RC 20372 (9/29/95)
Computer Science

IBM Research Report

Adaptive Source Routing in Multistage Interconnection Networks

Craig B. Stunkel, Bulent Abali

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

Yucel Aydogan, Cevdet Aykanat

Bilkent University
Dept. of Computer and Information Science
Ankara, Turkey

Research Division
Almaden · T.J. Watson · Tokyo · Zurich

page intentionally left blank for
double sided printing.

Adaptive Source Routing in Multistage Interconnection Networks

Yucel Aydogan†, Craig B. Stunkel‡, Cevdet Aykanat†, Bulent Abali‡*

† Bilkent University, Dept. of Computer Science, Ankara.

‡ IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598

Abstract

We describe the adaptive source routing (ASR) method which is a first attempt to combine adaptive routing and source routing methods. In ASR, the adaptivity of each packet is determined at the source processor. Every packet can be routed in a fully adaptive or partially adaptive or non–adaptive manner, all within the same network at the same time. We evaluate and compare performance of the proposed adaptive source routing networks and oblivious routing networks by simulations. We also describe a route generation algorithm that determines maximally adaptive routes in multistage networks.

1 Introduction

Interconnection networks play an important role in providing low latency, high bandwidth communication in multicomputers. Some examples of interconnection networks used in commercial machines are the IBM SP2 multistage interconnection network [1], Cray T3D 3–dimensional torus [2, 3], the Connection Machine fat tree [4, 5], and Intel Paragon mesh [6]. Routing in an interconnection network can be classified as adaptive or non-adaptive depending on the dynamics of route selection. In non-adaptive (or oblivious) routing, there is a fixed routing decision at each intermediate switching element (switch) along a path between a source node and a destination node—each switch can use only one output port for message packet forwarding. Adaptive routing methods allow more than one choice of output ports. Switches try to minimize network contention by exploring alternate routes to destinations [5, 7, 8, 9]. On the other hand, some networks employ oblivious routing methods such as the source–based routing (source routing) used in SP2 [1, 10, 11] due their flexible choice of network topology and simplicity of switch design. In the source routing method, the packet route is deterministic and is completely determined at the source processor which encodes the route in the packet header. Thus another way of classifying routing is source-based or destination-based according to the

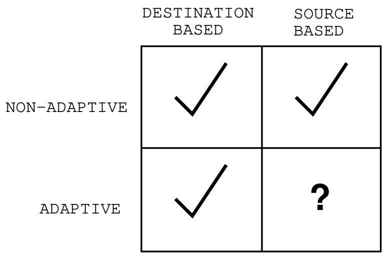

Figure 1: Routing Methods

method of addressing the destination processor, which results in a method space shown in Fig. 1.

Although many adaptive routing networks have been constructed and proposed to date, they have all been destination–based adaptive routing networks. This paper presents the first attempt to combine the source routing and adaptive routing, referred to as the *adaptive source routing* (ASR) method. The proposed combination has the advantages of both methods. The route and the adaptivity of each packet is determined at the source processor node. Every packet can be routed in a fully adaptive, or partially adaptive, or oblivious manner, all within the same network at the same time. The ASR method also provides support for multiple types of network traffic, in–order delivery of multiple packets, and network partitioning.

In the following, we give an overview of adaptive, destination–based, and source–based routing methods. In Section 2, we describe the proposed Adaptive Source Routing method. In Section 3, we present a performance comparison of the ASR networks and the oblivious routing networks by simulations. In Section 4, we present a route generation algorithm that finds *maximally adaptive* routes between the processor nodes. This algorithm enables the ASR method.

1.1 Background

In adaptive routing networks, message packets make use of multiple paths between source–destination node pairs [7]. Switches alleviate congestion by sending packets via less busy alternate routes. Typically, a busy output port will cause an adaptive routing switch to use another output in routing a packet to its destination. In a (destination–based) adaptive routing network, a switch element must therefore

*To whom correspondence should be addressed.

"know" which of its outputs lead to the intended destination. Therefore, a common characteristic of many adaptive routing networks is a regular and simply described network topology such as a hypercube, mesh, k–ary n–cube, or a fat tree [4, 5, 7, 8, 9]. The switches then have an implicit knowledge of the entire network topology, and therefore they can route packets accordingly. A disadvantage of adaptive routing is that it limits the choice of network topologies. In an alternative approach, each switch may have a routing table that maps destination processor addresses to the switch port numbers, however this will occupy real–estate on the switch (chips) and the bounded size of the tables may limit scalability of the networks.

In the destination–based routing, the address (e.g. position) of the destination processor in the network, or an address difference between source and destination, is encoded in the packet header and then the network decides how to route the message. The source processor has no influence on the routing decisions. This method also requires switches to have a global knowledge of the network topology in order to correctly route packets. Some examples of destination–based networks are the CM-5 fat tree [5] and Intel Paragon 2–dimensional mesh [6].

In the source routing method, unlike destination-based routing, switches need not know the topology; the source processor determines the route and encodes the routing instructions in the packet header. Switches then follow these instructions to forward the packet to its destination. Cray T3D [3] and IBM SP2 [1] systems are based on source routing networks. For example, in the SP2 multistage network, which consists of 8×8 switches, the packet header initially contains 3–bit routing words $R_1, R_2, \ldots, R_n$, where n is the number of network stages to travel. Each word indicates a switch port numbered from 0 to 7. The source processor determines the route and puts respective words in the header. Each switch forwards the packet through the output port indicated in the first route word and strips off the first word before forwarding the packet to the next level in the network [1]. Thus, the packet contains no routing information upon arriving at its destination. In the source routing method, typically routing headers are computed only once and then kept in a route table in each processor node. The route table approach enables faulty links and switches to be mapped out easily, and allows more choices of network topologies than destination–based routing, and allows multiple routes to be defined per destination.

2 Architecture

In the adaptive source routing method proposed here, the key idea is in the definition of the routing words in a packet header. Each routing word indicates a *set* of permitted output ports, rather than a specific output port. Each m–bit word has the format $R = r_{m-1}r_{m-2}\ldots r_0$, where m is the number of switch ports. One bits in the routing word indicate the set of outputs that the switch is permitted to use for forwarding the packet. The source processor is responsible for encoding the correct routing instructions in the header as in the source routing method.

Each switch examines the first word of each packet and (adaptively) selects an unused port from one of the permitted outputs to forward the packet to the next network stage. If none of the permitted ports are available, then the packet will be blocked and cannot proceed until at least one of the ports become available. The switch strips off the first route word before forwarding the packet as before. For example, in the 32 node network given in Fig. 14, the header of a packet from processor 4 to 30 may consist of words $R_1 = 11110000, R_2 = 11110000, R_3 = 10000000, R_4 = 01000000$. The header indicates to the first, second, third, and the last stage switches that they may forward the packet through one of four ports 4–7 (R_1), one of four ports 4–7 (R_2), port 7 (R_3), and port 6 (R_4), respectively. In general, the number of distinct paths a packet may follow from source to destination is

$$N_{\mathtt{path}} = |R_1| \times |R_2| \times \cdots\cdots \times |R_{n-1}| \times |R_n|$$

where $|R_i|$ is defined as the number of one bits in the routing word R_i. Obviously, not only N_{path} paths must exist between the source and the destination, but any combination of the outputs specified in the successive routing words in the header must correctly lead the packet to its intended destination.

Each source processor can choose the adaptivity of a message packet by varying $N_{\mathtt{path}}$. If $N_{\mathtt{path}} = \max$, then the adaptivity is maximum and packets may reap performance benefits of full adaptivity. This case is useful to minimize network contention due to non–uniform message traffic and difficult communication patterns.

If $N_{\mathtt{path}} = 1$, then the routing is oblivious. The packet is to be routed through a single deterministic path. The $N_{\mathtt{path}} = 1$ case may be useful in several applications. If interprocessor communication patterns are known in advance, such as for permutation routing on SIMD machines, an optimal set of oblivious routes may be selected to minimize contention [11]. The $N_{\mathtt{path}} = 1$ case is also valuable for avoiding the *over–taking* property of adaptive networks. Over–taking means that the successive packets that belong to the same message may arrive at their destination out of order, which requires the receiving processor to buffer and sort them. Thus, over–taking may result in more hardware resources being used and may offset performance gains due to adaptive routing. In a possible implementation that guarantees in–order delivery of packets (that belong to the same message), source processors may send the packets of a multiple packet message non–adaptively, i.e. use $N_{\mathtt{path}} = 1$, whereas they may send a single packet message adaptively,

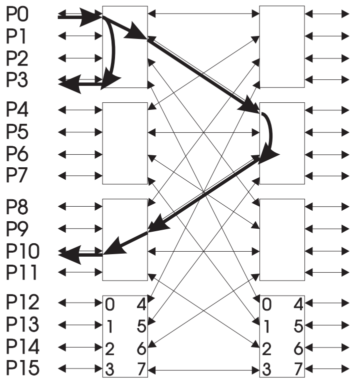

Figure 2: A 16 processor node bidirectional multistage network (BMIN)

i.e. use $N_{\mathtt{path}} = \max$, since single packet messages are not subject to over-taking.

If $1 < N_{\mathtt{path}} < \max$, then each packet is routed in a *partially–adaptive* manner, where only a subset of all possible paths is utilized. This case is valuable for network security or partitioning: When the network is to be logically partitioned among multiple parallel tasks so that their respective communications do not overlap in the network, then by using the partially–adaptive routing method each packet may be forced to remain in its partition, however routed adaptively within the partition.

3 Network Performance

We are primarily interested in the effect of adaptive routing on bidirectional multistage (BMIN) topologies similar to the topologies used in the IBM SP2, the Thinking Machine CM-5 [12], and the Meiko CS-2 [13]. Figure 2 illustrates a 16 processor node BMIN and shows sample routes from a source node 0 to destination nodes 3 and 10. The 16 ports on the right side are unused in this configuration. The BMIN switches—for this example 8 input, 8 output devices—could be permitted to forward packets from any input port to any output port (including ports on the same "side").

We have seen few studies directly comparing adaptive versus oblivious routing for BMIN's, although the CM-5 machine employs destination-based adaptive routing. In addition, we are interested in assessing the effects of adaptive routing when used in combination with switches that incorporate central buffers similar to SP2 switches [1].

To evaluate the performance of adaptive source routing, we conducted network simulations based upon a C++ model of SP2-like switches. These switches implement *buffered wormhole routing* [1] for flow-control and contain a 1 KB dynamically-shared central buffer. Under light to medium loading, a switch is typically able to buffer an entire arriving packet when that packet becomes blocked due to output port contention. Thus, in effect, the switch often operates in virtual cut-through [14] fashion, completely removing blocked packets from network links. However, under heavy loading the central buffer may become full, and packets may then be blocked across several switches, just as in wormhole routing [15].

In BMIN networks, adaptive choices can typically be made while the packet is traveling "away" from the processors until it reaches any switch which is a least common ancestor of both the source and the destination node. When more than one output port is both idle and permitted for adaptive routing, our simulations assume the choice of output port is made on a least-recently-granted basis. The path "back" to the destination from the least common ancestor switch is unique. We assume minimal paths—if there exists an h-hop path between source and destination, no $> h$-hop paths may be traversed for communication between them.

All simulations assume an open network model containing idealized processor nodes: the nodes contain an infinite transmit queue buffer, and packet flits are immediately pulled from the network as they arrive. We assume an exponential distribution for message injection time (message arrival time). We apply a range of loading to the network, where a load of 1.0 indicates that each node is injecting packets in the network at the maximum link data rate. Latency curves include input queueing time and are not shown after saturation (steady-state latency is infinite after saturation, assuming infinite input queues). The maximum packet size is 255, and messages longer than 255 bytes are broken into multiple packets before transmission.

The open network model makes it possible to "stress" the network to a far greater degree and cause more contention than might be possible in a "real" environment. For instance, in the SP2, the processor software and the network interface hardware control the injection of message packets via strategies such as end-to-end flow control and message interleaving that significantly reduce the possibility of network saturation and the creation of "hot-spots" [16]. Therefore the heavily-loaded simulation results shown here are extremely unlikely to be reproducible in an actual machine. However, the open network model simplifies analysis by removing the complex software and network interface factors, and makes it possible to examine a single issue: the effect of adaptive routing.

For each experiment, we compare adaptive routing with oblivious routing schemes. For instance, in SP2 systems, each node maintains a route table containing 4 valid minimal routes for each destination node. If there are less than 4 unique minimal routes, as when the source and destination node are connected to the same switch, then 2 or more of these routes are identical. Choosing between 4 routes reduces the effect of contention and reduces the probability of creating "hot-spots" in the network.

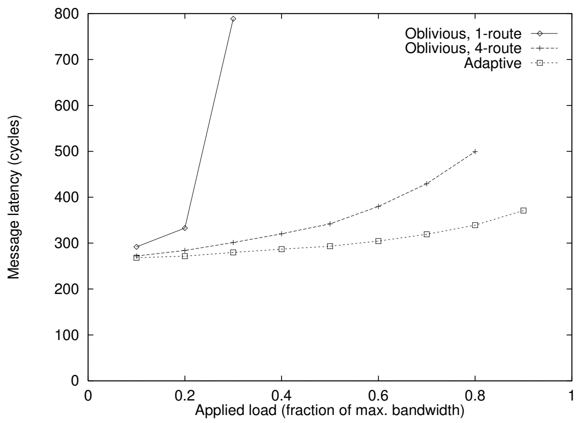

Figure 3: Bit-reversal permutation traffic on a 16-node BMIN topology

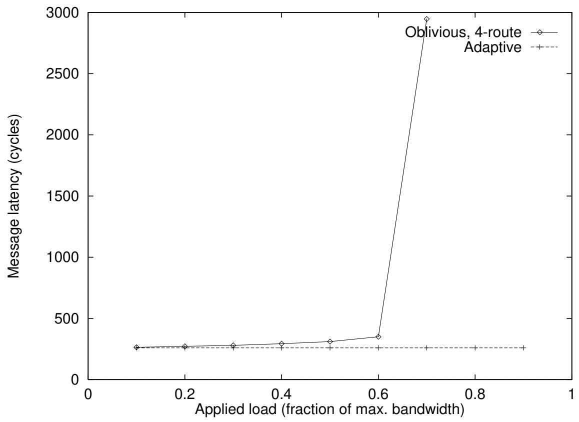

Figure 4: Transpose permutation traffic on a 16-node BMIN topology

3.1 Permutation traffic simulation

In this section we investigate the relative performance of adaptive routing when the communication pattern is a static permutation. We test 2 permutations: bit-reversal and transpose. In bit-reversal, a source processor represented in binary by $s_{n-1}s_{n-2}\dots s_1 s_0$ sends messages to destination $s_0 s_1 \dots s_{n-2}s_{n-1}$. In transposes for even n, the destination is $s_{\frac{n}{2}-1}s_{\frac{n}{2}-2}\dots s_1 s_0 s_{n-1}s_{n-2}\dots s_{\frac{n}{2}+1}s_{\frac{n}{2}}$. We simulate 16-way and 64-way BMIN's. For our simulations the 64-way BMIN is constructed from 4 of the 16-way BMIN's shown in Figure 2. For each 16-way BMIN, the 16 unused right-side bidirectional links are connected to a separate switch in a 3rd stage of 16 switches. Thus each 3rd stage switch is connected to each 16-way BMIN by one link.

Figure 3 displays simulation results for the bit-reversal permutation on a 16-way BMIN topology. Adaptive routing attained both the lowest latency and the highest saturation bandwidth for this difficult permutation. For our 1-route oblivious routing, each packet traverses a "straight" path to a least common ancestor switch, and then the packet proceeds on the unique path to the destination. This topology

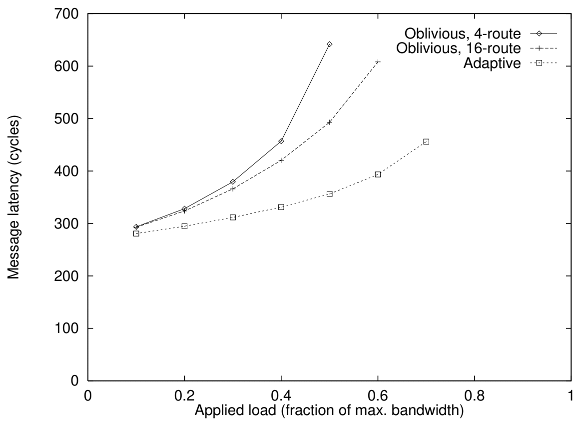

Figure 5: Transpose permutation traffic on a 64-node BMIN topology

has a maximum of 4 distinct paths between pairs of nodes, and thus 4-route oblivious routing is equivalent to randomized routing for the 16-way topology shown in Figure 2. In general, 1-route oblivious routing either performs very well or very poorly depending on the permutation. Its dismal worst-case performance and high variability make it a poor choice for a general routing strategy, and we will not consider it further in this paper.

For this 16-way topology, adaptive routing and 4-route oblivious routing have exactly the same paths available. However, with adaptive routing any packets traveling a 3-hop path are guaranteed *not* to contend with any other packets while traversing the first switch stage. Why? For this first hop, only 4 input ports (the "left" input ports in Figure 2) are contending for the 4 "right" output ports of the switching element (packets cannot enter and then exit the "right" side of the switching element, because the resulting path would not be minimal). Therefore if a packet is entering the "left" side, there are ≤ 3 other input ports currently sending packets to the "right" side, leaving at least one "right" output port open. The 4-route oblivious packets may often contend in the first stage, and this is the major cause of higher latency for this experiment.

Figure 4 illustrates the adaptive routing performance for the transpose permutation. Again adaptive routing performs better, and in fact encounters *no* contention. The 4-route method loses bandwidth principally because of contention in the first hop.

We have established that adaptive routing performs well for several types of permutation traffic on small systems. We now briefly examine the performance of one permutation on a larger system to illustrate that the benefits of adaptive routing extend over a range of system sizes. Figure 5 displays the latency curves for the transpose permutation on a 64-way BMIN topology. Adaptive routing still obtains lower latency and higher saturation throughput, although it no longer achieves the "no contention" curve of

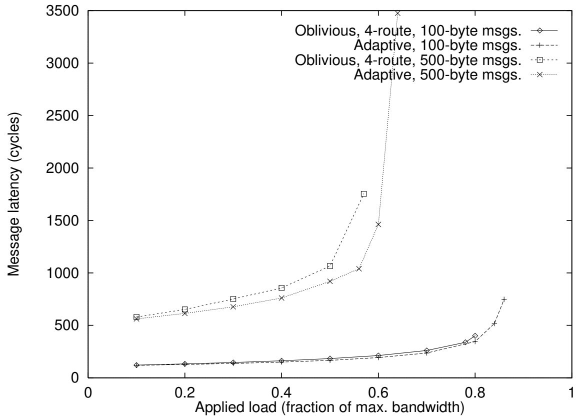

Figure 6: Short message random traffic on a 16-node BMIN topology

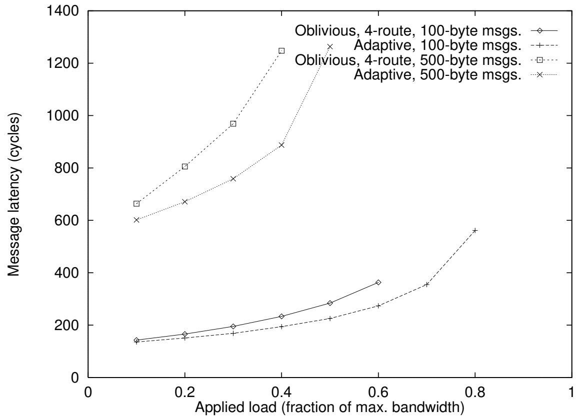

Figure 7: Short message random traffic on a 128-node BMIN topology

the 16-way system. For the 64-way system, packets with source and destination in different 16-way groups will traverse 5 switches and have 16 possible least common ancestors. Thus, 4-route oblivious routing no longer corresponds to random routing, and we include the 16-route oblivious case to demonstrate that adaptive routing maintains performance advantages over random routing as the system size grows. Other permutations and system sizes support similar conclusions, but we will not exhaustively detail results to further support these claims here.

3.2 Random traffic simulation

In other experiments, we injected traffic with a uniform destination distribution: for each message, the source randomly chooses any node except itself as the destination. Figure 6 plots message latency for adaptive routing and 4-route oblivious routing for short (100-byte and 500-byte) messages. Latency before saturation is lower and saturation load is higher for adaptive routing, although neither criteria is significantly better than that of oblivious routing.

To see how the effect of adaptive routing for random

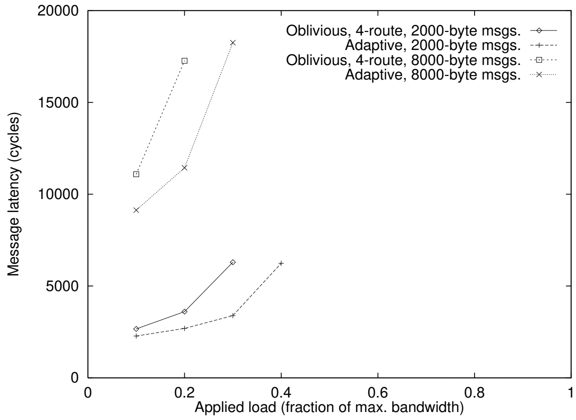

Figure 8: Long message random traffic on a 128-node BMIN topology

traffic changes with system size, Figure 7 shows the results of the same short message experiment conducted on a 128-way BMIN, an example of which can be found in [1]. For this larger topology, the positive effects of adaptive routing on random routing are more pronounced. There are more stages in which adaptive routing avoids contention compared with oblivious routing.

Figure 8 shows the results of the same 128-way experiment conducted with longer (2000-byte and 8000-byte) messages. For the longer messages, adaptive routing saturates the network at a 25% higher input load than 4-route oblivious routing. As messages become longer, the effect of hot-spots becomes greater, and adaptive routing tends to shift traffic away from heavily loaded parts of the BMIN network.

To summarize: for BMIN's, adaptive routing is generally superior to oblivious routing for both permutation and random routing. The advantages accrue for two reasons: (1) Adaptive routing does not contribute to contention on the path "away" from the processors, because for this portion of the path each packet always finds an output port link available. (2) Even in the absence of contention, adaptive routing randomizes traffic by choosing among several available output ports going "away" from the nodes.

4 Routing Algorithm

In this section, we describe an algorithm that generates the adaptive routing headers of the message packets. The algorithm maximizes the adaptivity, ($N_{\mathtt{path}}$), of the header. The problem of maximizing the adaptivity may be complicated by irregularities in the network topology, such as faulty links and switches. Here, we present an approach that is applicable to any multistage interconnection network, including networks with faults and partitioned networks.

We represent the topology of the network by a directed graph $G_T = (V_T, E_T)$, which is referred here as the *topology graph*. The vertex set V_T contains two types of nodes,

GENERATE_ROUTES(G_T, s)
1 $G_{\pi_s} \leftarrow$ BFS1(G_T, s);
2 **for** each processor $d \neq s$ **do**
3 $\quad G_R \leftarrow$ BFS2(G_{π_s}, d);
4 $\quad G_S \leftarrow$ ALL_FEASIBLE_ROUTES(G_R);
5 $\quad RT_{sd} \leftarrow$ MAX_ADAPTIVE_ROUTE(G_S);
6 **return** the routing table RT

Figure 9: Generating routes from a processor to other processors

namely processor nodes and switching nodes. The edge set E_T represents the interconnections between the switching nodes and between the processor and switching nodes. Each edge $e = < u, v >$ has an m-bit binary label $\ell_T[e]$ whose 1-bit position denotes the output port number of the switching vertex u it is sourced from. For the sake of efficiency, multiple edges between the same pair of vertices in the same direction are coalesced into a single edge. The label of the coalesced edge is obtained by bitwise OR'ing the labels of the individual edges.

We will work out an example on a 32 node network shown in Fig. 14. The topology graph $G_T = (V_T, E_T)$ contains 48 vertices which represent the switching nodes and processor nodes. The processors are indexed from 0 to 31 and switches are indexed from 32 to 47. In the examples to follow, the message source will be processor 4 and its destination will be processor 30.

Each processor node, $s \in V_T$, calls the GENERATE_ROUTES(G_T, s) function given in Fig. 9 to determine the set of adaptive routes from itself to every other processor $d \in V_T$ in the network. The first step (line 3) of the function finds all possible shortest paths from source to the destination processor on a *routability graph*. The second step (line 4) enumerates all feasible adaptive routing solutions on a *solution graph*. The last step (line 5) selects a route from the solution graph with the maximum adaptivity and stores the result in the routing table.

4.1 Routability Graph

A *routability graph* $G_R = (V_R, E_R)$ enumerates all possible shortest paths from a source to a destination processor. A routability graph contains only switching nodes and it is a subgraph of the topology graph with all switching nodes and edges that are not in the shortest paths from the source to destination node eliminated. For example, Fig. 15 shows the routability graph for the source–destination pair (4,30) of the network given in Fig. 14. Usable output ports are indicated by the edge labels.

Formally, $G_R = (V_R, E_R)$ for a given source–destination processor pair is defined to be a directed n-stage multistage graph [17], where n denotes the shortest path distance between the source and destination processors. Here, distance refers to the number of switching elements in a route. Each vertex $v \in V_R$ has an m-bit binary attribute

BFS1(G_T, s)
for each vertex $v \in V_T - \{s\}$ **do**
$\quad color[v] \leftarrow$ WHITE;
$color[s] \leftarrow$ GRAY; $depth[s] \leftarrow 0$; $Q \leftarrow \{s\}$;
while $Q \neq \emptyset$ **do**
$\quad u \leftarrow head[Q]$;
$\quad$ **for** each $v \in Adj_T[u]$ **do**
$\quad\quad$ **if** $color[v] =$ WHITE **then**
$\quad\quad\quad color[v] \leftarrow$ GRAY; $depth[v] \leftarrow depth[u] + 1$;
$\quad\quad\quad \pi[v] \leftarrow \{u\}$; $\ell_{\pi_s}[< v, u >] \leftarrow \ell_T[< u, v >]$;
$\quad\quad\quad$ FIFO_ENQUEUE(Q, v);
$\quad\quad$ **elseif** $color[v] =$ GRAY **and**
$\quad\quad\quad depth[v] = depth[u] + 1$ **then**
$\quad\quad\quad \pi[v] \leftarrow \pi[v] \cup \{u\}$; $\ell_{\pi_s}[< v, u >] \leftarrow \ell_T[< u, v >]$
$\quad$ FIFO_DEQUEUE(Q); $color[u] \leftarrow$ BLACK
return $G_{\pi_s} = (V_{\pi_s}, E_{\pi_s})$, where
$\quad V_{\pi_s} = \{v \in V_T : color[v] = \text{BLACK}\} - \{s\}$
$\quad E_{\pi_s} = \{< u, v >: u, v \in V_{\pi_s} \; and \; v \in \pi[u]\}$

Figure 10: The BFS–like algorithm proposed to construct the predecessors subgraph $G_{\pi_s} = (V_{\pi_s}, E_{\pi_s})$ for the source processors.

$ports_R[v]$ whose 1-bit positions denote the output ports allowed during routing to reach the destination processor. Vertices V_R^i at each stage i are indexed in decimal ordering from 0 to $|V_R^i| - 1$, for $i = 1, 2, \ldots, n$. Both first and last stages contain a single vertex v_0^1 and v_0^n which correspond to the source and destination switches, respectively. Here, source and destination switches refer to the unique switching nodes to which the source and the destination processors are connected, respectively. The routing word R_n for reaching the destination processor from the the destination switch is known in advance. Edges exist only between the vertices of the successive stages. That is, $< u, v > \in E_R$ only if $u \in V_R^i$ and $v \in V_R^{i+1}$ for some $i = 1, 2, \ldots, n-1$. Each edge $e = < u, v >$ is labeled with $\ell_R[e]$ similar to G_T.

The routability graph for a given source–destination processor pair (s, d) is constructed in two steps. In the first step, we use a modified Breadth-First-Search (BFS) algorithm on G_T starting from the source vertex s. The proposed BFS–like algorithm—BFS1(G_T, s) in Fig. 10—constructs the *predecessors subgraph* $G_{\pi_s} = (V_{\pi_s}, E_{\pi_s})$ which is different from the breadth-first tree generated during conventional BFS [18]. In G_{π_s}, V_{π_s} contains all processor nodes of G_T, and those switching nodes of G_T which are in the shortest route from the source processor s to at least one destination processor other than s. Similarly, E_{π_s} contains those edges (links) of G_T in reverse direction which are in the shortest route from the source processor s to at least one destination processor other than s. As seen in Fig. 10, each node $v \in V_{\pi_s}$ contains multiple parents stored in its $\pi_s[v]$ field which also denotes the adjacency list of vertex v in G_{π_s}. Hence, edge list E_{π_s} of G_{π_s} is constructed on G_T in adjacency list format by the π fields of the vertices in V_{π_s}.

In the second step, the routability graph for a processor pair (s, d) can easily be constructed by running another

BFS2(G_{π_s}, d)
$dsw \leftarrow \pi[d]$; /* dsw is the destination switch */
for each vertex $v \in V_{\pi_s} - \{dsw\}$ **do**
  $color[v] \leftarrow$ WHITE;
$color[dsw] \leftarrow$ GRAY; $Q \leftarrow \{dsw\}$;
while $Q \neq \emptyset$ **do**
  $u \leftarrow head[Q]$;
  for each $v \in \pi[u]$ **do**
    if $color[v] =$ WHITE **then**
      $color[v] \leftarrow$ GRAY; $Adj_R[v] \leftarrow \{u\}$;
      $ports_R[v] \leftarrow \ell_{\pi_s}[< u, v >]$;
      $stage_R[v] \leftarrow depth_{\pi_s}[v]$;
      FIFO_ENQUEUE(Q, v);
    else /* $color[v]$ should be GRAY */
      $Adj_R[v] \leftarrow Adj_R[v] \cup \{u\}$;
      $ports_R[v] \leftarrow ports_R[v] \bigvee \ell_{\pi_s}[< u, v >]$;
      /* "$\bigvee$": bitwise OR */
    $\ell_R[< v, u >] \leftarrow \ell_{\pi_s}[< u, v >]$;
  FIFO_ENQUEUE(Q); $color[u] \leftarrow$ BLACK
return $G_R = (V_R, E_R)$, where
  $V_R = \{v \in V_{\pi_s} : color[v] =$ BLACK $\}$
  $E_R = \{< u, v >: u, v \in V_R \text{ and } v \in Adj_R[u]\}$

Figure 11: The BFS–like algorithm proposed to construct the routability graph $G_R = (V_R, E_R)$ for the source destination pair (s, d).

BFS–like algorithm—BFS2(G_{π_s}, d) in Fig. 11—on G_{π_s} starting from destination processor d. In Fig. 11, each non–black (white and gray) vertex $v \in V_{\pi_s}$ encountered while scanning the adjacency list of a vertex u of depth j from the destination switch constitutes an edge from vertex v to u at stages i and $i+1$ of G_R, respectively, where $i = n - j - 1$.

4.2 Solution Graph

A *solution graph* $G_S = (V_S, E_S)$ enumerates every feasible adaptive route solution (route-word encoding) from a given source to a given destination. Formally, $G_S = (V_S, E_S)$ is defined to be a multistage graph with the same number of stages as in the routability graph G_R. The vertex set V_S^i of G_S at stage i is a subset of the power set of V_R^i of G_R excluding the empty set, i.e., $V_S^i \subseteq 2^{V_R^i} - \emptyset$. It is clear that both first and last stages (stages 1 and n) of G_S contain a single vertex v_1^1 and v_1^n which correspond to the source and destination switches, respectively.

In a straightforward implementation, we allocate $2^{|V_R^i|} - 1$ vertices for constructing the stage i vertices of G_S. Allocated vertices of each stage are indexed in decimal ordering starting from 1 to $2^{|V_R^i|} - 1$. The positions of the 1-bits in the binary representation of each vertex $v_k^i \in V_S^i$ determine the subset $\mathcal{S}_k^i$ of vertices (switches) at stage i of G_R that it represents. For example, at stage 2 of the routability graph shown in Fig. 15, there are 4 vertices 0, 1, 2, 3 corresponding to switches 36, 37, 38, 39, respectively. Therefore, at stage 2 of the corresponding solution graph shown in Fig. 16, there are $2^4 - 1 = 15$ vertices labeled in binary 0001 through 1111, representing all possible subsets of the set of 4 vertices in the routability graph. As is also seen in Fig. 16, $v_{13}^2 \in V_S^2$ of G_S represents the vertex subset $\mathcal{S}_{13}^2 = \{s_0^2, s_2^2, s_3^2\} = \{36, 38, 39\}$ of V_R^2 since 13 = "1101" in binary.

A vertex $v_k^i \in V_S^i$ only if there exists at least one feasible adaptive route $R_1 R_2 \ldots R_n$ which can forward the message initiated from the source processor to exactly one of the switches in $\mathcal{S}_k^i$ through $R_1 R_2 \ldots R_{i-1}$. Here, feasibility refers to the fact that the remaining $n - i$ route words $R_i R_{i+1} \ldots R_n$ can forward the message packets at all switches in $\mathcal{S}_k^i$ to the destination processor. An edge $e = < v_k^i, v_\ell^{i+1} > \in E_S$ only if there exists at least one feasible route whose stage-i routing word R_i forwards the message packets at all switches in $\mathcal{S}_k^i$ to exactly one of the switches in $\mathcal{S}_\ell^{i+1}$. Each edge e is associated with a label $\ell_S[e]$ which corresponds to the maximal routing word which achieves the above mentioned message forwarding. Here, maximality refers to the routing word with maximum number of 1's. Hence, the sequence of labels (routing words) on the edges of each distinct path from v_1^1 to v_1^n constitutes a feasible route from the source switch to the destination switch. Each one of these feasible routes appended with the pre-determined routing word R_n from the destination switch to the destination processor constitutes a feasible adaptive route from the source processor to the destination processor.

For example, in Fig. 16, the vertex $v_5^2 = 0101$ at stage 2 has an outgoing edge with the label 11110000 to the vertex $v_5^3 = 0101$. This edge indicates that from the set of switches $\mathcal{S}_5^2 = \{s_0^2, s_2^2\} = \{36, 38\}$ at stage 2 of G_R, we can reach the set of switches $\mathcal{S}_5^3 = \{s_0^3, s_2^3\} = \{40, 42\}$ at stage 3 of G_R by the routing word 11110000 which can be verified from Fig. 15. So, the set of switches reached from the switch set $\{36, 38\}$ by the routing word 11110000 is the *union* of the sets reached from all members.

Fig. 12 illustrates the pseudo-code of the algorithm for creating the solution graph. The first outer *for-loop* (lines 1–5) and statement at line 6 perform the necessary initializations. Here, $InAdj$ and $OutAdj$ denote the adjacency lists of the vertices for their incoming and outgoing edges in G_S, respectively.

The second outer *for-loop* (lines 7–21) performs a forward pass over the vertex stages starting from the only active vertex v_1^1 at stage 1 which corresponds to the source switch. In this *for-loop*, only active vertices are processed at each stage i to determine the active vertices at the following stage $i+1$ and create the edges between the active vertices at stages i and $i+1$. At line 9, P_a is an m-bit binary number whose 1-bit positions correspond to the common ports of the switches in $\mathcal{S}_a$ which can be used to reach the destination. In the *for-loop* at lines 10–21, all possible route words corresponding to the 1-bit positions of P_a are enumerated and processed. For a P_a with $1 \leq k \leq m$ 1-bits, $2^k - 1$ route

ALL_FEASIBLE_ROUTES(G_R)
1 **for** $i \leftarrow 1$ **to** n **do**
2 allocate $|V_S^i| = 2^{|V_R^i|} - 1$ nodes $\{v_j^i\}_{j=1}^{|V_S^i|}$ for V_S^i
3 **for** $j \leftarrow 1$ **to** $|V_S^i|$ **do**
4 $mark[v_j^i] \leftarrow$ INACTIVE;
5 $InAdj[v_j^i] \leftarrow \emptyset$; $OutAdj[v_j^i] \leftarrow \emptyset$;
6 $mark[v_1^1] \leftarrow$ ACTIVE;
7 **for** $i \leftarrow 1$ **to** $n-1$ **do**
8 **for** each ACTIVE vertex $a \in V_S^i$ **do**
9 $P_a \leftarrow \bigwedge_{u \in \mathcal{S}_a} ports_R[u]$;
/* "$\bigwedge$" : bitwise AND operation */
10 **for** each possible routing word $R_i \in$ {1-bit position combinations of P_a} **do**
11 $S_R^{i+1} \leftarrow \emptyset$;
12 **for** each stage-i vertex $u \in \mathcal{S}_a$ of V_R^i **do**
13 **for** each $w \in Adj_R[u]$ such that $\ell_R[< u, w >] \bigwedge R_i \neq 0$ **do**
14 $S_R^{i+1} \leftarrow S_R^{i+1} \cup \{w\}$;
15 find the vertex $v \in V_S^{i+1}$ where $\mathcal{S}_v = S_R^{i+1}$;
16 **if** $v \notin OutAdj[a]$ **then**
17 $mark[v] \leftarrow$ ACTIVE;
18 $OutAdj[a] \leftarrow OutAdj[a] \cup \{v\}$;
$InAdj[v] \leftarrow InAdj[v] \cup \{a\}$;
19 $\ell_S[< a, v >] \leftarrow R_i$;
20 **else** /* edge $< a, v >$ already exists */
21 $\ell_S[< a, v >] \leftarrow \ell_S[< a, v >] \bigvee R_i$;
/* "$\bigvee$" : bitwise OR operation */
22 **for** $i \leftarrow n-1$ **downto** 2 **do**
23 **for** each ACTIVE vertex $a \in V_S^i$ **do**
24 **if** $OutAdj[a] = \emptyset$ **then**
25 **for** each $u \in InAdj[a]$ **do**
26 remove vertex a from $OutAdj[u]$;
/* remove edge $< u, a >$ */
27 $InAdj[a] \leftarrow \emptyset$; $mark[a] \leftarrow$ INACTIVE;
28 **return** $G_S = (V_S, E_S)$, where
$V_S = \{v : mark[v] =$ ACTIVE $\}$
$E_S = \{< u, v >: u, v \in V_S \text{ and } v \in OutAdj[u]\}$

Figure 12: The algorithm for generating the solution graph $G_S = (V_S, E_S)$

words are generated by fixing the bit positions corresponding to the 0-bit positions of P_a to all 0's and enumerating $2^k - 1$ distinct non-zero binary numbers from 1 to $2^k - 1$ on the bit positions corresponding to the 1-bit positions of P_a. The set $S_R^{i+1} \subseteq V_R^{i+1}$ of switches reached from the switch set $\mathcal{S}_a \subseteq V_R^i$ by the routing word R_i is constructed in the *for-loop* at lines 12–14. The search operation at line 15 can be efficiently performed in constant time by exploiting the proposed vertex encoding in G_R and G_S. The *if-clause* at lines 16–19, adds the edge $e =< a, v >$ to E_S, activates vertex v at stage $i+1$ of G_S, and initializes the route-word label $\ell_S[e]$ of edge e. The *else* clause at lines 20–21 ensures the maximality of the route-word label $\ell_S[e]$.

The solution graph G_S generated at the end of the second outer *for-loop* (lines 7–21) may contain vertices and edges which are not involved in any feasible solution path from the source to the destination because of the vertices at later stages which do not have any outgoing edges. These infeasible vertices and edges are removed in the last outer *for-loop* (lines 22–27) in order to reduce the computational complexity of the dynamic programming algorithm to be executed in the next phase. The backward processing order over the vertex stages of G_S ensures the feasibility of all remaining vertices and edges.

MAX_ADAPTIVE_ROUTE(G_S)
1 $\text{ADP}[v_1^n] \leftarrow 1$;
2 **for** $i \leftarrow n-1$ **downto** 1 **do**
3 **for** each vertex $u \in V_S^i$ **do**
4 $\text{ADP}[u] \leftarrow 0$;
5 **for** each $v \in OutAdj[u]$ **do**
6 $adp \leftarrow |\ell_S[< u, v >]| \times \text{ADP}[v]$;
7 **if** $adp > \text{ADP}[u]$ **then**
8 $\text{ADP}[u] \leftarrow adp$;
9 $next[u] \leftarrow v$;
10 $u \leftarrow v_1^1$;
11 **for** $i \leftarrow 1$ **to** $n-1$ **do**
12 $v \leftarrow next[u]$;
13 $R_i \leftarrow \ell_S[< u, v >]$;
14 $u \leftarrow v$;
15 $R_n \leftarrow \ell_S[< v_1^n, d >]$; /* d : destination processor */
16 **return** $R = R_1 R_2 \ldots R_{n-1} R_n$ with $N_{\mathtt{path}} = \text{ADP}[v_1^1]$

Figure 13: Algorithm for determining maximum adaptive route in an n-stage solution graph $G_S = (V_S, E_S)$.

4.3 Maximizing Adaptivity

Once the solution graph is created, the maximally adaptive route may be found by finding a path from source to destination node in the solution graph that maximizes the *product* of the adaptivity values of edges. The adaptivity of an edge $e \in E_S$ is defined as the number of 1-bits (i.e., $|\ell_S[e]|$) in its edge label $\ell_S[e]$, representing the number of *common* output port choices of the switches in $\mathcal{S}_a$ that can be used to lead the messages at those switches to the destination. The adaptivity of a path from source to destination is the multiplication of the adaptivity values of edges on the path. Hence, the problem reduces to finding an optimal path from v_1^1 to v_1^n in G_S with maximum adaptivity. As an example, in Fig. 16, the top most path has a product cost (adaptivity) of $1 \times 4 \times 1 = 4$ (i.e., $|00010000| \times |11110000| \times |10000000|$), which indicates that the given sequence of routing words result in 4 different routes between source and destination processors. Likewise, the bottom most path has a product cost of $4 \times 4 \times 1 = 16$ (i.e., $|11110000| \times |11110000| \times |10000000|$), which shows that the given sequence of routing words result in 16 different routes between source and destination processors. Note that the bottom most path happens to be the solution with the maximum adaptivity; there are no more than 16 distinct shortest paths from processor 4 to 30, as can be verified from Figs. 14 and 15. Therefore, the route header encoding with the maximum adaptivity is $R_1 = 11110000$, $R_2 = 11110000$, $R_3 = 10000000$, and

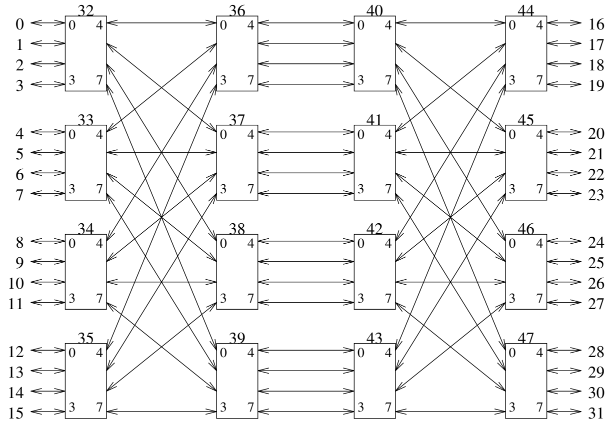

Figure 14: A 32 processor node bidirectional multistage network (BMIN)

$R_4 = 01000000$ in this example.

A *dynamic programming* [17, 18] formulation for an n-stage solution graph G_S is obtained by first noticing that every source to destination path is a result of a sequence of $n-2$ decisions. The i-th decision involves determining which vertex in V_S^i $(1 < i < n)$ is to be on an optimal path.

Let $\text{ADP}[v_k^i]$ denotes the adaptivity of the optimal path $p(v_k^i, v_1^n)$ from the stage-i vertex $v_k^i \in V_S^i$ to the destination switch v_1^n. Then, the optimal substructure property gives the recursive formulation

$$\text{ADP}[v_k^i] = \max_{v_\ell^i \in V_S^{i+1} \wedge <v_k^i, v_\ell^{i+1}> \in E_S} \{ |\ell_S[< v_k^i, v_\ell^{i+1} >]| \times \text{ADP}[v_\ell^{i+1}] \} \quad (1)$$

Since the adaptivity of the optimal path from destination switch v_1^n to the destination processor is 1, the adaptivity of optimal routes from all vertices of G_S can easily be computed by performing a backward pass over the vertex stages of G_S as shown in Fig. 13. $\text{ADP}[v_1^1]$ contains the adaptivity value of the optimal routing solution(s) when the first *for-loop* (lines 2–9) terminates. In this *for-loop*, $next$ attribute for each vertex is computed to enable the construction of an optimal routing in the second outer *for-loop* (lines 11–14). This *for-loop* constructs an optimal routing by simply following the *next* fields of the vertices in forward direction starting from the source switch at stage 1.

5 Conclusion

In this paper, we presented the first attempt to combine the source routing and adaptive routing methods, referred to as the *adaptive source routing* (ASR) method. We showed that the route and the adaptivity of message packets are determined at the source processor node, and that packets can be routed in a fully adaptive, or partially adaptive, or oblivious manner in the same network, at the same time. We described how the ASR method may support multiple types of network traffic, in–order delivery of multiple packets to avoid over–taking, and network partitioning. The source routing nature of the ASR method eliminates the need for routing tables on the switch chips which may limit scalability and occupy valuable real–estate on silicon. We presented performance comparison of adaptive versus oblivious routing networks. We found adaptive routing to be generally superior to oblivious routing for both permutation and random traffic. We presented an algorithm that generates maximally adaptive routing headers for the message packets. The algorithm is applicable to multistage networks in general, including faulty networks and irregular topologies.

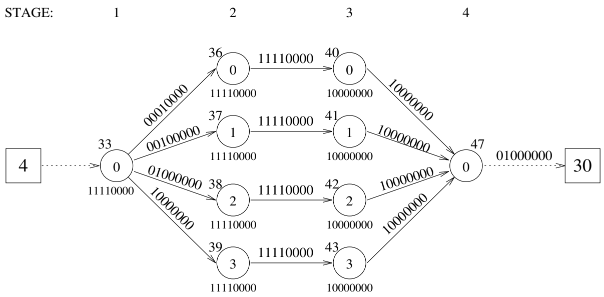

Figure 15: The routability graph $G_R = (V_R, E_R)$ for the processor pair $(4, 30)$.

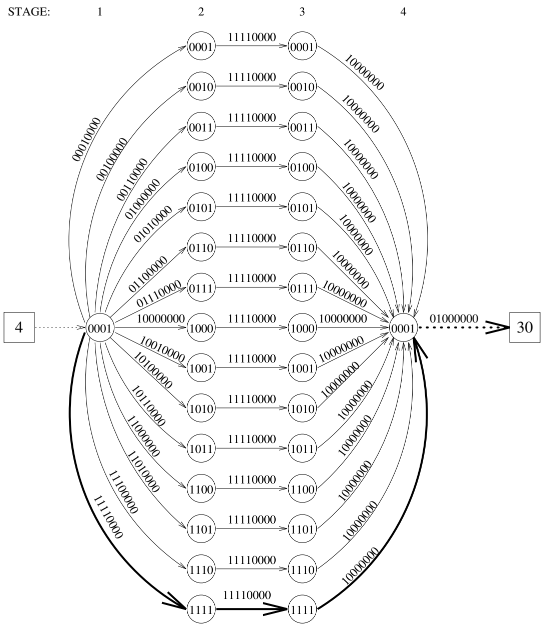

Figure 16: The solution graph $G_S = (V_S, E_S)$ for the processor pair $(4, 30)$.

Acknowledgements: We are thankful to Mike Rosenfield and anonymous reviewers for their helpful comments.

References

[1] C. B. Stunkel, D. G. Shea, and B. Abali et al., "The SP2 high-performance switch," *IBM Systems Journal*, vol. 34, no. 2, pp. 185–204, 1995.

[2] Cray Research Inc., *Cray T3D System Architecture Overview*, 1993.

[3] S. Scott and G. Thorson, "Optimized Routing in the Cray T3D," *Lecture Notes in Computer Science, Springer–Verlag*, vol. 853, pp. 281–294, 1994.

[4] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Trans. on Computers*, vol. C-34, pp. 892–901, Oct. 1985.

[5] Thinking Machines Corporation, *Connection Machine CM-5 Technical Summary*, November 1993.

[6] Intel Corporation, *Paragon XP/S Product Overview*, 1991.

[7] S. A. Felperin, L. Gravano, G. D. Pifarre, and L. C. Sanz, "Routing techniques for massively parallel communication," *Proc. IEEE*, vol. 79, pp. 488–503, April 1991.

[8] S. Konstantinidou and L. Snyder, "Chaos router: architecture and performance," in *Proc. 18th Ann. Int. Symp. on Computer Architecture*, pp. 212–221, 1991.

[9] R. V. Boppana and S. Chalasani, "A comparison of adaptive wormhole routing algorithms," in *Proc. 20th. Ann. Int. Symp. on Computer Architecture*, pp. 351–360, May 1993.

[10] C. B. Stunkel, D. G. Shea, B. Abali, M. M. Denneau, P. H. Hochschild, D. J. Joseph, B. J. Nathanson, M. Tsao, and P. R. Varker, "Architecture and implementation of Vulcan," in *Proc. 8th Int. Parallel Processing Symp.*, pp. 268–274, April 1994.

[11] B. Abali and C. Aykanat, "Routing Algorithms for IBM SP1," *Lecture Notes in Computer Science, Springer–Verlag*, vol. 853, pp. 161–175, 1994.

[12] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak, "The network architecture of the Connection Machine CM-5," in *Proc. 1992 Symp. Parallel Algorithms and Architectures*, pp. 272–285, ACM, 1992.

[13] J. Beecroft, M. Homewood, and M. McLaren, "Meiko CS-2 interconnect Elan-Elite design," *Parallel Computing*, vol. 20, pp. 1627–1638, Nov. 1994.

[14] P. Kermani and L. Kleinrock, "Virtual cut-through: A new computer communications switching technique," *Computer Networks*, vol. 3, pp. 267–286, Sept. 1979.

[15] W. J. Dally, "Performance analysis of k-ary n-cube interconnection networks," *IEEE Trans. on Computers*, vol. 39, pp. 775–785, June 1990.

[16] M. Snir, P. Hochschild, D. D. Frye, and K. J. Gildea, "The communication software and parallel environment of the IBM SP2," *IBM Systems Journal*, vol. 34, no. 2, pp. 205–221, 1995.

[17] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*. Maryland: Computer Science Press, 1989.

[18] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. NY: The MIT Press, 1991.