast(X, n);
  BEGIN
    FOR i = 0 to n-1 DO
      BEGINFOR
        config(n, i);
        IF (MSONE(p) = i) THEN X(p) <- X(left(p));
      ENDFOR;
  END;
```

Fig. 2. Broadcast operation.

The simple broadcast operation for $RMRN_3$ is depicted in Fig. 3. We describe below each step in the broadcast operation for $RMRN_3$. For each step we list the addresses of the source and destination processor(s). Note that for step i, $0 \le i < n$, the corresponding configuration is $config(RMRN_n, i)$, the destination processor(s) p is (are) such that $MSONE(p) = i$ and the corresponding source processors(s) is (are) $q = p - 2^i = left(p)$.

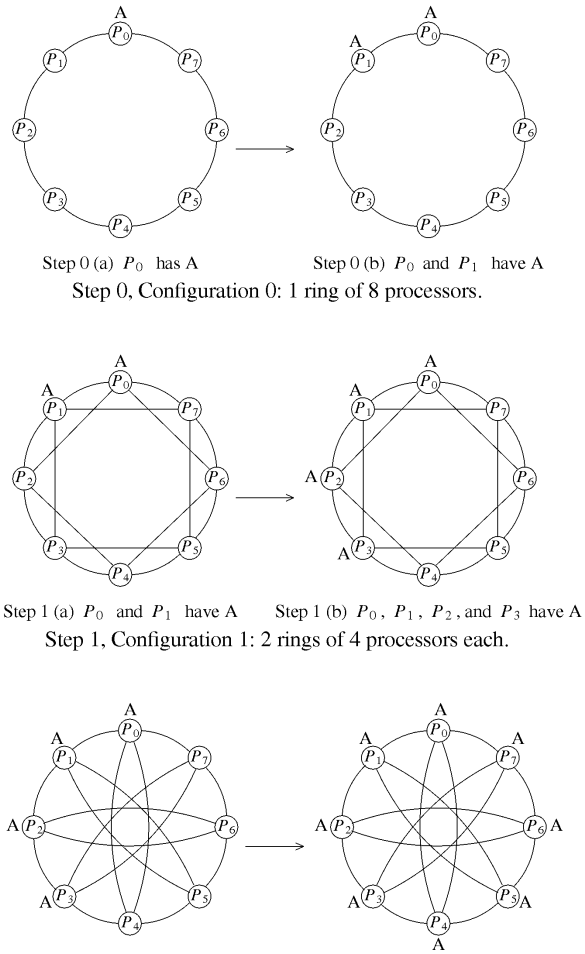


Fig. 3. The simple broadcast operation for $RMRN_3$.

Step 0: Configuration 0, $MSONE(p) = 0$
0 (000) $\rightarrow$ 1 (001)

Step 1: Configuration 1, $MSONE(p) = 1$
0 (000) $\rightarrow$ 2 (010)
1 (001) $\rightarrow$ 3 (011)

Step 2: Configuration 2, $MSONE(p) = 2$
0 (000) $\rightarrow$ 4 (100)
1 (001) $\rightarrow$ 5 (101)
2 (010) $\rightarrow$ 6 (110)
3 (011) $\rightarrow$ 7 (111)

One can also design variants of the simple broadcast operation for image processing or computer vision applications. These operations could be listed as:

1) **Image Tile Broadcast:** In this operation, an image is decomposed into tiles. All the tiles are initially resident on a single processor of the RMRN. These tiles are then distributed via the broadcast operation to individual processors such that each processor has a single tile. The image tile broadcast is depicted in Fig. 4.
2) **All-To-All Broadcast (Gossiping):** A data item d_i is resident on each of the processors p_i of the $RMRN_n$. The all-to-all broadcast enables *each* processor p_i to have a copy of *all* the data items d_i, $0 \le i < N$. The all-to-all broadcast is depicted in Fig. 5.

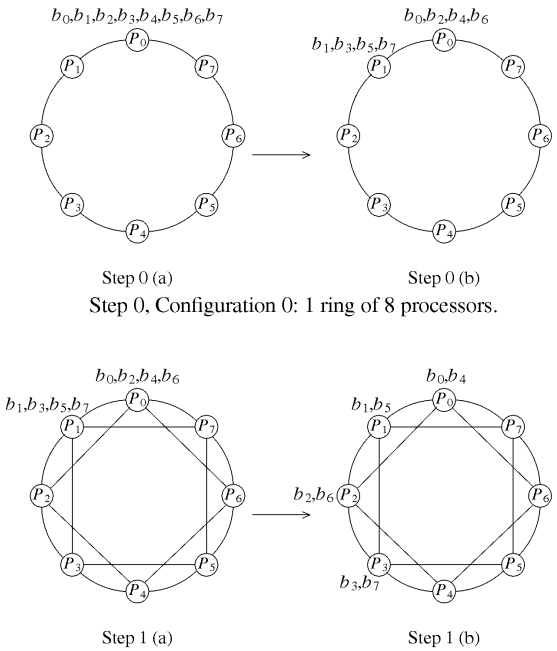


Fig. 4. The image tile broadcast operation for $RMRN_3$.

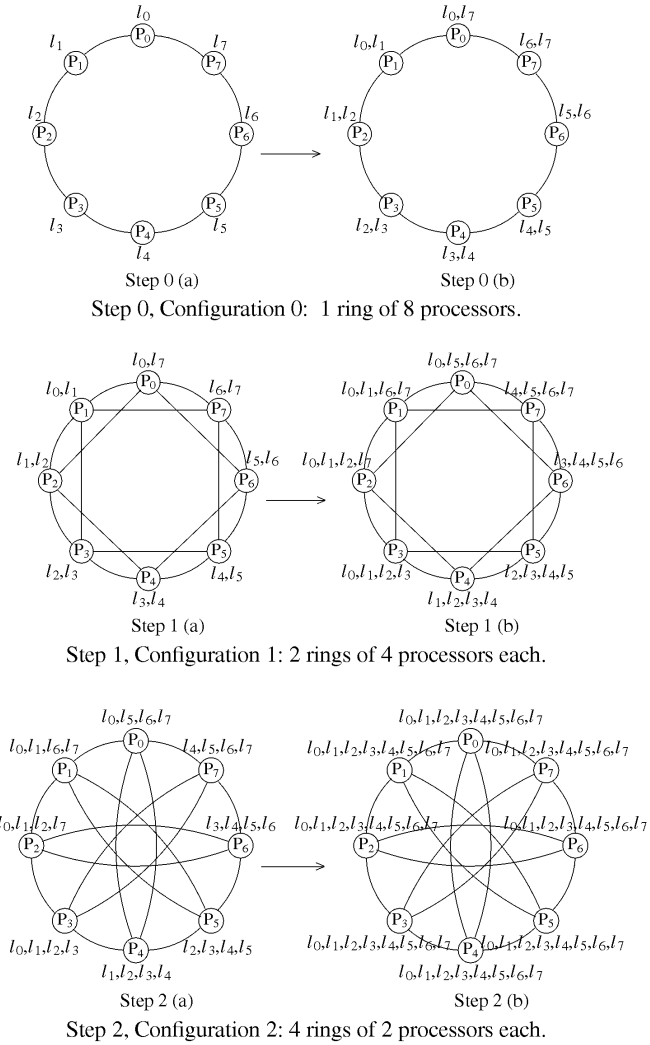


Fig. 5. The all-to-all broadcast operation for $RMRN_3$.

From Figs. 4 and 5, the image-tile broadcast and the all-to-all broadcast can be seen to take $n = \log_2 N$ reconfiguration steps on the $RMRN_n$. In the case of the image-tile broadcast operation, at step i, where $0 \le i < n$, $N/2^{i+1}$ image tiles are transferred over the interprocessor communication links. If each tile is B bytes and the link capacity is C bytes/sec then the total time taken for the image tile broadcast is given by:

$$T_{IT} = \sum_{i=0}^{n-1} \frac{N}{2^{i+1}} \frac{B}{C} = \frac{BN}{C} \sum_{i=1}^{n} \frac{1}{2^i} = \frac{B}{C}(N-1) = O(N) \quad (1)$$

In the case of the all-to-all broadcast, at step i, where $0 \le i < n$, 2^i image tiles are transmitted over each of the interprocessor communication links (Fig. 5). The total time taken for the all-to-all broadcast is given by:

$$T_{AB} = \sum_{i=0}^{n-1} \frac{B}{C} 2^i = \frac{B}{C} \sum_{i=0}^{n-1} 2^i = \frac{B}{C}\left(2^n - 1\right) = \frac{B}{C}(N-1) = O(N) \qquad (2)$$

Although the image tile broadcast and the all-to-all broadcast take the same amount of time on the RMRN, in the case of the all-to-all broadcast, unlike the image tile broadcast, each interprocessor communication link is busy all the time (Figs. 4 and 5). The complexity of the image tile broadcast and all-to-all broadcast is $O(N)$ on the $RMRN_n$. In practice, techniques like *wormhole routing* or *cut-through routing* could be used to cut down on the actual broadcast time. The algorithms for the image-tile broadcast and the all-to-all broadcast are similar to the algorithm for the simple broadcast in Fig. 2.

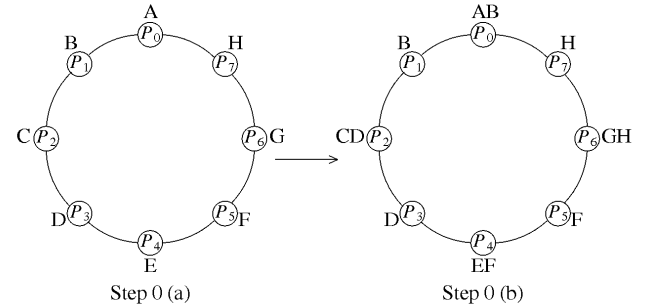
3.1.2 Combine Operation

Let $\oplus$ denote any associative binary operation such as MAX, MIN, logical AND, logical OR, sum, or product. Let each processor contain a data item in its X register. We are interested in the $\oplus$ (i.e., combine) of all these data items, that is, $\oplus_{p=0}^{N-1} X(p)$ where $N = 2^n$. In this case, reconfigurability allows the combine operation to be performed in $O(\log_2 N)$ unit hops where each hop involves $O(1)$ amount of data transfer over the interprocessor communication links. The algorithm for the combine operation is given in Fig. 6 in which $F(x, y)$ is assumed to be an associative binary operation. The function $LSONE(p)$ denotes the position of the least significant 1 in the binary processor address p. For example, $LSONE(5) = LSONE(0101) = 0$ and $LSONE(4) = LSONE(0100) = 2$. We define $LSONE(0) = -1$. At the end of the combine operation processor 0 contains the final result of the combine operation.

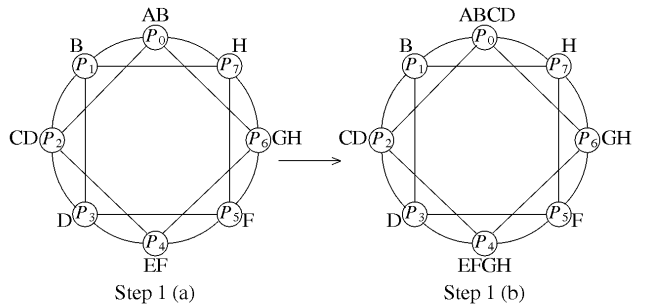
```
Combine(X, n, F);
  BEGIN
    FOR i = 0 to n-1 DO
      BEGINFOR
        config(n,i);
        IF (LSONE(p) = i) THEN X(p) -> Y(left(p));
        IF (p[i] = 0) THEN X(p) := F(X(p),Y(p));
        {F is an associative binary operation}
      ENDFOR;
  END;
```

Fig. 6. Combine operation.

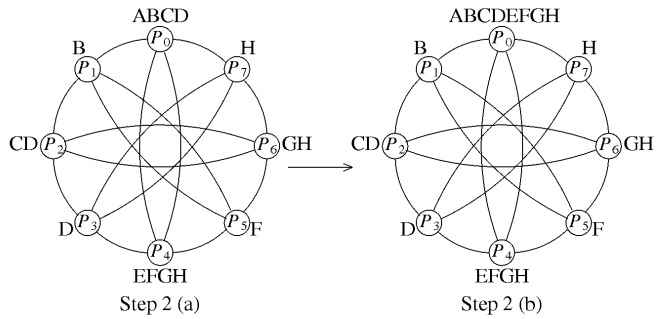
The combine operation for $RMRN_3$ is depicted in Fig. 7. We describe below each step in the combine operation for $RMRN_3$. For each step we list the addresses of the source and destination processor(s). Note that for step i, $0 \le i < n$, the corresponding configuration is $config(RMRN_n, i)$, the source processor(s) p is (are) such that $LSONE(p) = i$ and the corresponding destination processors(s) is (are) $q = p - 2^i = left(p)$. The destination processor q evaluates the associative binary operation $F(x, y)$ at each step.



Step 0, Configuration 0: 1 ring of 8 processors.



Step 1, Configuration 1: 2 rings of 4 processors each.



Step 2, Configuration 2: 4 rings of 2 processors each.

Fig. 7. The combine operation for $RMRN_3$.

Step 0: Configuration 0, $LSONE(p) = 0$
- 1 (001) → 0 (000)
- 3 (011) → 2 (010)
- 5 (101) → 4 (100)
- 7 (111) → 6 (110)

Step 1: Configuration 1, $LSONE(p) = 1$
- 2 (010) → 0 (000)
- 6 (110) → 4 (100)

Step 2: Configuration 2, $LSONE(p) = 2$
- 4 (100) → 0 (000)

3.1.3 Data Circulation

Let us consider the operation of circulating the data in the X register of each processor in $RMRN_n$ through the remaining $N - 1$ processors. We define an exchange sequence X_n as follows [11]:

$$X_i = X_{i-1}, i-1, X_{i-1};\ 1 < i \le n$$
$$X_1 = 0 \qquad (3)$$

Note that X_n is a palindromic sequence of length $2^n - 1 = N - 1$ where each integer in the sequence is in the range $[0, n - 1]$. The sequence X_n is such that by successively complementing the processor address along each bit in the sequence, each data item in each processor is made to pass through all the remaining processors in the $RMRN_n$. Also, since each X_i is a palindrome, X_n can be computed in $O(n) = O(\log_2 N)$ time and stored in a stack of height $N - 1$ [11]. Let $f(i, j)$ denote the jth member in the sequence X_i (from left to right). The procedure for data circulation in $RMRN_n$ is given in Fig. 8. The data circulate operation takes $O(N)$ unit hops on the $RMRN_n$.

```
Circulate(X, n)
  BEGIN
    FOR i:= 0 to (2**n) - 1 DO
      BEGINFOR
        config(n,f(n,i));
        IF (p[f(n,i)] = 0) THEN X(p) -> X(right(p))
        ELSE X(p) -> X(left(p));
      ENDFOR
END;
```

Fig. 8. Data circulation using reconfiguration.

Alternatively, the data circulation operation can be accomplished using a single configuration of the $RMRN_n$, i.e., $config(RMRN_n, 0)$ which consists of a single ring of $N = 2^n$ processors. The $O(N)$ algorithm for the data circulate operation which uses a single configuration of the $RMRN_n$ is outlined in Fig. 9. The advantage of this algorithm over the one in Fig. 8 is that the latter entails $O(N)$ calls to $config(\cdot, \cdot)$ whereas the former involves a single call to $config(\cdot, \cdot)$. Since reconfiguration in a practical system involves some overhead, the algorithm in Fig. 9 could be expected to run faster than the one in Fig. 8 although both algorithms have an asymptotic complexity of $O(N)$.

```
Circulate(X, n)
  BEGIN
    config(n, 0);
    FOR i:= 0 to (2**n) - 1 DO
      X(p) -> X(right(p));
  END;
```

Fig. 9. Data circulation using a single configuration.

3.2 Some Basic Imaging Operations on the RMRN

Given a 2D image $G = \{G(i, j); i, j \in [0, N - 1]\}$ and an RMRN with $N^2 = 2^{2n}$ processors where the value of the pixel (i, j) is stored in register $R(p)$ of processor $p = iN + j$ (i.e., row-major mapping). We will occasionally refer to the processor $p = iN + j$ by the ordered pair (i, j) keeping in mind that the row-major mapping is a bijection since $i = p$ div N and $j = p$ mod N. Similarly, we will also occasionally refer to the register $R(p)$ as $R(i, j)$.

3.2.1 Rotate Operation Within a Window

One of the basic imaging operations that could be performed on the RMRN is the *rotate* or the *cyclic shift* operation performed in a subimage or window within the entire image. The window essentially defines a subconfiguration within the RMRN. The rotate operation is then carried out in parallel in each window (i.e., by the processors in each subconfiguration). If we are interested in $config(w, k)$ in a window of size $W = 2^w$, then $RMRN_n$ has to be placed in $config(n, n - w + k)$. This ensures that $config(w, k)$ is a proper subconfiguration of $config(n, n - w + k)$ (Property 3). In $RMRN_n$ with $N = 2^n$, processors a window of size $W = 2^w$ is defined by appropriately assigning values to $(n - w)$ bits in the address of the processors. The address of the processor within the window is given by appropriately masking the lower order $(n - w)$ bits in the address of the processor (Property 3). Let $p\{w\}$ denote the address of the processor after the masking operation for a window of size $W = 2^w$ has been carried out for $RMRN_n$. We can see that $p\{w\}$, $right(p\{w\})$, and $left(p\{w\})$ in $config(w, k)$ refer to processors p, $right(p)$, and $left(p)$ respectively in $config(n, n - w + k)$ since the connectivity pattern is preserved in subconfigurations of the $RMRN_n$ (Property 3).

In $RMRN_{2n}$ with $N^2 = 2^{2n}$ processors a window of size $W \times W = 2^{2w}$ is defined by appropriately assigning values to $2(n - w)$ bits in the address of the processors. The address of the processor within the window is given by appropriately masking the lower order $2(n - w)$ bits in the address of the processor. The address of the processor after the masking operation for a window of size $W \times W = 2^{2w}$ has been carried out is denoted by $p\{2w\}$. The kth bit in the masked address is then denoted as $p\{2w\}[k]$.

The rotate operation within a window could be described thus:

A 1D rotate operation by 2^k pixels in a window of size $W^2 = 2^{2w}$ can be described as:

for rotate *left*: $R(p) \leftarrow R((p + 2^k) \bmod W^2)$
for rotate *right*: $R(p) \leftarrow R((p - 2^k) \bmod W^2)$

A 2D rotate operation by 2^k pixels can be described as:

for rotate *right*: $R(i, j) \leftarrow R(i, (j - 2^k) \bmod W)$
for rotate *left*: $R(i, j) \leftarrow R(i, (j + 2^k) \bmod W)$
for rotate *up*: $R(i, j) \leftarrow R((i + 2^k) \bmod W, j)$
for rotate *down*: $R(i, j) \leftarrow R((i - 2^k) \bmod W, j)$

A generic rotate operation within a window is described in Fig. 10. For a 1D rotation in a window of size $W^2 = 2^{2w}$ the *RotateWin* procedure would have to be invoked thus:

Rotate left by 2^k pixels: **`RotateWin(R, 2n, k, 2w-1, 2w, 0)`**
Rotate right by 2^k pixels: **`RotateWin(R, 2n, k, 2w-1, 2w, 1)`**

For a 2D rotation in a window of size $W^2 = 2^{2w}$ the *RotateWin* procedure would have to be invoked thus:

Rotate down by 2^k pixels: **`RotateWin(R, 2n, w+k, 2w-1, 2w, 1)`**
Rotate up by 2^k pixels: **`RotateWin(R, 2n, w+k, 2w-1, 2w, 0)`**
Rotate left by 2^k pixels: **`RotateWin(R, 2n, k, w-1, 2w, 0)`**
Rotate right by 2^k pixels: **`RotateWin(R, 2n, k, w-1, 2w, 1)`**

```
RotateWin(R, n, s, r, w, flag)
  BEGIN
    config(n, n-w+s);
    IF (p{w}[s] = 0) THEN R(p) -> R(right(p))
    ELSE R(p) -> R(left(p));
    FOR b:= s+1 to r DO
      BEGINFOR
        config(n, n-w+b);
        IF (flag = 0) THEN
          IF (p{w}[b-1] = 1) AND... AND
              (p{w}[s] = 1) THEN
            IF (p{w}[b] = 0) THEN R(p) ->
              right(R(p))
            ELSE R(p) -> left(R(p));
          ENDIF;
        ELSE
          IF (p{w}[b-1] = 0) AND... AND
                (p{w}[s] = 0) THEN
            IF (p{w}[b] = 0) THEN R(p) ->
                right(R(p))
            ELSE R(p) -> left(R(p));
            ENDIF;
          ENDIF;
      ENDFOR;
  END;
```

Fig. 10. Rotate operation within a window.

The rotate operation within a $W \times W = 2^{2w}$ window contains $O(w) = O(\log_2 W)$ unit hops. The broadcast, combine and circulate operations can be similarly performed within a predefined window. The broadcast and combine operations would contain $O(w) = O(\log_2 W)$ unit hops and the circulate operation would contain $O(W)$ unit hops for a window of size $W = 2^w$. The algorithms for the broadcast, combine and circulate operations within a window are similar to the ones already described and hence will not be repeated here.

3.2.2 Accumulate Operation

Each processor j has an array $A[0 \ldots M-1]$ of size M. $A[i](j)$ refers to the element $A[i]$ in processor j. In addition, each processor has a value in its I register. After the accumulate operation, the M elements of the array A in each processor j are such that:

$$A[i](j) = I((j + i) \bmod N),\ 0 \le i < M,\ 0 \le j < W$$

where $W = 2^w$ is the window size.

The accumulate operation can be performed as a minor modification of the circulate operation within a window using the following lemma from Ranka and Sahni [33]

LEMMA 3.1. *Let I_0, I_1, …, I_{W-1} denote the initial values in $I(0)$, $I(1)$, …, $I(W-1)$. Let index(j, i) be such that index(j, i) is in $A(j)$ following the* i*th iteration of the* `for` *loop in the data circulation algorithm shown in* Fig. 8. *Initially index$(j, i) = j$. For every $i > 0$, index(j, i) = index$(j, i-1) \oplus 2^{f(w,i)}$ where $\oplus$ denotes the EXOR operation and $f(w, i)$ is as defined in Section* 3.1.3.

The value of *index*(j, i) essentially conveys the address (within the window) of the originating processor during each iteration of the circulate operation. The algorithm for the accumulate operation is given in Fig. 11. The data accumulate operation *Accumulate*(A, I, M) can be performed in $O(M + \log_2 (N/M))$ unit hops on the $RMRN_n$.

```
Accumulate(A, I, M)
{Each processor accumulates in array A the I
values of the next M processors. The window
size is M = 2**m}
  BEGIN
    A[0] = I;
    index = p;
    FOR i:= 1 to 2**m - 1 DO
    BEGINFOR
      config(n, f(m,i) + n - m);
      IF (p{m}[f(m,i)] = 0) THEN I(p) ->
        I(left(p))
      ELSE I(p) -> I(right(p))
      index := index EXOR 2**f(m,i);
      location := (index - p{m}) mod M;
      A[location] := I;
    ENDFOR;
END;
```

Fig. 11. Accumulate operation within a window.

4 LOW- AND INTERMEDIATE-LEVEL VISION OPERATIONS ON THE RMRN

The primitive operations defined in the previous section can be extended to design operations for low- and intermediate-level vision on the RMRN. Typical examples of low-level vision operations are image transforms such as the Fast Fourier Transform (FFT), and convolution for edge detection, image filtering, image smoothing, and feature detection via template matching. A typical example of intermediate-level vision is feature extraction via the Hough Transform. We consider the implementation of these operations on the RMRN.

4.1 The Fast Fourier Transform

For the purpose of discussion, we have selected one of the most widely known *decimation-in-frequency* FFT algorithms described in [16]. First, we describe how the one-dimensional FFT algorithm can exploit the RMRN system and then address the problem of expanding the algorithm to handle the two-dimensional case. Let $s(m)$, $m = 0, 1, \ldots, M-1$ be M samples of a time function. The Discrete Fourier Transform of $s(m)$ is defined to be the discrete function $X(j)$, $j = 0, 1, \ldots, M-1$ given by:

$$X(j) = \sum_{m=0}^{M-1} s(m) \cdot W^{jm}$$

where $W = e^{2\pi i/M}$, $i = \sqrt{-1}$, and $j = 0, 1, \ldots, M-1$.

In the $RMRN_n$ where $N = 2^n = M/2$, processor p initially contains $s(p)$ and $s(p + M/2)$, where $0 \le p < N$. The FFT algorithm consists of $\log_2 M$ stages of computation and $\log_2 (M/2) = \log_2 N$ stages of parallel data transfers (unit hops). A process known as the *butterfly* (Fig. 12a) is executed $M/2$ times (in parallel) at each stage of computation. For example, for a 16-point FFT, eight butterfly processes are performed in parallel at each stage of computation where each process receives two inputs and produces two results. Each processor, however, outputs only one of the two results to another processor, keeping the other in its memory. At the

*i*th parallel data transfer stage a processor *p* outputs its *X* value if *p*[*i*] = 1 else it outputs its *Y* value. It is only at the very last stage of computation that each processor outputs both results of its butterfly process (to the outside world). The actual interconnection network (i.e., butterfly network) required to perform the four stages of the 16-point FFT is shown in Fig. 12b where the data is considered to flow from left to right.

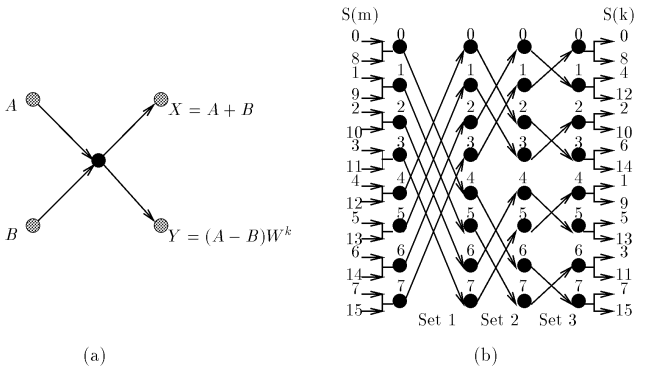


Fig. 12. (a) Butterfly process (b) Interconnection pattern for the 16-point FFT on the $RMRN_3$.

The FFT algorithm described in Fig. 13, performs the *M*-point FFT calculations using $\log_2 (M/2) = \log_2 N$ parallel data transfers (unit hops) on the $RMRN_n$ with $N = 2^n$ processors. This is a lower bound on the number of parallel data transfers required to perform an *M*-point FFT when the *M* points are initially distributed over $M/2$ processors [16]. The number of parallel butterfly operations performed is $\log_2 M$, where each butterfly involves two complex additions and one complex multiplication in each processor. The asymptotic complexity of the algorithm is $O(\log_2 (M/2)) = O(\log_2 N)$. The 2D FFT of an $N \times N$ image can be computed by first computing the 1D FFT of each row of the image followed by the 1D FFT of each column of the resulting image.

4.2 Convolution for Edge and Feature Detection

The convolution operation is a fundamental operation used in both edge and feature detection. In both cases, the underlying image is convolved with a template or a set of templates and the edges or features of interest are deemed to be points in the image where the output of the convolution has a maximum. A one-dimensional convolution is given by the relation:

$$C[i] = \sum_{k=0}^{M-1} I[(i+k) \bmod N] \cdot T[k] \quad 0 \le i \le N \qquad (4)$$

where $I[0 \ldots N-1]$ is the 1D image and $T[0 \ldots M-1]$ is the template. The convolved output is $C[0 \ldots N-1]$. A two-dimensional convolution is given by the relation:

$$C[i,j] = \sum_{k=0}^{M-1} \sum_{l=0}^{M-1} I[(i+k) \bmod N, (j+l) \bmod N] \cdot T[k,l] \qquad (5)$$

where $0 \le i, j \le N$, $I[0 \ldots N-1, 0 \ldots N-1]$ is the $N \times N$ image, $T[0 \ldots M-1, 0 \ldots M-1]$ the $M \times M$ template and $C[0 \ldots N-1, 0 \ldots N-1]$ the $N \times N$ image resulting from the convolution. The two-dimensional convolution can be decomposed into a series of 1D convolutions and summations [31], [32]:

$$C'[i,j,k] = \sum_{l=0}^{M-1} I[(i+k) \bmod N, (j+l) \bmod N] \cdot T[k,l]$$
$$C[i,j] = \sum_{k=0}^{M-1} C'[i,j,k] \qquad (6)$$

For the 1D convolution operation we assume that there are *N* processors in the RMRN and the vector *I* is mapped onto the RMRN using the identity mapping, i.e., *I*(*i*) is mapped onto processor *i*. We also assume that there are (N/M) copies of the template *T* on the RMRN with one copy in each subconfiguration of *M* processors. Within each subconfiguration, the mapping of *T* is identical to the mapping of *I*. Since each processor has $O(M)$ memory, the most effective strategy is to perform a data accumulate operation on the *I* values such that each processor has all the *I* values necessary to compute a single value of the output vector *C*. The mapping of the output vector *C* on the RMRN is identical to the mapping of the input vector *I*. Let *f*(*n*, *i*) denote the *i*th number in the palindromic exchange sequence in (3). The code for 1D convolution with $O(M)$ memory per processor is given in Fig. 14. The template is stored in the *T* register and the final result in the *C* register of each processor.

For the 2D convolution operation we assume that an $RMRN_{2n}$ with $N^2 = 2^{2n}$ processors is available. We assume an identity mapping for the image *I*, i.e., *I*(*i*, *j*) is contained in the *I* register of the (*i*, *j*)th processor. We also assume that there are $(N/M)^2$ copies of the template *T*; one copy in each subconfiguration of M^2 processors. Within each subconfiguration the mapping of *T*(*I*, *j*) is identical to the mapping

```
FFT(M, n)
{There are N=2**n processors and N=M/2}
  BEGIN
    config(n, n);
    A(p) := s(p);
    B(p) := s(p + M/2);
    k := p;
    X(p) := A(p) + B(p);
    Y(p) := (A(p) - B(p)) * W**k; {** denotes
          exponentiation}
    FOR i = n-1 DOWNTO 0 DO
      BEGINFOR
        config(n, i);
        IF (p[i] = 1) THEN
          X(p) -> A(left(p));
          B := Y;
        ELSE
          Y(p) -> B(right(p))
          A := X;
        ENDIF;
        k := (2*k) mod N;
        X(p) := A(p) + B(p);
        Y(p) := (A(p) - B(p)) * W**k;
    ENDFOR;
  END;
```

Fig. 13. M-point FFT on the M/2 processor RMRN.

of $I(i, j)$. The mapping of the output image $C(i, j)$ is also identical to that of $I(i, j)$. The code for the 2D convolution can be derived by decomposing it into a series of 1D convolutions and summations using (6). Both the 1D and 2D convolutions can be performed on the *RMRN* in with the same asymptotic order of complexity as would result by implementing them on the hypercube [31], [32], [20], [12] in an SIMD or SPMD mode parallelism. The 1D convolution can be performed in $O(M + \log_2 M + \log_2 N)$ unit hops whereas the 2D convolution can be performed in $O(M^2 + \log_2 M + \log_2 N)$ unit hops.

```
procedure Convolve1D(M)
  BEGIN
    ACCUM(A, I, M);
    C := 0;
    index := p mod M;
    FOR j:= 1 to M DO
      BEGINFOR
        C := C + A[index] * T;
        m := log2(M); {log2() is log to the
            base 2}
        l := f(m, j);
        config(n,l);
        IF (p[l] = 0) THEN T(p) <- T(right(p))
        ELSE T(p) <- T(left(p));
        index := index EXOR 2**l; {** denotes
            exponentiation}
    ENDFOR
  END;
```

Fig. 14. 1D Convolution on the RMRN.

4.3 The Hough Transform

The Hough Transform is known to be a very important though computationally intensive operation in computer vision and image processing. The conventional Hough Transform is used to detect and extract features with well defined parametric descriptions such as lines, circles, ellipses, etc., in an image. Serial implementation of the Hough Transform on a uniprocessor architecture is computationally intensive and is not feasible for real-time applications. Thus, parallelization of the Hough Transform is imperative and has been attempted by several researchers on a variety of architectures, such as the mesh [34], [10], hypercube [33], pyramid [35], butterfly network [7], tree network [17], systolic array [9], and shared memory architecture [8]. In this section, we consider the parallelization of the Hough Transform for line detection on the RMRN.

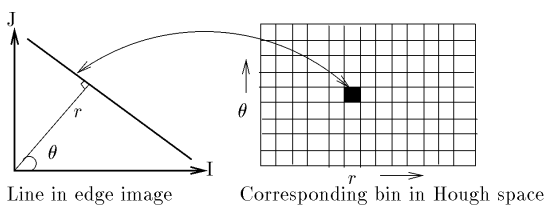


Fig. 15. Hough Transform for line detection.

Let $E = E(i, j)$, $0 \le i, j < N$ be an edge image such that $E(i, j) = 1$ if the point (i, j) is an edge point and $E(i, j) = 0$ otherwise. Formally, the Hough Transform $H(x, y)$ (Fig. 15) is given by:

$$H(x, y) = |\{(i, j) \text{ such that } x = \lfloor i\cos\theta_y + j \sin \theta_y \rfloor\}|$$

where $\theta_y = \frac{\pi y}{Y}$, $0 \le y < Y$, $E(i, j) = 1$ and $|P|$ denotes the cardinality of set P. Here, Y denotes the total number of levels in which the angle θ_y is quantized in the range $[0, \pi)$. Consequently, the resulting transform is said to be the *Y-angle* Hough Transform of the edge image $E(i, j)$. For an $N \times N$ image, x lies in the range $[0, \sqrt{2}N)$ whenever y lies in the range $[0, Y)$. Let $N = 2^n$ and let $Y = 2^m$ such that $m < n$. Since x lies in the range $[0, \sqrt{2}N)$ we use an RMRN that has $2N^2 = 2^{2n+1}$ processors, i.e., the smallest number of processors that are a power of 2 and greater than $N \times \sqrt{2}N = \sqrt{2}N^2$. We view the *RMRN* as an $N \times 2N$ array. The Hough Transform $H(x,y)$ is computed in three phases:

Phase 1: Consider the subconfiguration $RMRN_{n+m+1}$ consisting of $Y \times 2N = 2^{n+m+1}$ processors. Each such subconfiguration $RMRN_{n+m+1}$ will be used to compute the Hough Transform $h(x, y)$ in a window of size $Y \times 2N = 2^{n+m+1}$. That is,

$$h(x, y) = |\{(i, j) \text{ such that } x = \lfloor i\cos\theta_y + j \sin \theta_y \rfloor\}|$$

where $\theta_y = \frac{\pi y}{Y}$, $0 \le y < Y$, and $E(i, j) = 1$ and (i, j) lies within the subconfiguration.

Phase 2: Compute the window sum of each of the $h(x, y)$s.

$$H(x, y) = \sum_W h(x, y)$$

where W is a window of size $N / Y = 2^{n-m}$.

Phase 3: Compute the maximum in $H(x, y)$.

Phase 1 dominates the overall complexity of the algorithm. Phases 2 and 3 are straightforward since they each involve a simple combine operation.

Let each processor $p(i, j)$ in $RMRN_{n+m+1}$ generate a triple (x, y, q) where x is the column number of $p(i, j)$ in $RMRN_{n+m+1}$, y is the row number of $p(i, j)$ in $RMRN_{n+m+1}$, and q is defined as follows:

$$q = |\{(i, j) \text{ such that } x = \lfloor i\cos\theta_y + j \sin \theta_y \rfloor\}|$$

where $\theta_y = \frac{\pi y}{Y}$, $0 \le y < Y$, and $E(i, j) = 1$. We assume that q has been initialized to zero. The triple (x, y, q) is stored in the *VOTES* register of each processor. Each element within the triple is referred to as *VOTES.x*, *VOTES.y*, and *VOTES.q*, respectively. Let W denote the window corresponding to the subconfiguration $RMRN_{n+m+1}$. The key factor in designing an efficient algorithm for Phase 1 is to realize that not all pixels $E(i, j)$ in a given row of the $Y \times 2N$ Hough accumulator array $h(x, y)$ contribute towards the vote count of a given value of (x, y) (i.e., a given bin in the Hough accumulator array) [10], [33]. We subdivide Phase 1 in the Hough Transform algorithm in the following subphases:

Phase 1(a): $\frac{\pi}{4} \le \theta_y < \frac{\pi}{2}$

Phase 1(b): $\frac{\pi}{2} \le \theta_y < \frac{3\pi}{4}$

Phase 1(c): $0 \le \theta_y < \frac{\pi}{4}$

Phase 1(d): $\frac{3\pi}{4} \le \theta_y < \pi$

For each of the subphases 1(a)–1(d), the $RMRN_{2n+1}$ is configured as composed of $\frac{4N}{Y}$ windows (i.e., subconfigurations) of size $\frac{Y}{4} \times 2N$ each. In particular, we exploit the following two lemmas from Ranka and Sahni [33]:

LEMMA 4.1. *When* $\frac{\pi}{4} \le \theta_y < \frac{\pi}{2}$, *two pixels* (i, j) *and* $(i, j + k)$, $k > 0$ *can contribute to the count of the same* $H(x, y)$ *only if* $k = 1$.

LEMMA 4.2. *When* $\frac{\pi}{4} \le \theta_y < \frac{\pi}{2}$, *two pixels* (i, j) *and* $(i + 1, k)$ *can contribute to the count of the same* $H(x, y)$ *only if* $k \in \{j, j - 1\}$.

Using Lemmas 4.1 and 4.2, one can implement the vote accumulation in a single row by a sequence of two *rotate* operations; a *rotate* operation to the right by one position followed by another *rotate* operation to the left by two positions. Further improvement can be realized by replacing the *rotate* operations along the rows by the *left* and *right* operations. This can be done by visualizing the $RMRN_{2n+1}$ as composed of $N = 2^n$ rings each containing $2N = 2^{n+1}$ processors (i.e., *config*$(2n + 1, n)$). Each $\frac{Y}{4} \times 2N$ window can then be looked upon as subconfiguration $RMRN_{n+m-1}$ composed of $\frac{Y}{4} = 2^{m-2}$ rings of $2N = 2^{n+1}$ processors each (i.e., *config*$(n + m - 1, m - 2)$). In order that each of the $N = 2^n$ rings in $RMRN_{2n+1}$ contain a single row of the image $E(i, j)$, the mapping of the image $E(i, j)$ has to be modified from the row-major mapping described earlier to a column-major mapping where the value of the pixel (i, j) is stored in register $R(p)$ of processor $p = jN + i$ in $RMRN_{2n+1}$. The rotate operations along the rows are followed by a rotate operation by a single position along the columns. The pseudocode for subphase 1(a) is shown in Fig. 16. The subphases 1(b)–1(d) are tackled in an identical manner.

The overall asymptotic complexity of the Hough Transform on the *RMRN* is $O(Y \log_2 Y + \log_2 N)$ in either the SIMD or SPMD mode of parallelism. This compares favorably with the optimal $O(Y + \log_2 N)$ Hough Transform algorithm on the MIMD *n-cube* reported by Ranka and Sahni [33].

5 Implementation and Performance Evaluation of the RMRN

We have built a prototype RMRN using a reconfigurable network of INMOS T400 transputers. The switches needed for input/output and reconfiguration were built using off-the-shelf standard TTL components and cross-bar switches. The transputers were used in the SPMD mode of parallelism and were programmed using *Logical Systems C*. Each node in the prototype RMRN is an INMOS T400 Transputer which has a 32 bit 10 MIPS RISC processor, 2 KB of internal (i.e., on-chip) RAM, and an external memory interface for off-chip memory access. The image data is one byte per pixel and resides entirely in the external memory.

```
procedure votecount(W)
  BEGIN
    VOTES.x := x; {column address of p[n+m+1]
          in the window W}
    VOTES.y := y; {row address of p[n+m+1] in
          the window W}
    VOTES.q := 0; {initialize the number of
          accumulated votes to 0}
    FOR count1 := 1 to Y/4 DO
      BEGINFOR
        config(2n+1, n);
        theta := ((VOTES.y + 1) * PI)/Y + (PI/4);
        x1 = trunc(i * COS(theta) + j *
            SIN(theta));
        IF (x1 = VOTES.x) THEN VOTES.q :=
            VOTES.q + 1;
        VOTES(p) -> VOTES(right(p)); {Rotate
            one position to the right along the
            row}
        theta := ((VOTES.y + 1) * PI)/Y + (PI/4);
        x1 = trunc(i * COS(theta) + j *
            SIN(theta));
        IF (x1 = VOTES.x) THEN VOTES.q :=
            VOTES.q + 1;
        VOTES(p) -> VOTES(left(p));
        VOTES(p) -> VOTES(left(p)); {Rotate two
            positions to the left along the
            row}
        RotateWin(VOTES, 2n+1, n+1, n+m, n+m+1,
            1); {Rotate one position down along
            the column}
      ENDFOR
  END
```

Fig. 16. Algorithm for the Hough Transform.

5.1 Hardware Implementation of the Input/Output Switches

As mentioned in Section 2.2, in *config*$(RMRN_n, i)$ the input control switch needs to check whether or not the least significant i bits of the processor address are all 0. Conversely, in *config*$(RMRN_n, i)$ the output control switch needs to check whether or not the least significant i bits of the processor address are all 1. The circuit for the input control switch is given in Fig. 17. The bit select signal $(s_{n-1}, s_{n-2}, \ldots, s_0)$ in Fig. 17 determines the value of i (i.e., the number of bits to be tested). In *config*$(RMRN_n, i)$ we set $s_{i-1} = s_{i-2} = \ldots = s_0 = 1$ and $s_{n-1} = s_{n-2} = \ldots = s_i = 0$. It is easy to see that the circuit in Fig. 17 determines whether the last i bits of p are all 0 or not. If the last i bits of p are all 0 and the input control signal *IN* is high, then the circuit in Fig. 17 connects p to the input channel otherwise it connects p to the output of $(p - 1) \bmod N$.

Similarly, the output control switch shown in Fig. 18 checks to see whether the last i bits of p are all 1 or not. If the last i bits of p are all 1 and the control signal *OUT* is high, then the output control switch connects p to the output channel else it connects p to the input of $(p + 1) \bmod N$. The setting for the bit select signal $(s_{n-1}, s_{n-2}, \ldots, s_0)$ is the same as that for the input control switch. In the current prototype of the RMRN, the input/output control switches are implemented using standard TTL components.

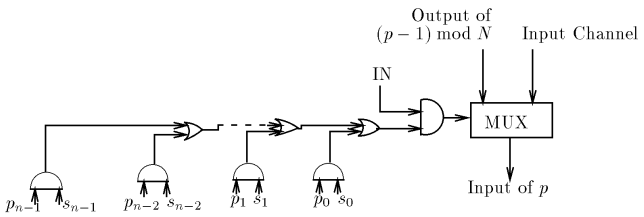


Fig. 17. The input control switch.

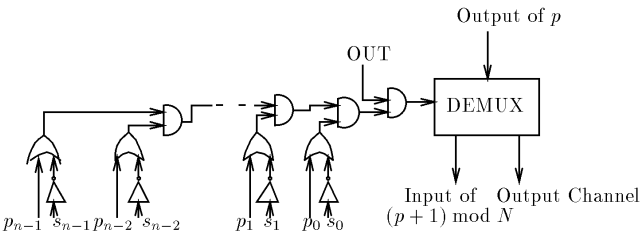


Fig. 18. The output control switch.

5.2 Hardware Implementation of the Reconfiguration Switching Network

As mentioned in Section 2.3, the reconfiguration switching network for the RMRN is a modular, multistage network similar to the butterfly network. Consider the circuit SR_1 shown in Fig. 19a made up of standard multiplexer elements. The truth table of SR_1 can be seen to be:

S	Z_0	Z_1
0	A_0	A_1
1	A_1	X

With $Z_0 = A_0$, $Z_1 = A_1$, and $X = A_0$, the switch SR_1 can be converted to SW_1 which is, in fact, the switching network for $RMRN_1$ (Fig. 19b).

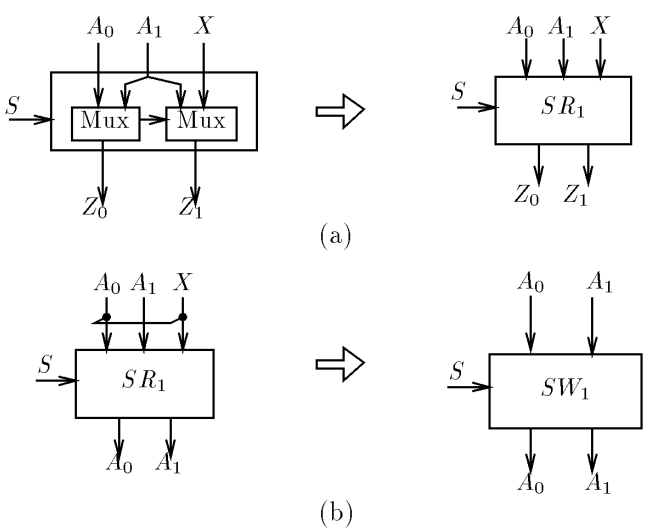


Fig. 19. (a) The switch SR_1 (b) The switch SW_1.

We now give a recursive definition for the switch SW_n that could be used in the switching network for $RMRN_n$. We assume that we have constructed a switch SR_n by cascading $\frac{N}{2}$ SR_1 switches as shown in Fig. 20. We also assume that we have constructed a switch SW_{n-1} which is used in the switching network for $RMRN_{n-1}$ with $2^{n-1} = \frac{N}{2}$ processors. Fig. 21 shows how the switch SW_n could be constructed from a single SR_n switch and two SW_{n-1} switches. Here, *SHFL(n)* denotes a *perfect shuffle* of 2^n elements. The switch SW_n followed by a *perfect unshuffle* of its 2^n outputs (denoted by *UNSHFL(n)*) constitutes the reconfiguration switch RSW_n for $RMRN_n$ as shown in Fig. 22. Since switch SW_1 consists of a single SR_1 switch, the solution of the recurrence depicted in Fig. 21 shows that we need $\frac{N}{2}\log_2 N$ SR_1 switches to construct a switch SW_n (or RSW_n) which could be used in the switching network for $RMRN_n$. Although the asymptotic complexity of $O(N \log_2 N)$ is not very attractive for very large values of N, for moderate values of N it is manageable. Fig. 23 depicts $RMRN_3$ with eight processors configured around the RSW_3 switch. The inputs I_0, ..., I_7 and the outputs O_0, ..., O_7 of the switch RSW_3 enable the reconfigurable multi-ring structure whereas the ring R provides the inter-ring connections.

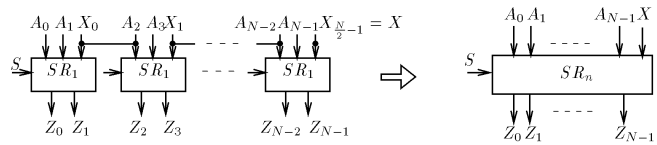


Fig. 20. The switch SR_n.

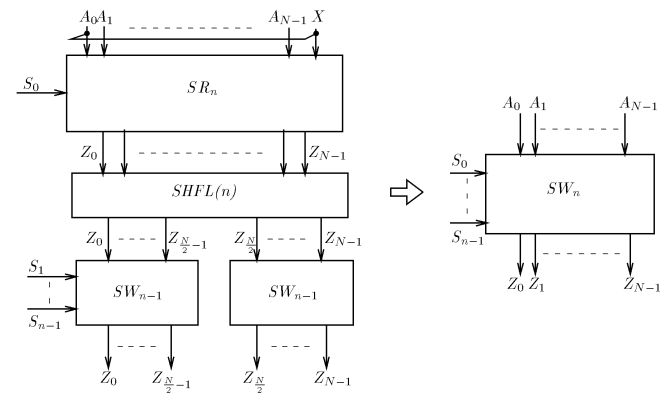


Fig. 21. The reconfiguration switch SW_n.

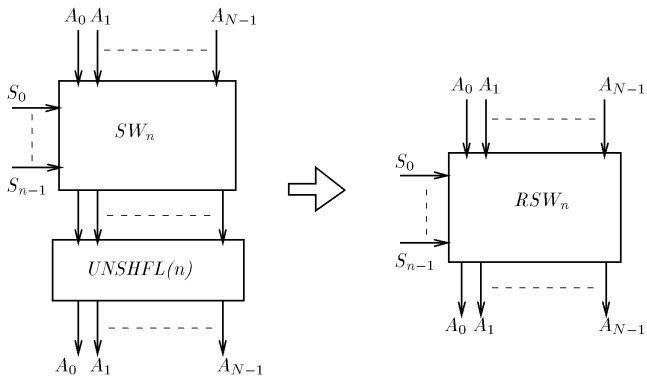


Fig. 22. The reconfiguration switch for $RMRN_n$.

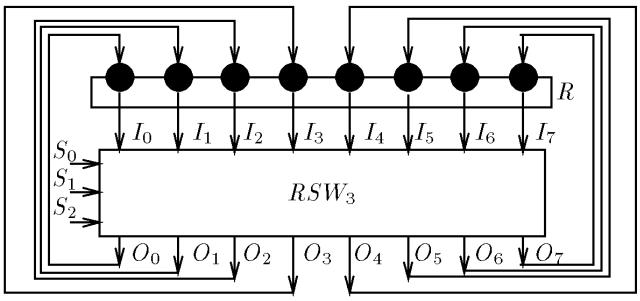


Fig. 23. The $RMRN_3$ configured around the RSW_3 switching network.

The switching network for the RMRN provides point-to-point connections without contention, is modular, composed of fairly simple components, hierarchical and incrementally upgradable. We have sized the RSW_4 switch made from TTL gates (50 packages, 80 nsec total delay) and from TTL MSI logic [38] (16 packages, 72 nsec total delay). Since increased integration makes the switching network both smaller and faster, we have examined the possibility of a custom integrated circuit to provide the switching function [3]. According to the timing analysis from the simulation of the VLSI implementation, the total delay introduced by the switching network is less than 15 nsec from input to output for a 16 processor RMRN. The speed, simplicity, and capability of the VLSI implementation of the switching network make a cost-effective reconfigurable parallel multiprocessor system possible. The reconfiguration switching network in the current prototype has been constructed using TTL MSI logic and standard crossbar switches and therefore leaves room for improvement using a custom CMOS VLSI implementation.

5.3 Performance Evaluation

We have obtained the timings for various operations on the prototype RMRN. These are tabulated in Tables 1, 2, 3, 4, 5, 6, 7. The tile size in the case of the image-tile broadcast and the all-to-all broadcast, and the window size in the case of the 1D rotate operation and the Hough Transform were decided by the size of the image and the number of processors in the RMRN. For example, for a $1K \times 1K$ image and a 16 processor RMRN, the resulting tile (window) size was $64K = 256 \times 256$. In the case of the Hough Transform, the angle $\theta \in [0, \pi)$ was quantized in steps of 1 degree or $\pi/180$ radians resulting in a total of 180 quantization levels.

We note that the timings that we have obtained from our prototype have been fairly encouraging in spite of the fact that the hardware used in the prototype is not the latest or the fastest available. We estimate that if each node of the RMRN is a T9000 transputer (the latest in the INMOS transputer series), the timings for each of the operations discussed in the previous subsections would reduce by a factor in the range [8, 20]. Operations such as the Hough Transform that are computation intensive would exhibit speedup factors close to 20 by switching over to the T9000 transputer whereas broadcast operations that are interprocessor communications intensive would exhibit speedup factors that are closer to 8. The RMRN need not be limited to a transputer-based implementation; one could also envisage an RMRN implemented using a cluster of workstations or personal computers interconnected via the reconfiguration switch shown in Fig. 22. However, if one is interested in a physically compact multiprocessor system that could be used for computer vision on a mobile, autonomous or dextrous robotic platform, a transputer-based implementation of the RMRN seems to be an appropriate choice.

In summary, the RMRN offers an interconnection network for parallel/distributed processing for computer vision problems that is:

1) *general* and *flexible* since the compute nodes can be processors or computing platforms of any type, manufacture or make and since software written for other prototypical architectures such as the hypercube and the 2D toroidal mesh can be readily implemented on the RMRN,
2) *powerful* since the RMRN can implement the standard parallelization strategies such as process decomposition, data decomposition, and functional decomposition (i.e., pipelining) and any combination(s) thereof,
3) *efficient* since the diameter scales of the RMRN scales as $O(\log_2 N)$ ensuring that the algorithms designed for the RMRN exhibit good speedup characteristics and high efficiency,
4) *scalable* since the interprocessor communication links scales as $O(N)$ ensuring thereby that the cost of the system also scales well with increasing network size,
5) *cost-effective* since the RMRN can be built to exploit readily available state-of-the-art processor and switching technology, and
6) *upgradable* since the RMRN can be readily upgraded to take advantage of advances in VLSI and processor technology and faster interprocessor communication links.

A fixed topology architecture such as a *mesh* or an *n-cube* would, in general, prove more efficient than the RMRN for those image processing and computer vision operations that are tailored for that specific architecture since no reconfiguration overhead is entailed. The RMRN on account of its reconfigurability, however, would provide greater flexibility in algorithm design than would a fixed topology architecture.

TABLE 1
TIMINGS FOR SIMPLE IMAGE BROADCAST

Image Size	RMRN Size			
	16 nodes	32 nodes	64 nodes	128 nodes
$1K \times 1K$ pixels	3.3 sec.	4.2 sec.	5.1 sec.	5.9 sec.
$2K \times 2K$ pixels	13.5 sec.	16.8 sec.	20.2 sec.	23.5 sec.
$4K \times 4K$ pixels	53.7 sec.	67.2 sec.	80.6 sec.	94.1 sec.

TABLE 2
TIMINGS FOR IMAGE TILE BROADCAST AND ALL-TO-ALL BROADCAST

Image Size	RMRN Size			
	16 nodes	32 nodes	64 nodes	128 nodes
$1K \times 1K$ pixels	1.01 sec.	1.03 sec.	1.04 sec.	1.05 sec.
$2K \times 2K$ pixels	4.04 sec.	4.12 sec.	4.16 sec.	4.18 sec.
$4K \times 4K$ pixels	16.16 sec.	16.48 sec.	16.63 sec.	16.71 sec.

TABLE 3
TIMINGS FOR THE 1D ROTATE OPERATION

Image Size	RMRN Size			
	16 nodes	32 nodes	64 nodes	128 nodes
$1K \times 1K$ pixels	147 ms.	92 ms.	55 ms.	32 ms.
$2K \times 2K$ pixels	586 ms.	368 ms.	220 ms.	128 ms.
$4K \times 4K$ pixels	2.34 sec.	1.47 sec.	880 ms.	512 ms.

TABLE 4
TIMINGS FOR THE ACCUMULATE OPERATION

Image Size	Window Size	RMRN Size			
		16 nodes	32 nodes	64 nodes	128 nodes
$1K \times 1K$ pixels	4×4 pixels	105 ms.	53 ms.	26 ms.	13 ms.
	8×8 pixels	420 ms.	210 ms.	105 ms.	53 ms.
	16×16 pixels	1.68 sec.	840 ms.	420 ms.	210 ms.
	32×32 pixels	6.72 sec.	3.36 sec.	1.68 sec.	840 ms.
$2K \times 2K$ pixels	4×4 pixels	420 ms.	210 ms.	105 ms.	53 ms.
	8×8 pixels	1.68 sec.	840 ms.	420 ms.	210 ms.
	16×16 pixels	6.72 sec.	3.36 sec.	1.68 sec.	840 ms.
	32×32 pixels	26.88 sec.	13.44 sec.	6.72 sec.	3.36 sec.
$4K \times 4K$ pixels	4×4 pixels	1.68 sec.	840 ms.	420 ms.	210 ms.
	8×8 pixels	6.72 sec.	3.36 sec.	1.68 sec.	840 ms.
	16×16 pixels	26.88 sec.	13.44 sec.	6.72 sec.	3.36 sec.
	32×32 pixels	107.52 sec.	53.76 sec.	26.88 sec.	13.44 sec.

TABLE 5
TIMINGS FOR THE 2D FFT OPERATION

Image Size	RMRN Size			
	16 nodes	32 nodes	64 nodes	128 nodes
$1K \times 1K$ pixels	1.43 sec.	894 ms.	536 ms.	313 ms.
$2K \times 2K$ pixels	5.72 sec.	3.58 sec.	2.14 sec.	1.25 sec.
$4K \times 4K$ pixels	22.91 sec.	14.31 sec.	8.58 sec.	5.00 sec.

TABLE 6
TIMINGS FOR THE CONVOLUTION OPERATION

Image Size	Window Size	RMRN Size			
		16 nodes	32 nodes	64 nodes	128 nodes
$1K \times 1K$ pixels	4×4 pixels	2.10 sec.	1.05 sec.	524 ms.	262 ms.
	8×8 pixels	8.39 sec.	4.20 sec.	2.10 sec.	1.05 sec.
	16×16 pixels	33.55 sec.	16.78 sec.	8.39 sec.	4.20 sec.
	32×32 pixels	134.21 sec.	67.11 sec.	33.55 sec.	16.78 sec.
$2K \times 2K$ pixels	4×4 pixels	8.39 sec.	4.20 sec.	2.10 sec.	1.05 sec.
	8×8 pixels	33.55 sec.	16.78 sec.	8.39 sec.	4.20 sec.
	16×16 pixels	134.2 sec.	67.11 sec.	33.55 sec.	16.78 sec.
	32×32 pixels	529.6 sec.	264.8 sec.	132.4 sec.	66.20 sec.
$4K \times 4K$ pixels	4×4 pixels	33.55 sec.	16.78 sec.	8.39 sec.	4.20 sec.
	8×8 pixels	134.2 sec.	67.11 sec.	33.55 sec.	16.78 sec.
	16×16 pixels	529.6 sec.	264.8 sec.	132.4 sec.	66.2 sec.
	32×32 pixels	2118 sec.	1059 sec.	529.6 sec.	264.8 sec.

TABLE 7
TIMINGS FOR THE HOUGH TRANSFORM OPERATION

Image Size	RMRN Size			
	16 nodes	32 nodes	64 nodes	128 nodes
$1K \times 1K$ pixels	63.32 sec.	34.72 sec.	20.89 sec.	14.32 sec.
$2K \times 2K$ pixels	253.4 sec.	138.9 sec.	83.34 sec.	57.17 sec.
$4K \times 4K$ pixels	1014 sec.	555.5 sec.	333.2 sec.	228.9 sec.

Also, the RMRN has advantage of being able to *simultaneously* provide computational efficiency due to its $O(\log_2 N)$ network diameter and cost effectiveness due to its $O(N)$ number of interprocessor communication links for a system of size N. The *n-cube* has a $O(\log_2 N)$ network diameter but the number of interprocessor communication links scales as $O(N \log_2 N)$ whereas in the case of the k-D mesh, the number of interprocessor communication links scales as $O(kN)$ but the network diameter scales as $O(N^{\frac{1}{k}})$ for a system of size N. In the case of the *n-cube* and the mesh, one is confronted with a trade-off between computational efficiency and cost effectiveness which one can circumvent in the case of the RMRN.

6 CONCLUSIONS

In this paper, we have described a reconfigurable multiring network (RMRN) and highlighted some important properties of the RMRN structure. We have shown that a broad class of algorithms for the *n-cube* can be mapped to the $RMRN_n$ in a simple and elegant manner. We have designed and analyzed a general class of procedural primitives for the RMRN and shown how these primitives can be used as building blocks for more complex operations for computer vision. The RMRN is shown to be a highly scalable architecture with a $O(\log_2 N)$ diameter and requiring $O(N)$ communication links for a network size of N. We have shown that the RMRN can support major parallelization strategies such as functional parallelism (i.e., pipelining), data parallelism, and control parallelism, and various combinations thereof. The RMRN can be used in both the SIMD (Single Instruction Multiple Data) and the SPMD (Single Program Multiple Data) modes of parallelism. We have demonstrated the usefulness of the RMRN for computer vision by considering important operations in low- and intermediate-level vision such as the FFT, edge detection, and the Hough Transform. A prototype RMRN using T400 transputers as compute nodes in the RMRN was discussed. Timing results for the aforementioned low- and intermediate-level vision operations on the prototype RMRN indicate that the RMRN is a viable architecture for problems in computer vision. The RMRN results in a cost-effective, versatile and physically compact multiprocessor system amenable to VLSI implementation. Efforts are currently underway towards building the hardware for the RMRN using INMOS T9000 transputers and custom VLSI. Complex problems in intermediate- and high-level vision such as binocular stereo, surface reconstruction, image segmentation, and object recognition are also being considered.

APPENDIX
PROOF OF PROPERTY 2

Consider the Gray code G_n of 2^n elements defined as

$$G_1 = (0, 1)$$
$$G_n = (0G_{n-1}, 1rev(G_{n-1}))$$

where $0G_{n-1}$ denotes the operation whereby we write the elements of G_{n-1} and concatenate a 0 to the left of each element of G_{n-1}. Similarly, $1rev(G_{n-1})$ denotes the operation whereby we write the elements of G_{n-1} in reverse order and concatenate a 1 to the left of each element G_{n-1}.

It can be shown that $G_n(i)$ and $G_n(i + 1) \bmod N$ differ in a single bit position for all i and are hence directly connected (i.e., neighboring nodes) in the *n-cube*. That is to say, the Gray code mapping G_n defines a Hamiltonian cycle of length $N = 2^n$ in the *n-cube*. Also, let g_n denote the inverse / reverse of G_n. If $g_n(p) = g_n(q) + 1 \bmod N$, then p and q differ in a single bit position thus making p and q adjacent nodes in the *n-cube*.

Consider the mapping $M_n^i(p) = 2^i g_{n-1}(p_{n-i}) + g_i(p_i)$. In order to simplify the proof that the mapping M_n^i indeed embeds *config*($RMRN_n$, i) in the *n-cube* we propose the following lemma:

LEMMA A.1. *Let p be an n bit number and let p_i be the i higher order bits of p and p_{n-i} be the $n - i$ low order bits of p; that is, $p = p_i p_{n-i}$. Then, $M_n^i(p) \bmod 2^i = g_i(p_i)$ and $M_n^i(p) \operatorname{div} 2^i = g_{n-i}(p_{n-i})$ where*

$$M_n^i(p) = 2^i g_{n-i}(p_{n-i}) + g_i(p_i).$$

PROOF. The proof follows directly from the definition of M_n^i.

$$\begin{aligned} M_n^i(p) \bmod 2^i &= \left(2^i g_{n-i}(p_{n-i}) + g_i(p_i)\right) \bmod 2^i \\ &= \left(2^i g_{n-i}(p_{n-i})\right) \bmod 2^i + g_i(p_i) \bmod 2^i \\ &= 0 + g_i(p_i) \bmod 2^i \\ &= g_i(p_i) \end{aligned}$$

$$\begin{aligned} M_n^i(p) \operatorname{div} 2^i &= \left(2^i g_{n-i}(p_{n-i}) + g_i(p_i)\right) \operatorname{div} 2^i \\ &= \left(2^i g_{n-i}(p_{n-i})\right) \operatorname{div} 2^i + g_i(p_i) \operatorname{div} 2^i \\ &= g_{n-i}(p_i) + 0 \\ &= g_{n-i}(p_i) \end{aligned}$$

Both the equalities hold since $g_i(p_i) < 2^i$. □

We can now prove the following theorem which embodies the result we are interested in.

THEOREM A.1. *$M_n^i(p)$ embeds config($RMRN_n$, i) in the n-cube.*

PROOF. What we wish to prove is that if

$$M_n^i(p) = \left(M_n^i(q) \pm 1\right) \bmod N$$

or

$$M_n^i(p) = \left(M_n^i(q) \pm 2^i\right) \bmod N,$$

then p and q are adjacent in the *n-cube*. Recall that two processors p and q are adjacent in the cube if and only if the binary addresses p and q differ in a single bit position.

Case I: Consider the case where $p = p_i\, p_{n-i}$ and $q = q_i\, q_{n-i}$ and $M_n^i(p) = (M_n^i(q) \pm 2^i) \bmod N$. Applying Lemma A.1,

$$M_n^i(p) \bmod 2^i = g_i(p_i) \tag{7}$$

also,

$$\begin{aligned} M_n^i(p) \bmod 2^i &= \left(M_n^i(q) \pm 2^i\right) \bmod 2^i \\ &= M_n^i(q) \bmod 2^i \\ &= g_i(q_i) \end{aligned} \tag{8}$$

From (7) and (8) $g_i(q_i) = g_i(p_i)$. Therefore, $p_i = q_i$ since g_i is a bijective mapping. Thus, p and q have the same i higher order bits. Further,

$$M_n^i(p) \text{ div } 2^i = g_{n-i}(p_{n-i}) \tag{9}$$

also,

$$\begin{aligned} M_n^i(p) \text{ div } 2^i &= \left(M_n^i(q) \pm 2^i\right) \text{ div } 2^i \\ &= g_{n-i}(q_{n-i}) \pm 1 \end{aligned} \tag{10}$$

From (9) and (10) we conclude that $g_{n-i}(p_{n-i}) = g_{n-i}(q_{n-i}) \pm 1$. Hence, p_{n-i} and q_{n-i} differ in a single bit position. Since $p_i = q_i$ and p_{n-i} and q_{n-i} differ in a single bit position, we conclude that p and q differ in a single bit position.

Case II: Consider the case where $p = p_i\, p_{n-i}$ and $q = q_i\, q_{n-i}$ and $M_n^i(p) - (M_n^i(q) \pm 1) \bmod N$, i.e., processors p and q are in the same position in two different rings R_j and R_{j+1}. Note that if $p < 2^i - 1$ then

$$M_n^i(p) \bmod 2^i = g_i(p_i) \tag{11}$$

and

$$M_n^i(p_i) \bmod 2^i = \left(M_n^i(q) \pm 1\right) \bmod 2^i = g_i(q_i) \pm 1 \tag{12}$$

From (11) and (12), we get $g_i(p_i) = g_i(q_i) \pm 1$. Hence, p_i and q_i only differ in a single bit position. Further

$$M_n^i(p) \text{ div } 2^i = g_{n-i}(p_{n-i}) \tag{13}$$

and

$$M_n^i(p) \text{ div } 2^i = \left(M_n^i(q) + 1\right) \text{ div } 2^i = g_{n-i}(q) \tag{14}$$

From (13) and (14) we get $g_{n-i}(p_{n-i}) = g_{n-i}(q_{n-i})$. Thus $p_{n-i} = q_{n-i}$ since g_{n-i} is a bijective mapping. Since $p_{n-i} = q_{n-i}$ and p_i and q_i only differ in a single bit position, we conclude that p and q differ in a single bit position.

Since, in either case, the binary numbers p and q differ in a single bit position, the processors p and q are adjacent in the *n-cube*. This concludes the proof that $M_n^i(p)$ embeds *config*($RMRN_n$, *i*) in the *n-cube*. □

ACKNOWLEDGEMENTS

This work was partly supported by the University of Georgia Research Foundation through the Faculty Research Grants to Dr. Suchendra Bhandarkar (1991 and 1992) and Dr. Hamid Arabnia (1991, 1992, and 1993), and the Office of Instructional Development at the University of Georgia through the Instructional Technology Grant to Dr. Hamid Arabnia (1991). The assistance of the U.S. National Science Foundation is acknowledged (CCR-8717033 and CDA-8820544). The authors wish to thank Mr. Sarmad Abbasi for his comments and discussions on some of the algorithms and the proofs of some of the theorems in this paper. The authors wish to thank the anonymous reviewers for their insightful comments and suggestions which greatly improved the paper.

REFERENCES

[1] M.G. Albanesi, V. Cantoni, U. Cei, M. Ferreti, and M. Mosconi, "Embedding Pyramids into Mesh Arrays," *Reconfigurable SIMD Parallel Processors*, H. Li and Q. Stout, eds. , pp. 123-141. Englewood Cliffs, N.J.: Prentice Hall, 1991.

[2] H.R. Arabnia, "A Transputer-Based Reconfigurable Parallel System," *Proc. Transputer Research and Applications—NATUG 6*, pp. 153-167, 1993.

[3] H.R Arabnia and J.W. Smith, "A Reconfigurable Interconnection Network for Imaging Operations and Its Implementation Using a Multi-Stage Switching Box," *Proc. 1993 Conf. High Performance Computing: New Horizons*, pp. 349-357, Alberta, Canada, 1993.

[4] BBN Advanced Computers Inc., Cambridge, Mass., TC2000 Technical Product Summary, Nov. 1989.

[5] S.M. Bhandarkar and H.R. Arabnia, "The Multi-Ring Reconfigurable Multiprocessor Network for Computer Vision," *Proc. IEEE Int'l Workshop Computer Architectures for Machine Perception* (CAMP 93), pp. 180-190, New Orleans, Dec. 15–17, 1993.

[6] S.M. Bhandarkar, H.R. Arabnia, and J.W. Smith, "A Reconfigurable Architecture for Image Processing and Computer Vision," *Int'l J. Pattern Recognition and Artificial Intelligence*, vol. 9, no. 2, pp. 201-229, 1995.

[7] S. Chandran and L. Davis, "The Hough Transform on the Butterfly and the Ncube," Technical Report CAR-TR-226, Center for Automation Research, Univ. of Maryland, Sept. 1986.

[8] A.N. Choudhary and R. Ponnusamy, "Implementation and Evaluation of Hough Transform Algorithms on a Shared-Memory Multiprocessor," *J. Parallel and Distributed Computing*, vol. 12, pp. 178-188, 1991.

[9] H.Y.H. Chaung and C.C. Li, "A Systolic Array Processor for Straight Line Detection by Modified Hough Transform," *Proc. IEEE Workshop Computer Archictecture for Pattern Analysis and Image Database Management*, pp. 300-304, 1985.

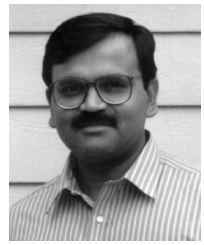
[10] R.E. Cypher, J.L.C. Sanz, and L. Snyder, "The Hough Transform has $O(N)$ Complexity on SIMD $N \times N$ Architectures," *Proc. IEEE Workshop Computer Architectures for Computer Vision and Pattern Recognition*, pp. 115-121, 1987.

[11] E. Dekel, D. Nassimi, and S. Sahni, "Parallel Matrix and Graph Algorithms," *SIAM J. Computing*, vol. 10, no. 4, pp. 657-675, Nov. 1981.

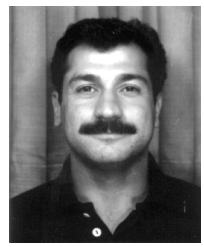
[12] Z. Fang, X. Li, and L.M. Ni, "Parallel Algorithms for Image Template Matching on Hypercube SIMD Computers," *IEEE Workshop Computer Architecture for Pattern Analysis and Image Database Management*, pp. 33-40, 1985.

[13] T.J. Fountain and V. Goetcherian, "CLIP7 Parallel Processing System," *Proc. IEE*, Part E, vol. 127, no. 5, pp. 219-224, 1980.

[14] R. Goke and G.J. Lipovski, "Banyan Networks for Partitioning on Multiprocessor Systems," *Proc. First Ann. Symp. Computer Architecture*, pp. 21-30, 1973.
[15] A. Gottleib, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolf, and M. Snir, "The NYU Ultracomputer—Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. Computers*, vol. 32, no. 2, pp. 175-189, Feb. 1986.
[16] K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing*, pp. 325-388. Mc Graw Hill, 1985.
[17] H. Ibrahim, A. Hussien, J.R. Kender, and D.E. Shaw, "The Analysis and Performance of Two Middle-level Vision Tasks for a Fine-Grained SIMD Tree Machine," *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, pp. 248-256, 1985.
[18] P.P. Jonker, "An SIMD-MIMD Architecture for Image Processing and Pattern Recognition," *Proc. IEEE Int'l Workshop Computer Architectures for Machine Perception*, New Orleans, pp. 222-231, Dec. 1993.
[19] D.J. Kuck, E.S. Davidson, D.H. Lawrie, and A.H. Sameh, "Parallel Supercomputing Today—The Cedar Approach," *Science*, vol. 231, no. 4741, pp. 967-974, Feb. 1986.
[20] V.K.P. Kumar and V. Krishnan, "Efficient Image Template Matching on Hypercube SIMD Arrays," *Proc. Int'l Conf. Parallel Processing*, pp. 765-771, 1987.
[21] D.H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Trans. Computers*, vol. 24, no. 12, pp. 1,145-1,155, Dec. 1975.
[22] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures—Arrays, Trees, Hypercubes*. San Mateo, Calif.: Morgan Kaufmann, 1992.
[23] H. Li and M. Maresca, "Polymorphic Torus Network," *IEEE Trans. Computers*, vol. 38, no. 9, pp. 1,345-1,351, Sept. 1989.
[24] T. Matsuyama, N. Asada, and M. Aoyama, "Parallel Image Analysis on a Recursive Torus," *Proc. IEEE Int'l Workshop Computer Architectures for Machine Perception*, pp. 202-214, New Orleans, Dec. 1993.
[25] C. Mead and L. Conway, *VLSI Systems Design*. Addison-Wesley, 1980.
[26] R. Miller, V.K. Prasanna Kumar, D. Reisis, and Q.F. Stout, "Image Computations on Reconfigurable VLSI Arrays," *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, pp. 925-930, 1988.
[27] R. Miller, V.K. Prasanna Kumar, D. Reisis, and Q.F. Stout, "Parallel Computations on Reconfigurable Meshes," *IEEE Trans. Computers*, vol. 42, no. 6, pp. 678-692, June 1993.
[28] J.H. Patel, "Performance of Processor–Memory Interconnections for Multiprocessors," *IEEE Trans. Computers*, vol. 27, no. 10, pp. 771-780, Oct. 1981.
[29] M.C. Pease, "The Indirect Binary n-Cube Microprocessor Array," *IEEE Trans. Computers*, vol. 25, no. 5, pp. 458-473, May 1977.
[30] G.F. Pfister and V.A. Norton, "Hot Spot Contention and Combining in Multistage Interconnection Networks," *Proc. Int'l Conf. Parallel Processing*, pp. 790-797, Aug. 1985.
[31] S. Ranka and S. Sahni, "Image Template Matching on SIMD Hypercube Multicomputers," *Proc. Int'l Conf. Parallel Processing*, pp. 84-91, 1988.
[32] S. Ranka and S. Sahni, "Odd Even Shifts in SIMD Hypercubes," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, no. 1, pp. 77-76, Jan. 1990.
[33] S. Ranka and S. Sahni, "Computing Hough Transforms on Hypercube Multicomputers," *J. Supercomputing*, vol. 4, pp. 169-190, 1990.
[34] A. Rosenfeld, J. Ornelas, and Y. Hung, "Hough Transform Algorithms for Mesh Connected SIMD Parallel Processors," *Computer Vision Graphics and Image Processing*, vol. 41, no. 3, pp. 293-305, 1988.
[35] J.M. Jolion and A. Rosenfeld, "A $O(\log N)$ Pyramid Hough Transform," *Pattern Recognition Letters*, vol. 9, pp. 343–349, 1989.
[36] D.B. Shu and J.G. Nash, "The Gated Interconnection Network for Dynamic Programming," *Concurrent Computations*, S.K. Tewsburg et al., eds. , pp. 645-658. Plenum, New York, 1988.
[37] H.S. Stone, *High Performance Computer Architecture*. Addison-Wesley, 1990.
[38] Texas Instruments, *TTL Logic Data Book*. Texas Instruments, 1988.
[39] C.L. Wu and T.Y. Feng, "On a Class of Multistage Interconnection Networks," *IEEE Trans. Computers*, vol. 26, no. 8, pp. 696-702, Aug. 1980.



Suchendra M. Bhandarkar received a BTech in electrical engineering from the Indian Institute of Technology, Bombay, India, in 1983, and MS and PhD degrees in computer engineering from Syracuse University, Syracuse, New York, in 1985 and 1989, respectively. He is currently an associate professor in the Department of Computer Science at the University of Georgia, Athens, Georgia. He was a Syracuse University Fellow for the academic years 1986-1987 and 1987-1988. He is a member of the IEEE, AAAI, ACM, and SPIE, and the honor societies Phi Kappa Phi and Phi Beta Delta. He is a coauthor of the book *3-D Object Recognition from Range Images* (Springer Verlag, 1992). His research interests include computer vision, pattern recognition, image processing, artificial intelligence, and parallel algorithms and architectures for computer vision and pattern recognition. He has published more than 50 research articles in these areas including more than 25 articles in refereed archival journals.



Hamid R. Arabnia received a BSc Honors degree in mathematics and computing in 1983 from the Polytechnic of Wales, Pontypridd, United Kingdom, and a PhD degree in computer science from the University of Kent, Canterbury, England, in 1987. For nine months in 1987, he worked as a computer science consultant for Caplin Cybernetics Corporation, London, England, where he helped in the design and implementation of a number of image processing algorithms that were targeted at a transputer-based machine. Dr. Arabnia is currently an associate professor of computer science at the University of Georgia, Athens, Georgia, where he has been since 1987. His research interests include parallel and distributed algorithms, reconfigurable networks, and application of parallel processing in remote sensing and medical imaging. He has published more than 40 technical papers in these areas. Dr. Arabnia has chaired a number of international conferences, including the 1995 and 1996 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '95) and PDPTA '96).