Routing Techniques for Massively Parallel Communication

SERGIO A. FELPERIN, LUIS GRAVANO, GUSTAVO D. PIFARRÉ, AND JORGE L.C. SANZ, FELLOW, IEEE

Invited Paper

In this paper, a survey of some packet-switched routing methods for massively parallel computers is presented. Some of the techniques are applicable to both shared-memory and message-passing architectures. These routing methods are compared in terms of their efficiency in mapping to parallel machines, network delays and interconnection topologies, deadlock and livelock freedom, and adaptivity to network congestion.

I. Introduction

A *parallel computer* or *multiprocessor* is a network of processors, coprocessors, memory banks, switches, and links. The topology of the network is described by a directed graph G. The arcs of G are called (external) *links*. Each node of G contains one or more *switches* that connect incoming links to outgoing links (see Section II for details). As it shall be seen later, switches can be either very simple or quite elaborate. For example, they may have buffers (input and/or output queues) and an *arithmetic and logic unit*. Some nodes of G (possibly all) contain processors, and some contain memory banks.

The graph G describes the logical structure of the interconnection network used for machine-wide interprocessor communication. Each external link consists of k wires (typical values for k are 1, 4, or 8) and is used for sending messages between two switches residing at distinct nodes, k bits at a time. Processor-to-processor and/or processor-to-memory communication is accomplished by sending messages that are routed between the source and destination nodes across links in the network, often passing through multiple switches on the way. It is assumed that special coprocessors at each switch take care of interprocessor communications: both programmers and processors are relieved of the burden of routing messages through the network, including handling the intermediate hops of messages traveling to their destination. A node is not restricted to contain a single processor or memory bank. For example, a node can contain several processors that share a common coprocessor for their machine-wide communication. As this work focuses on *routing* over the network, only the case of each node having one memory bank and one processor will be considered here.

A network is said to be *fully populated* if there is at least one processor at every node. Otherwise it is *partially populated*. A *multistage network* is one in which the nodes can be partitioned into *stages* (or *columns*), and all links from a node go to nodes which are either in the next stage or in the previous one (with possible wrap-around). This classification has been suggested in [1].

A *constant-degree* network is a network for which the *degree* of each node (i.e., the number of links touching it) is a constant (i.e., it is not dependent on the size of the network). It is good for networks to be constant-degree, to allow them to be extended. In addition, from a technological point of view, a feasible network can only have a small number of links per node.

Either each processor fetches its own instructions from memory (a *MIMD* machine) or all processors execute the same instruction, broadcast from a central control processor (a *SIMD* machine). Examples of MIMD machines include the IBM RP3 [2], the Intel iPCS [3],[4], Cedar [5], and the BBN Butterfly [6],[7] . The Connection Machine [8] is an example of an SIMD machine.

Parallel computers can be distinguished by whether they support a shared memory or a message passing model of processor communication. A *shared memory parallel computer* is one in which the processors communicate by reading and writing data words to a global memory. The "messages" sent from a processor to a memory location (and back) are short and simple, and the machine is

S.A. Felperin, L. Gravano, and G.D. Pifarré are with the Computer Research and Advanced Applications Group, IBM Argentina, 1300 Buenos Aires, Argentina and Escuela Superior Latino Americana de Informática (ESLAI), CC 3193, (1000), Buenos Aires, Argentina.

J.L.C. Sanz is with the Computer Science Department, IBM Almaden Research Center, San José, CA 95120 and with the Computer Research and Advanced Applications Group, IBM Argentina, Ing. E. Butti 275, (1300), Buenos Aires, Argentina.

IEEE Log Number 9142815.

0018–9219/91/1000–0488$01.00 © 1991 IEEE

streamlined accordingly.

The *Parallel Random Access Machine* (PRAM) is one of the most popular programming models for parallel, shared-memory machines. In this model, the address space is global and accessible by all the processors. There exist many variants of this model, depending on whether processors are allowed or not to perform concurrent accesses to a single memory location. The most common versions are *Exclusive-Read Exclusive-Write* (EREW), *Concurrent-Read Exclusive-Write* (CREW), and *Concurrent-Read Concurrent-Write* (CRCW) PRAM's.

In the synchronous PRAM model, a powerful primitive, the *multiprefix* (*MP*) operation can be defined. An efficient implementation of *MP* is a powerful programming tool. See [9] for the definition of *MP* and for some programming examples on its use.

In a *message passing parallel computer*, the processors communicate by passing messages that can have rich semantics (such as in an object-oriented programming style). The messages are viewed as sent between processors, and they can be long and invoke complicated actions (beyond simply loading and storing a data word).

When the processors want to communicate, they inject their messages in the network. If all of them do so at the same moment, with the network clear of messages, the injection model is *static*. It is well suited for SIMD machines. Otherwise, if every processor can inject messages at an arbitrary time, the injection model is *dynamic*.

Given a network, a distance measure D can be defined on it. A routing algorithm is said to be *minimal* if for every sequence of nodes $a_0, \ldots, a_k$ such that they conform a feasible path from a_0 to a_k, it holds that $D(a_i, a_k) > D(a_j, a_k)$ if $i < j$, i.e., every hop brings the message closer to its destination.

A routing technique is *adaptive* if for some pair of nodes a, b it can use more than a path when routing messages from a to b. Note that not only must these paths exist physically, but the routing technique must be able to make use of them. The choice of the path to be taken by a particular message may depend on many factors, e.g., faulty links or congestion of the network.

A routing algorithm such that the route a message takes depends *only* on its source and destination nodes is said to be *oblivious*.

Usually, two kinds of routing techniques are defined.

- In *packet-switching* routing, the messages are of constant size, and they are called *packets*. In this kind of routing, packets are moved from node to node in every hop of their path, and they are stored either in the nodes or in the links.
- The technique just defined does not work if the messages are of variable size. For this kind of messages *wormhole* routing can be used instead. In this technique, a message m is divided into a sequence of constant-size *flits*. The first flit (the head) of the sequence must hold the destination's address because it is used to determine the path the message must take. Once a link is occupied by the head, it cannot be used for other messages until the last flit of m has left it. If the head of m discovers that the next link it has to traverse is being used, it must wait in the buffers of the link until its next link is freed. For a more detailed description, see [10].

In this paper, a survey of packet-switching routing techniques for parallel computers is presented. Although wormhole routing has importance on its own, it will not be analyzed here. Nevertheless, many of the techniques described can also be used for wormhole routing.

An extended version of this survey can be found in [11].

Interconnection network topologies are defined briefly in Section II. The techniques are analyzed following the criteria stated in Section III.

In Sections IV, V, and XI, three nonadaptive techniques are discussed.

Sections VI, VII, and VIII describe some adaptive routing techniques. Finally, Sections IX and X show some adaptive, nonminimal routing algorithms.

II. Some Topologies

In this section, some usual network topologies will be defined. They will be used in the rest of this work. A *topology* is a graph $G = (V, E)$ where $V = \{0, \ldots, \mid V \mid -1\}$. The nodes of the graph stand for the nodes of the network, and the edges stand for its links.[1] In the following, if x and p are integers, $x \parallel p$ will denote the integer whose binary representation is the same of x's except for the pth bit.

Definition 1: In an n-**hypercube** $\mid V \mid = 2^n$. E is defined to be the set

$$\{(x, x \parallel i), x \in V, i \in \{0 \ldots n-1\}\}$$

Every node in the hypercube has n links incident to it. Its diameter is $\log \mid V \mid = n$.

Definition 2: In an n-**cube-connected cycles** (n-CCC) $\mid V \mid = n2^n$. Every address in V can be thought of as a pair (c, p) where c is n bits long and p is $\lceil \log n \rceil$ bits long. E is the set

$$\{< (c,p), (c, p+1 \pmod n) >, (c,p) \in V\}$$
$$\bigcup \{< (c,p), (c \parallel p, p) >, (c,p) \in V\}$$

The n-CCC is an n-hypercube in which every node has been decomposed into a ring of n nodes, p denotes the place of the node in the cycle and c the cycle it belongs to. Its diameter is $\mathbf{O}(n)$. Every node in the CCC has three links incident to it.

Definition 3: In an n-**butterfly**, $\mid V \mid = n2^n$. Every node in V can be thought of as a pair (c, r) where r is n bits long and c is $\lceil \log n \rceil$ bits long. r stands for the node's *row* and c for its *column*. E is the set

$$\{< (c,r), (c+1 \pmod n, r) >, (c,r) \in V\}$$
$$\bigcup \{< (c,r), (c+1 \pmod n, r \parallel c) >, (c,r) \in V\}$$

[1] If the graph is *undirected* the edges should be considered as bidirectional links. In a *directed* graph they stand for unidirectional links. Unless otherwise stated, graphs will be undirected.

The n-butterfly is related to the n-hypercube in the following way: if q_r stands for the set $\{< r, c > \in V\}$, then the topology defined by the set $\{q_r, r \in \{0, \ldots, n-1\}\}$ is an n-hypercube. The diameter of the n-butterfly is $\lceil 1.5n \rceil$ and the degree of each node is 4.

Definition 4: The **n-shuffle-exchange** network is defined by taking $\mid V \mid = 2^n$. There is an edge between x and y if either $x = y \parallel 0$ (exchange link) or y is obtained by a circular, one-bit right shift of the binary representation of x (shuffle link). The diameter of the network is $2n$ and the degree of each node is 3.

The hypercube, CCC, shuffle-exchange and butterfly form the so-called **hypercube** family in [12, p. 4]. As it has been shown in their definitions, they are closely related.

Definition 5: In a two-dimensional **n-mesh,** $\mid V \mid = n^2$. Every node is considered as a pair (r, c), $r, c \in \{0, \ldots, n-1\}$. There is a link between (r_1, c_1) and (r_2, c_2) if $r_1 = r_2$ and $\mid c_1 - c_2 \mid = 1$ or $c_1 = c_2$ and $\mid r_1 - r_2 \mid = 1$. If r, c are thought of as *row* and *column* of the node respectively, this network represents a square grid where every node is connected only to its direct neighbors. An **n-torus** can be defined by changing the connections in the following way. Node (r, c) is connected to nodes $(r, c+1 \pmod n), (r, c-1 \pmod n), (r+1 \pmod n, c)$, and $(r-1 \pmod n, c)$.

These definitions may also be generalized in an obvious way to k-dimensional meshes and toruses.

The diameter of a k-dimensional n-mesh is $k(n-1)$ and in a k-dimensional n-torus it is $\lceil kn/2 \rceil$. In the meshes, the degree of every node is $2k$, the borders being a special case, whereas in the torus it is the same for all nodes.

A generalization may be made in order to consider hypercubes and bidimensional toruses as instances of the same class of topologies [10, p. 549].

Definition 6: A k-ary n-cube can be defined as follows: $\mid V \mid = k^n$. Every node of the network is identified with an n-digit base-k number. If node p is identified with number $k_{n-1}, \cdots, k_0$ then p is connected to the nodes numbered $k_{n-1}, \cdots, k_i + 1 \pmod k, \cdots, k_0, 0 \leq i \leq n-1$. If k is fixed to be 2, then the family of n-hypercubes is obtained, and if n is fixed to be 2, the family of two-dimensional k-toruses is obtained. The diameter of the network is $\lceil kn/2 \rceil$.

Definition 7: Valiant and Brebner defined the **d-way n-shuffle** in the following way for any integer d: $\mid V \mid = d^n$. There is a link between x and y if the first $n-1$ digits of the base d representation of x are equal to the last $n-1$ of the representation of y in the same base. Every node has degree d and the diameter is $\log_d \mid V \mid = n$ [13, p. 269].

A. How to Route

Many of the routing techniques that will be described in this work are built on top of an oblivious routing algorithm that sends a message to its destination following a shortest path. Here, an example on how to compute such paths is given.

A message going from a to b on an n-hypercube may follow its shortest path computing a ticket a *XOR* b. A 1 in the kth bit of the ticket means that this dimension has to be corrected. When a node receives the message, with a nonzero ticket t, it looks deterministically for a k such that the kth bit of t is set to 1. It sets that bit to 0 and sends the packet along the link corresponding to the kth dimension. Obviously, when a node receives a message with a zero ticket, it is the destination of the message.

III. Some Important Features of Routing Algorithms

The main criteria for evaluating routing techniques strategies are efficiency in supporting programming models, efficiency in mapping to parallel machines, and practicality. The criteria suggested in [1, pp. 23–26] will be partially followed in this paper, with three important additions concerning deadlock, starvation, and livelock. In this section, some items of these criteria will be commented on. Other criteria used in this paper involve synchronization overheads, fault tolerance and generality of the routing techniques, and use of randomization.

1. **Network delay** (latency): In order to be practical, each routing algorithm should have low latency to deliver the messages to their destinations. Not only should theoretical bounds be proved regarding this low latency, but also the practicality of the routing algorithm should be shown.

Theoretical bounds can be either deterministic (a bound on the number of routing cycles the packets remain in the network) or probabilistic (a bound on the probability that any message stays in the network after some number of routing cycles).

Also, the routing algorithm should behave well for a number of different communication patterns. There are two main communication patterns for static injection. The first one is routing a permutation, where each node sends one message and receives one message.[2] The second pattern is one in which one processor is requested by many others. Therefore, the so-called bank conflicts arise. This pattern is very useful in the shared memory programming models (see Section V).

In the case of dynamic injection, routing algorithms should perform well for random routing (when the destination of messages are chosen randomly) and for some fixed permutations arising in practice [14].

Borodin and Hopcroft proved [15] that every deterministic oblivious routing scheme has a worst case latency of $\Omega\left(\sqrt{N/d^3}\right)$ where d is the degree of the nodes and N is the number of nodes in the network. Note that the best latency bound potentially achievable on a given network is precisely the *diameter* of that network, i.e., the maximum distance between any pair of nodes.

2. **Deadlock freedom**: The possibility of deadlock has already been addressed in a variety of different areas. As it will be seen, guaranteeing deadlock freedom is also a problem when designing a routing technique. Every algorithm must have either some mechanism in order to

[2] A generalization of this pattern are the so-called h-permutations, where each memory bank is requested by exactly h processors, and by the partial h-permutations, where each memory bank is requested for at most h processors [13].

avoid deadlock or a way to recover from this type of situation. Recovery is an important topic but it is beyond the scope of this paper.

This paper will only focus on packet switching deadlock. In packet switching routing, the critical resources are the buffers at the nodes that are used to store the messages during their way toward their destination. Deadlock of this type will arise if and only if there exists a set of buffers occupied by messages that have not yet arrived at their destinations such that all of these messages need a buffer that belongs to the set in order to continue their way.

A description of the different deadlock situations that may arise within the context of packet switching routing, as well as a collection of techniques to avoid them, can be found in [16].

If preemption of messages from the buffers is allowed, then deadlock configurations may be avoided. There are two possible policies regarding what to do with a preempted message. Such a message can either be discarded [17] with the consequent overhead of recovering the message, or it can be derouted. The latter policy gives raise to adaptive nonminimal routing techniques. These techniques will be studied later. (See Sections IX and X.)

Other techniques can be developed if preemption of the critical resources is not allowed. In general, these techniques arise as an application of a well-known concept within the framework of Operating Systems: the definition of an ordering of the critical resources in order to avoid cyclic wait. The users ask for the resources in a strictly monotonic order. The technique developed by Günther in [16], the one in [18], and the virtual channels technique of Dally and Seitz presented in [10] should be mentioned as instances of this method.

3. **Livelock freedom**: The livelock problem arises whenever a message can be denied getting delivered to its destination forever. It should be clear that livelock cannot take place in a static injection model if the queue policy is *fair* and the routes are minimal. The first condition means that no message can be forever prevented from routing if it can be routed. All the sensible queue policies are fair, so livelock is not a problem when injection is static in minimal routing techniques, as those of Sections IV and V.

The real problem appears with dynamic injection. Imagine a routing algorithm that routes first those messages that are closest to their destination. Furthermore, suppose that node n has a message m that is at distance 2 from its destination. If n injects an infinite sequence of messages with distance 1 to their destination, m will wait forever in the queue. In nonminimal routing, livelock can also take place if a message can be derouted forever.

Different policies have been proposed in order to avoid livelock. In general, they are based upon the following.

Let P be a set of priorities which is totally ordered. Whenever a packet is injected into the network, some priority is assigned to it. It must hold that:

a) packets are routed respecting their priority;

b) once a packet has been injected, only a *finite* number of packets will be injected with a higher or equal priority.

A related policy is one that assigns every packet some minimum priority when injected and increases it as the packet remains undelivered. Both of these policies are actually the same.

These policies have been criticized because they have at least two drawbacks. The first one is the fact that priority queues must be implemented to maintain the packets ordered, and such queues are more expensive and slower than the simple FIFO queues. The second drawback is the need for more bits in the packets in order to store the priorities. It is also costly to update them.

No policy against livelock without these drawbacks is known by the authors. Reference [19] proposes a routing technique that has no priority queues but is livelock free only *probabilistically* (see Section IX for details).

4. **Starvation freedom**: A node suffers *starvation* when it has a packet to inject in the network and it is never allowed to do so. Obviously, starvation cannot arise if the model of injection is static. The way starvation is avoided is related to the injection control policy. The goal is to assure that every node can eventually inject its packets into the network. The main policies proposed in order to avoid starvation are the following.

a. **Injection competition**: Every node has an injection queue, where it stores the messages it wants to inject into the network. This queue is considered in the same way as the queues of the incoming links to that node, and it competes with them. As the queue management is fair, this method avoids starvation. See [20] and Section VI for an instance of this technique. Its main advantage is also its main drawback: its simplicity. This is so because a node with a high injection rate can slow down all the others in the network.

b. **Private-buffer recirculation**: This policy allows every node to reserve some fraction of its own internal message queue for injection. Each packet must have a mark saying if it was injected in the private fraction or in the public one. When a packet injected in the internal fraction of its sender is delivered to its final destination, the receiver uses the buffer used by the delivered packet to inject a *null* message destined to the sender. When a node needs to inject a message in the network, it looks for a free buffer in its queue. If it finds one, the packet is injected there. Otherwise, the node must wait for the return of some *null* packet sent to it, and then, it uses that buffer to inject its packet [21], [22]. This policy has been censored in [19] because it makes the number of circulating messages with packets that have no information too large.

c. **Injection-token recirculation**: This policy has been proposed in [19] as an alternative to the policy described above. A function $Next : V \rightarrow V$ must be built defining a cycle involving all the nodes of the network. There exists a number of *token* packets in the network, with null content and a destination. When a token packet arrives at its destination node n, this node determines whether it has been prevented from injecting for longer than a fixed quantum

q_n. If it has and it wants to inject, then it uses the buffer occupied by the token to inject its packet. Otherwise, it sends the token packet to $Next(n)$. Whenever a packet arrives at its final destination, the receiver must check if the message has been injected over a token packet. If it has, then the receiver must regenerate the token packet, with destination equal to the *Next* node of the packet's sender.

This policy can be generalized letting *Next* define many disjoint cycles covering all the nodes of the network instead of just one with at least one token packet in every cycle.[3]

Finally, it should be noted that Private-Buffer Recirculation can be thought of as a particular instance of this policy, when *Next* is the identity permutation and $q_n = 0$ for all n in the network.

d. **Packet-injection control** [21], [22]: It is based on putting a bound to the difference between a node's injection rate and that of its neighbors. This is done via some kind of information exchanged between a node and its neighbors. A node is allowed to inject if the difference between the number of packets it has injected itself and the number of packets injected by its neighbors is greater than $-k$ for all its neighbors, for some fixed k. This policy is more adaptive to changing injection rates than the previous ones and does not have the drawback of augmenting the number of circulating packets in the network. As a disadvantage, it is complex and its implementation may be expensive.

IV. Random Routing

A great part of the delay in communications networks is due to the fact that there are conflicts among the packets for the use of the network's resources. The number of conflicts increases if the communication pattern is highly structured, e.g. transposing messages (i.e. sending a message from every node p_{ij} to p_{ji}) will involve a high number of conflicts over the diagonal of a two-dimensional mesh. To avoid this kind of structural clashes, Valiant and Brebner developed and studied random routing [13], [12]. Shortly, this is their scheme. Let $\mathcal{N}$ be a network with N nodes and an oblivious routing scheme. Every link between nodes has a queue where it stores its messages. As the routing is oblivious, for every pair of nodes a and b a ticket τ_{ab} is computed with the route that should be traversed when going from a to b, e.g., τ_{ab} may be a list containing the path that should be traversed. The initialization consists of the choice of an intermediate random destination q for every packet p and the computation of $\tau_{aq}(p)$ and $\tau_{qb}(p)$. It is followed by two phases: in the first phase (Phase A) the message is sent from a to q. In the second phase (Phase B), from q to b. The random choice of q allows to study the overall behavior of the network probabilistically, assuming no structure in the communication pattern.

It should be defined how to pass from Phase A to Phase B and two policies have been suggested. The first one separates completely the phases: at the end of its Phase A, every packet is delivered to its intermediate destination q, where it waits for the beginning of Phase B. Phase B starts when *all* packets have finished Phase A. The second policy allows every packet to begin its Phase B immediately after it has finished its Phase A route. This can be achieved by using just one ticket $\tau(p) = \tau_{aq}(p).\tau_{qb}(p)$, where the dot means concatenation. If the choice of q is made with uniform probability, then the order of the delay does not depend on the way in which the phase is changed. Reference [23] simulated a variety of different techniques for computing τ on various structured communications patterns over a hypercube using the latter policy and compared their results with those of random traffic patterns with good results.[4]

A. Analysis of the Method

1. **Network delay**: Let D be the random variable that measures the number of routing cycles it takes the last message to end either phase. Then, in [12] it has been proved that there exist α_0, β, $\delta > 0$ such that for all $\gamma > 0$ and $\alpha > \alpha_0$

$$P(D > \alpha\gamma n) \leq N^{-\alpha\beta\gamma+\delta} \tag{1}$$

either for routing γ-permutations on n-butterflies and n-CCC or for routing γn-permutations on n-hypercubes.[5] This result is stronger than the one presented in [13] because the routing model is weaker and in [12] good time bounds are proved even for some constant-degree networks. The fact that routing γ-permutations in the constant-degree topologies cited above and routing γn-permutations on hypercubes have related time bounds should be obvious because of the relationship stated among these topologies in Section II. When routing on a two-dimensional n-mesh, direct analysis has been used in [13] to prove that there exists some $C < 1$ such that for all $K > 1$

$$P(D > 3n + 2Kn^{\frac{3}{4}}) < C^{Kn^{\frac{1}{2}}}$$

This is a good result since it is $\mathbf{O}(n)$ which is the diameter of the network. Also, the experimental results given in [13] support the claim that delays are proportional to the diameter of the network for some constant-degree networks such as shuffle-exchange, CCC and l-way-shuffles, for small values of l.

Upfal [24] developed a similar random routing that was the first technique to achieve the optimal asymptotical latency for some constant-degree networks.

Random routing has been criticized because it doubles the expected path length [19]. This criticism is right, but the reader should note that it has been proved in [25] that for the d-way shuffle graphs, all oblivious algorithms that realize permutations with optimal latency must send packets along routes doubling the network diameter. The simulations of [23] on an n-hypercube showed that even if fewer random bits are chosen, network congestion is reduced.

2. **Synchronization overheads:** No mechanism is provided by the routing scheme for processor synchronization.

[3] This means that *Next* needs to be only a permutation of the nodes' set.

[4] The simulations considered that every processor sends out only one message for transposing and complementing patterns.

[5] In the first case, $N = n2^n$ and in the second one $N = 2^n$.

In an SIMD environment, the controller would detect the termination of the routing.

3. **Fault tolerance:** This routing scheme does not include any analysis of fault tolerance. Since the underlying routing algorithm must be oblivious (called *greedy* in [12]) the routing strategy is useless in a dynamic fault model. A point that requires further development is the way of dealing with a static fault model.

4. **Generality:** This method has been shown useful over a great variety of topologies. The programming model of this scheme supposes that all the communications begin in the same moment with the network clear of messages. This is best suited for SIMD computers.

5. **Use of randomization:** The method uses randomization in an obvious way when q is selected. In [23], randomization is also used in the order in which dimensions are corrected when routing on an n-hypercube, with good results.

6. **Deadlock freedom:** Deadlock freedom is not proved in general for any network with bounded-size queues. Reference [13] experimentally showed that on an n-hypercube a queue size of $\mathbf{O}(n)$ avoided deadlock. No proof has been presented of constant-size queues for avoiding deterministically deadlock. Pippenger [26] devised a random routing scheme with constant-size queue that avoids deadlock with high probability.

7. **Livelock freedom:** The routing in every phase is minimal, so there is no livelock if the queue policy is fair. In [12], this method is shown invariant under many queuing policies such as priority queues, FIFO, and LIFO.

8. **Starvation freedom:** Since this method has static injection, no starvation is possible.

V. Routing Based on Combining Messages

A. The Fluent Machine

In this section, the Fluent Machine, presented in [9] and [27], will be described. It is a machine model that can be classified as a "combining with holding" model, designed with the goal of supporting efficiently the CRCW PRAM's. In the model, all processors synchronize at each communication step. The injection of messages is static. Each processor issues at most one message at each communication step. References [9] and [27] will be followed in the rest of this description.

This routing technique implements efficiently the multiprefix primitive on a fully-populated n-butterfly (see Section II). It is oblivious, deadlock free, and uses constant-size queues. The combination of messages is deterministic and mandatory.

The routing algorithm guarantees that all messages destined to a same memory location are combined into a single message while these requests are being routed. In one of the phases of the routing algorithm, requests directed to the same physical location will eventually meet at one node and they will be combined in that node. Only one request will continue its way to the target address.

Each node of the n-butterfly network has six routing switches. These routing switches are necessary because each physical node plays the role of six "virtual" nodes, one for each of the six phases of the routing algorithm. These phases will be described below. Each switch has 1 or 2 inputs and 1 or 2 outputs, depending on the phase it belongs to. Every input into a switch has associated with it a FIFO queue of constant size.

B. Routing Algorithm

Suppose a given processor $< c, r >$ wants to access physical location m that is in node $< c', r' >$. This request will traverse a path from its source to its destination node, and backward. This path is divided into six phases.

Phase 1: The message issued at node $< c, r >$ is directed forward to node $< 0, r >$. This phase is used to begin the routing algorithm and provides a way to start the messages flowing while keeping them sorted as will be explain below. Messages waiting for being injected in the network at a given node can be combined with messages already flowing in the network, if they are both directed to the same physical memory location.

Phase 2: The message follows the unique (forward) path in the network from node $< 0, r >$ to node $< 0, r' >$. The message has to traverse the network even though it is already in the correct row. During this phase, the combination of the different messages directed to the same address is finished. These messages will necessarily meet one another during their path to node $< 0, r' >$ because all messages with physical destination m must pass through node $< 0, r' >$ during this phase.

Phase 3: The message eventually reaches node $< c', r' >$ from node $< 0, r' >$. Then, it continues its way to $< 0, r' >$, so as to be able to start Phase 4. No combination is performed during this phase. The aim of this phase is to provide the network with full connection, allowing every node to communicate with every other node.

Phases 4 to 6: The message traverses the path of phases 1 to 3 but in the reverse direction, carrying the replies to the request back to the issuing processor ($< c, r >$ in this case).

All memory requests are assigned priorities according to their physical destination, i.e., the target memory address. Messages are kept sorted according to this priority. This is achieved by allowing a message to go out of a given node once the node knows that all the messages that may want to go through it have greater target memory addresses. This is done by keeping an input queue associated with each input channel of the node. The queues are served using a FIFO policy. The head of a given queue will be allowed to go out of the node and continue its route to its destination once the head of the other queue has been occupied by a message with greater destination address. If the head of the other queue holds a message with equal destination address, the messages at both heads are combined and the resulting message goes out of the node.

The process just outlined takes place during phases 1 and 2. Switches belonging to phase 3 and 4 have only one

input queue. Phases 6 and 5 are analogous to phases 1 and 2, respectively, except that no combination takes place.

Replies to requests take exactly the same path as the corresponding requests. Messages pass back through a node in exactly the same order as they have passed during their first traversal. Therefore, each node belonging to phases 1 and 2 only needs to store two bits of information about each message it routes. These two bits indicate if the corresponding reply will have to be routed only through the top link, only through the bottom link or through both. The latter case corresponds to the reply to two messages that have been combined into a single one in that node. These pairs of bits are stored in a FIFO queue. Information regarding the values that must be sent to each processor in order to implement the multiprefix primitive can be stored and used in a similar way.

An important point is that in order for this algorithm to work correctly, each processor must issue a message at each communication step. Moreover, every processor must issue its request followed by a special message, named End of Stream (**EOS**), that has the least priority (physical destination ∞). These **EOS** messages are necessary in order to allow all the messages to arrive at their targets.

There is yet another type of messages: **GHOST** messages.

Suppose node A is connected to node B through its upper link, and to node C through its lower link. Suppose A selects a certain message m to route through the upper link. As every node routes messages sorted with respect to their physical destination, no message with physical destination smaller than that of m will go out of A after A sends m out. So, when forwarding m to B, A can issue a **GHOST** message with the same label as m to C. In this way, C will know that no message with smaller tag will arrive through that link during the current communication phase. Therefore, C will presumably be able to forward messages waiting at its other input queue that would otherwise have to be held for more routing steps.

An important point is that node A sends a **GHOST** message with the label of m to node C even if it is not able to route message m through the upper channel during that routing step as a result of congestion. In this way, it is guaranteed that queues do not become empty from a certain instant and afterwards till an **EOS** message leaves them. **GHOST** messages are essential when proving properties regarding deadlock freedom.

C. Analysis of the Method

1. **Network delay**: There are not "good" cases: every request takes at least $4n$ communication steps to complete. Nevertheless, "bad" cases are extremely rare. Memory bank collisions are a problem. Random hashing is used to solve it, as will be analyzed below. It is proven in [27], assuming a perfect random address map, that the probability that any memory reference takes more than $15 \log N$ steps is less than N^{-20}, where $N = n2^n$. Therefore, the provable latency, which is achievable with overwhelming probability, is only slightly higher than the best case.

Simulation results show [9] that it is more efficient to perform concurrent reads and writes. Concurrent accesses are faster than exclusive ones.

2. **Synchronization overheads**: As **EOS** messages are used, every processor is able to stop the rest of the processors simply by not sending the **EOS** message corresponding to a given communication step. Furthermore, **EOS** messages enforce instruction separation, thus supporting synchronous programming models efficiently. This fact has good consequences regarding the ability to implement virtual barriers, for example.

As a drawback, every message must pay for this characteristic of the algorithm, as messages are held in the switches when being routed.

In [28], an extension of this technique is presented in order to implement synchronization barriers in general multistage networks.

3. **Fault tolerance**: The routing strategy depends heavily on the processors issuing the **EOS** messages during each communication step. If one of these messages were lost, all the algorithm would break down if no further improvements are added to it. Static node or link failures can be avoided by adding an extra column to the network and connecting it to the nth column following the pattern of connection between columns 0 and 1. By doing so, alternative paths are created. Message combination will still be guaranteed and deterministic. The return bits will ensure that messages return using the same path as in their forward phases. Some problems seem to arise regarding the multiprefix primitive. This primitive depends on the ordering of the processors and on the topology of the butterfly network.

4. **Generality**: Every processor must issue an **EOS** message at each communication phase even though it has no message to inject in the network. This need restricts the sort of programming models that can be "directly" implemented on top of the Fluent machine to synchronous communication models.

Regarding hardware needs, each node must have support for the combining of messages. Comparisons and arithmetic operations are involved in this task. The Fluent Machine seems to have been designed with the butterfly network in mind. Nothing has been said about it being transferred to other topologies in [9] and [27].

The authors of this paper are currently working on extending Ranade's technique to other topologies over which acyclic routing can be performed [29].

5. **Use of randomization**: The implementation outlined above makes use of a hash function in order to distribute the shared variables throughout the local memories of the different nodes of the network. By doing so, the algorithm is able to destroy many conflicts arising when many nodes try to access logically neighboring memory locations. By using a hash function, there will probably be considerably fewer network conflicts. The Fluent Machine router can properly handle many accesses involving a single variable by combining these requests into a single one in

a deterministic fashion. This is exactly the case that is not solved by using hash functions, and it is handled efficiently. A possible drawback of using hash functions to map the logical address space to physical locations is that it makes it impossible to exploit locality of reference. This is not a problem because of the way the routing algorithm manages requests. There are not any "good" cases: every memory access requires $\Omega(n)$ steps to reach its target. If memory addresses are allowed to be computed at run time then the hash function must be computed extremely quickly as not to delay all the routing algorithm. Otherwise, all the algorithm will become useless.

6. **Deadlock freedom**:The routing algorithm is deadlock free because the logical routing graph (i.e., the graph that results from splitting each physical node into the six logical nodes it represents) is acyclic and because of the way **GHOST** messages are handled.

7. **Livelock freedom**: The injection policy is static, i.e., the number of messages of each routing phase is finite. Although **GHOST** messages are injected dynamically, only a finite number of them are generated at each routing phase because of the characteristics of the network and the routing algorithm. Moreover, even though nodes are allowed to pipeline their consecutive instructions, thus allowing each node to inject messages in a "pseudo continuous" way, no message from a later routing phase will ever delay or compete with a message from an earlier phase. Therefore, this algorithm is livelock free.

8. **Starvation freedom**: The injection of messages is performed in phases. Each of these phases is separated from the others by a wave of **EOS** messages. So, as each processor is only allowed to inject a finite number of messages in each phase, starvation is not a problem at all.

VI. Adaptive, Minimal Routing

In this section, a minimal adaptive packet-switched routing algorithm for the n-hypercube is described (see Section II). This algorithm is deadlock, livelock, and starvation free. In addition, it requires only a constant number of bounded length queues at every node and it allows nodes to inject messages continuously. This algorithm is presented in [20]. A related routing technique resulting from "hanging" the hypercube by its node 0 was independently presented in [28].

A. Routing Algorithm

1) Definitions: Let M be a message with source node S and destination node D. Then, $Zenith(M) = (S\ OR\ D)$. If P is a node of the network, then $Nadir(M, P) = (P\ AND\ D)$.

2) Informal description of the routing algorithm: Messages are divided into two different classes: Class 1 and Class 2 messages.

Both Class 1 and Class 2 messages follow a minimal path from their source to their destination. Class 1 messages first turn the "incorrect" zero bits into one bits (ascending phase) and then turn the "incorrect" one bits into zero bits (descending phase). Class 2 messages do exactly the converse process.

To guarantee deadlock freedom, this algorithm requires that each node have three queues: Queue 0, Queue 1 and Queue 2. Queue 0 holds only Class 1 messages during their ascending phase. Queue 1 holds Class 1 and Class 2 messages during their descending phase and Queue 2 holds Class 2 messages during their ascending phase. In principle, every queue may have only constant size. The size of the queues does not affect the deadlock freedom of the algorithm. As will be seen, it does affect the latency of the messages.

Every message M is injected into the network as a Class 1 member in Queue 0 of the source node. It then starts moving from node to node through Queue 0 of the visited nodes. When message M arrives at $Zenith(M)$, it is moved from Queue 0 to Queue 1 of $Zenith(M)$ and starts its descending phase moving through Queue 1 of the visited nodes. If at a certain node P, message M is at its ascending phase (i.e., climbing up to $Zenith(M)$) and it cannot make progress through any of the possible dimensions due to congestion problems, and if Queue 1 of that node has free space, then M is moved from Queue 0 to Queue 1 of node P. Therefore, M turns into a Class 2 message and starts its descending phase to $Nadir(M, P)$. Once it has arrived at $Nadir(M, P)$, it is moved from Queue 1 to Queue 2 of $Nadir(M, P)$ and it starts its ascending phase until it reaches the destination node D. Notice that a message can only change from Class 1 to Class 2, and only if it is performing the ascending phase of Class 1, i.e. if it has not reached its $Zenith$ yet. Once it has reached its $Zenith$, it must remain a Class 1 message until it arrives at its destination node.

There are two ways in which congestion can influence a message's route toward its destination. First, during any of the phases of the algorithm, each message M has, at a given node P, a set of dimensions S that can be corrected by means of the next hop. P can route M along any of the links corresponding to the dimensions in S. Second, if a message M is a Class 1 message during its ascending phase, it can switch from Class 1 to Class 2, as explained above.

Therefore, one message can change class only at a certain phase, and only once. At any other moment, it will have to adapt to congestion only by means of the first way described above.

B. Implementation Issues

In order to avoid communication delays, especially when implementing an asynchronous communication model, each node should have buffers associated with each of its n channels.

The implementation suggested in [20] requires $3n + 2$ buffers plus $3\,\mathbf{O}(1)$ queues in each node. In addition, each node requires an injection buffer (where new messages waiting for place in Queue 0 are injected) and a delivery buffer (where messages that have arrived at their destination are kept stored until they are consumed).

C. Analysis of the Method

1. **Network delay**: Notice that hot-spots are likely to arise around node 0 and node $2^n - 1$. The reason for this congestion is that all messages start routing as Class 1 messages, thus traveling toward node $2^n - 1$ when seeking to reach their *Zenith*. If a message is stopped by congestion before it has reached its *Zenith*, it starts heading to its *Nadir*, thus moving toward node 0. By making some messages turn into Class 2 messages, contention around node $2^n - 1$ is partially alleviated, while congesting node 0 and its surroundings.

Although congestion can be partly relieved by the possibility of randomly choosing which dimension to correct, as pointed out above, no message can avoid moving to either of these two potential hot-spots. Some of the messages will have their "particular" hot-spot changed: they will head toward 0 instead of $2^n - 1$.

There are no latency results proven in [20]. Instead, some simulation results are presented. These results assume a synchronous communication model. One of the most interesting points mentioned is that the average latency of the messages is not a monotonically decreasing function of the queue size. The explanation given to this phenomenon is that by increasing the queue size farther from the optimal size, (as determined by the experiments) more messages will be allowed to arrive at the neighborhood of node $2^n - 1$ before being switched from Class 1 to Class 2. Therefore, smaller queues will switch messages from Class 1 to Class 2 before, thus making the congestion around node $2^n - 1$ remain relatively smaller.

2. **Synchronization overheads**: Nothing is said in [20] about the implementation of virtual barriers, for instance. Concurrent accesses to locks, for example, are not facilitated by the algorithm either.

Nevertheless, machine-wide barrier synchronization may be implemented following the technique described in [28]. The routing algorithm on top of which this synchronization technique is implemented is very similar to the one in [20], except that there is no changing of messages from Class 1 to Class 2. All messages are Class 1 members "forever". The important point of this sort of algorithms is that routing is monotonic with respect to the node numbers. Consequently, barriers can be efficiently implemented as a set of waves that sweep the network [28].

3. **Fault tolerance**: It is claimed that adaptive routing techniques are more fault tolerant than oblivious ones. In this case, a damaged link can be viewed as a link that remains busy forever. So, in such a case, a Class 1 message in its ascending phase may be turned into a Class 2 message and start its descending phase. A message in Class 2 or in the descending phase of Class 1 cannot change class. So, it will be allowed to go through a nonfaulty link as long as there are other links among the set from which it may choose. Otherwise, the message will not be able to reach its destination. A similar reasoning follows for faulty nodes. The scheme outlined above will work for both dynamic and static faults.

In summary, this algorithm is more fault tolerant than an oblivious routing but not completely fault tolerant.

4. **Generality**: This routing algorithm has been presented as designed specifically with the hypercube network in mind. One of the main features hypercube networks have is that minimal paths from any node to any other node can be determined directly from the nodes' indexes. This is an important property when allowing a message to change its route adaptively because it is cheap to recalculate the path it must follow. As virtual barriers might be efficiently implemented on top of this technique, as mentioned above, this scheme may support semisynchronous programming models as well as asynchronous ones.

Recently, the authors have extended this hypercube method to other networks and have also generalized the routing, thus widening the range of applicability of the techniques. In addition, the algorithms developed do not have serious hot-spots as in this technique [27].

5. **Use of randomization**: As mentioned above, at any step, a given message can choose arbitrarily what dimension to correct among a given set. This free choice may be regarded as allowing some sort of randomization within the algorithm.

6. **Deadlock freedom**: This algorithm is deadlock free. The proof given in [20] numbers all the queues in the network in such a way that all messages occupy queues in a strictly ascending order with respect to the queues' indexes.

The minimal adaptive routing routing shown in [20] may be regarded as what results from routing as if the hypercube were "hanged" from node 0 and messages must pass two or three times through the network. Each of these passes corresponds to a phase within a Class of the routing algorithm described above. Each phase will use, according to the terminology used in [10] , different "sets" of virtual channels. In this way, the channel dependency graph [10] results acyclic, thus avoiding deadlocks. So, each physical link is considered by the algorithm as three different virtual channels, and the use of the link by these channels is multiplexed in time.

7. **Livelock freedom**: This is a minimal routing algorithm. So, as long as all shared resources are assigned in a fair way to their users, no livelock may arise. Channels and queues are the shared resources. For example, Queue 0 is accessed by messages waiting for being injected in the network and by Class 1 messages in their ascending phase.

8. **Starvation freedom**: Starvation freedom of the algorithm is guaranteed by the fairness with which shared resources are assigned to their users. Messages waiting for being injected compete with Class 1 messages in their ascending phase in order to get access to Queue 0. So, as long as Queue 0 is administered fairly, no starvation can possibly arise.

VII. Multibutterflies

In this section, a new topology and a routing algorithm, presented in [37], will be described. This topology, dubbed *multibutterfly*, which is based upon the *n*-butterfly (see

Section II), has constant degree. Furthermore, the routing algorithm presented in [37] can route *any* permutation on an $N = (n+1)2^n$ processor multibutterfly-based network jusing constant-sized buffers in time O(log N). It should be emphasized that this result is *deterministic*, i.e., any permutation is guaranteed to finish being routed in time proportional to log N.

Next, the multibutterfly topology as described in [37] and [38] will be depicted. A (d, n)-*multibutterfly* is formed by merging together d n-butterflies in a special way. In the resulting network, there will be O(d^n) paths between any node in the first column and any node in the last column of the network, instead of just one as with the n-butterfly topology. Nevertheless, the *logical path* [38] taken by a message from the first stage to the last one is unique, and the same one as in the n-butterfly. This logical path will be a sequence of n steps. Each of these steps will be associated to fixing one bit of the address of the destination's row, and will be either an *up*-step, if the corresponding bit in the address of the destination's row is a 0, or a *down*-step, otherwise.

The d butterflies that form the multibutterfly will be superimposed in such a way that each of these logical steps will have d different physical links to be realized, regardless of the step's being an up or down step. Each message will decide which of these links it will take at each step adaptively depending on local congestion. So, in the resulting network, each node will have both indegree and outdegree $2d$. There will be d outgoing up-links and d outgoing down-links incident to each node of the network. In addition, the way the d butterflies are merged should be such that the connections between the different stages observe a so-called *expansion property* [37], [38] that is essential to obtain deterministic logarithmic performance.

Up until now in this section, the multibutterfly network has been regarded as a partially populated multistage network in which processors in the first stage send messages to the last stage. In [37], a topology based upon the multibutterfly has been presented. This is a constant-degree fully populated network of N processors that can still route any permutation in O(log N) time in a deterministic sense.

A. Analysis of the Method

1. **Network delay**: As pointed out previously, the routing algorithm presented in [37] can route any N permutation in O(log N) steps in a deterministic sense. Nevertheless, the constants hidden by the O notation are big. These constants have been lowered in [38] for the partially populated multibutterfly, but are still high for small values of d. Experimental results showing good performance of this partially populated network even in the presence of faulty switches have been presented in [39].

2. **Synchronization overheads**: Nothing has been said in either [37] or [38] regarding the implementation of virtual barriers. Even so, the methodology presented in [28] may be combined with this routing algorithm to implement virtual barriers efficiently.

3. **Fault tolerance**: In [38], the partially populated multibutterfly network and the greedy routing algorithm over it have been shown to tolerate a reasonable amount of switch faults without significant performance degradation. Specifically, for $d \geq 5$, it has been proved that if *any* k switches are faulty in a (d, n)-multibutterfly, then there are always at least $2^n - O(k)$ (input) nodes in the first stage and $2^n - O(k)$ (output) nodes in the last stage through which any permutation on these inputs and outputs can be routed in O(n) steps. The fault model is static [38]. Furthermore, if the faulty switches are randomly located, then the (d, n)-multibutterfly will tolerate $2^{\gamma n}$ random "interior" switch faults ($\gamma \geq 1/2$ constant) without losing any inputs or outputs, with high probability [38]. Experimental results on performance of this partially populated multistage network in the presence of faulty nodes can be found in [39].

4. **Generality**: These routing algorithms have been designed, focused on one topology with very special properties.

5. **Use of randomization**: Randomization can be used to generate the d n-butterflies to form a (d, n)-multibutterfly with the desired properties with high probability.

6. **Deadlock, livelock, and starvation freedom**: Because of the characteristics of the algorithm, these are not problems at all.

VIII. Fully-Adaptive Routing

As it has been said in Section III, a usual technique to define deadlock-free routing functions consists of ordering the critical resources and defining the function in such a way that messages use these critical resources strictly monotonically according to this ordering. In this way, cyclic wait is avoided, and so, deadlock can not arise involving these critical resources. This technique has been widely studied and used [16], [18], [10], and can be relaxed for packet-switching routing to adopt a *dynamic* strategy to avoid deadlock [18]. This idea has been applied [40] to develop adaptive routing algorithms for packet-switching routing on a variety of networks, and with very moderate resources. This technique requires the definition of an acyclic Queue Dependency Graph representing the use of the critical resources (queues in this case), as in the static technique presented in [10]. After this, new *dependencies* are allowed to be established between the different queues. These dependencies cause static cycles to appear in this Queue Dependency Graph. A message will be allowed to take one of these new dependencies provided that from the queue to which it arrives there is still a path towards the destination of the message in the acyclic Queue Dependency Graph. In this way, every message can always follow a path to its destination following the acyclic Queue Dependency Graph built first, and so, deadlock situations are avoided.

While the new techniques apply to a wide variety of networks, routing algorithms have been shown for the hypercube, the two-dimensional mesh, and the shuffle-exchange in [40]. The techniques presented for hypercubes

and meshes are fully adaptive and minimal. A fully adaptive and minimal routing is one in which *all* possible minimal paths between a source and a destination are of potential use at the time a message is injected into the network. Minimal paths followed by messages ultimately depend on the local congestion encountered in each node of the network. The shuffle-exchange algorithm is the first adaptive and deadlock-free method that requires a small (and independent of N) number of buffers and queues in the routing nodes for that network.

In contrast to other approaches in which adaptivity, deadlock and livelock freedom can be guaranteed at the expense of complex architectures, the algorithms presented in [40] require a very moderate amount of routing hardware. Only two central queues per routing node of the network are necessary for the cases of the two-dimensional mesh and the hypercube, and four queues for the shuffle-exchange.

A. Analysis of the Method

1. **Network delay**: Neither deterministic nor probabilistic bounds have been proven regarding latency for these algorithms. Some situations have been reported in [40] and [14]. These simulations show good results for the hypercube and the mesh networks. For the mesh, the routing algorithm outperforms oblivious and partially adaptive algorithms when dynamic injection is considered. The fully adaptive routing algorithm allows the injection rate to extend without saturating the network.
2. **Synchronization overheads**: There is no simple way to implement virtual barriers applying the technique presented in [28].
3. **Fault tolerance**: Nothing has been analyzed in [40] regarding fault-tolerance.
4. **Generality**: This technique can be applied to any topology, but only for packet switching routing.
5. **Use of randomization**: The choice of which path message eventually takes among all the possible ones depends on congestion. Randomization could be introduced to choose between all the possible paths available at a given instant.
6. **Deadlock freedom**: The algorithms are deadlock-free, as explained above.
7. **Livelock and starvation freedom**: The algorithms are minimal, and so, if the critical resources are handled with fairness, livelock and starvation are not a problem.

IX. The Chaos Router

In this section, the Chaos Router introduced in [19], is described. It is an asynchronous, nonminimal, adaptive, packet-switched routing technique for the n-hypercube (see Section II). This routing algorithm is related to one of the variants presented in [22]. As it is a nonminimal adaptive routing, messages can be sent farther from their destination as a result of local congestion problems. Therefore, livelock arises as a problem. Another thing is that the Chaos Router allows nodes to inject messages continuously. It requires $O(n)$ buffers of size 1 and only one queue at each node. In this section, [19] will be followed.

A. Main Features of the Algorithm

One of the main aims of the designers of this algorithm was to simplify the management of the queues of messages at each node. So, the resulting algorithm is only probabilistically livelock free. No complex mechanism to guarantee eventual delivery of the messages (e.g., aging of the messages) is provided. When derouting is necessary, the message to be derouted is selected randomly among the messages in a given set. Another important point is that the router is fully asynchronous. All routers are independent of one another.

B. Hardware Requirements

Every node must have an input and an output buffer per channel, called the `Input Frame` and `Output Frame`, respectively. Each of these buffers can hold only one message. In addition, every node must have an injection and a delivery buffer, as well as a queue. In order to be able to select what message to deroute in a random fashion, every node needs also a source of randomness.

C. Routing Algorithm

The routing algorithm presented in [19] is as follows.

```
current_dim=0;
while (True) do
begin
  current_dim=(current_dim+1)   (mod n);
  while (Full(Output Frame(current_dim)))
  begin
    current_dim=(current_dim+1,   (mod n);
  end
  Match(current_dim);
  if (Full(Queue) AND no match AND
  Full(InputFrame(current_dim)))
  begin
    Deroute;
  end
  Send(current_dim);
  Read(current_dim);
end.
```

`Match`, `Deroute`, `Send`, and `Read` perform the following tasks.

- `Match` selects the first—in FIFO order—message of the queue that can be routed along the current dimension, if such a message exists.
- `Deroute` randomly selects a message from the queue. This is the message that will be derouted by the algorithm.
- `Send` removes from the queue the message that has been selected by the previous `Match` or `Deroute` operation, and puts it in the `Output Frame` of the current dimension.
- `Read` processes the message at the `Input Frame` of the current dimension, if there is one. If the message has already arrived at its destination, it is delivered. Otherwise,

it is added at the end of the queue, in FIFO order. This operation will take a message from the injection buffer (if any) and move it to the queue, if it finds an empty slot in the queue.

Dimensions are processed cyclically. Adaptivity is expressed basically in two ways. A message waiting at a queue can either be routed correctly along any of the possible dimensions it needs to correct, or be derouted as a result of local congestion problems.

D. Analysis of the Method

1. **Network delay**: No theoretical bounds are given in [19]. Some simulation results are shown, but for an 8-hypercube network (with only 256 nodes). These results are used to compare the Chaos Router with two other algorithms that differ from it only in the way in which the selection of the message to deroute is performed. These techniques are the "Priority Router", that deroutes messages according to priorities related to the age of the messages and therefore is livelock free and is the "Natural Router" that always deroutes the last message of the queue in FIFO order. Several traffic patterns have been tested. The results indicate that in almost all cases the "Natural Router" got slightly less average and worst case delay than the "Chaos Router"; these two performed much better than the "Priority Router". The maximum number of times any message was derouted was relatively small in all cases (between 1 and 4 times).

2. **Synchronization overheads**: Nothing has been said in [19] about the implementation of virtual barriers, for instance. Concurrent accesses to locks, for example, are not facilitated by the algorithm either.

3. **Fault tolerance**: Messages can move around faulty links or nodes by traversing alternative paths. If a link breaks down, the output buffers associated with it must appear as busy forever. Similar considerations follow for faulty nodes. So, if a message is waiting at a node's queue for traversing a faulty link, then it will either be routed through another link (if possible) or it will wait for being derouted. In principle, a message that has only to traverse a faulty link could remain in the queue forever. If the traffic remains relatively heavy, so as to keep the queue full, and derouting is performed infinitely often, the probability of its never being chosen for derouting goes to zero as time approaches infinity. So, it will probabilistically be derouted and so, another route will be found (if possible). But if the queue of the node at which the message is waiting does not become completely full any longer, for example, no derouting will ever again be performed, making this message stay in the node forever.

4. **Generality**: This routing algorithm has been presented as designed specifically with the hypercube network in mind. It is claimed that it can be adapted to work with any k-ary n-cube (see Section II). One of the main features hypercubes network have is that minimal paths from any node to any other node can be determined directly from the nodes' indexes.

As no synchronization mechanism is provided by this technique, it is best suited to supporting asynchronous programming models.

5. **Use of randomization**: Randomization is used when selecting a message to be derouted.

6. **Deadlock freedom**: The routing algorithm is deadlock free. The basis of the proof given in [19] is that any message that wants to enter a given node will eventually succeed in doing so in a finite amount of time, even if that implies that another message has to be derouted.

7. **Livelock freedom**: The Chaos Router is probabilistically livelock free, as pointed out above, i.e. message delivery is guaranteed with high probability. This result emerges from the fact that every message has a nonzero probability of avoiding derouting at each "derouting step".

8. **Starvation freedom**: The Chaos Router is probabilistically starvation free. In the `Read` operation, a given node is allowed to inject a message in the network if it finds at least one empty slot in the queue. Starvation may arise because the model of injection is continuous and so, infinite message streams are possible.

In [19] the use of Injection-Token Recirculation is proposed to solve this problem. Although they are criticized, Private-Buffer Recirculation and Packet-Injection Control are suggested there to avoid starvation.

By using Injection-Token Recirculation, starvation freedom is guaranteed only probabilistically, as message delivery is guaranteed only probabilistically and the injection mechanism relies on the delivery of token messages.

X. The Exchange Model

Exchange is the model presented by Ngai and Seitz for adaptive routing [21], [22]. They use a multicomputer network with bidirectional links, every node with at least as many buffers as incident links[6]. Every node n_i has a predefined *routing relation* $\mathcal{R}_i$ that tells which of its neighbors is the next on the route of the packets currently in n_i with destination n_j, for every n_j in the network. The transfer of packets between adjacent nodes is accomplished via an *exchange* operation: if n_i has a packet p in a buffer b and n_j is selected to be the next node in the route of p then any of the following cases may arise.

1. n_j has a packet p' in a buffer b' such that it also wants to exchange it with n_i. Then they use the common link, p is allocated in b' and p' in b.

2. n_j does not want to exchange any packet with n_i. Then

   a. n_j has an empty buffer b' that is not being used for another exchange operation. Then n_j receives p in b' and b receives the *null* packet, i.e., it is freed.

   b. n_j either has no empty buffer or all its empty buffers are being used to exchange through other links. Then, a buffer b' not currently being used for another exchange is chosen. The packet p' contained in b' is moved (along the link) to b and p is moved to b'.

[6] Here every buffer has place for only one packet.

It is assumed that the exchange operation is performed in a finite amount of time in all cases.

The exchange is a nonminimal adaptive routing. It allows to use a variety of injection policies, and works for both static and dynamic injection.

A. Analysis of the Method

1. **Network delay:** No theoretical bounds have been proved for this method. This is partially justified by the fact that this technique is very general (e.g., it is independent of the topology). As no combination is performed, the exchange routing technique cannot deal with many-to-one communications efficiently.

2. **Synchronization overheads:** No synchronization mechanism is provided.

3. **Fault tolerance:** This model is fault tolerant if the network has enough redundant paths among nodes. It can support both a dynamic and a static fault model, if faulty links are considered as links that cannot be used for exchanging. If a packet that finds no next link to go on its route is used for exchanging, many kinds of failures may be avoided. This point is more deeply discussed in [21, p. 90].

4. **Generality:** The exchange routing algorithm can be used over all kind of topologies, if every node has at least as many buffers as links incident to it. The routing relation should be able to be easily recalculated to deal with derouting.

5. **Use of randomization:** This technique uses randomization in the choice of the buffer to be preempted in the exchange operation, if there is more than one candidate.

6. **Deadlock freedom:** Deadlock freedom can be easily proved given that the number of buffers is greater than or equal to the number of links in all nodes and assuring that all buffers have some chance of exchanging. Every exchange is satisfied: if a node n wants to exchange with some other node p then the only way in which this could not happen would be if p had all of its buffers exchanging with other nodes. But this is not possible because of the above condition. It should be noted that this method allows deadlock free routing with a constant number of buffers if the degree of the nodes is constant.

7. **Livelock freedom:** Livelock freedom is assured by using the technique explained in the Livelock Freedom point in Section III. If the injection model is static, the packets' distance to destination can be used as their priority. On the other hand, if the injection model is dynamic, [22, pp. 10–15] suggests a priority based on the pair (age-of-packet, distance-to-destination) with the pairs ordered lexicographically.

8. **Starvation freedom:** If the injection model is static, no starvation can arise. So, only the injection policy needed in the exchange model to avoid starvation in a dynamic injection environment will be analyzed. Private-Buffer Recirculation and Packet-Injection Control are both proposed in [22] to solve this problem.[7] To implement the first of these policies, every node n_i must have b_i buffers and c_i links and it must hold that $b_i > c_i$. Every node owns a number of private tokens that is between 1 and $b_i - c_i$. To implement Packet-Injection Control, every node informs its neighbors about the number of injections it has performed by smuggling this information in the packets while performing exchange operations (see the Starvation Freedom point in Section III for details).

[7] In [21], it is called *Buffer Token Recirculation.*

XI. Routing by Sorting

Some methods have been developed that route messages, avoiding any conflict in the network. In order to achieve this, conflict-free paths have to be constructed for every pair of messages involved in a communication phase.

Unfortunately, it is not easy to build such paths. Even so, if the destinations of the messages involved in a communication phase define a (possibly partial) permutation of the indexes of the processors in the network, and if this permutation is known in advance, then it is possible to precompute the sequence of steps that each processor must execute in order to obtain conflict-free communication [30], [31].

Whenever the communication pattern does not define a permutation or this pattern is not known in advance, messages should be processed in such a way that those with the same node as destination are combined and then routed so as to avoid conflicts.

An example of such a technique is presented in [32], where the classification of the messages mentioned above is performed by sorting the messages according to their destination node. In [32], two common problems in communication are solved: **Random Access Read** and **Random Access Write**.

A. Description of the Algorithm

It is assumed that each processor has a unique index that is called *processor number*. The ith processor will be referred to as PE_i.

1) Definition of the problem: Random Access Read (RAR): Each processor wants to read the contents of a register of other processor. If it does not want any data, it must issue a request with infinite destination.

Random Access Write (RAW): Each processor wants to write data into other processor. Again, if it does not want to send any data to other processor, it must issue a request with infinite destination.

2) Auxiliary primitives: The following communication primitives are necessary to implement routing-by-sorting methods.

1. `RANK`: Initially, some processors are marked in some way. The `RANK` operation assigns each marked processor the number of marked processors that have lower indexes.

2. `CONCENTRATE`: Suppose `RANK` has been executed. The result corresponding to processor PE_i has been assigned to the field C_i of the record R_i (in PE_i). The `CONCENTRATE` operation sends each record R_i to the processor whose number is held in C_i.

3. DISTRIBUTE: Consider now an "initial" (with respect to the indexes) subset of the processors, i.e., those processors PE_i such that $0 \leq i \leq k$ for some $1 \leq k < N$. Each of these processors has a record R_i with a field H_i such that $H_i < H_j$ if $i < j$. The DISTRIBUTE operation sends each record R_i to processor H_i.

4. GENERALIZE: Consider an initial subset of the processors, each with a field H_i, as described just above. The GENERALIZE operation copies record R_i onto those processors with indexes between $H_{i-1}+1$ and H_i for every i in the initial subset. (For convenience, $H_{-1} = 0$).

5. SORT: Suppose that every processor PE_i has a record with a field C_i that will be used as a key by the sorting algorithm. After the sorting algorithm has been performed, records have been moved in such a way that, if $i < j$ then $C_i < C_j$.

3) Brief overview of the RAR and RAW algorithms: These techniques can be viewed as a combination of the subalgorithms described above. To start the routing algorithm, it is required that all processors meet at the communication step. Every processor has a request for some other processor. A succinct description of the algorithms is stated as follows [32].

Random Access Read. First, all the requests are sorted according to the processor to which they are directed. The number of the issuing processor is used to break ties. Therefore, all requests corresponding to a same processor will be at consecutive processors at the end of this phase. Among all the messages destined to a same processor, the previously stated, that has been issued by a processor with the least index is selected. The selected requests are **ranked** and then **concentrated** using this rank. Then, the requests are **distributed** using their destination. Therefore, all the requests arrive at their destination node where the required value is fetched. The replies are **concentrated** again using the previous ranks, and then **generalized** so that each request gets the reply from its "leader". After this, the replies are sorted with respect to their issuing processor. As a result of this, each processor receives the value it has asked for.

Random Access Write. As in the RAR, all the requests are sorted according to the processor to which they are directed. The number of the issuing processor is used to break ties. There are two possible policies, among others, regarding what to do to solve write conflicts (i.e., when two processors want to write data on a same processor). The first one chooses only the request with the least source index, as stated above. The rest of the requests destined to that processor are simply ignored. The selected requests are **ranked** and **concentrated** using the rank. After this, these requests are **distributed** according to their destination index. So, the **write** requests arrive at the destination node. No conflict arises, as at most one request arrives at a single processor.

The second policy allows all the requests to arrive at their destination processor. The algorithm is similar to the one of the first policy. The difference is that each nonselected request is given the rank of the selected request that represents it, i.e., the request with the same destination that has been selected as in the first policy. After this, **all** the requests are **concentrated** using this rank. Conflicts will arise as more than one request will try to enter the same node, if there are write conflicts. These conflicts are solved by combining the conflicting requests when they meet each other. Then, the requests are **distributed** as before. As will be mentioned below, the performance of the algorithm will depend on the number of write conflicts.

B. Analysis of the Method

1. **Network delay**: In the above algorithms, the distribution of the messages throughout the network is controlled all the time. Therefore, conflicts are avoided. So, the delay messages will suffer until delivery will not depend on the congestion of the network. In [32], algorithms for the subproblems of Section XI-A-2) are presented. They are $\mathbf{O}(qn)$ for an n-ary q-cube, and $\mathbf{O}(n)$ for the n-hypercube and for the n-shuffle-exchange networks. Nevertheless, no algorithm to sort N items in $\mathbf{O}(\log N)$ is known. Therefore, the latency of the whole routing algorithm will be determined by the order of the sorting algorithm selected. In [31], Batcher presents an algorithm for sorting 2^n numbers in $\mathbf{O}(n^2)$ in an n-hypercube and in an n-shuffle-exchange network. This algorithm performs bitonic sort. More recently, an algorithm with worst-case $\mathbf{O}(n(\log n)^2)$ performance was presented for the n-hypercube and related networks in [34]. However, for all practical values of n, bitonic sort will behave better.

If only a small subset of the processors participate in a given communication step, then other more efficient algorithms can be used for sorting the requests. In [35], Nassimi and Sahni present an algorithm that sorts P keys, $2^n = P^{1+\frac{1}{k}}$, in $\mathbf{O}(k \log P)$ time in an n-hypercube or n-shuffle-exchange, where k is a constant between 1 and $\log P$.

When managing one request per processor, Nassimi and Sahni [32] implement the RAR algorithm in $\mathbf{O}(n^2)$ time on an n-hypercube or an n-shuffle-exchange, and in $\mathbf{O}(q^2 n)$ time on an n-ary q-cube. They also implement the RAW algorithm in $\mathbf{O}(n^2 + dn)$ time on an n-hypercube or an n-shuffle-exchange and in $\mathbf{O}(q^2 n + dqn)$ time on an n-ary q-cube. d is the maximum number of data items written into any one PE.

An important point is that these routing algorithms finish a given communication phase in a certain number of steps that is known in advance. So, although the latency of this technique is worse than that of other techniques outlined in this paper, the algorithm is guaranteed to terminate in a given amount of time, whereas in the rest of the techniques this is only guaranteed with high probability. This is well-suited for synchronous programming models.

2. **Synchronization overheads**: The algorithm needs to have the requests issued in synchronous phases. All the messages participating in a given communication step must

be in the network when the sorting step of the algorithm starts.

3. **Fault tolerance**: In general, the algorithms cited above do not take into account the possibility of faults in the network. Neither do they consider it the algorithms presented in [32] for solving the subproblems described in Section XI-A-2).

4. **Generality**: This machine model has been designed to support synchronous programming models. All processors must synchronize at communication phases. In this way, there is no need of secondary buffers. The switches must have hardware to perform the different complex tasks (e.g., comparisons) required during the different phases of the algorithms.

5. **Use of randomization**: The algorithms described here do not use randomization at all. Nevertheless, the Valiant-Reif randomized sorting algorithm can be used to sort requests [36]. It is a probabilistical algorithm.

6. **Deadlock, livelock and starvation freedom**: Because of the characteristics of the algorithm, these are not problems at all.

XII. Acknowledgment

The authors wish to thank Lelia A. Vázquez and R. A. Alvez for their helpful comments and discussions on earlier versions of this paper.

References

[1] P. Gibbons, D. Soroker, and J. Sanz, "A study on massively parallel shared memory computing," RJ 7218 (67985) Computer Science, IBM Almaden Research Center, Dec. 1989.

[2] G. Pfister, W. Brantley, D. George, "An introduction to the IBM Research Parallel Processor Prototype (RP3)," *Experimental Parallel Computing Architectures*, J.J. Dongarra, Ed. Amsterdam, The Netherlands: North-Holland, 1987.

[3] C. Seitz, "The Cosmic Cube," *CACM*, vol. 28, no. 1, pp. 22–33, 1985.

[4] W. Athas, and C. Seitz, "Multicomputers: Message-Passing Concurrent Computers," *IEEE Comput.*, vol. 21, pp. 9–24, 1988.

[5] D. Kuck, E. Davidson, D. Lawrie, and A. Sameh, "Parallel Supercomputing Today and the Cedar Approach," *Experimental Parallel Computing Architectures*, J.J. Dongarra, Ed. Amsterdam, The Netherlands: North-Holland, 1987.

[6] BBN, "Butterfly-TM Parallel Processor Overview," 6149, BBN, June, 1986, version 2.

[7] T. Leblanc, M. Scott, C. and Brown, "Large-Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor," in *ACM Symp. on Parallel Programming*, pp. 161–172, July, 1988.

[8] D. Hillis, *The Connection Machine*. Cambride, MA: MIT Press, 1985.

[9] A. Ranade, S. Bhat, S. Johnson, "The Fluent Abstract Machine," *Fifth MIT Conference on Advanced Research in VLSI*, J. Allen and F.T. Leighton, Eds. Cambridge, MA: MIT Press, Mar. 1988, pp. 71 – 93.

[10] W. Dally and C. Seitz, "Deadlock-free routing in multiprocessor interconnection network," Computer Science Dep., Calif. Inst. Technol., Tech. Rep. 5206:TR:86, 1986.

[11] S. Felperin, L. Gravano, G. Pifarré, and J. Sanz, "Routing techniques for massively parallel communication," Computer Research and Advanced Applications Group, IBM Argentina, Tech. Rep. TR:90-01, 1990.

[12] L. Valiant, "General Purpose Parallel Architectures," *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Amsterdam, The Netherlands: North-Holland, 1988.

[13] L. Valiant, G. Brebner, "Universal Schemes for Parallel Communication," *ACM STOC*, pp. 263–277 1981.

[14] S. Felperin and J. Sanz, "Simulation results on the performance of fully-adaptive minimal deadlock-free routing for meshes and hypercubes," IBM Argentina, CRAAG, Tech, Rep., Mar. 1991.

[15] A. Borodin, and J.E. Hopcroft, "Routing, merging and sorting on parallel models of computation," in *Symp. Theory of Computing*, pp. 338–344, 1982.

[16] K. Gunther, "Prevention of deadlocks in packet-switched data transport system," *IEEE Trans. Commun.*, vol. COM-29, Apr. 1981.

[17] D. Gelernter, "A DAG-based algorithm for prevention of store-and-forward deadlock in packet networks," *IEEE Trans. Comput.*, vol. C-30, pp. 709–715, Oct. 1981.

[18] P. Merlin and P. Schweitzer, "Deadlock avoidance in store-and-forward networks. 1: Store-and-forward deadlock," *IEEE Trans. Commun.*, vol. 28, Mar. 1980

[19] S. Konstantinidou and L. Snyder, "The chaos router: A practical application of randomization in network routing.," in *2nd. Ann. ACM SAAP*, pp. 21–30, 1990

[20] S. Konstantinidou, "Adaptive, minimal routing in hypercube," in *6th. MIT Conf. Advanced Research in VLSI*, pp. 139–153, 1990.

[21] J. Ngai and C. Seitz, "Adaptive routing in multicomputers," in *Opportunities and constraints of parallel computing*, J. Sanz, Ed. New York: Springer Verlag, 1989.

[22] J. Ngai and C. Seitz, "A framework for adaptive routing," Computer Science Dep., Calif. Instit. Technol., Tech. Rep. 5246:TR:87, 1987.

[23] M. Fulgham, R. Cypher, and J. Sanz, "A comparison of SIMD hypercube routing strategies," IBM Almaden Research Center, 1990.

[24] E. Upfal, "Efficient schemes for parallel communication," *JACM*, vol. 31, pp. 507–517, July 1984.

[25] L.G. Valiant, "Optimality of a two-phase strategy for routing in interconnection networks," Mar. 1982.

[26] N. Pippenger, "Parallel communication with limited buffers," in *Foundations of Computer Science*, pp. 185–194, 1985.

[27] A. Ranade, "How to emulate shared memory," in *Foundations of Computer Science*, pp. 185–194, 1985.

[28] Y. Birk, P. Gibbons, D. Soroker, and J. Sanz, "A simple mechanism for efficient barrier synchronization in MIMD machines," Computer Science, IBM Almaden Research Center, RJ 7078 (67141), Oct. 1989.

[29] G. Pifarré, S. Felperin, L. Gravano, and J. Sanz, "New techniques for combination, adaptivity, deadlock-freedom and synchronization in massively parallel routing," to be published, 1990.

[30] D. Nassimi and S. Sahni, "Parallel algorithms to set up the Benes Permutation Network," *IEEE Trans. Comput.*, vol. C-31, pp. 148–154, Feb. 1982.

[31] J. Schwartz, "Ultracomputers," *ACM Trans. Programming Languages and Systems*, vol. 2, pp. 484–521, Oct. 1980.

[32] D. Nassimi and S. Sahni, "Data broadcasting in SIMD computers," *IEEE Trans. Comput.*, vol. C-39, 2, pp. 101 – 107, Feb, 1981.

[33] K.E. Batcher, "Sorting networks and their applications," in *AFIPS Conf. Proc.*, pp. 307–314, 1968.

[34] R. Cypher and C. Plaxton, "Deterministic sorting in nearly logarithmic time on the hypercube and related computers," in *22nd Ann. Symp. Theory of Computing*, pp. 193–203, 1990.

[35] D. Nassimi and S. Sahni, "Parallel permutation and sorting algorithms and a new generalized connection network," *JACM*, vol. 29, pp. 642–667, July 1982.

[36] J. Reif and L. Valiant, "A logarithmic time sort for linear size networks," *JACM*, vol. 34, pp. 60–76, Jan. 1987.

[37] E. Upfal, "An O(log *N*) deterministic packet routing scheme," in *21st Ann. ACM-SIGACT Symp. Theory of Computing*, May 1989.

[38] T. Leighton and B. Maggs, Expanders might be practical: Fast algorithms for routing around faults on multibutterflies," in *30th Ann. Symp. Foundations of Comput. Science* (IEEE, ed.), pp. 384–389, Oct. 1989.

[39] T. Leighton, D. Lisinski, and B. Maggs, "Empirical evaluation of randomly wired multistage networks," in *ICCD'90*, 1990.

[40] G. Pifarré, L. Gravano, S. Felperin, and J. Sanz, "Fully-adaptive minimal deadlock-free packet routing in hypercubes, meshes, and other networks," IBM Almaden Research Center, Tech. Rep., 1991.

Sergio A. Felperin was born in Buenos Aires, Argentina in 1965. In 1984 he began studying computer science at the University of Buenos Aires, Argentina. In 1988, he was admitted to the Escuela Superior LatinoAmericana de Informatica (ESLAI) and graduated in April 1991.

In 1990, he joined the Computer Research and Advanced Applications Group in IBM Argentina as a researcher. His current areas of interest are massively parallel routing and programming languages.

Luis Gravano was born in Buenos Aires, Argentina in 1967. In 1986 he began studying computer science at the University of Buenos Aires, Argentina. In 1988, he was admitted to the Escuela Superior LatinoAmericana de Informatica (ESLAI) and graduated in April 1991.

In 1990, he joined the Computer Research and Advanced Applications Group in IBM Argentina as a researcher. His main current areas of interest are massively parallel routing and programming languages.

Gustavo D. Pifarré was born in Buenos Aires, Argentina in 1964. In 1985 he began studying computer science at the University of Buenos Aires, Argentina. In 1988, he was admitted to the Escuela Superior LatinoAmericana de Informatica (ESLAI) and he graduated in April 1991.

In 1990, he joined the Computer Research and Advanced Applications Group in IBM Argentina as a researcher. His current area of interest is massively parallel routing.

Jorge L. C. Sanz was born in Buenos Aires, Argentina in 1955. He received the M.S. degree in computer science in 1977 and in mathematics in 1978, both from the University of Buenos Aires, Argentina. In 1981 he received the Ph.D. degree in applied mathematics, working on complexity of algorithms, from the same university.

During 1978 and 1980, he was with the University of Buenos Aires at the Department of Mathematics as an instructor. He conducted research as a scholar member of the National Council of Scientific and Technical Research of Argentina for four years. He was the recipient of many scholarships and a member of the Argentinian Institute of Mathematics. During 1981 and 1982 he was a visiting scientist at the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign. In 1983 he was an assistant professor at the same university in the Electrical Engineering Department and the Coordinated Science Laboratory. His areas of broad professional interest are computer science and applied mathematics. Specifically, he is interested in multidimensional signal and image processing, image analysis and machine vision, parallel processing, numerical analysis, and computer architectures. Since 1984, he has been with the Computer Science Department, IBM Research Laboratory, San Jose, CA, as a research staff member. He conducts research work on industrial machine vision, parallel computing, and multidimensional signal processing. He was the technical manager of the machine vision group during 1985 and 1986. Since 1985, he has also been an adjunct Associate Professor with the University of California at Davis, where he conducts research as the Associate Director of the Computer Vision Research Laboratory. He has served as a consultant of several companies in the United States. Since August 1987, he has been an associate editor of the IEEE Transactions on Acoustics, Speech and Signal Processing. He is the editor of the IEEE Pattern Analysis and Machine Intelligence 1988 special issues on Industrial Machine Vision and Computer Vision Technology. Also, he is an editor-in-chief of *Machine Vision and Applications, an International Journal*. He is an author of the book *Radon and Projection Transform-Based Computer Vision*.

Dr. Sanz is a member of ACM. In 1986, he received the IEEE Acoustic Speech and Signal Processing Society's Paper Award. He is a committee member of the Multidimensional Signal Processing Group of the IEEE Acoustic, Speech and Signal Processing Society. He has been a chair and organizer of the 1988 "IEEE Workshop on Machine Vision" and the chair and organizer of the IBM Almaden-NSF Workshop on "Opportunities and Constraints of Parallel Computing." He was a program chair of the "VI IEEE Acoustic Speech and Signal Processing Workshop on Multidimensional Signal Processing." He was the program committee chair of the Computer Architecture chapter at the "1990 International Conference on Pattern Recognition."