ADAPTIVE SOURCE ROUTING AND ROUTE GENERATION FOR MULTICOMPUTERS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER

ENGINEERING AND INFORMATION SCIENCE

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BİLKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Yücel Aydoğan

July, 1995

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Cevdet Aykanat (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. İlyas Çiçekli

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Tuğrul Dayar

Approved for the Institute of Engineering and Science:

Prof. Mehmet Baray
Director of the Institute

ABSTRACT

ADAPTIVE SOURCE ROUTING AND ROUTE GENERATION FOR MULTICOMPUTERS

Yücel Aydoğan
M.S. in Computer Engineering and Information Science
Advisor: Assoc. Prof. Cevdet Aykanat
July, 1995

Scalable multicomputers are based upon interconnection networks that typically provide multiple communication routes between any given pair of processor nodes. In such networks, the selection of the routes is an important problem because of its impact on the communication performance. We propose the adaptive source routing (ASR) scheme which combines adaptive routing and source routing into one which has the advantages of both schemes. In ASR, the degree of 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. The ASR scheme permits any network topology to be used provided that deadlock constraints are satisfied. We evaluate and compare performance of the adaptive source routing and non–adaptive randomized routing by simulations. Also we propose an algorithm to generate adaptive routes for all pairs of processors in any multistage interconnection network. Adaptive routes are stored in a route table in each processor's memory and provide high bandwidth and reliable interprocessor communication. We evaluate the performance of the algorithm on IBM SP2 networks in terms of obtained bandwidth, time to fill in the route tables, and efficiency exploited by the parallel execution of the algorithm.

Keywords: Adaptive Routing, Multicomputers, Interconnection Networks, Parallel Processing

ÖZET

ÇOKİŞLEMCİLİ BİLGİSAYARLARDA UYARLANABİLİR KAYNAK DAĞITIMI VE YOL ÜRETİMİ

Yücel Aydoğan
Bilgisayar ve Enformatik Mühendisliği, Yüksek Lisans
Danışman: Doç. Dr. Cevdet Aykanat
Temmuz, 1995

Ölçeklenebilir çokişlemcili bilgisayarlar herhangi iki işlemci arasında birden fazla haberleşme yolu sağlayan bağlantı ağları üzerine kurulan sistemlerdir. Bu tür ağlarda yol seçimi haberleşme performansını etkileyen önemli bir etkendir. Uyarlanabilir Kaynak Dağıtımı (UKD), uyarlanabilir dağıtım ve kaynak dağıtımı yöntemlerini birleştiren ve her ikisinin de avantajlarına sahip olan bir dağıtım yöntemi olarak önerilmiştir. Her paket tam uyarlanabilir, kısmi uyarlanabilir yada uyarlamasız şekilde yöneltilir. UKD yöntemi kilitlenme sınırlamalarının sağlandığı herhangi bir ağ topolojisi kullanımına izin verir. Uyarlanabilir kaynak dağıtımı ve uyarlamasız rastlantısal dağıtım yöntemleri benzetim yapılarak karşılaştırılmıştır. Ayrıca çokişlemcili bilgisayar ağlarında işlemciler arasında uyarlanabilir yollar üreten bir yöntem önerilmiştir. Üretilen uyarlanabilir yollar her işlemcinin belleğindeki yol çizelgelerinde saklanır. Bu yöntem yüksek veri iletişim kapasitesi ve işlemciler arası güvenilir iletişimi sağlar. Önerilen yöntem ile IBM SP2 çokişlemcisi ağları kullanılarak deneyler yapılmış ve sağlanan veri iletişim kapasitesi ve işlemcilerde yol çizelgesi oluşturma zamanları ölçülmüştür. Yöntemin çokişlemcili bilgisayarlarda paralel işlemesi ile elde edilen verim de deneysel olarak sunulmuştur.

Anahtar Sözcükler: Uyarlanabilir Dağıtım, Çokişlemcili Bilgisayarlar, Bağlantı Ağları, Paralel İşleme

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to Dr. Bülent Abalı for his invaluable guidance, suggestion, and encouragement throughout the development of this thesis. I would like to thank my advisor Dr. Cevdet Aykanat for his guidance, suggestions, and contributions. I would like to thank Dr. İlyas Çiçekli for reading and commenting on the thesis. I would also like to thank Dr. Tuğrul Dayar for reading and commenting on the thesis. I owe special thanks to Dr. Craig B. Stunkel at IBM T.J. Watson Research Center for providing figures for the thesis.

Bu çalışmamı
anneme ve *babama*
adıyorum

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Scalable multicomputers are based upon interconnection networks that typically provide multiple communication routes between any given pair of processor nodes. Interconnection networks [2, 7] can be classified according to their topology. A *static network topology* is one that does not change after the machine is built. Ring, star, mesh, and hypercubes are some of the examples for static interconnection topologies. Parallel computers employing static interconnection networks can have very good performance on specific problems to which their network topologies are well matched. However, it is hard to achieve a multipurpose highly parallel system using a fixed interconnection topology short of an all-to-all network. This difficulty has given rise to much work on *dynamic interconnection networks*. Bus networks, multistage switching networks, and crossbar networks are examples for dynamic interconnection topologies. A bus network is very much like a party-line telephone. A crossbar network, on the other hand, is like a private exchange that allows any processor to contact any other non busy processor at any time. A multistage switching network falls in between these two extremes.

Multiple routes provided by interconnection networks and routing algorithms play important role in providing low latency, high bandwidth, and reliable interprocessor communication. Examples of interconnection networks used in commercial machines are the IBM SP2 multistage interconnection network [1, 27], Cray T3D 3–dimensional torus [12], and the Connection Machine fat tree [4, 16].

Given an interconnection network, a distance measure D can be defined on it. A routing algorithm is said to be *minimal* [22] if for every sequence of nodes

$a_0, ..., a_k$ 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 algorithm 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 algorithm 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 in the network. Minimal fully adaptive algorithms do not impose any restrictions on the choice of shortest paths to be used in routing messages; in contrast, partially adaptive minimal routing algorithms allow only a subset of available minimal paths in routing messages. The well known e-cube [5] algorithm is an example of non-adaptive routing algorithms [5, 6] since it has no flexibility in routing messages.

Usually, two kinds of routing algorithms 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. If the messages are of variable size, *wormhole* routing can be used instead. In wormhole routing, 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 until the link is freed.

Adaptive routing schemes are employed in some networks to eliminate congestion by finding alternate routes to destinations [3, 4, 6, 13]. On the other hand, some networks trade off performance for simplicity of switch design between flexible choice of topology by employing non–adaptive routing schemes such as the *source routing* scheme used in SP2 [1, 27]. In the source routing scheme, the packet route is deterministic and it is completely determined at the source processor sending the packet. In the first part of this thesis, we propose the *adaptive source routing* (ASR) scheme which combines adaptive routing and the source routing to exploit the advantages of both schemes. In ASR, the degree of adaptivity of each packet is determined at the source processor node. 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. Adaptive

source routing is a superset of the source routing scheme used in IBM SP2 multicomputer, thus ASR is backward compatible with the SP2 routing scheme. The ASR scheme also permits any network topology to be used provided that deadlock constraints are satisfied, unlike other adaptive routing schemes.

The ASR scheme has the advantages of both adaptive routing and source routing schemes as it combines both. However, the problem we address when we make use of adaptivity is the assignment of outputs to the packets in the switches. The switch must –adaptively and in a conflict free manner– assign an output to each packet from a set of permitted outputs specified in the packet header, with the consideration that multiple packets may be waiting for an output assignment. This problem can be formulated as a *maximum matching problem* in a bipartite graph [19, 23, 28]. Polynomial time algorithms exist for solving maximum matching problem [19, 23] however these algorithms require sophisticated data structure that are difficult and impractical to implement in switch hardware. We propose a maximum matching heuristic that can be implemented in terms of primitive logic operations AND, OR, NOT, and Rotate which makes it possible to implement in switch hardware.

The performance of the ASR scheme is evaluated by a network simulator. We describe the network simulator and present the experimental results of simulations on a sample network. We compare the ASR scheme with non–adaptive random routing scheme by giving the average latency as a function of average load in the network for different sized networks.

The second part of this thesis is on route table generation for multicomputers based upon any interconnection network. Packets in interconnection networks that have a regular structure, make use of the regular structure in the interconnection topology to determine the possible ports that lead the packet to correct destination at each stage. The main disadvantage of such networks is the restriction on the number of processors that can be connected to maintain the interconnection structure. The requirement is that the number of processors should generally be a power of 2. IBM SP1 and SP2 multicomputers make use of multistage interconnection networks that provides a wide flexibility in the number of processors connected because of the interconnect technology used. However such networks need not have any structure in the interconnection topology which complicates route decision at each stage.

We propose an algorithm for route generation in any multistage interconnection network regardless of the regularity in the topology. Generated routes for each pair of source–destination processors are adaptive routes that provide multiple distinct paths and are stored in a route table in each processor's memory. We implemented and evaluated the performance of the proposed algorithm on IBM SP2 [26] interconnection networks. The SP2 switch architecture and the network implementations are introduced. The experimental results show how much the generated adaptive routes make use of the physically existing paths with the execution times on different sized networks. We also give an improvement in the algorithm and the results of the improvement. The parallel version of the proposed algorithm is also presented.

The organization of the thesis is as follows: we describe the proposed adaptive source routing scheme and the maximum matching heuristic in a bipartite graph in Chapter 2. The network simulator and the simulation results of ASR on a sample network are given in Chapter 3. The proposed route generation algorithm for any interconnection network and experimental results on IBM SP2 network samples with the parallel route generation algorithm are presented in Chapter 4. Finally, conclusions are given in Chapter 5.

Chapter 2

Adaptive Source Routing (ASR)

In adaptive routing networks, message packets make use of multiple paths between source–destination node pairs [6]. Switches alleviate the congestion problem by sending packets from less busy alternate routes. For example, a busy output port will cause an adaptive routing switch to use another output port in routing a packet to its destination. This means that the adaptive routing switch must know which of its outputs lead to the intended destination. For this reason, a common requirement for all adaptive networks is a regular, simply described network topology such as a hypercube, mesh, k–ary n–cube, or a fat tree [3, 4, 6, 13, 16]. The switches then have an implicit knowledge of the topology, and therefore can route packets using shortest paths. For example, in a 2–dimensional mesh topology, each switch knows that a node at the upper right corner of the network can be reached by sending a packet either in the North or East direction. In an alternative approach, routing tables may be put in each switch, however this would be impractical since it would occupy valuable real–estate on the switch chips.

In the source routing scheme, unlike adaptive routing, switches need not know the topology; the source processor determines the route and encodes the routing information in the packet header, which is then used by the switches. Thus, switches make routing decisions purely based on local information. For example, in the SP2 multistage network, which consists of 8×8 switches [27], the packet header for an n–hop message initially contains 3–bit routing bytes $R_1, R_2, \ldots, R_n$ as shown in Fig. 2.1. Each routing byte indicates a switch port numbered from 0 to 7. The source processor determines the route and puts

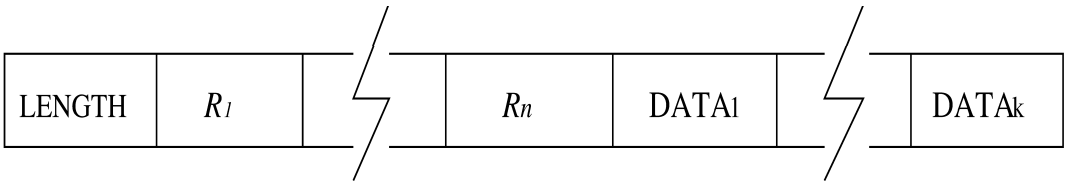

Figure 2.1. Message Packet Format

respective bytes in the header. As the message packet proceeds in the network, each switch examines the first byte and forwards the packet through the indicated output port. The switch also strips off that first byte before forwarding the packet to the next level in the network. Thus the packet contains no routing information upon arriving at its destination. In SP2, routing bytes are computed only once and then kept in a route table in each processor node. Keeping route tables in processors is inexpensive since processors already have large memory. The algorithm for creating the routing tables is described in [1]. The route table approach enables routing to be done in a *topology independent* fashion which is important in practice. Any network topology is possible to implement without having to change the hardware or the routing algorithms, provided that cost, performance, and deadlock constraints are satisfied. Furthermore, faulty links and switches are handled easily by modifying routing tables. In that respect, source routing is more flexible than adaptive routing.

2.1 Adaptive Source Routing Scheme

In the adaptive source routing scheme proposed in this thesis, the packet format is similar to that of SP2. However, each routing byte indicates a set of possible output ports, rather than a specific output port. Each m–bit byte has the format $R = r_{m-1}r_{m-2}\ldots r_0$, where m is the number of switch ports. One bits indicate the set of outputs that the switch is permitted to route the packet through. Routing header is determined by the source processor sending the message packet, as in source routing. Each switch examines the first byte and adaptively selects from one of permitted outputs by considering the local traffic, and then forwards the packet to the next level in the network. The switch also strips off that first byte before forwarding the packet as in source routing. For example, in a network constructed of 8×8 switches such as in SP2, a packet header may consist of bytes $R_1 = 00001111, R_2 = 11000000, R_3 = 01000000$, which tells to the first switch that the packet may be routed through one of

the four ports 0–3, and to the next switch that through one of the ports 6, 7, and to the last switch that through the port 6. Thus, the number of distinct paths a packet may follow from source to destination is

$$N_{\text{path}} = |R_1| \times |R_2| \times \cdots\cdots \times |R_{n-1}| \times |R_n| \tag{2.1}$$

where $|R_i|$ is defined as the number of ones in the routing byte R_i. Obviously N_{path} paths must exist between the source and destination, and any combination of the outputs specified in the header must correctly lead the packet to its destination. In Chapter 3 of the thesis, we describe only the switch architecture and simulations of the proposed routing scheme. The algorithms we proposed for determining routing headers for multistage interconnection networks will be described in the later chapters and the experimental results on SP2 interconnection networks are also presented.

Each source processor can determine the degree of adaptivity of each message packet by varying N_{path}. If $N_{\text{path}} = 1$, then the adaptivity is zero; the packet is to be routed through a single deterministic path. This case is equivalent to the routing scheme used in SP2 [27]. Furthermore, $N_{\text{path}} = 1$ case may be useful for several other applications. When interprocessor communication patterns are known in advance, optimal route between each processor pair may be selected to minimize congestion. A heuristic for solving that optimization problem is described in [1]. When operating in the SIMD mode such that permutations to be realized by the network are known in advance, single deterministic routes may be selected. $N_{\text{path}} = 1$ case may also be useful for diagnosis of the interconnection network, where faulty links or switches are to be determined; for example a source processor may identify faulty elements by circulating packets through deterministic paths. If $N_{\text{path}} = \max$, then the adaptivity is maximum and packets may reap performance benefits of full adaptivity. This case is useful when some switches get congested due to non–uniform message traffic and difficult communication patterns. If $1 < N_{\text{path}} < \max$, then each packet is routed in a *partially–adaptive* manner, where only a subset of all possible paths is utilized. This case may be useful when the network is to be logically partitioned among multiple parallel tasks so that their respective communications do not influence each other; using the ASR scheme, each packet may be forced to remain in its partition, however routed in a fully adaptive manner within the partition.

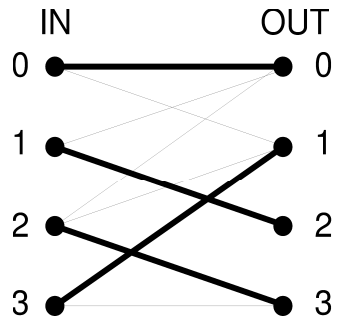

Figure 2.2. A bipartite graph and its matching

2.2 The Matching of Packets and Outputs

In this section, we address the problem of assigning outputs to the packets. Each packet in a switch has a set of permitted outputs specified in the packet header leading the packet to its destination in an adaptive manner. The switch must assign an output to each packet considering the permitted set of outputs. The switch must also consider that multiple packets may be waiting for an output assignment. The assignment of outputs to packets must be adaptive and conflict free. This problem can be formulated as a *maximum matching problem* in a bipartite graph [19, 23, 28].

2.2.1 Maximum Matching Problem

A graph $G(V_1, V_2, E)$ is called a *bipartite graph* if its vertex set V is the disjoint union of sets V_1 and V_2, and every edge in E has the form (v_1, v_2), where $v_1 \in V_1$ and $v_2 \in V_2$. If $G(V_1, V_2, E)$ is a bipartite graph, a matching in G is a set of edges in G such that no two edges share a vertex. A *maximum matching* in G is defined as the matching that has as many vertices in V_1 as possible with the vertices in V_2.

The problem of matching outputs to packets can be formulated as a maximum matching problem as follows. Let $G(IN, OUT, E)$ be a bipartite graph with a set of vertices IN, OUT, and a set of edges E. Each vertex in IN represents a packet waiting to be assigned an output. Each vertex in OUT represents an output. Each edge in E represents a permitted output assignment specified in the routing byte of the packet. Let M be the set of edges in

a matching in G. In maximum matching problem, we try to maximize the cardinality of M, i.e., the number of successful output assignments in our case, so that the message bandwidth through the switch is maximized. Fig. 2.2 shows an example bipartite graph where the matching is maximum.

Note that a matching scheme is also described for the Chaos router in [13, 14]. Our scheme differs in that we try to maximize matching, whereas in their scheme, packets are assigned without consideration for the other packets waiting in the switch. Their justification was that for the hypercube topology they considered, only one packet would be in the switch even under heavy traffic conditions.

MATCH(R,passes)

```
1   Let M be an m × m matrix representing the
        matching, and M_i denote the i-th row of M,
    Let R be an m × m matrix representing the request
        matrix, and R_i denote the i-th row of R,
    Let C be an m–bit row vector
2   Initialize M using R
3   for k = 1 to passes
4       for i = 0 to m − 1
5           C ← ColumnOR(M)
6           C ← C OR R_i (complement)
7           M_i ← Rotate_Until_Zero(M_i, C)
8       endfor
9   endfor
10  return M
```

Figure 2.3. The Matching Heuristic

2.2.2 Maximum Matching Heuristic

Polynomial time algorithms exist for solving the maximum matching problem [19, 23]. However, these algorithms require sophisticated data structures which would be difficult to implement in hardware. Here, we describe a heuristic that can be implemented in terms of primitive logic operations AND, OR,

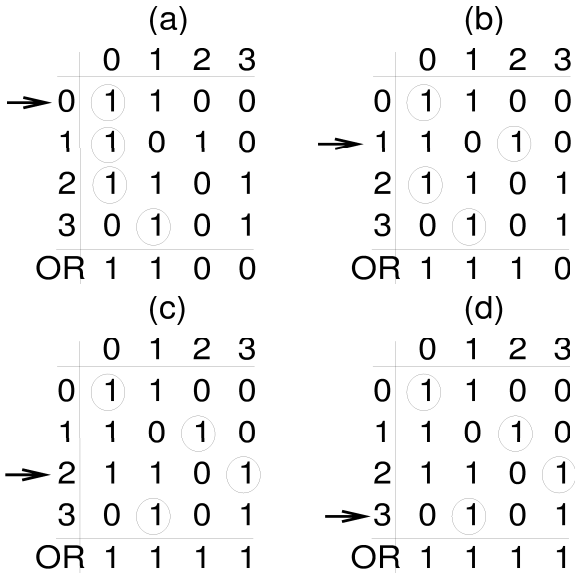

Figure 2.4. A request matrix R and finding the maximum matching

NOT, and Rotate.

The set of packets waiting for an assignment is represented by an $m \times m$ binary request matrix R as shown in Fig. 2.4(a), where m is the number of outputs. Matrix R is constructed from packets' routing bytes. Each row of R corresponds to a packet, and each column corresponds to an output. One bits in a row indicate the set of outputs that the respective packet may be routed through. An $m \times m$ binary output assignment matrix M is defined such that each row of M comprises at most 1 one bit. A one bit M_{ij} in M indicates that output j is assigned to packet i for routing. By definition M should have one bits only at places where R has one bits. In Fig. 2.4(a), the M matrix is superimposed over R, indicated by circled one bits of R. A $ColumnOR$ operation on M is defined such that M's rows are ORed column-wise, whose m–bit result C gives the set of assigned outputs (ones) and unassigned outputs (zeros) for the given M matrix. An operation called $Rotate\_Until\_Zero(M_i, C)$ is defined on m–bit row vectors M_i and C such that the one bit in M_i is aligned to a zero bit in C, i.e., M_i is rotated until the result of $M_i \;\; AND \;\; C$ is all zeros. Using the primitive operations defined, the heuristic shown in Fig. 2.3 attempts to find a maximum matching. The heuristic starts with an arbitrary matching M, then for each row M_i ($i = 0, 1, \ldots, m-1$), it does $ColumnOR$ on M finding unused outputs, and then rotates M_i to an unused output with the condition that R_i (the routing byte) has a one in that column position.

Fig. 2.4(a)–(d) illustrates the procedure: in step (a) M_0 cannot be rotated because there is no permitted free output. In step (b) M_1 is rotated to output 2. In step (c) M_2 is rotated to output 3, resulting in a maximum matching since no free outputs are left. In step (d) no change is made.

The heuristic doesn't find a matching in the strict sense because it may assign multiple packets to the same output. In that case, we assume that the switch will employ some fair arbitration policy to choose one of those packets for routing. Note that the cardinality of the matchings found by the heuristic is monotonically increasing; in each step a better solution is found or there is no change. Note also that the heuristic does not always find a maximum matching. However, at the expense of increased execution time, the procedure may be repeated few more times to improve the solution (the variable *passes* is the repeat count). The number of repetitions for finding the maximum matching depends on the request instance and there is not a bound on the number of repetitions that will yield the maximum matching.

2.2.3 Performance of Maximum Matching Heuristic

We evaluated the performance of the matching heuristic on pseudo–randomly generated request matrices R. To be able to evaluate how good the matching found by the heuristic is, we must determine the cardinality of the maximum matching that is possible in a bipartite graph G. We use the idea in [8] to determine the maximum number of vertices that can be matched in a bipartite graph as follows. Let $G = (V_1, V_2, E)$ be a bipartite graph. If $A \subseteq V_1$, then $\delta(A) = |A| - |R(A)|$, where $R(A)$ is the subset of V_2 consisting of those vertices that are adjacent to the vertices in A, is called the *deficiency of* A. The *deficiency of graph* G, denoted $\delta(G)$, is given by $\delta(G) = max\{\delta(A) \mid A \subseteq V_1\}$. The following theorem, proved in [8], gives the cardinality of the maximum possible matching in a bipartite graph.

Theorem 2.1 *Let $G = (V_1, V_2, E)$ be a bipartite graph. The maximum number of vertices in V_1 that can be matched with those in V_2 is $|V_1| - \delta(G)$. Moreover, a matching of size $|V_1| - \delta(G)$ exists.*

To illustrate the theorem, consider the bipartite graph in Fig. 2.5. Note that $\delta(\{a, b, d\}) = 2$ and this is maximum, so $\delta(G) = 2$. So $|X| - \delta(G) = 4 - 2 = 2$.

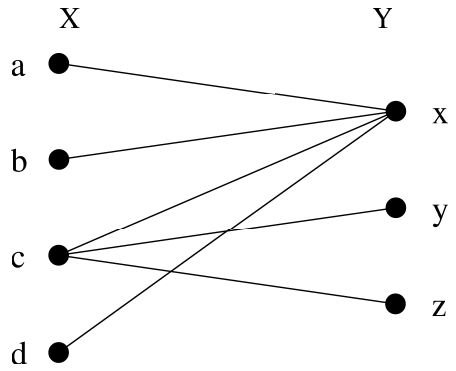

Figure 2.5. A bipartite graph with $\delta(G) = 2$

The largest subset of X that can be matched has two elements. An example of such a set is $\{a, c\}$.

We generated a number of request matrices for the heuristic and compared the matching found by the heuristic with the possible maximum matching given by Theorem 2.1. Table 2.1 shows that the heuristic finds a maximum matching over 88% of the time using one pass and 98% of the time using two passes for 4×4 switches. For 8×8 and 16×16 switches, our matching heuristic finds a maximum matching over 86% of the time using two passes. It is worth noticing that the percentage of finding a maximum-2 matching is very low (2%) using one pass and is 0% using two passes. So the matching found by the proposed heuristic is either a maximum matching with a very high probability or a maximum-1 matching with a considerably low probability.

Implementation of the heuristic in terms of primitive logic operations AND, OR, NOT, and Rotate makes it possible to implement the heuristic algorithm in switch hardware unlike the algorithms for solving maximum matching problem which require sophisticated data structures.

Switch Size	4×4		8×8		16×16	
Matching	1 pass	2 pass	1 pass	2 pass	1 pass	2 pass
maximum	0.88	0.98	0.59	0.86	0.59	0.87
maximum-1	0.12	0.02	0.39	0.14	0.39	0.13
maximum-2	0.0	0.0	0.02	0.0	0.02	0.0

Table 2.1. Performance of the matching heuristic. Percentage of the time a maximum, or a maximum-1, or a maximum-2 matching is found.

Chapter 3

Simulation of Adaptive Source Routing

In Section 2.1 we described the adaptive source routing (ASR) scheme. We developed a network simulator for evaluating the performance of the ASR scheme and we present the simulation results. In this chapter we introduce the switch architecture used in the network simulator. We present the algorithm for the simulator and describe how packets are generated to be able to simulate different message traffic and load in the network. Simulation results are given at the end of the chapter.

3.1 The Switch Architecture

In the simulations we used 2×2 switches. The switch consists of a buffer at each input and output port, and a 2×2 crossbar interconnecting input buffers to output buffers. The main operation of the switch is to forward the packets in the input buffers to the output buffers in a profitable manner. The unit of transfer between the buffers is a packet. A *cycle* is defined here as the time required for a packet to move from one buffer to another. In each cycle, either a *forwarding* or a *blocking* operation takes place. In forwarding, a packet moves forward entirely from an input buffer to the assigned output buffer in a switch or through the links between the switches i.e., from an output buffer of a switch to the input buffer of the connected one. In blocking, a packet is blocked in the buffers waiting for the availability of the buffer it is assigned to. The 2×2 size of the crossbar in the switch simplifies the matching heuristic described in

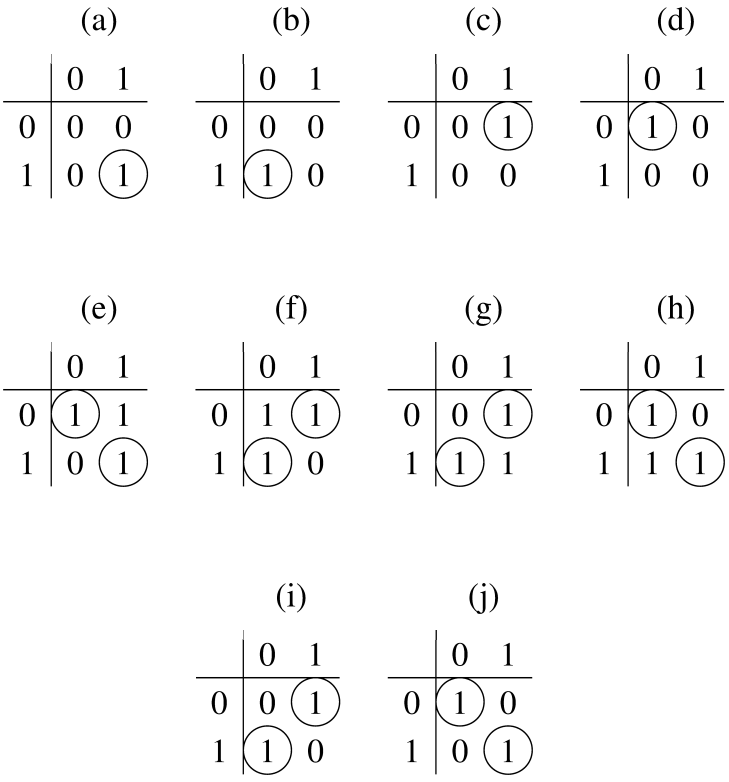

Figure 3.1. Maximum matchings for some of the possible request matrices for 2×2 switches

Section 2.2.2 considerably; routing decision is made by a table lookup since the number of possible cases is small, and the matchings are always the maximum. The set of packets waiting for an assignment is represented by a 2×2 binary request matrix (see Fig. 3.1). Each row of the request matrix corresponds to a packet, and each column corresponds to an output. One bits in a row indicate the set of outputs that the respective packet may be routed through. The assignment matrices are superimposed over the request matrices in Fig. 3.1, indicated by circled one bits. Some entries of the table used to make the assignment of outputs to packets are in Fig. 3.1. In these request matrices, the packets' permitted set of outputs make it possible to make a maximum matching of outputs to packets in a deterministic way. The assignment for each case for obtaining a maximum matching of outputs to packets is unique and straight forward. However, assignments in the remaining entries of the table are not unique. The switch must make a decision considering the local traffic and the starvation problem of some packets. These entries are in Fig. 3.2. When only one packet is waiting for an assignment, as in Fig. 3.2(a)–(b), two

(a)

	0	1
0	0	0
1	1	1

(b)

	0	1
0	1	1
1	0	0

(c)

	0	1
0	1	1
1	1	1

(d)

	0	1
0	0	1
1	0	1

(e)

	0	1
0	1	0
1	1	0

Figure 3.2. Request matrices for 2×2 switches for which the maximum matchings may change

different assignments can be made. The switch decides which output to assign to the packet according to the local traffic i.e., the available output buffer is assigned to the packet. In case both output buffers are available, the output buffer is chosen in a round robin fashion for uniform distribution of packets to all links and switches in the network. There may be conflicting requests of output buffers. More than one packet may demand the same output buffer as in Fig. 3.2(d)–(e). These conflicts are resolved in a round robin fashion to prevent starvation of some packets. Fig. 3.2(c) shows the case that all the packets are permitted to use all output buffers. In this case, a maximum matching is found according to the available output buffers resolving the conflicts among the packets in a round robin fashion.

3.2 The Network

In the simulations, we used the Beneš interconnection network since it has been extensively studied for synchronous and asynchronous communication [7], and since it is a multistage network which provides multiple paths between source–destination pairs as in the SP2 interconnection network. Although, Beneš networks are generally considered for synchronous communication in SIMD machines with a centralized network control [2], here we will consider it for asynchronous communication in MIMD machines with a distributed network

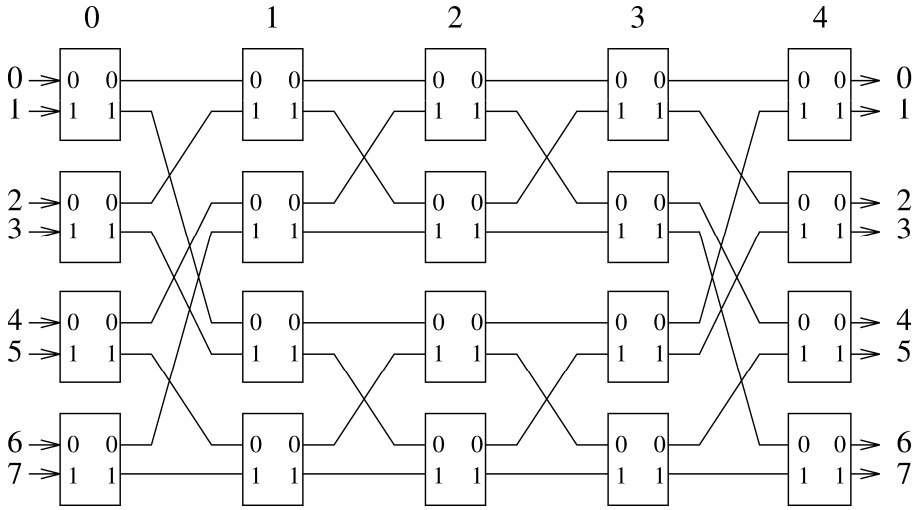

Figure 3.3. 8×8 Beneš network

control, such that each switch makes its own routing decisions, as described in Section 2.1. An N input N output Beneš network consists of $2(\log N)-1$ stages of switches interconnected as shown in Fig. 3.3 for $N = 8$. The Beneš network may be viewed as concatenation of a *baseline* network $B(N)$ that consists of stages 0,1,2 in Fig. 3.3, and its mirror image $B^{-1}(N)$ that consists of stages 2,3,4 in Fig. 3.3, with the middle stage (stage 2) shared between $B(N)$ and $B^{-1}(N)$. This construction is well known. The $N \times N$ Beneš network provides $N/2$ different paths between any given input–output port pair as explained in the following. In the baseline network $B(N)$, there is a single path from a given input to a given output. From a given input of the Beneš network, $N/2$ different switch inputs in the middle stage of the Beneš network may be reached, and from that point there exists a single path to reach the required network output. Therefore, there exists $N/2$ different paths between any given input–output port pair in the Beneš network.

3.3 The Simulator

We implemented a network simulator which simulates the behavior of adaptive source routing and non–adaptive random routing schemes under different loads using a number of communication patterns. The simulator has two major components which are the component for controlling the insertion of packets into the network and the component for controlling the flow of packets in the

network. These two major components, their functions, and algorithms are given in the following sections. The main algorithm used in the simulator is defined just after the following two sections.

3.3.1 Packet Generator

In order to be able to evaluate the performance of a routing scheme, we must provide different communication patterns and different loads to the network. These are the functions of the packet generator.

Packet destinations for *uniform communication* pattern are randomly generated at each input port to reach to every output with a uniform distribution. The packet generator also allows generating packet destinations for a number of *structured communication* patterns like *cyclic-shift-left communication*, *cyclic-shift-right communication*, and *reverse communication* patterns. In cyclic-shift-left communication pattern, the destination for the packet is calculated by shifting the binary representation of the source processor sending the packet one bit position to the left in a cyclic manner. For example, in an 8×8 Beneš network, processor 6 (110 in binary) sends packets to processor 5 (101 in binary). The cyclic-shift-right communication pattern is similar. For the preceding example, processor 6 (110 in binary) sends packets to processor 3 (011 in binary). In reverse communication pattern, the sum of the source and destination processors must sum up to $N - 1$ in an $N \times N$ network. For the 8×8 Beneš network example, processor 6 sends packets to 1 and processor 1 sends packets to 6. These are the uniform and some examples of the structured communication patterns implemented. Packet generator also permits implementation of packet destination calculations for other structured communications in a very modular way, by just describing the relationship between the source processor sending the packet and the receiving processor.

In addition to providing different communication patterns, the packet generator must also provide a way to generate packets at random time instants such that the inter–arrival times between successive packets are in control of the user to provide different loads to the network in simulations. We generate packets at random instants with geometric inter–arrival times using the *probability density function* (*pdf*)

$$\frac{1-a}{a} \times a^t \tag{3.1}$$

```
POISSON(a)
1   Let random() return a real number
    between 0.0 and 1.0 with uniform distribution
2   r ← (1 − a) × random()
3   t ← (log r − log((1 − a)/a)) / log a
4   return (int)t
```

Figure 3.4. Function defined for generating an inter–arrival time between two successive packets using *Poisson distribution*

where $0 \leq a < 1$. This function satisfies the property that all probabilities sum up to 1, i.e.,

$$\sum_{t=1}^{\infty} \frac{1-a}{a} \times a^t = 1 \tag{3.2}$$

a is the parameter for the distribution function which determines the inter–arrival times of the randomly generated packets. This distribution is known as the *Poisson distribution* [24]. The algorithm used to generate a time interval for the next packet to be inserted in to the network is in Fig. 3.4. Note that a simpler exponential random number generator [20] can also be used.

The relationship between the poisson distribution function parameter a and the average inter–arrival time between successive packet generation, t, is given by the equality

$$t = \frac{1}{1-a} \tag{3.3}$$

For example, for $a = 0.5$, the average inter-arrival time between two successive packets is 2 time units. In fact this means that if function POISSON(0.5) is repeated enough number of times, the average of the values returned by the function equals 2.

We described how to determine the time instants to generate the next packet arrival into the network. All the processors must insert packets into the network at random instants using the defined algorithm. This is achieved by keeping the time to generate the next packet in each processor, which we call $Packet\_Issue\_Time$. Our simulator is clock driven and a global clock is used. $Packet\_Issue\_Time$ for each processor is initialized at time 0 by using poisson distribution function in Fig. 3.4 which determines the time for the first

```
PACKET_GENERATION_PROCESS(N, a)
 1  for i = 0 to N − 1
 2      if CLOCK = Packet_Issue_Time[i]
 3          Insert_Packet_into_Network(i)
 4          Collect_Statistics()
 5          Packet_Issue_Time[i] ← Packet_Issue_Time[i]+ POISSON(a)
 6      endif
 7  endfor
```

Figure 3.5. Algorithm used for generating packets into the network at an arbitrary time

packet to be generated for each processor. The algorithm used for determining which processors will inject packets into the network at an arbitrary time is given in Fig 3.5. The function $Insert\_Packet\_into\_Network(i)$ creates a packet at the source processor i, determines the destination processor according to one of the communication patterns used as described at the beginning of Section 3.3.1, and places the generated packet into the source processor's buffer to be delivered to the destination processor. $Collect\_Statistics()$ is the function used for collecting statistics like the number of packets generated at each input processor, the average inter-arrival times of packets, and current load in the network.

3.3.2 Control of Packet Flow in the Network

Our network simulator is derived by a global clock. The packets in the network are forwarded towards destination or blocked waiting for the needed buffers to be available during each clock cycle. The operations of packet propagation or blocking during one clock cycle are controlled by the algorithm given in Fig. 3.6. $Move\_Packet()$ moves the packet from one buffer to the destination buffer. Whenever a movement of a packet occurs denoted by the variable CHANGE, the loop is iterated since the buffer emptied by the packet may accept a packet waiting for it. The loop terminates when there are no more possible moves of packets in the network. The order of the processors or the switches processed does not affect the result of this algorithm.

```
PACKET_FLOW_CONTROL_PROCESS()
 1  repeat
 2      CHANGE ← FALSE
 3      for all destination processors i
 4          if processor i can accept a packet AND
              there is a packet waiting for processor i
 5            Move_Packet()
 6            CHANGE ← TRUE
 7          endif
 8      endfor
 9      for all switches i in the network
10          Perform output to packet assignment for switch i
11          for each packet p in the switch
12            if assigned buffer for p is available
13                  Move_Packet()
14                  CHANGE ← TRUE
15            endif
16          endfor
17          for each packet p in output buffers of switches
18            if connected input buffer is available
19                  Move_Packet()
20                  CHANGE ← TRUE
21            endif
22          endfor
23      endfor
24  until CHANGE = FALSE
```

Figure 3.6. Algorithm of packet flow control during one clock cycle. Movements of all packets in the network during one clock cycle is handled by this algorithm.

```
NETWORK_SIMULATOR(N, MAX_PACKETS, a)
 1   Let MAX_PACKETS be the total number of packets to
         be inserted into the network for simulation
     Let a be the Poisson distribution parameter for network load
 2   Initialize processor and switches using the network
     topology description file (N × N network)
 3   for i = 0 to N − 1
 4       Packet_Issue_Time[i] ← POISSON(a)
 5   endfor
 6   CLOCK ← 0
 7   repeat
 8       if PACKETS_IN_NETWORK < MAX_PACKETS
 9           PACKET_GENERATION_PROCESS(N, a)
10           PACKET_FLOW_CONTROL_PROCESS()
11           CLOCK ← CLOCK + 1
12       endif
13  until    PACKETS_IN_NETWORK = MAX_PACKETS
             AND all packets are delivered to their destinations
```

Figure 3.7. Algorithm for the network simulator

We described how the packets are inserted into the network and how the packet moves are controlled in the simulator. The main algorithm of the simulator is as in Fig. 3.7 using the defined algorithms. Initialization of the instants of first packet generation for each processor are performed in lines 3–5 of Fig. 3.7. Generation of packets into the network and the control of the packet moves are iterated until a given number of packets are inserted in the network and all packets in the network are delivered to their destinations. When the number of packets generated reaches the given constant, PACKET_GENERATION_PROCESS() stops generating new packets. Delivery of all packets in the network to their destinations is signaled by the availability of all input and output buffers of all switches in the network.

3.4 The Routing Schemes in the Simulator

3.4.1 Random Routing

We implemented a random routing scheme based on the ideas described in [6] for comparison with ASR. Random routing has been devised to reduce congestion that may occur in the network when communication patterns are highly structured. In this scheme, the packet is first routed to a randomly chosen intermediate destination, and from that destination the packet is routed to its final destination. Here, we use this idea in the following way: suppose a packet is to be routed from input a to output b of the Beneš network. We first route the packet from a to a randomly chosen middle stage input of the Beneš network. From that middle stage input we route the packet to b. There exists a single path to accomplish this task and therefore the random routing scheme is non–adaptive.

3.4.2 Adaptive Routing

In the ASR scheme, we encode routing headers such that packets are routed in a fully–adaptive manner in the first $(\log N) - 1$ stages of the network (stages 0 and 1 in Fig. 3.3). That is the first $(\log N) - 1$ bytes of the packet's routing header consists of all ones indicating all output ports in the first $(\log N) - 1$ stages lead the packet to its destination. Once the packet reaches an input of the middle stage (stage 2 in Fig. 3.3), there exists a single path to reach to the required network output. Therefore, the packet will be routed in a non–adaptive manner in the last $\log N$ stages of the network. For computing routing bytes in the last $\log N$ stages, the destination–tag method is used [2]. In this method, the destination port number in binary, $b_{n-1}b_{n-2}\ldots b_0$, indicates the switch ports that should be used to reach to the required network output. The first switch routes the packet through its port numbered b_{n-1}, the next switch through b_{n-2} and so on. For example in Fig. 3.3, to reach from any input of stage 2 to network output 6 (110 in binary), the packet must be routed through port 1 of a switch in stage 2, then through port 1 of a switch in stage 3, then through port 0 of the switch connected to output 6 in stage 4.

3.5 Simulation Results

Simulation results of the adaptive source routing scheme and the non–adaptive random routing scheme for different network loads and communication patterns are presented, giving the average latency as a function of the average load. *Latency* is defined as the number of cycles that takes a packet to cross the network. Latency includes queuing delays at the source processor. *Load* is defined as the average number of packets injected to an input port of the network per cycle. 1.0 packet/cycle (100% load) is the upper bound for the Beneš network. For both routing schemes, we used identical seeds for the pseudo-random number generators. We ran simulations until at least 1500 packets were generated at each input port. The latency of the delivered packets in a network having only a small population (packets currently in the network), do not reflect the exact behavior of latency in terms of load. Packets are delivered to their destinations without queuing delays and blocking when the network is initially clear of packets. For this reason, various statistics were gathered starting from the time the network population has reached a steady state. The number of packets that reached their destinations and that are currently in the network are controlled at each clock cycle to determine whether the network population is in a steady state or not. Whenever the packets in the network reach a predetermined amount, the network population is said to be in a steady state.

	UNIFORM		NON–UNIFORM	
Network	Adaptive	Non-adaptive	Adaptive	Non-adaptive
16×16	0.48	0.40	0.58	0.40
32×32	0.46	0.38	0.53	0.37
64×64	0.44	0.37	0.55	0.36
128×128	0.43	0.37	0.51	0.34
512×512	0.41	0.35	0.50	0.34

Table 3.1. Throughput under uniform and non–uniform packet traffic

In the simulations, uniform loads were used; equal loads were applied to every network input. Figures A.1 through A.5 in Appendix A show the simulation results under uniform packet traffic. Packet destinations were randomly generated at each input port to reach to every output with a uniform distribution. Figures A.6 through A.10 show the simulation results using a structured

communication pattern, cyclic–shift–right communication. This communication pattern introduces a non–uniform packet traffic in the network. Packet destinations were generated as described in Section 3.3.1. Table. 3.1 gives the throughput of random routing scheme and the adaptive source routing scheme under uniform and non–uniform packet traffic in the network. The adaptive routing scheme increases the throughput by a factor of 18% on the average under uniform packet traffic. When the packet traffic is non–uniform, the increase in the throughput that adaptive source routing provides is about 45% on the average as expected. Another noteworthy observation is that the throughput decreases with increasing network size.

Chapter 4

Route Generation in Multicomputers

Scalable multicomputers are based upon interconnection networks that typically provide multiple communication routes between any given pair of processor nodes. Multiple routes provide low latency, high bandwidth, and reliable interprocessor communication. There are multistage interconnection networks (MIN's) [18, 25] which have a regular structure, such as Omega [15], Banyan [9], and indirect binary n-cube [21] networks. Using the inherent knowledge of the interconnection topology, each switch in the network knows which output ports lead a packet to its destination at each stage. Route generation for such networks makes use of the structure in the topology to determine possible output ports to reach to the destination at each stage of the network. An example is the Beneš network given in Section 3.2. In an $N \times N$ Beneš network, all output ports in the first $(\log N) - 1$ stages lead the packet to its destination. For the last $\log N$ stages, the network provides a deterministic route for each destination processor, determined by the destination–tag method.

Regular structure in the interconnection topology of the network provides easy route generation. However a common restriction for such networks is the number of processors that can be connected. Number of processors must generally be a power of 2. This requirement restricts the scalability of the multicomputer in terms of the processors and the interconnection network. The only possible amount of increase in the number of processors in an N processor network is N. Besides, the interconnection network must also be scaled according to the structure in the interconnection topology. Thus, any upgrade in the size of the parallel system will necessitate large amount of funding. These disadvantages have given rise to research on interconnection networks

that provide a wide flexibility in the number of processors and the connections between these processors. Any number of processors may be connected with such interconnection networks and different interconnection topologies may be provided for a given number of processors. The topology used depends on many factors like the communication structure in the applications, number of different routes that must be provided by the network for all source-destination processor pairs, and the possible future increases in the number of processors connected to the network. Flexibility in the number of processors and the interconnection topology addresses the problem of route generation for the packets to reach their destinations. The interconnection network need not have any structure which complicates the decision of which output ports lead a packet to its destination in each stage. Common approach for route generation for such interconnection networks is selecting a single shortest path between each pair of processor nodes, although multiple shortest paths may exist [1]. The routes are stored in a route table in each processor's memory.

We propose a new approach for route generation in any interconnection network. We give an algorithm that generates adaptive routes for all pairs of processors. Generated routes for each source–destination processor node pair determine the maximum adaptivity possible and are stored in a route table in each processor's local memory. The route table approach enables the routing to be done in a topology independent fashion. Our approach also enables the applicability of the adaptive source routing scheme regardless of the interconnection network topology.

4.1 Route Table Generator

We propose an algorithm which generates route tables containing adaptive routes for each pair of processor nodes in any interconnection network. The topology of the network is given by the connections between the ports of the switching nodes and processor nodes. Switching nodes have p ports indexed from 0 to $p-1$. For example, a 2×2 switch is denoted by a 4 ports ($p = 4$) switching node. Processor nodes have just one port used to be attached to the network. Any interconnection network can be defined in terms of the defined switching nodes, processor nodes, and the connections in between. The interconnection topology of the network can also be represented by an undirected graph $T = (V_T, E_T)$, which is referred here as the *topology graph*. The vertex

set $V_T[T]$ contains two types of nodes, namely processor nodes and switching nodes. The edge set $E_T[T]$ represents the interconnections between the switching nodes and between the processor and switching nodes. In general, each processor is connected to a single switching node.

ROUTE_TABLE_GENERATOR(T)
1 Let $T = (V_T, E_T)$ be the topology graph of the network
2 **for** each processor node $u \in V_T[T]$
3 GENERATE_ROUTES(T, u)
4 **endfor**

Figure 4.1. Route Table Generator

The main algorithm for the route table generator is illustrated in Fig. 4.1. As is seen in the algorithm, the function GENERATE_ROUTES(T, u) determines the set of adaptive routes from processor u to all other processors, and stores these routes in the local memory of processor u. For an N node network, each processor keeps route tables with $N - 1$ entries.

Each processor's route table holds routing information for all destination processors which provide a set of possible output ports, rather than a specific output port, at each stage of the interconnection network that leads the packets to the correct destination. The number of distinct paths between a source and a destination processor is determined by the routing information kept in the route table and is given by

$$N_{\text{path}} = |R_1| \times |R_2| \times \cdots\cdots \times |R_{n-1}| \times |R_n| \tag{4.1}$$

where $|R_i|$ is defined as the number of ones in routing byte R_i which determine the set of possible output ports at stage i that lead packets to the destination. Here, n is the number of stages in the shortest paths from the source to the destination processor.

At each stage of the interconnection network, a set of output ports that lead packets to the destination correctly can be determined. However, the problem is that if there are m such output ports at stage i, there are $2^m - 1$ possible subsets of output ports at stage i, all of which are meaningful adaptive routes.

The decision of which subset to choose seems to be obvious at first sight: choosing the maximal set, but in fact this is not the case. The choice of the set of output ports at stage i affects the possible routes that can be found at stage $i+1$. The problem is to maximize the number of distinct paths which is an *optimization problem* because the choice of a subset of possible output ports at any stage i for maximizing N_{path} depends on the other stages. We seek to maximize N_{path} in all routes for all pairs of source destination processor pairs.

The process of generating routes from a processor node (line 3 in Fig. 4.1) has three major steps. The first step is to find all possible shortest paths to destination processor nodes. The second step is to enumerate all possible adaptive routes that lead packets to their destinations. The last step is to select one of the enumerated routes which satisfy the maximum adaptivity criteria. The algorithm of the process is given in Fig. 4.2 and the three main steps of the algorithm are described in the following sections. We used IBM SP2 interconnection networks as sample network which are described in Section 4.2. After introducing the general properties of SP2 networks, we will present an example route generation for a source–destination processor pair on a sample network implementation in Section 4.3.1 which gives the results of each stage in the process.

```
GENERATE_ROUTES(T, u)
1   MODIFIED_BFS(T, u)
    // Generate all shortest paths from u to v ∈ V_T[T] s.t. v ≠ u //
2   for each processor node v ∈ V_T[T] s.t. v ≠ u
3       Create a routability graph R = (V_R, E_R) from u to v
4       Create a solution graph S = (V_S, E_S)
5       MAX_ADAPTIVE_PATH(S)
        // Find the route that provides maximum adaptivity //
6       Store route for (u, v) pair in Route_Table[u, v]
7   endfor
```

Figure 4.2. Generating routes from a processor to other processors

4.1.1 Routability between Processors

In this section, we address the problem of finding all possible shortest paths between all processor pairs. We define a *routability graph* $R = (V_R, E_R)$ for a source–destination processor pair to be a multistage graph [10] as follows: $R = (V_R, E_R)$ is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets V_R^i for $0 \leq i < k$. Each vertex $v \in V_R$ represents a switching node or a processor in the topology graph and vertices at each stage V_R^i are indexed from 0 to $|V_R^i| - 1$. If $< u, v >$ is an edge in E_R then $u \in V_R^i$ and $v \in V_R^{i+1}$ for some i, $0 \leq i < k - 1$ and the label of the edge represents the output port of switch u at stage i which is connected to switch or processor v at stage $i + 1$. The sets V_R^0 and V_R^{k-1} are such that $|V_R^0| = |V_R^{k-1}| = 1$. Let s and d respectively be the vertices in V_R^0 and V_R^{k-1}, s is the *source* and d the *destination*. The number of stages in the routability graph denotes the number of stages in the shortest paths between the source and the destination processors.

The routability graph for a given pair of source–destination processor nodes is created in two steps. In the first step, we use a modified version of the breadth first search algorithm in [17] to find all shortest paths from the source processor, *src*, to all other processors. We apply the modified breadth first search algorithm given in Fig. 4.3 on the topology graph T, rooted at the source processor, *src*. The resulting breadth first tree has all shortest paths from the source processor to all other processors. In a routability graph for a source–destination processor pair what we need is a multistage graph which contains all shortest paths from the source to the destination. The second step uses the resulting breadth first tree of the modified breadth first search and creates a routability graph as follows: for creating the routability graph for source–destination processor pair (src, dst), we run a breadth first search algorithm rooted at the destination node *dst* on the created breadth first tree. When discovering the nodes of the graph during the search, new edges are created such that each node in the multistage graph keeps outgoing edges to the vertices in the next stage. The resulting graph is a multistage graph which contains all shortest paths between (src, dst) processor pair.

The first step which is the application of the modified breadth first search (Fig. 4.3) is executed only once when generating routes from a source processor to all other processors (line 1 in Fig. 4.2). The resulting breadth first tree contains information about all shortest paths from the source processor to all

```
MODIFIED_BFS(T, src)
 1  for each vertex v ∈ V_T[T] − {src}
 2      v.visit ← WHITE
 3      v.depth ← ∞
 4      v.parent ← NIL
 5  endfor
 6  src.visit ← GRAY
 7  src.depth ← 0
 8  src.parent ← NIL
 9  FIFO_ENQUEUE(Q, src)
10  while Q ≠ ∅
11      u ← head[Q]
12      for i = 0 to p − 1    // p is the number of ports //
13          Let v ∈ V_T[T] be the vertex connected to the i-th port of u
14          if v.visit = WHITE
15              v.visit ← GRAY
16              v.depth ← u.depth + 1
17              v.parent ← {< v, u > | l(v, u) = i}
18              if v.type = SWITCH
19                  FIFO_ENQUEUE(Q, v)
20              endif
21          elseif v.visit = GRAY AND v.depth = u.depth + 1
22              v.parent ← v.parent ∪ {< v, u > | l(v, u) = i}
23          endif
24      endfor
25      FIFO_DEQUEUE(Q)
26      u.visit ← BLACK
27  endwhile
```

Figure 4.3. Modified Breadth First Search algorithm. The algorithm finds all shortest paths from a source processor node to other processor nodes in a topology graph.

other processors. All routability graphs from the source processor to the others are generated using this breadth first tree. Thus, the breadth first tree created at the first step is kept unchanged throughout the process.

4.1.2 Generating All Adaptive Routes

In this section, we describe how to generate all possible adaptive routes between a source and a destination processor using the routability graph of the processor pair. We define a *solution graph* for all possible adaptive routes as follows: a solution graph $S = (V_S, E_S)$ for a source–destination processor pair is a multistage graph with the same number of stages as in the routability graph, R, for the same processor pair. Each vertex in the i-th stage, $v \in V_S^i$, represents a subset of the vertices in the i-th stage in the routability graph R except the empty set i.e., $v \in V_S^i$ represents a set x s.t. $x \subseteq V_R^i$ and $x \neq \emptyset$. So the partitions of the vertex set $V_S[S]$ are such that $|V_S^i| = 2^{|V_R^i|} - 1$, for $0 \leq i < k$. Vertices at each stage i, $v \in V_S^i$, are indexed starting from 1 to $|V_S^i|$ and the ones in the binary representation of each vertex v determines the set of vertices at stage i in routability graph R that it represents. This encoding provides direct access to all subset of vertices and an easy way to determine the set members. For example, if the number of vertices at stage i of a routability graph R is 4, the number of vertices at stage i of the corresponding solution graph S will be 15. Each vertex at stage i has an index between 1 and 15. The vertex indexed 13 (1101 in binary) represents the set of vertices $\{u_0, u_2, u_3\} \subseteq V_R^i$ at the i-th stage of the routability graph R. Each edge $e \in E_S[S]$ has a label l associated with it. The meaning of each edge and its label is as follows: let $< u, v > \in E_S[S]$ be an edge from stage i to $i+1$, $u \in V_S^i$ and $v \in V_S^{i+1}$. Let $x \subseteq V_R^i$ be the set of vertices at stage i in R which is represented by u and similarly $y \subseteq V_R^{i+1}$ be the set of vertices at stage $i+1$ in R represented by v. Label $l(u, v)$ is the routing byte at stage i such that the vertices in set y are the ones those of which are reached in the routability graph R, from the vertices in set x using the allowed ports in the routing byte.

The algorithm for creating the solution graph $S = (V_S, E_S)$ from a routability graph $R = (V_R, E_R)$ is given in Fig. 4.4. The vertices of the solution graph are created first. The next job is to create edges in the solution graph. The edges are created stage by stage starting from the first one. The only vertex in the first stage is marked as active. For each active vertex in the current stage,

```
1   Let R = (V_R, E_R) be a routability graph with V_R = ⋃_{i=0}^{k-1} V_R^i
2   Let S = (V_S, E_S) be the solution graph for R, V_S = ⋃_{i=0}^{k-1} V_S^i
3   Create vertex set V_S s.t. |V_S^i| = 2^{|V_R^i|} − 1, 0 ≤ i ≤ k − 1
    // Create edges in E_S //
4   Mark the vertex in V_S^0 as ACTIVE
5   for i = 0 to k − 2      // for all stages //
6       for all ACTIVE vertex v ∈ V_S^i
7           Let v represent the set of vertices x ⊆ V_R^i in R
8           for RB = 1 to 2^p    // for all possible routing bytes //
9               if ∀x_k ∈ x, < x_k, y_k > ∈ E_R^i AND l(x_k, y_k) = RB
10                  Let vertex u represent the set of vertices y = ⋃ y_k
11                  Add edge < v, u > with label RB to v.edge_list
12                  Mark u ∈ V_S^{i+1} as ACTIVE
13              endif
14          endfor
15      endfor
16  endfor
```

Figure 4.4. The algorithm for generating the solution graph $S = (V_S, E_S)$ for a routability graph $R = (V_R, E_R)$

we check whether an edge exists or not for all possible labels which are in fact all possible routing bytes. The existence of edges from a vertex v with label l is examined as follows: remember that a vertex in the solution graph represents a subset of the vertices at the same stage in the corresponding routability graph. For the subset of the vertices of the routability graph represented by the vertex v, we find the vertices reachable in R using the permitted ports in label l, and if none of them fail to reach somewhere using the permitted set of ports, we add an edge $< v, u >$ with label l where u is the set of vertices in the routability graph reached by the set of vertices in v. Whenever an edge $< v, u >$ is added to the solution graph S, the vertex u is marked as active. The active vertices in the solution graph are in fact the sets of switches that can be reached using the routing bytes so far. This is the reason why we only check for the active vertices at each stage. The vertices which are not marked as active can never be reached so there is no need to check edges from those vertices.

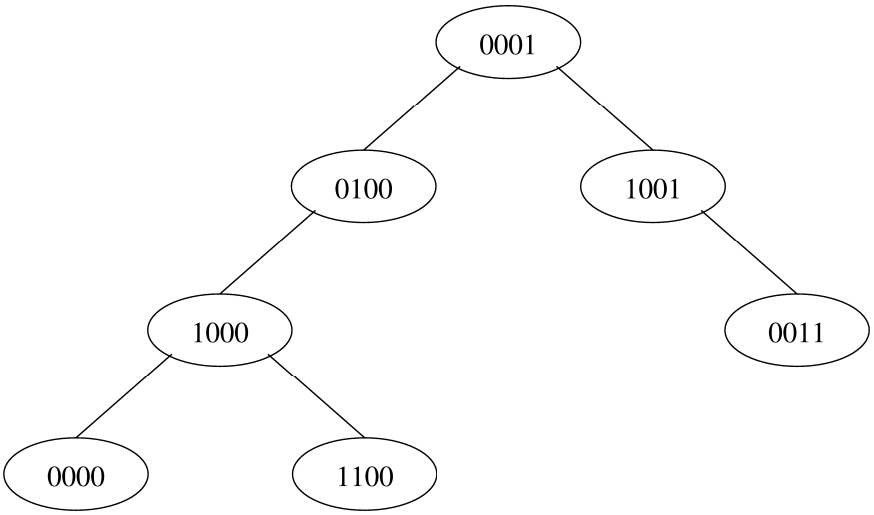

Figure 4.5. Example digital search tree

A limitation of the given algorithm to create a solution graph is the requirement that all the vertices must be created first. The number of vertices in each stage is a power of 2 which may be impossible to handle both in terms of memory and algorithmic complexity for large powers of 2. When the routability graph R is created for a source–destination pair, the number of vertices in each stage is determined. For stages in R which have up to 16 vertices, the vertices at the same stage in the solution graph $S = (V_S, E_S)$ are created and edges for those vertices are found as explained above. Stages in R which have more than 16 vertices are handled in a different way. For those stages in the solution graph S, we use *digital search trees* [11].

A *digital search tree* is a binary tree in which each node contains one element. The element to node assignment is determined by the binary representation of the element keys. Suppose we number the bits in the binary representation of a key right to left beginning at zero. All keys in the left subtree of a node at level $i+1$ have bit i equal to zero while those in the right subtree of nodes at this level have bit $i = 1$. Fig. 4.5 shows an example digital search tree. A search in a digital search tree is performed in the following way. If we are to search for the key k, k is first compared with the key in the root. If there is no match, the subtree to move is determined by the value of bit in the zero bit position, which is the level of the current node of the tree. If the bit equals zero, search is continued in the left subtree, else the search proceeds on the right subtree recursively. Insertion and deletion operations are also similar to the ones for binary search trees. The essential difference is that the subtree to move is determined by a bit in the search key rather than by the result of

the comparison of the search key and the key in the current node. Search and insertion operations can be performed in $O(h)$ time where h is the height of the digital search tree. If each key in a digital search tree has $KeySize$ bits, then the height of the digital search tree is at most $KeySize + 1$.

For those stages in routability graph R which have more than 16 vertices, a digital search tree is used in the solution graph S. The key of the digital search tree is a binary number where the one bit positions represent a set of vertices at the same stage in R. Each node of the tree also keeps a list of outgoing edges to the next stage in the graph. When an edge to a vertex in a stage using digital search tree is created, the vertex index is searched in the tree, if it does not exist, a new node with that key is inserted. When creating edges from a stage using digital search tree, the tree is traversed and for each vertex represented by the key of the nodes in the tree, the edge calculations to the next stage are carried on similarly. The use of digital search trees provide an efficient use of memory. In addition, the experimental results show that the number of elements in the digital search trees constitute a small percentage of the total number vertices that would be created otherwise. The only overhead introduced by the use of digital search trees is the search time when a vertex is to be reached or to be created. However the height of a digital search tree for stage i in $S = (V_S, E_S)$ is at most $|V_R^i| + 1$ because of the encoding used for vertex indices. This adds only a constant complexity.

4.1.3 Selection of an Optimal Route

The solution graph $S = (V_S, E_S)$ contains all possible adaptive routes for a pair of source–destination processors. Each route has a value which is the adaptivity defined as the number of distinct paths provided by the route. We wish to find a route with optimal value, i.e., with maximum adaptivity. This problem is an optimization problem which is well suited for the application of the *dynamic programming* [10, 17] paradigm. The development of a dynamic programming algorithm can be broken into a sequence of four steps.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom–up fashion.
4. Construct an optimal solution from computed information.

We call a solution with the optimal value *an* optimal solution, since there may be several solutions that achieve the optimal value. Steps 1–3 form the basis of a dynamic programming solution to a problem. Step 4 can be omitted if only the value of an optimal solution is required.

The solution graph $S = (V_S, E_S)$ is a k-stage graph. V_S^0 and V_S^{k-1} are such that $|V_S^0| = |V_S^{k-1}| = 1$. Let s and d respectively be the vertex in V_S^0 and V_S^{k-1}. s is the *source* and d the *destination*. We define $c(u, v)$ to be the cost of an edge $< u, v >$ as the number of ones in the label $l(u, v)$ of the edge, i.e., $c(u, v) = |l(u, v)|$. The adaptivity of a path from s to d is the multiplication of the costs of edges on the path, which is the number of distinct paths provided by the route. A dynamic programming formulation for a k-stage solution graph S is obtained by first noticing that every s to d path is a result of a sequence of $k-2$ decisions. The i-th decision involves determining which vertex in V_S^{i+1}, $0 \leq i \leq k-3$, is to be on the path. It is easy to see that the principle of optimality holds. Let $P(i, j)$ be a maximum adaptive path from vertex j at stage i to vertex d. Let $ADP(i, j)$ be the adaptivity of this path. Then, using the forward approach [10], we obtain

$$ADP(i,j) = \max_{m \in V_S^{i+1} \ \wedge \ <j,m> \in E_S[S]} \{c(j,m) \times ADP(i+1,m)\ \} \qquad (4.2)$$

Since

$$ADP(k-2,j) = \begin{cases} c(j,d) & < j,d > \in E_S[S] \\ 0 & < j,d > \notin E_S[S] \end{cases}$$

(4.2) may be solved for $ADP(0, s)$ by first computing $ADP(k-2, j)$ for all $j \in V_S^{k-2}$, then $ADP(k-3, j)$ for all $j \in V_S^{k-3}$, etc., and finally $ADP(0, s)$.

Before giving the algorithm to solve (4.2) for a general k-stage graph, let us impose an ordering on the vertices in $V_S[S]$. This ordering will make it easier to write the algorithm. We shall require that the n vertices in $V_S[S]$ are indexed 1 through n. Indices are assigned in order of stages. First, s is assigned index 1, then vertices in V_S^1 are assigned indices, then vertices from V_S^2 and so on. d has index n. Hence, indices assigned to vertices in V_S^{i+1} are bigger than those assigned to vertices in V_S^i. As a result of this indexing scheme, ADP may be computed in the order $n-1, n-2, \ldots, 1$. The first subscript in ADP and P identifies only the stage number and is omitted in the algorithm. The resulting algorithm is given in Fig. 4.6. D is used to record the decision made at each stage (vertex) so that the maximum adaptive path can be determined easily.

```
MAX_ADAPTIVE_PATH(S)
// S = (V_S, E_S) is a k-stage graph with n vertices indexed in order of stages. //
// c(i,j) is the cost of edge < i,j >. //
// P(1 : k) is a maximum adaptive path. //
1   ADP(n) ← 1
2   for j = n − 1 down to 1    // compute ADP(j) //
3       Let r be a vertex s.t. < j,r > ∈ E_S[S] AND
            c(j,r) × ADP(r) is maximum
4       ADP(j) ← c(j,r) × ADP(r)
5       D(j) ← r
6   endfor
7   // find a maximum adaptive path //
8   P(1) ← 1; P(k) ← n
9   for j = 2 to k − 1    // find j-th vertex on path //
10      P(j) ← D(P(j − 1))
11  endfor
12  return P
```

Figure 4.6. Algorithm for determining maximum adaptive path in a k-stage multistage graph $S = (V_S, E_S)$. It also constructs and returns the maximum adaptive path.

We have implemented the route table generator for any interconnection network provided that the topology of the network is given. We used multistage interconnection networks of IBM SP2 multicomputer, which is commercially available, for performance evaluations. In the next section we describe the SP2 network architecture, the switch chip used, and topologies in the network implementations. The experimental results on SP2 networks are presented in the following section.

4.2 IBM SP2 Network Architecture

The IBM SP2 [26] is a commercially available multicomputer whose communication architecture is based upon the Vulcan architecture. The SP2 processor nodes attach to a multistage interconnection network consisting of 8 input 8 output non–blocking switches [27].

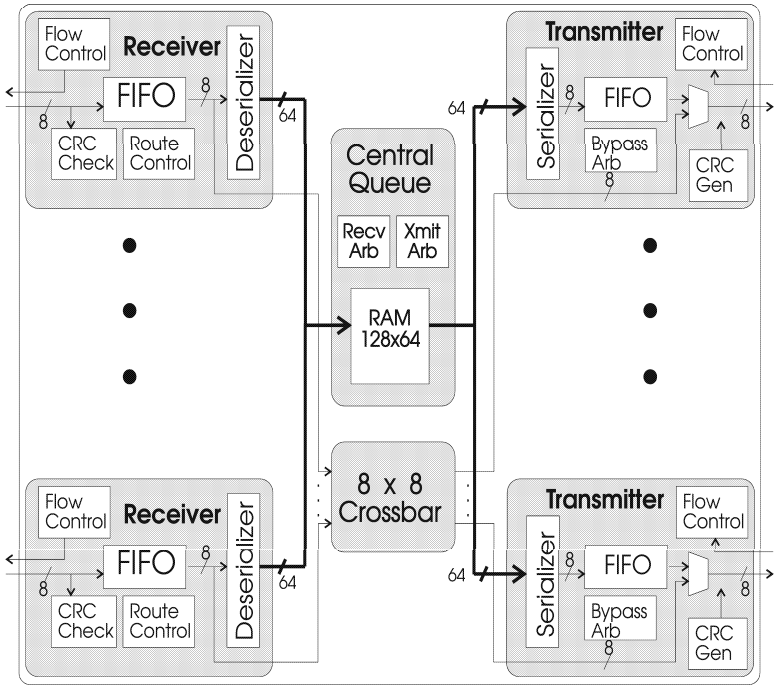

Figure 4.7. The Switch chip organization. Courtesy Dr. Craig. B. Stunkel, IBM T.J. Watson Research Center.

4.2.1 The Switch Chip

The Switch chip, illustrated in Fig. 4.7, contains 8 receiver modules and 8 transmitter modules, an unbuffered crossbar, and a central queue. All ports are one byte wide. In the absence of contention, packet bytes incur 5 cycles of latency cutting through the chip via the crossbar path.

Receivers: The switch chip contains eight identical receiver modules, one associated with each of the eight input ports. The receiver module performs five major functions: (1) administrating the link flow-control protocol, (2) checking the link CRC codes, (3) buffering incoming data, (4) decoding packet routing information, and (5) deserializing incoming packets into 8-byte *chunks* when the packet is blocked. Buffering is accomplished with a first–in–first–out (FIFO) queue. When an incoming packet encounters no contention for the selected output port, packets are immediately forwarded via the Switch chip's crossbar. When the packet is blocked, 8-byte packet chunks are sent to the central queue for temporary buffering.

Crossbar Routing: The switch chip incorporates an unbuffered logical crossbar that allows packets to pass directly from the receivers to the transmitters. These byte-serial crossbar paths permit packets to pass through the chip with low latency whenever there is no contention for the output port. As soon as a receiver decodes the routing information carried by an incoming packet, it asserts a crossbar request to the appropriate transmitter. If the crossbar request is not granted by the time the entire first chunk of the packet has been received, the crossbar request is dropped (and hence the packet will go to the central queue). Each transmitter arbitrates crossbar requests on a least-recently-served basis. A transmitter will honor no crossbar request if it is already transmitting a packet or if it has packet chunks stored in the central queue.

Transmitters : There are eight transmitter modules, one corresponding to each output port. When the central queue contains packet chunks destined for a transmitter, that transmitter requests the next packet chunk. Transmitter modules are served by the central queue in a least-recently-served fashion. As long as data is available, one transmitter is served each clock cycle. The transmitter accepts packet chunks from the central queue, serialize them, buffers them in a 7-byte output FIFO, and transmits them to the link in accordance with the link flow-control protocol. The transmitter is also responsible for computing and transmitting the CRC codes.

Central Queue : The central queue implements a buffered, time-multiplexed 8-way router. It accepts packet chunks from the receivers, stores them, and eventually passes them to the appropriate transmitters. The central queue stores packets until they can be transmitted. The storage, a 128 by 64-bit dual-port RAM, holds up to 128 eight-byte packet chunks. The queue does not reserve a fixed amount of space for each output port; storage is allocated dynamically according to demand.

Stored packets are queued in FIFO order on eight linked lists, one list corresponding to each of the eight switch output ports. As long as queue space is available, one receiver can be served every cycle. As long as data is available, one transmitter is also served every cycle. Thus the bandwidth through the central queue matches the bandwidth into and out of the switch chip: eight bytes per cycle.

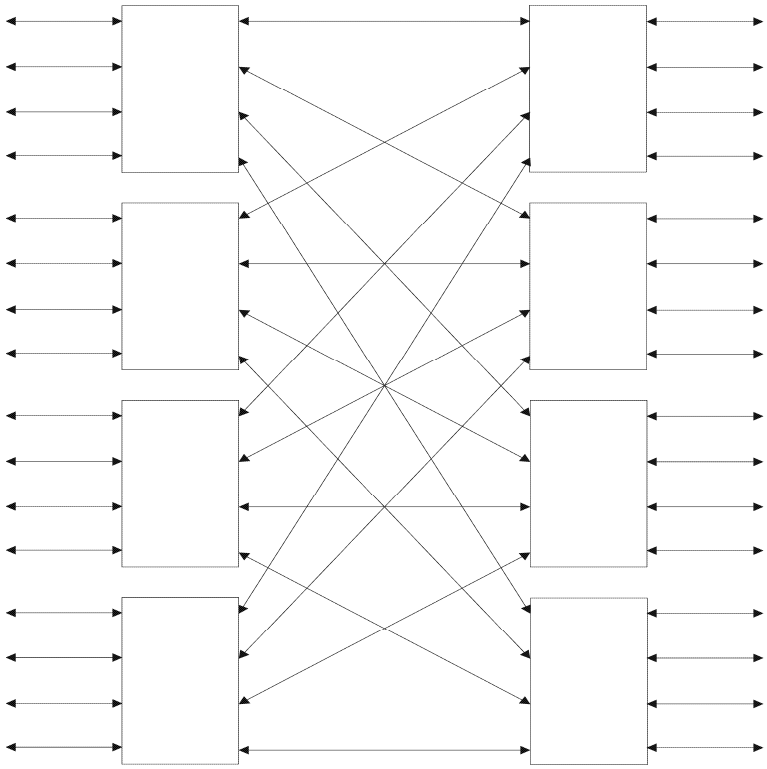

Figure 4.8. The Switch Board consisting of 8 Switch Chips (an SP2 frame)

4.2.2 IBM SP2 Network Topology

In the network implementations, the switch chip input port i and output port i are paired together to form a full duplex bidirectional channel. The resulting 4×4 bidirectional switch element can forward a packet to any of the 8 output ports, including the output ports on the same side with the input port (called "turn-around routing"). In that respect, the SP2 network topologies differ from more commonly known unidirectional multistage interconnection networks such as Omega [15] and indirect binary n-cube [21]. Bidirectionality enhances the modularity, fault–tolerance, and diagnosis of the network. Eight switches placed in a 2–stage configuration interconnected with a shuffle form the *switch board* as shown in Fig. 4.8. The switch board provides full connectivity; it can route a packet from any 32 input ports to any 32 output ports.

Switch boards may be interconnected in various ways to construct larger networks. A 16 node network is constructed using only one switch board with the 16 processor nodes attached to the left hand side of the board and the 16

ports on the right hand side unused. A 32 node network is constructed using two switch boards interconnected as seen in Fig. 4.10. 128 node and 256 node network examples are shown in Appendix B. Custom network topologies of any size can be constructed very easily due to the interconnect technology used. An example of a custom network is a 48 node network as shown in Fig. 4.9.

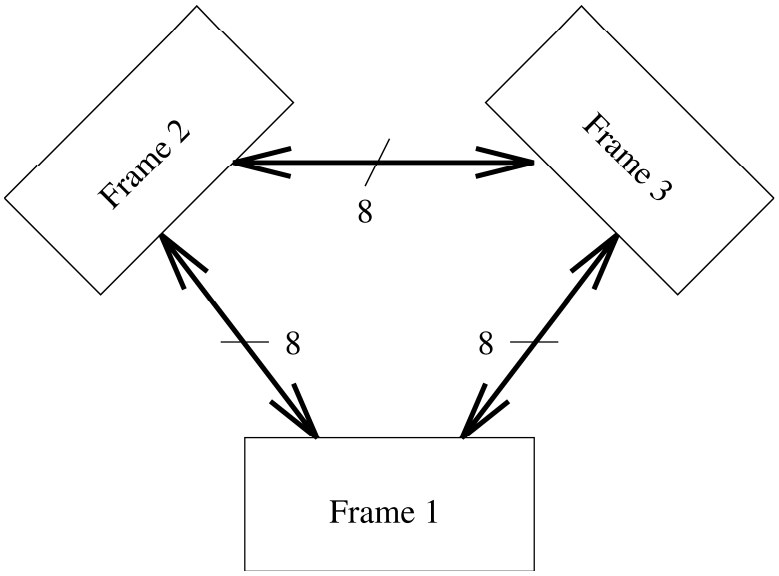

Figure 4.9. SP2 48 way system interconnection

4.3 Route Generation in SP2 Networks

In Section 4.1, we have proposed an algorithm for generating route tables for any multistage interconnection network. We have used the IBM SP2 interconnection networks for test and performance evaluations of our algorithm. In the following sections we will present how the route table generation process works on a sample SP2 network implementation, adaptation of the proposed algorithm to SP2 networks, and experimental results on SP2 network examples.

4.3.1 An Example Route Generation

We will present the route generation for a source–destination processor pair using a 32 node SP2 network which is given in Fig 4.10. In the topology

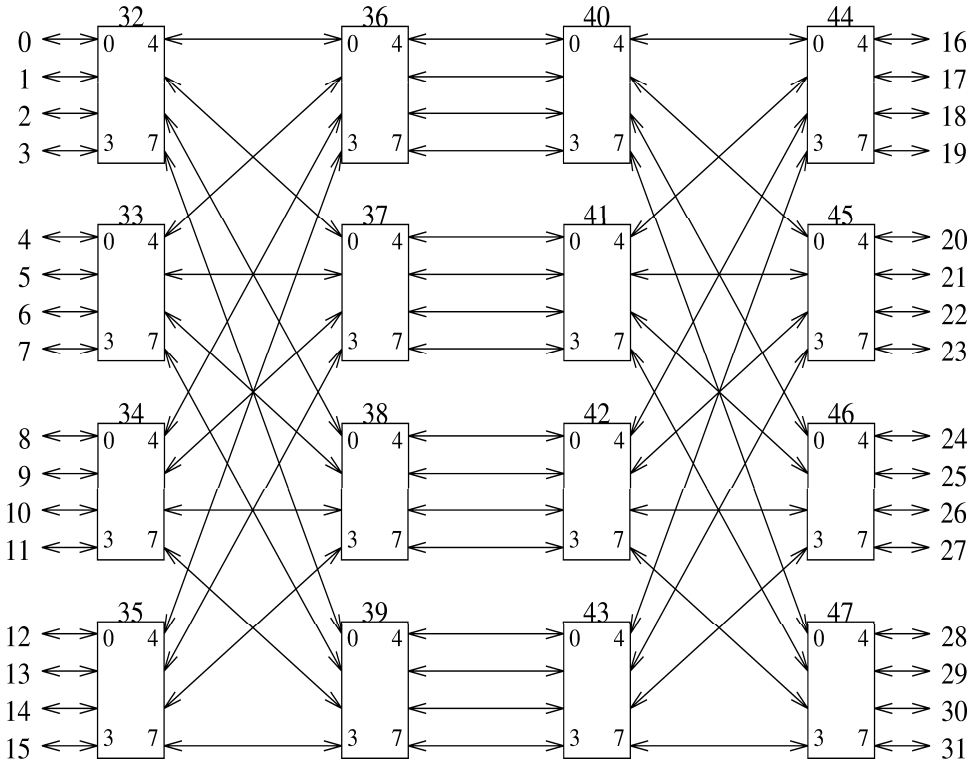

Figure 4.10. A 32 node SP2 network

graph $T = (V_T, E_T)$ for this network, there are a total of 48 vertices which represent the switching and the processor nodes in the network. The processor nodes are represented by the vertices with the index same as the processor number i.e., processor 5 is represented by a vertex indexed 5. The switching nodes are indexed as shown in the network implementation. Topology graph is implemented by keeping edge lists for all vertices in the graph. Each vertex in the graph keeps undirected outgoing edges from the vertex whose labels denote the port number of the vertex that the edge goes out from. For example, the vertex representing the switching node 32 has 8 outgoing edges and an example of them is the edge to the vertex representing the switching node 38 with label 6. For the sake of simplicity, we will not give the topology graph because the network implementation is easier to follow when we keep the meaning of the edges and the vertices in mind.

The route generation for the processor pair (4,30) in the given network will be demonstrated by giving the results at each stage of the algorithm. The first step of the algorithm is to generate the routability graph $R = (V_R, E_R)$ which contains all possible shortest paths between the processor pair. As seen

in Fig. 4.11, vertices at each stage are indexed starting from 0. Each vertex has also the index of the node in the network that it represents. For example, vertex 0 at stage 0 represents the switching node 33 in the network.

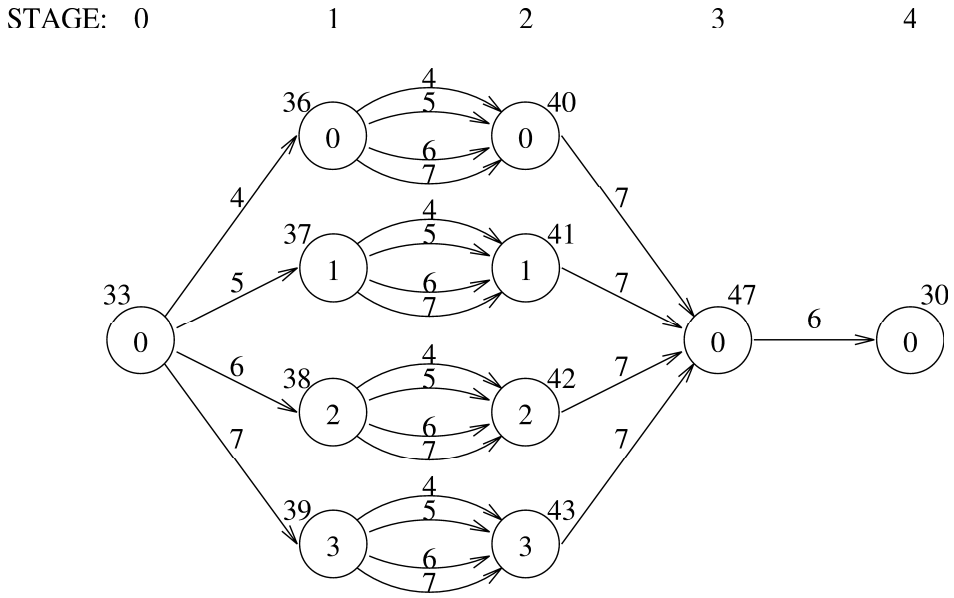

Figure 4.11. $R = (V_R, E_R)$ for processor pair $(4, 30)$

The next step is to generate a solution graph $S = (V_S, E_S)$ for the processor pair (4,30) which is given in Fig. 4.12. Vertices at each stage i have n-bit indices where n is the number of vertices at stage i in R. For example, vertices at stage 1 have 4-bit indices. One bit positions in the indices represent a subset of vertices at the same stage in R. For example, vertex with index 1101 at stage 1 represents the set of vertices $\{0, 2, 3\}$ at stage 1 in R. The labels are routing bytes such that the one bit positions determine the allowed ports to be used. For example, the vertex 0101 at stage 1 has an edge to vertex 0101 at stage 2 with label 11110000. The vertex 0101 at stage 1 represents the set of vertices $\{0, 2\}$ at stage 1 in R. We can reach to the set of vertices $\{0\}$ from vertex 0 and to $\{2\}$ from vertex 2 by the routing byte 11110000 as seen in the routability graph R. So the set of vertices reached from the set $\{0, 2\}$ by the routing byte 11110000 is the union of the sets reached from all members, which is the set $\{0, 2\}$. The maximum adaptive path in S is given by the bold edges in Fig. 4.12 where the labels of the edges on the path determine the route.

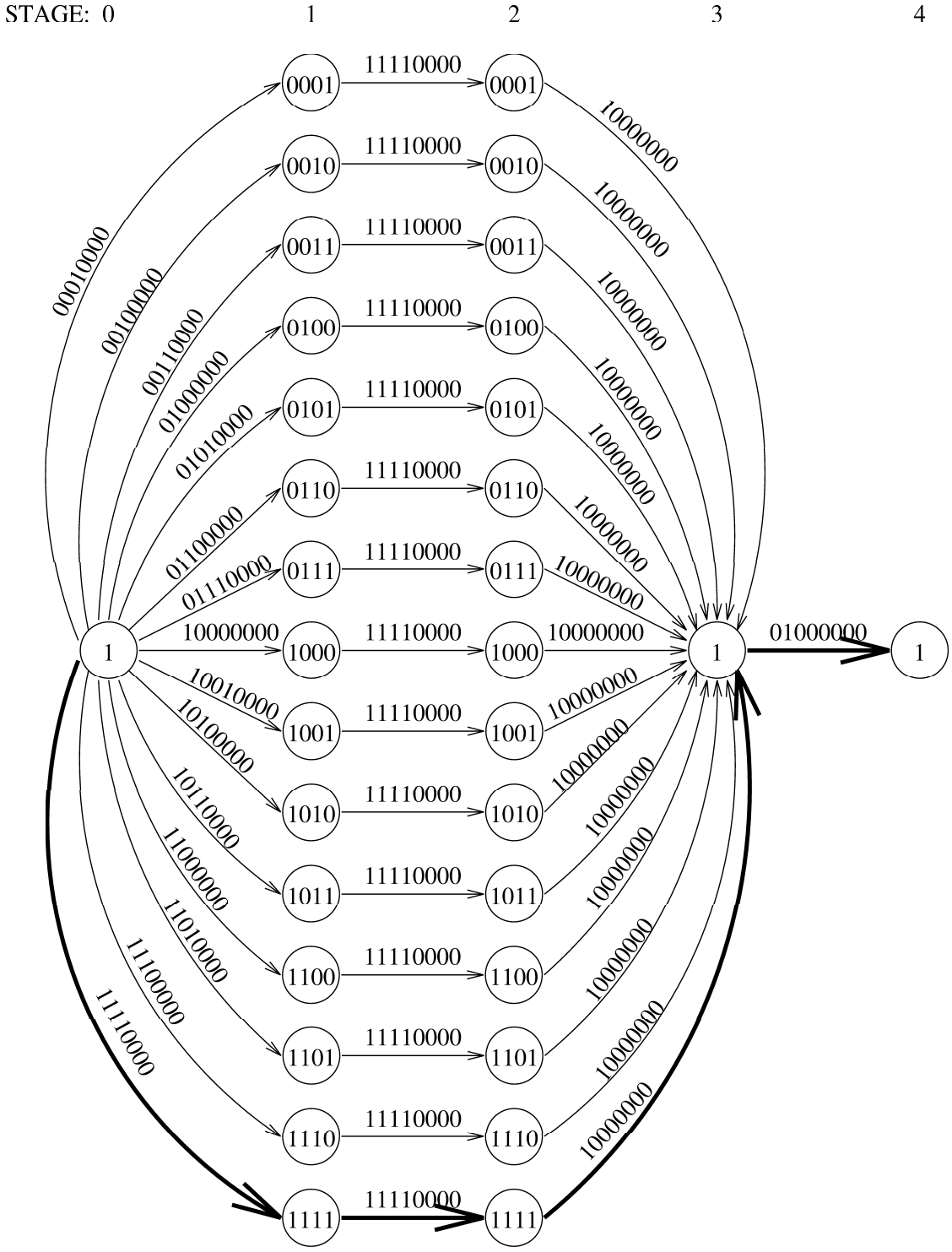

Figure 4.12. $S = (V_S, E_S)$ for processor pair $(4, 30)$

4.3.2 Adapting the Algorithm to SP2 Networks

As seen in Fig. 4.8, 4 processors are connected to a switch in the switch board in SP2 networks. The algorithm we proposed is given in terms of source–destination processor pairs. However, the route tables for all processors connected to the same switch in a switch board are same. The algorithm can be executed for source–destination switch pairs for the switches which are connected to processors for SP2 networks. The routes generated determine the routes to reach to the destination switch from the the source switch. To complete the routes to all processors connected to the destination switch, we use the port of the switch the destination processors are connected to. For example, in the 32 processor network given in Fig. 4.10, we can execute the algorithm to generate routes between switching node pairs (32,47) which can be used to generate routes from the set of processor $\{0, 1, 2, 3\}$ to the set of processors $\{28, 29, 30, 31\}$. Note that the algorithm given in Fig. 4.1 is for one processor to generate all route tables in the system just to give the idea.

4.3.3 Experimental Results

We evaluated the performance of the route table generator in terms of the percentage of the physically existing paths those of which can be used by the generated adaptive routes and the running times for different sized networks. Table 4.1 gives the average ratio of the distinct paths usable by the generated adaptive routes to the total number of physically existing paths in the network for all pairs of source–destination processors. This is in fact the average of the maximum adaptivity of the routes provided by the route table generator for each pair of processor nodes in the network. Generated adaptive routes make use of all the physically existing paths in all networks except the 64 processor network. In 64 processor network, routes for source–destination processor pairs, where the source and the destination processors are on the same switch board, make use of all physical paths. Processor pairs those of which are on different

Network Size	16	32	48	64	128	256	512
Average Adaptivity	1.0	1.0	1.0	0.53	1.0	1.0	1.0

Table 4.1. Average adaptivity for different sized networks

switch boards can not make use of much of the physical paths because of the interconnection of the switch boards in the network implementation. So, on the average generated routes make use of more than half of the underlying physical paths, which is better than using only one or a few of them.

	Sun	IBM SP2	
Network Size	Sparc 5	Host Processor	Node Processor
16	0.066	0.06	0.06
32	0.366	0.28	0.28
48	0.516	0.37	0.37
64	0.533	0.39	0.39
128	33.190	24.98	24.52
256	663.74	498.43	479.22
512	3209.90	2394.87	2298.28

Table 4.2. Average route table generation times for one processor

Table 4.2 shows the average time to create route table for one processor in different size networks. The table contains timings on Sun Sparc 5 workstations, IBM SP2 host and node processors. Times are given in seconds.

4.3.4 An Improvement in the Algorithm

An observation about the routes between processor pairs is that the routes in the first $\lceil n/2 \rceil$ stages are adaptive while the routes in the remaining stages are not in an n stage network. This is also a characteristics of other multistage interconnection networks. We used this fact to improve our algorithms.

Packets reach the middle stage of the network by adaptive routes and in the remaining stages there is only one path to reach their destinations which is determined by the destination processor. Remember that the solution graph $S = (V_S, E_S)$ contains all possible routes between a source–destination processor pair. Edges $e \in E_S$ represent the possible routing bytes between stages of the network. The fact that there is a unique path from the middle stage to the destination implies that all the edges from the middle stage to the next stage have the same label, and the edges in the later stages have the same property. That is, in an n stage solution graph S, all edges $< u, v > \in E_S$, $u \in V_S^i$ and $v \in V_S^{i+1}$, have same labels $l(u, v)$ for $\lceil n/2 \rceil \leq i < n$. Using this property we

Network Size	Sun	IBM SP2	
	Sparc 5	Host Processor	Node Processor
16	0.016	0.01	0.01
32	0.233	0.19	0.18
48	0.283	0.21	0.21
64	0.366	0.32	0.27
128	20.699	15.57	14.88
256	101.645	47.43	47.13
512	1846.276	1347.19	1304.23

Table 4.3. Average route table generation times for one processor for the improved algorithm

can improve the algorithm for creating the solution graph as follows: the edges up to the middle stage are created as before. The edges in the stages starting from the middle stage are created using the fact that edges in the same stage have the same labels. The first active vertex v is considered first in the middle stage. An edge from vertex v exists with label l, and in addition this edge is the only one from v to the next stage. We find this edge $< v, u >$ with label l and add it to the outgoing edges of vertex v. All edges in the same stage will have the same label l since it defines the unique routing byte at that stage. So we just add an edge from all active vertices in the same stage with label l to vertex u in the next stage. For the next stages, there should only be one active vertex and for this vertex we find only one edge with the label being the only routing byte possible in the unique path to the destination. This prevents the overhead of finding the same edges over and over again and decreases the time spent in the solution graph generation process, because creating the edges in the solution graph S dominates overall execution time (since all possible adaptive routes are enumerated). The execution time for finding the maximum adaptive path in S is also reduced since the number of edges created in the last $\lfloor n/2 \rfloor$ stages are reduced compared to the unimproved version. Table 4.3 shows route table generation times for one processor using the improved version of the algorithm.

4.4 Parallel Route Table Generator

The algorithm we experimented on SP2 networks did not make use of the parallelism in the process. Every processor executed the same algorithm on its own to fill in its route table. The common jobs for the processors connected to the same switch are not distributed but repeated by each processor. The inherent parallelism in the process is that the routes are created for source–destination switch pairs and 4 processors are connected to each switch in the network. The processors connected to the same switch have the same route tables so instead of repeating the same processes on all 4 processors connected to the same switch, we can distribute the job to the processors. Upon completion of the jobs each processor is assigned to, processors send each other the partial route tables to form the complete route table. The main steps in the parallelized route table generation are as follows:

1. Make job assignments for 4 processors connected to a switch such that each processor is responsible for a portion of the entries in the route table.
2. Let each processor complete its job.
3. Processors connected to the same switch send their results to each other to fill in their route tables.

We could not implement the proposed parallel route table generator since the only available SP2 multicomputer we can use had 8 processors. Instead of implementing it as a parallel program on the SP2 multicomputer, we implemented an algorithm which distributes jobs to the processors connected to a switch and collects the statistics about the times spent at each processor. These statistics allow us to give the behavior of the proposed parallel algorithm for any number of processors without the need to run it on a real implementation of the network. We neglect the times for the processors to send the partial tables to each other since the amount of data exchanged is considerably small. The algorithm for each processor is as seen in Fig. 4.13. The assignment of destination switches to the processors connected to a source switch is achieved as follows: the 4 processors connected to the source switch generates all shortest paths from the switch to other switches in the network those of which are connected to processors by using the modified breadth first search algorithm given in Fig. 4.3. The algorithm is deterministic and creates the same breadth

PARALLEL_GENERATE_ROUTES(T, u)
Let src be the switch that processor u is connected to
1 MODIFIED_BFS(T, u)
// Generate all shortest paths from u to $v \in V_T[T] - \{u\}$ //
2 Distribute processors $v \in V_T[T] - \{u\}$ among the 4 processors connected to switch src in a *scattered* manner
Let the set Ass_Sw contain the processors assigned to processor u
3 **for** all $v \in Ass\_Sw$
4 Create a *routability graph* $R = (V_R, E_R)$ from u to v
5 Create a *solution graph* $S = (V_S, E_S)$
6 MAX_ADAPTIVE_PATH(S)
// Find the route that provides maximum adaptivity //
7 Store route for (u, v) pair in $Route\_Table[u, v]$
8 Collect timing statistics
9 **endfor**
10 **send** $Route\_Table$ **to** other processors connected to switch src
11 **receive** $Route\_Tables$ **from** other processors
12 **combine** the received tables **to** complete the $Route\_Table$

Figure 4.13. A parallel algorithm for generating routes at a processor to other processors in the network

first tree in all 4 processors. The processors seem to do redundant work at this step but if we were to parallelize this step, the overhead of communication between processors should be a great percentage of the overall execution of the algorithm. We introduce a global ordering on the destination switches discovered by the breadth first search algorithm which is known by all processors. The global ordering indexes the destination switches starting from 1 to $N/4 - 1$ in an N processor network. The processors are assigned destination switches in a scattered manner. The processors connected to the source switch are also indexed starting from 1 to 4. The first switch is assigned to the first processor, the second switch to the second processor, and so on. The next assignment after the last processor is done to the first processor and the cycle is repeated. This distribution will distribute nearly equal work to all processors. Each processor has a set of assigned destination switches. All processors create the route table entries for the set of destinations they are responsible of. Upon completion of route generation at each processor, partial route tables are sent

to other processors. As soon as the processors receive all table entries, they compose their route tables.

4.4.1 Experimental Results

In this section, we give the experimental results of the proposed parallel algorithm for route table generation. The algorithm is experimented on a Sun Sparc 5 workstation neglecting the communication times needed at the end since the data to be exchanged are small in size. In order to evaluate the performance of the parallel algorithm, we first generate all route tables in the system using only one processor and measure the time on one processor, namely $\mathcal{T}_1$. Next, we run the parallel algorithm such that a processor only fills in its own route table according to the job distribution and we collect statistics of the times spent at each processor neglecting the communications. If t_i is the completion time for processor i in a $\mathcal{P}$ processor network, then

$$t_{max} = \max_{0 \leq i < \mathcal{P}} t_i,$$

$$t_{min} = \min_{0 \leq i < \mathcal{P}} t_i.$$

Table 4.4 gives t_{max} and t_{min} in seconds. We evaluated the speedup ($\mathcal{S}$) by the formula

$$\mathcal{S} = \frac{\mathcal{T}_1}{t_{max}} \tag{4.3}$$

where $\mathcal{T}_1$ is the time spent on one processor. We evaluated the efficiency (η) of the parallel algorithm which is equal to the ratio of speedup achieved on $\mathcal{P}$ processors to $\mathcal{P}$, i.e.,

$$\eta = \frac{\mathcal{S}}{\mathcal{P}} \tag{4.4}$$

We measured the degree of work evenly distributed amongst available processors by load imbalance ($\mathcal{L}$), which is the ratio of the difference between the finishing times of the last and first processors to complete their portion of the computation to the time taken by the last processor given by the formula

$$\mathcal{L} = \frac{t_{max} - t_{min}}{t_{max}} \tag{4.5}$$

Speedup, efficiency, and load imbalance in the SP2 network experiments are given in Table 4.4. The speedup and efficiency are also plotted as graphs in Fig. 4.14 and Fig. 4.15 respectively.

Network Size	t_{min}	t_{max}	$\mathcal{S}$	η	$\mathcal{L}$
16	0.0	0.016	3.0	0.187	1.0
32	0.05	0.083	22.8	0.228	0.40
48	0.066	0.083	38.4	0.800	0.20
64	0.1	0.133	41.75	0.652	0.25
128	5.233	5.566	120.072	0.938	0.059
256	26.565	27.248	222.913	0.871	0.025
512	482.597	501.713	477.675	0.933	0.038

Table 4.4. Statistics for parallel route table generator

The maximum time of the execution times of the processors, t_{max}, also determines the overall execution time to fill all route tables in the multicomputer. The route tables are created just once at the startup time and stored in each processor's memory. Hence, time spent in route table generation is acceptable.

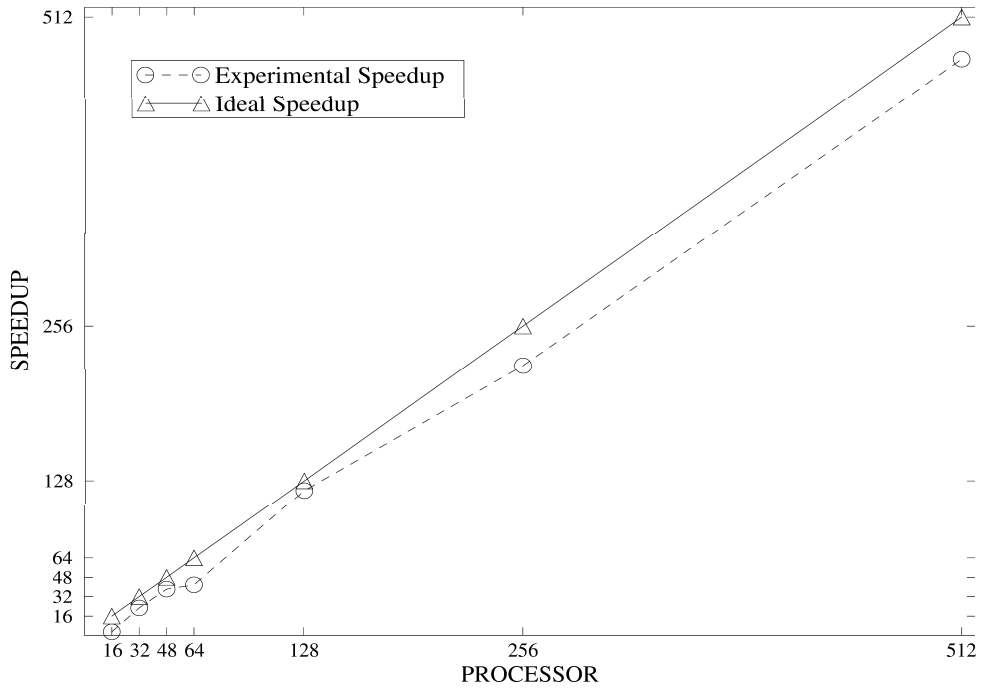

Figure 4.14. Speedup graph for parallel route table generator

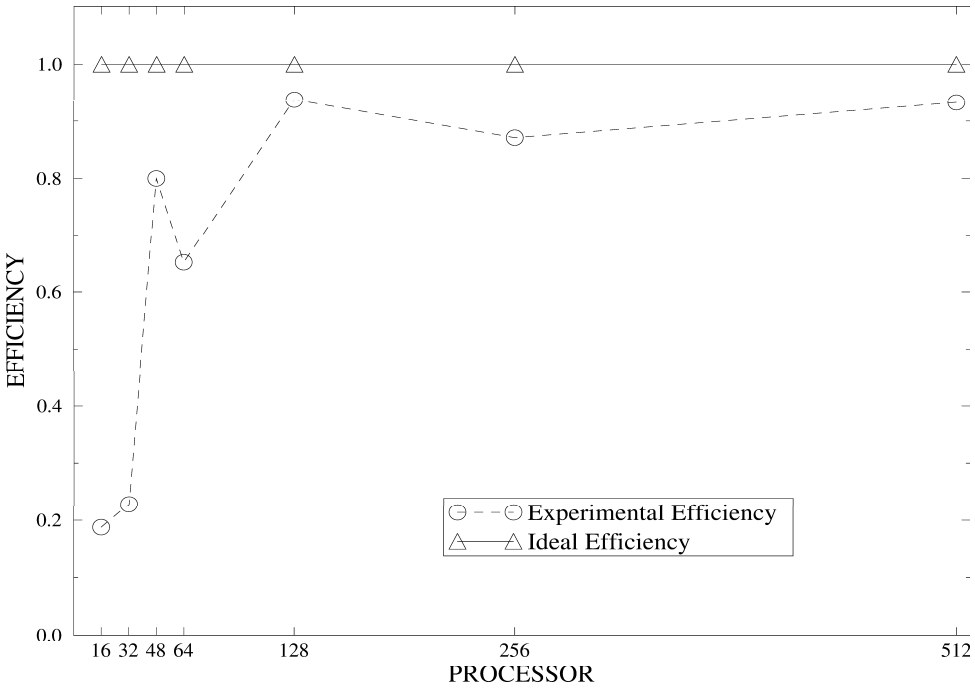

Figure 4.15. Efficiency graph for parallel route table generator

Chapter 5

Conclusion

Scalable multicomputers are based upon interconnection networks that typically provide multiple communication routes between any given pair of processor nodes. Multiple routes provide low latency, high bandwidth, and reliable interprocessor communication. In such networks, the selection of the routes is an important problem because of its impact on the communication performance.

In adaptive routing networks, messages make use of multiple paths between source–destination processor pairs. Switches alleviate the congestion problem by sending packets from less busy alternate routes. In the source routing scheme, the packet route is deterministic and it is completely determined at the source processor sending the packet. In the first part of this thesis, we proposed the *adaptive source routing* (ASR) scheme which combines adaptive routing and the source routing into one. ASR has the advantages of both schemes. The degree of 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.

When we make use of adaptivity, we have the problem of assignment of output ports to the packets in the switches. Each packet in a switch has a permitted set of output ports and the switch must adaptively and in a conflict free manner assign an output to each packet. We formulated this as a *maximum matching problem* in a bipartite graph. Polynomial time algorithms for solving the maximum matching problem exist but they require sophisticated data structures which makes them impractical to implement in switch hardware. We described a heuristic that can be implemented in terms of primitive

logic operations. The experimental results showed that the matching heuristic performs well in practice and it can be implemented easily in switch hardware.

We implemented a network simulator for performance comparisons of the ASR scheme and the non–adaptive random routing scheme. Simulations of a sample network showed that the adaptive source routing performs well under uniform and non–uniform message traffic as expected.

In this thesis we also proposed a route generation algorithm for any interconnection network. Route generation in regular structured networks is easy since each packet makes use of the inherent knowledge of the network topology to reach a destination. However, the main disadvantage of such regular networks is the restriction on the number of processors connected to the network. Generally the number of processors are required to be a power of 2. There are examples of interconnection networks which provide a flexibility in the number of processors connected to the network and the used interconnection topology. Such networks need not have any regular structure in topology which complicates the route generation. We proposed an algorithm which generates routes for all pairs of source–destination processors in any interconnection network. The generated routes are stored in a route table in each processor's memory. We implemented the algorithm and evaluated the performance on SP2 network examples. We give the performance in terms of the distinct paths provided by the generated adaptive routes compared with the physically existing paths and the execution times for different sized networks.

To improve the performance of the algorithm, we also proposed a parallel route table generation algorithm. We implemented a job distribution algorithm to be able to evaluate the performance of the parallel algorithm that makes the job assignments to processors and collects statistics for each processor. This allows us to give the performance without the need to execute the algorithm on a real implementation of the network. The experimental results show that the efficiency increases as the size of the network increases. The job distribution algorithm also provides good load balance for large networks.

The advantages provided by the proposed algorithms are that they can be used for any interconnection network regardless of the interconnection topology. In addition, for the case of faulty links or switches in the network, new routes that take care of the faults can be generated at any time updating the route table.

Appendix A

Simulation Results of ASR

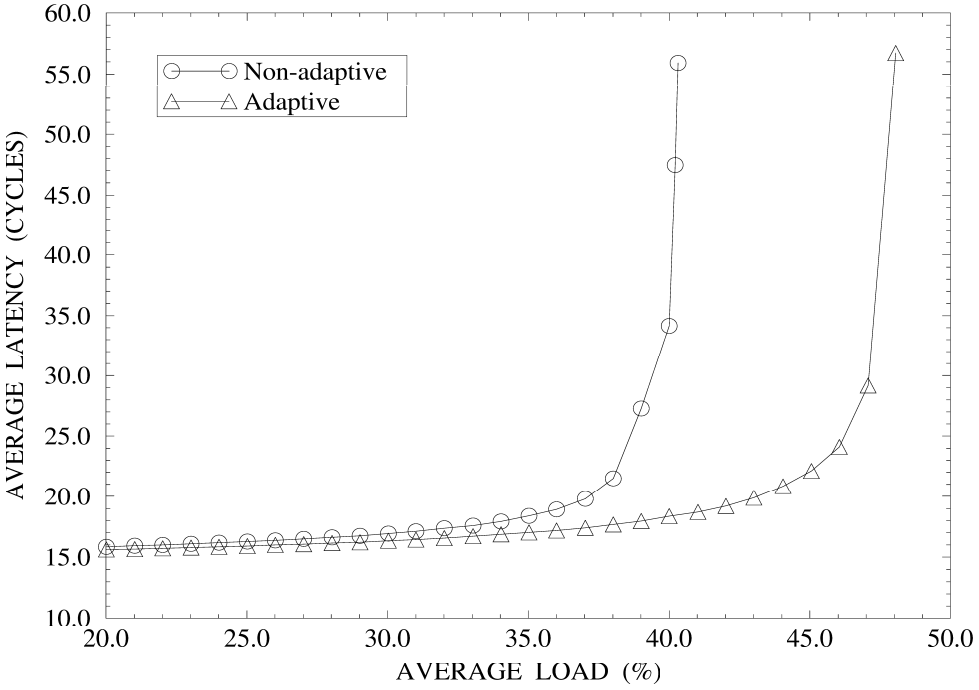

Figure A.1. Performance of adaptive source routing and non–adaptive random routing on a 16×16 network with uniform communication pattern

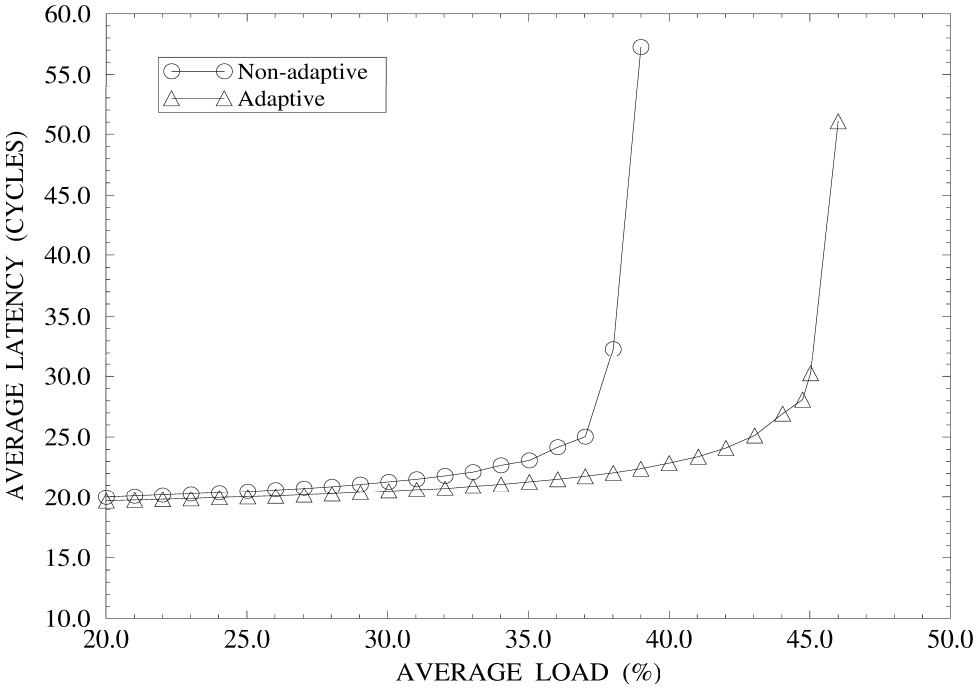

Figure A.2. Performance of adaptive source routing and non–adaptive random routing on a 32×32 network with uniform communication pattern

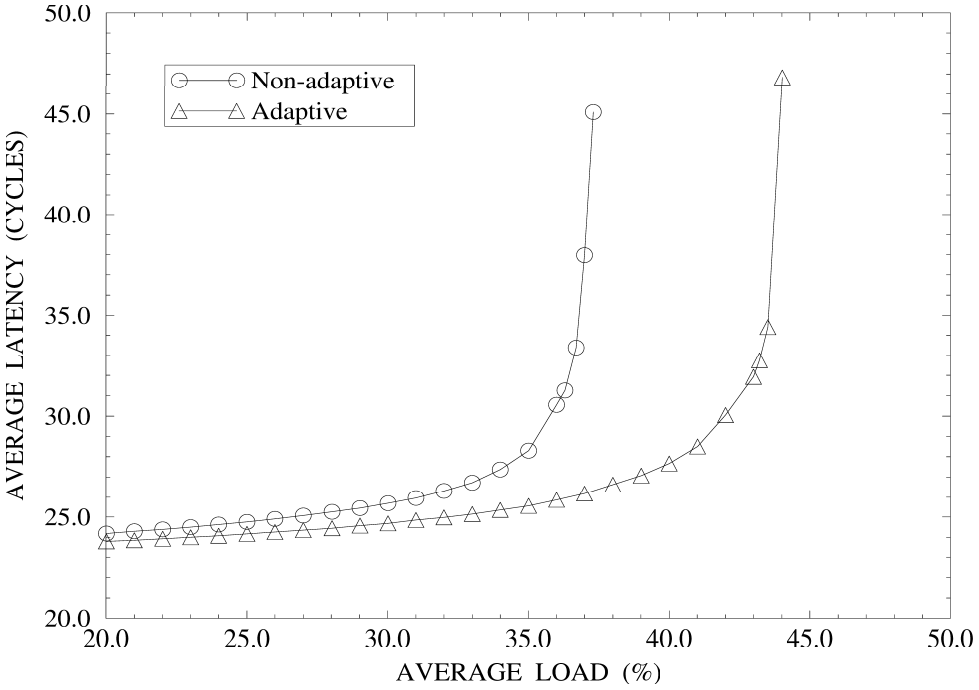

Figure A.3. Performance of adaptive source routing and non–adaptive random routing on a 64×64 network with uniform communication pattern

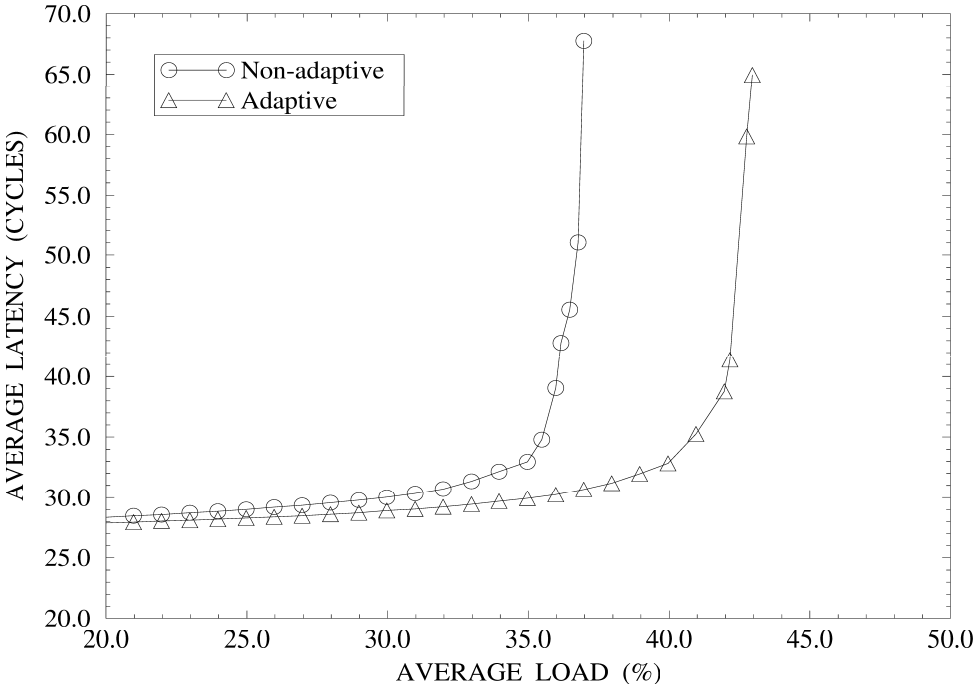

Figure A.4. Performance of adaptive source routing and non–adaptive random routing on a 128×128 network with uniform communication pattern

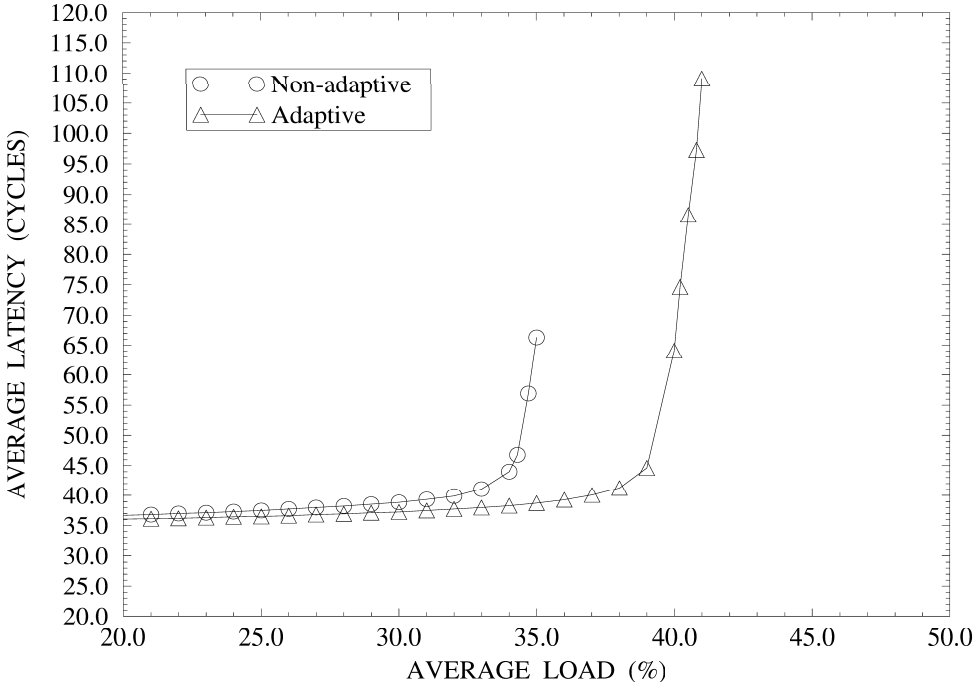

Figure A.5. Performance of adaptive source routing and non–adaptive random routing on a 512×512 network with uniform communication pattern

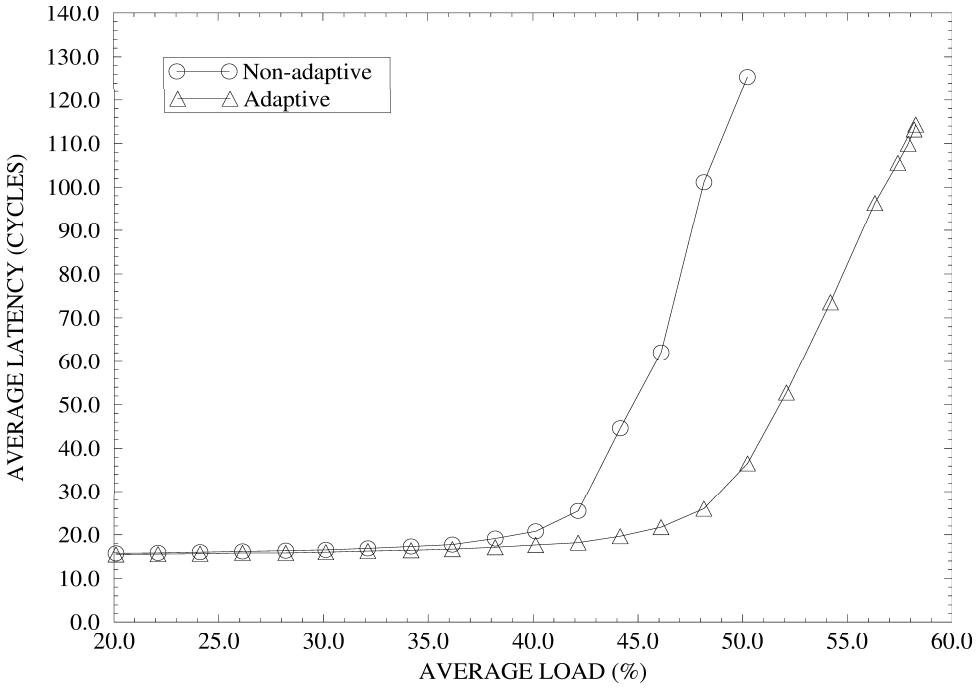

Figure A.6. Performance of adaptive source routing and non–adaptive random routing on a 16×16 network with shift–right communication pattern

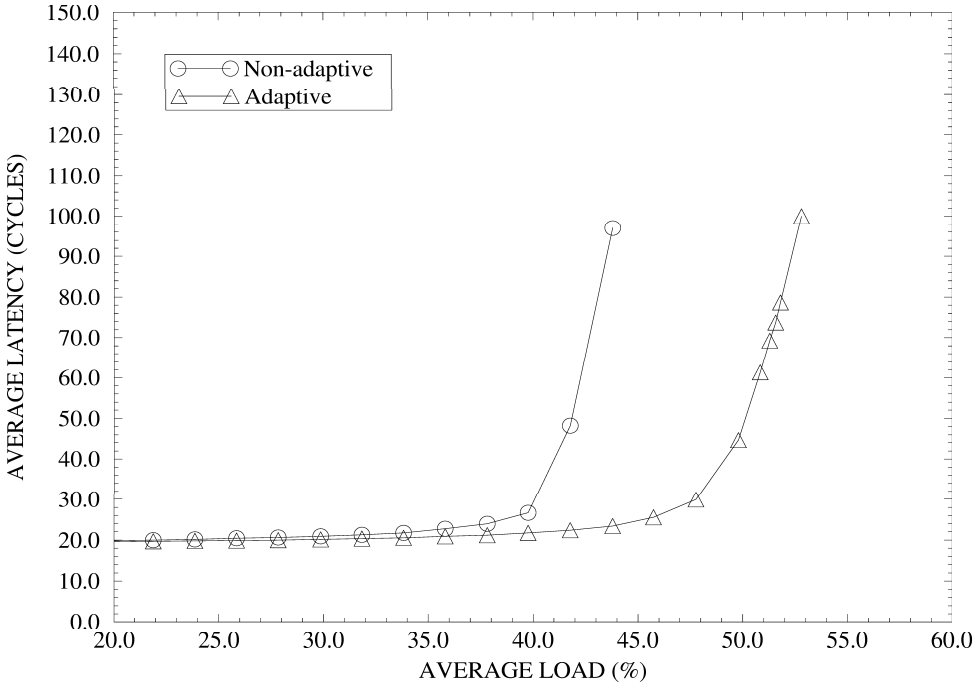

Figure A.7. Performance of adaptive source routing and non–adaptive random routing on a 32×32 network with shift–right communication pattern

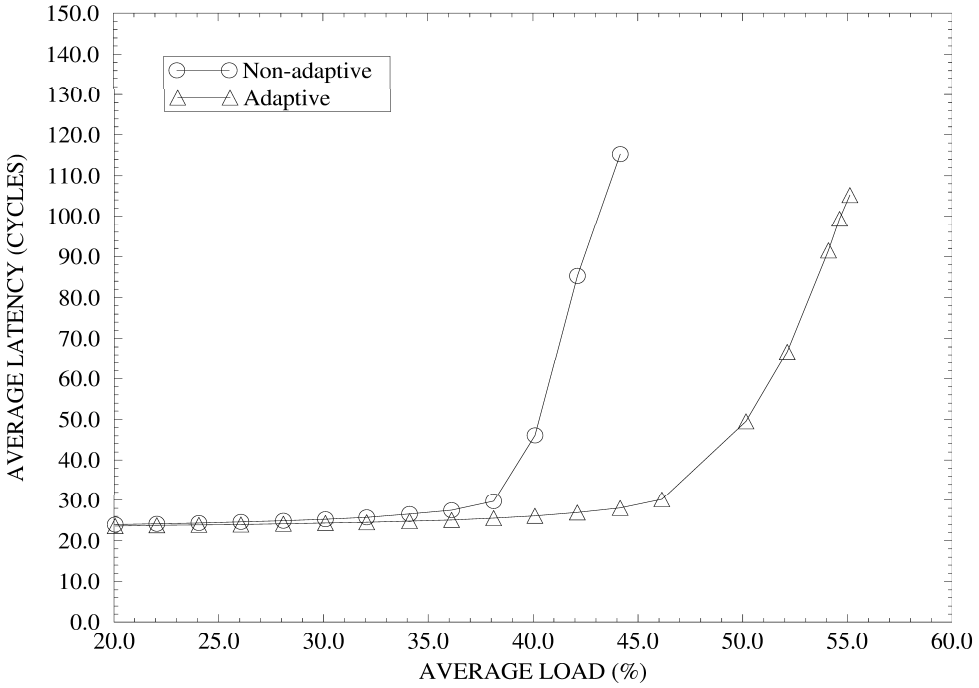

Figure A.8. Performance of adaptive source routing and non–adaptive random routing on a 64×64 network with shift–right communication pattern

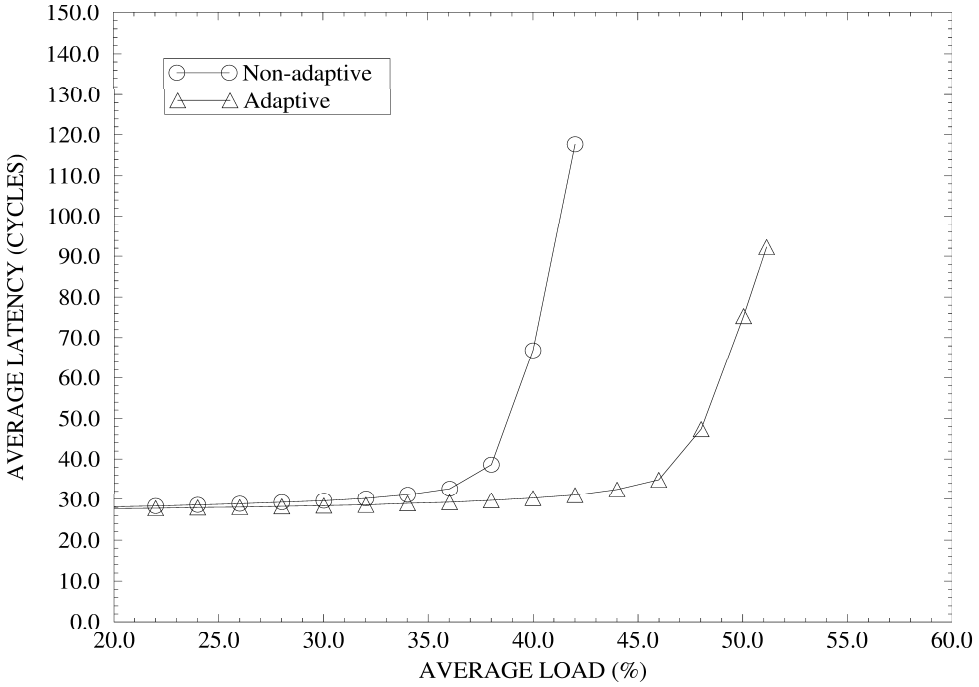

Figure A.9. Performance of adaptive source routing and non–adaptive random routing on a 128×128 network with shift–right communication pattern

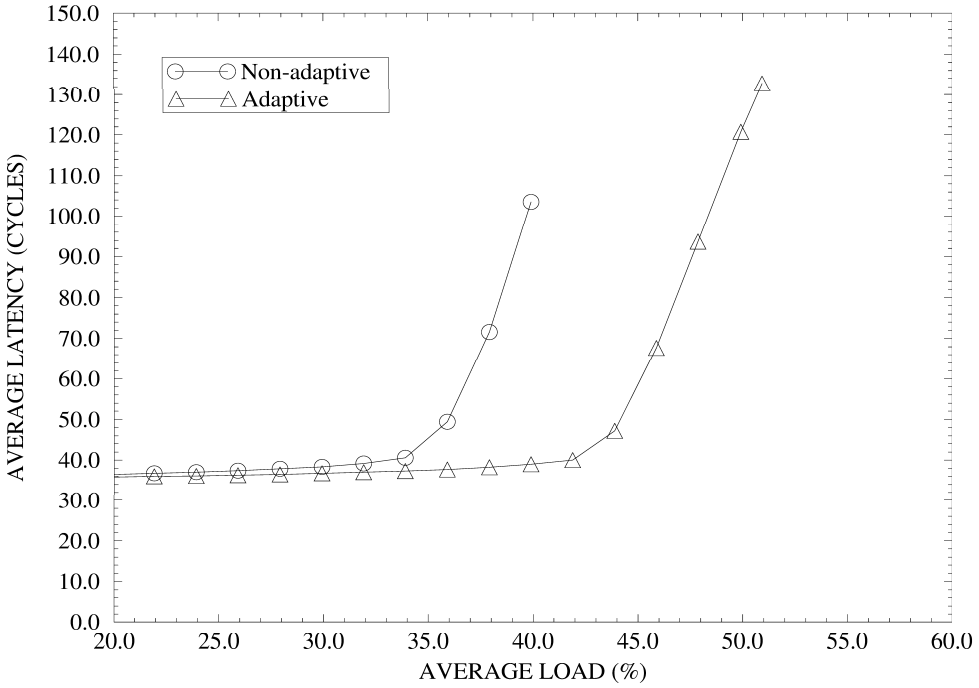

Figure A.10. Performance of adaptive source routing and non–adaptive random routing on a 512×512 network with shift–right communication pattern

Appendix B

IBM SP2 Network Examples

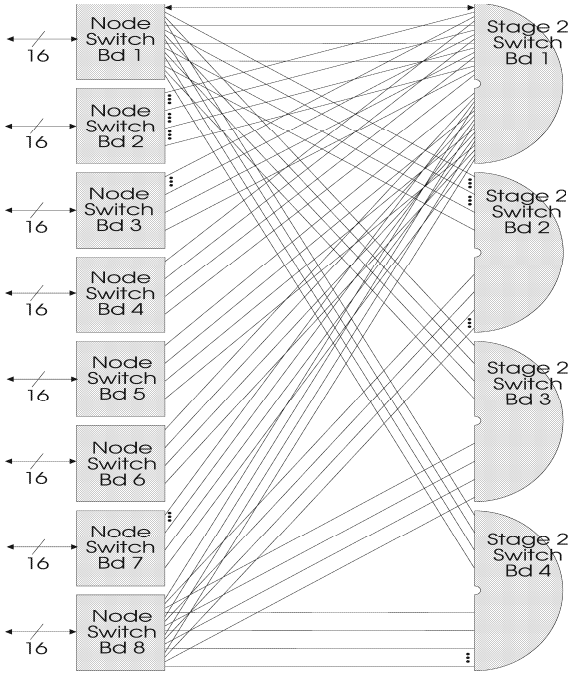

Figure B.1. A 128 node network consisting of 8 first stage and 4 second stage switch boards. Courtesy Dr. Craig. B. Stunkel, IBM T.J. Watson Research Center.

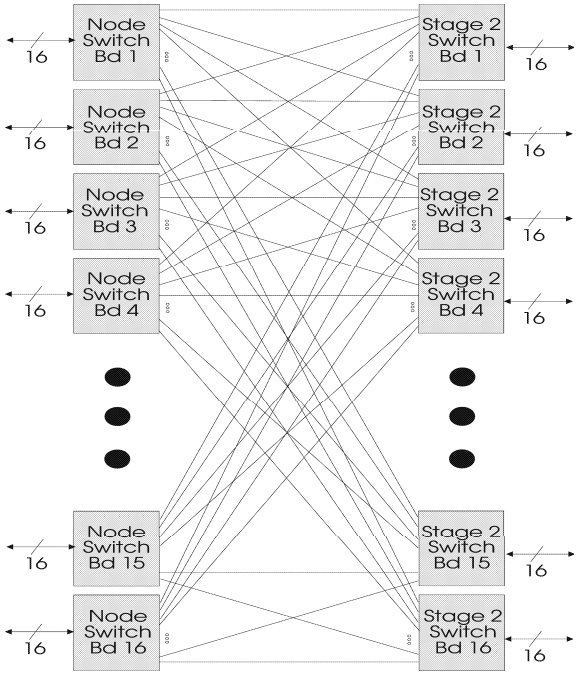

Figure B.2. A 256 node network consisting of 16 first stage and 16 second stage switch boards. Courtesy Dr. Craig. B. Stunkel, IBM T.J. Watson Research Center.

Bibliography

[1] B. Abali and C. Aykanat. Routing algorithms for IBM SP1. *Lecture Notes in Computer Science, Springer–Verlag*, 853:161–175, 1994.

[2] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings Publishing, CA, 1989.

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

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

[5] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. on Computers*, C-36(5):547–553, May 1987.

[6] S. A. Felperin, L. Gravano, G. D. Pifarre, and L. C. Sanz. Routing techniques for massively parallel communication. *Proc. IEEE*, 79(4):488–503, April 1991.

[7] T. Y. Feng. A survey of interconnection networks. *IEEE Computer*, pages 12–27, Dec. 1981.

[8] R. P. Garimaldi. *Discrete and Combinatorial Mathematics.* Addison–Wesley, NY, 1989.

[9] L. R. Goke and G. J. Lipovski. Banyan networks for partitioning multiprocessor systems. In *Proc. 1st Ann. Symp. on Computer Architecture*, pages 21–28, 1973.

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

[11] E. Horowitz and S. Sahni. *Data Structures in Pascal.* Computer Science Press, NY, 1990.

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

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

[14] S. Konstantinidou and L. Snyder. The Chaos Router. *IEEE Trans. Computers*, 43(12):1386–1397, December 1994.

[15] D. H. Lawrie. Access and alignment of data in an array processor. *IEEE Trans. on Computers*, C-24(12):1145–1155, Dec. 1975.

[16] C. E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. on Computers*, C-34(10):892–901, Oct. 1985.

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

[18] G. J. Lipovski and M. Malek. *Parallel Computing: Theory and Comparisons.* Wiley & Sons, New York, NY, 1987.

[19] J. A. McHugh. *Algorithmic Graph Theory.* Prentice Hall, New Jersey, 1990.

[20] M. K. Molloy. *Fundamentals of Performance Modeling.* Macmillan Publishing Company, NY, 1989.

[21] M. C. Pease, III. The indirect binary n-cube microprocessor array. *IEEE Trans. on Computers*, C-26(5):458–473, May 1977.

[22] G. D. Pifarre, L. Gravano, S. A. Felperin, and L. C. Sanz. Fully adaptive minimal deadlock-free packet routing in hypercubes, meshes, and other networks: Algorithms and simulations. *IEEE Trans. on Parallel and Distributed Systems*, 5(3):247–263, March 1994.

[23] F. S. Roberts. *Applied Combinatorics.* Prentice–Hall, NJ, 1984.

[24] D. H. Sanders. *Statistics.* McGraw–Hill, NY, 1990.

[25] I. D. Scherson and C. H. Chien. Least common ancestor networks. In *Proc. 7th Int. Parallel Processing Symp.*, pages 507–513, 1993.

[26] C. B. Stunkel, D. G. Shea, B. Abali, M. G. Atkins, C. A. Bender, D. G. Grice, P. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swets, R. F. Stucke, M. Tsao, and P. R. Varker. The SP2 high-performance switch. *IBM Systems Journal*, 34(2):185–204, 1995.

[27] 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.*, pages 268–274, April 1994.

[28] K. Thulasiraman and M. N. S. Swamy. *Graphs: Theory and Algorithms.* Wiley–Interscience Publication, NY, 1992.