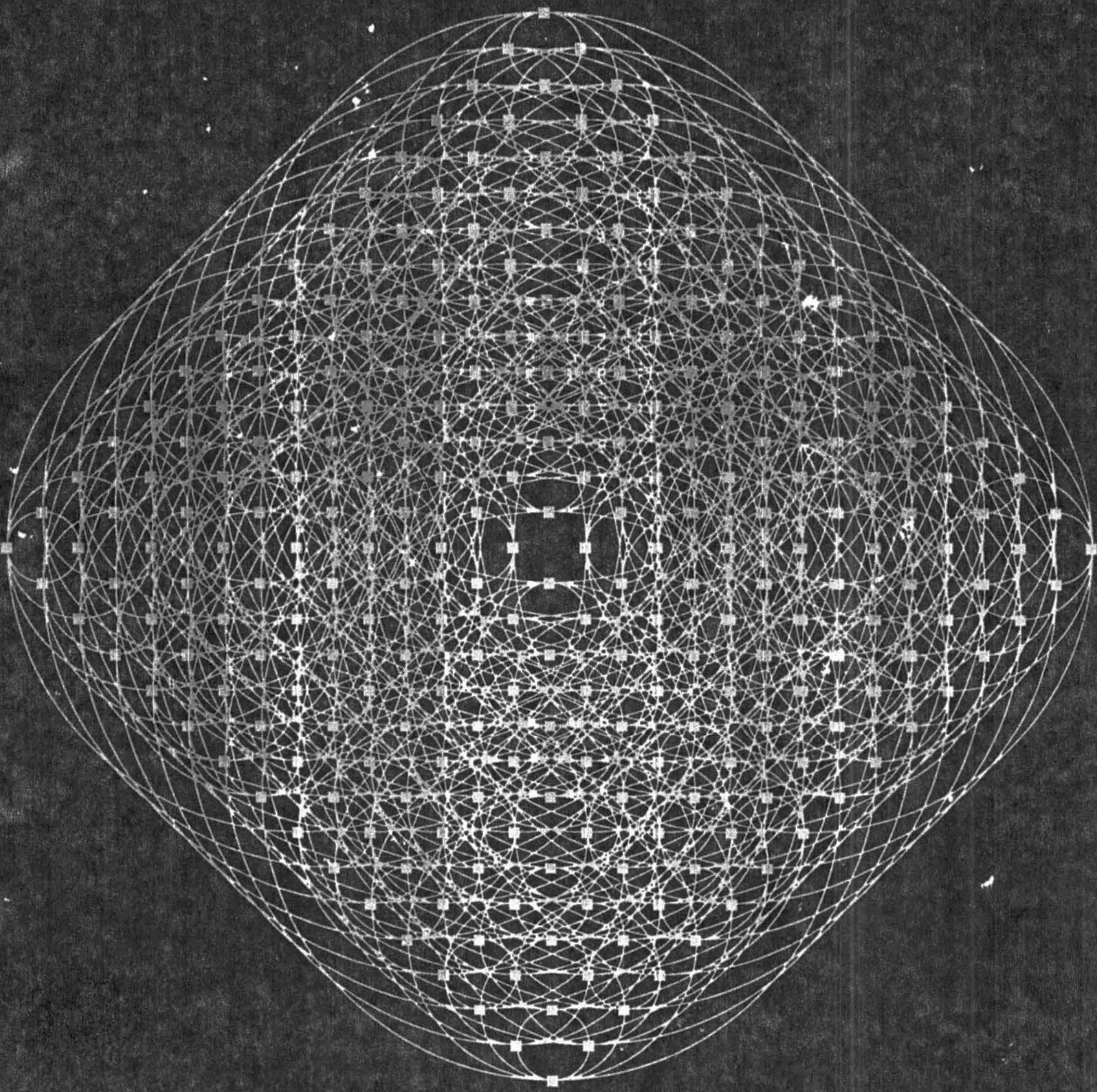
90-HC

THE PROCEEDINGS OF



THE FOURTH CONFERENCE ON HYPERCUBES, CONCURRENT COMPUTERS, AND APPLICATIONS

AS PRESENTED
MARCH 6–8, 1989
HYATT REGENCY MONTEREY'S CONVENTION CENTER
MONTEREY, CALIFORNIA

VOLUME I

90-HC

PREFACE

John L. Gustafson, *Program Chairman*

The Fourth Conference on Hypercubes, Concurrent Computers, and Applications (HCCA4) was held March 6–8, 1989, in Monterey, California. Over 600 people attended, and about 250 papers were presented in this fast-growing area. The number of institutions actively pursuing distributed-memory computing has grown from about 10 in 1983 to over 100 presently.

The corresponding growth in the size of this conference led to the need to referee submitted papers by abstract. Previous HCCA conferences have had a 100% acceptance rate, but only 70% of the submitted papers were accepted for presentation at HCCA4. Originality and relevance to distributed-memory computing were the main filtering criteria.

ORGANIZATION OF THE PROCEEDINGS

These Proceedings are organized along the same lines as the Conference: the Introduction by Geoffrey Fox, followed by three major divisions by topic: Hardware, Software, and Applications. Within each major division are more specific topics such as Fluid Dynamics or Neural Networks. Within each specific topic, papers are ordered alphabetically by first author, except for Mini-Symposia. To preserve the organization of the Mini-Symposia, papers are ordered in the sequence in which they were given.

The distinction between both major and specific topics is frequently difficult. Should Matrix Algebra be placed in Software or in Applications? If a paper deals with the performance of a graphical PDE solver on a novel architecture, should it be classified under Performance Evaluation, PDE Solvers, Input/Output, or New Hardware? The reader is cautioned that the investigation of any subject within this Proceedings, such as Fast Fourier Transforms, might require perusal of several scattered sections.

HARDWARE

The Hardware section includes Decomposition Methods, Fault Tolerance, Input/Output, New Hardware, Performance Evaluation, Routing and Topologies, and Shared Memory. Many authors who certainly do not consider themselves electrical engineers or computer designers might be surprised to find their papers classified under "Hardware." The guiding rule for putting a paper in this section was that it depended on a knowledge of a specific underlying computer architecture, whether that architecture was the subject of the paper or not. Papers on Decomposition Methods or Routing and Topologies, for example, usually deal with optimizing the mapping of application topologies to hardware interconnection topologies. Fault Tolerance can be done with either hardware or software, but in all cases the faults being tolerated are in the hardware, not the software.

Perhaps the majority of the papers at HCCA discuss performance in some respect, but as a *means* to understanding the value of some approach. The Performance Evaluation section includes those papers which centered on the *problem* of evaluating computer performance.

The Shared Memory category deserves some explanation. While its existence might seem contradictory in a conference dedicated to distributed memory, several researchers have endeavored to provide a shared memory software environment on hypercubes and similar computers. The HCCA focus does not include computers with hardware for shared memory, since many other forums exist for exploring that approach to parallelism.

SOFTWARE

The Software section includes Algorithms, Databases, Languages, Libraries and Tools, Load Balancing, Matrix Algebra, MCC Minisymposium, NP-Hard Problems, and Parallel Environments. Although some of these topics seem more like Applications (Databases especially), papers in Software tend to focus on the underlying issues (kernel operations, operating systems, user interfaces, techniques for efficiency) rather than complete solutions for a particular application.

The Parallel Environments was the single largest category of papers; having shown that hypercubes and similar computers work and for some things work very well, many people are now turning their energies to making them easier to program and use. The conflict between performance via novelty and ease-of-use via compatibility is probably more intense now than at any time in the history of computing.

APPLICATIONS

Dozens of applications were presented at HCCA4, adding strength to the view that "special purpose" might not be an accurate adjective for distributed-memory computers any more. Among the clearest successes have been Fluid Dynamics, Image Processing, Neural Networks & Vision, PDE Solvers, and Structural Analysis. The Databases papers in Software imply that hypercubes might soon be ready to expand from scientific applications to mainstream business applications such as transaction processing.

Where three or more papers on a particular application were accepted, a separate session was organized on that application at HCCA4. Other papers on applications were simply categorized as "Other Applications," and that same subdivision exists in the Proceedings.

To echo the sentiments of Geoffrey Fox in his Introduction, this is the first year that hypercubes have really made a difference. They are being used to make scientific discoveries that could not be made by other means; they are being used for production computing at Fortune 500 companies; third-party vendors are committing to distributed-memory versions of major software packages. In general, distributed-memory computers are starting to achieve their long-promised higher performance and superior price/performance compared to conventional computers. After several false starts, parallel computing is finally blooming.

—*John Gustafson*

The Program Committee wishes to extend their appreciation to the following sponsors for their financial support of the Conference:

U.S. Department of Energy, Applied Mathematical Sciences Program
Strategic Defense Initiative Organization, Office of Innovative Science and Technology
Joint Tactical Fusion Program Office
U.S. Air Force, Electronic Systems Division
Air Force Office of Scientific Research
NASA Ames Research Center

And to all the members of the Organizing Committee who devoted their precious time in the planning and implementation of the technical program:

Don Austin
Department of Energy

Tony Chan
University of California at Los Angeles

Terry Cole
Jet Propulsion Laboratory

Erik DeBenedictis
ATT/Bell Labs

Geoffrey Fox
California Institute of Technology

Michael Heath
Oak Ridge National Laboratory

Paul Messina
California Institute of Technology

Gary Montry
Sandia National Laboratories

Ed Oliver
Air Force Weapons Lab

Quentin Stout
University of Michigan

And to Sandia National Laboratories, the Host Institution for this Conference, whose support and guidance were most valued.

The Program Committee:

Gil Weigand
General Chairman
Sandia National Laboratories

John Gustafson
Program Chairman
Sandia National Laboratories

Bill Hickey
Conference Administration

A Hypercube Control Network for a Circuit-Switching Interprocessor Network

Qiang Li and Naphtali Rishe

School of Computer Science
Florida International University–
The State University of Florida at Miami
Miami, Florida 33199

Abstract

This paper presents a hypercube network of control processors employed to control a passive circuit-switching network, namely the three-stage Clos network[2] which is used to interconnect a large number of processors in a highly parallel distributed-memory system. The hypercube network relaxes the bottleneck caused by using a single control processor in both the speed of processing the circuit setup requests and the communication capacity between the control processors and others processors. The combination of the hypercube network and the circuit-switching network is a component of the implementation of a massively parallel database machine described in [4].

Keywords: Parallel architectures, Distributed memory, Inter-processor network, Circuit-switching, Parallel control of circuit-switching.

1. Introduction

We are developing a massively parallel database machine, LSDM[4]. LSDM is a distributed memory and secondary storage system and is to consist of thousands of processing units. Each processing unit consists of a processor, a memory module and a secondary storage device. A hybrid of the packet-switching network and the circuit-switching network is developed to connect the processors. The main characteristic of the network is that it shows a fast response to the short messages and a high bandwidth to the long messages. In the authors' opinion, this is a promising network architecture for large scale distributed memory systems.

A part of the inter-processor network is a circuit-switching network, namely a three stage CLos network[2]. In this network, a dedicated path can be set up simultaneously for each pair of processors. Since the circuit-switching network is a passive network, a controller is needed to setup the circuits upon request. The speed of the controller imposes a limit on the performance of the network. This paper presents a control processor network consisting of 32 processors, instead of one controller, connected into a 5-dimensional hypercube. The circuit setup requests will arrive at one of the 32 processors depending on where the requests are originated. A processor receiving a request will handle the request with cooperation of the other processors. In this way, the requests arrive through 32 independent channels and are processed in parallel by 32 processors. Thus the controller bottleneck is significantly relaxed. To clarify the discussion, we call the processors in the hypercube network the "control processors" and the other processors the "data processors".

This research has been supported in part by a grant from the Florida High Technology and Industry Council.

2. The Circuit-Switching Network

Figure 1 shows the structure of the three-stage Clos network. Each box in the figure is a crossbar switching component. If a switching component has n input lines and m output lines, we say that it is an $n \times m$ switch. There are three columns of switches, the input switches, the output switches and the intermediary switches. The links between the switches are one-directional. The left-

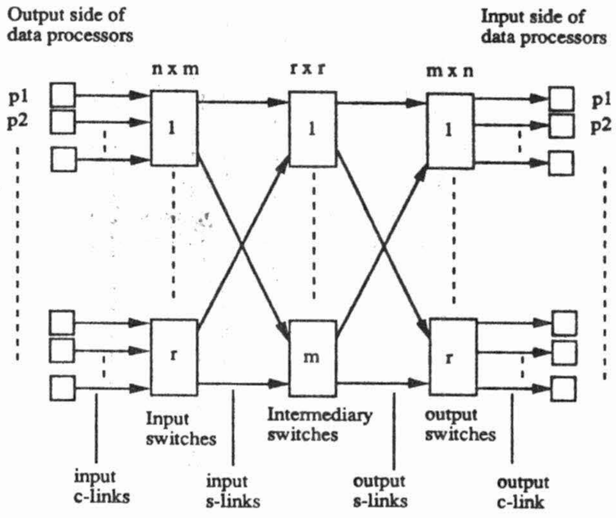


Figure 1: The three-stage Clos network

most column is the output side of the data processors and the rightmost column is the input side of the data processors. For convenience, we call the links between the data processors and the input switches the "input c-links"; the links between the data processors and the output switches the "output c-links"; the links between the input switches and the intermediary switches the "input s-links" and the links between the intermediary switches the "output s-links".

The input switches are $n \times m$ switches; the output switches are $m \times n$ switches; the intermediary switches are $r \times r$ switches. Such a network is denoted $N(m, n, r)$. There are two important properties of an $N(m, n, r)$ network proven in [1,2]:

1. If $m \geq n$ then for every partitioning of the set of all processors into pairs there exist connection configurations where all the pairs talk simultaneously, i.e., there is no bottleneck in the switches.
2. When $m \geq 2n - 1$, the network is non-blocking, meaning that there is always a path available between any idle input c-link and any idle output c-link, independently of the connection sequence, i.e., there is no need to prearrange a special connection configuration of Property (1) above in order to avoid bottleneck.

At the hardware level we are concerned with, the switching components are normally passive, i.e., the connections have to be made by an outside controller.

We assume that the circuits between data processors are always bidirectional. Thus, whenever a path from a data processor i to a data processor j is set up, a path from j to i is also set.

When a data processor *orig* needs to communicate with another data processor *dest*, the former will send a circuit set up request to a controller. The controller will make the necessary connections and then inform *orig*. Then *orig* will send its data through the dedicated circuit to *dest*. Once the circuit is setup, the bandwidth of the circuit can be fully utilized and very large data sets can be sent efficiently.

Assume that x data processors are connected to the network and $m \geq n$ where m, n and r are the parameters of $N(m, n, r)$. Then, $x/2$ pairs of processors can communicate simultaneously. If one circuit has bandwidth of b bits per second, the total bandwidth of the network is $b \times (x/2)$ bits per second.

3. The Hypercube Control Network

The circuit controller of the circuit-switching network is responsible for selecting an available route for setting up the circuit, keeping track of the current status of the network, maintaining a queue of the connection requests unable to be satisfied for the time being, etc. The circuit controller is also responsible for sending hardware signals to actually set up the circuit.

The speed of the circuit controller imposes a limit on circuit setup time and circuit setup rate, i.e., the number of circuits that can be setup per time unit. The problem comes in two aspects. First, the large amount of requests coming from the data processors have to be converged to the controller, which presents a communication bottleneck. Second, the processing speed of the controllers

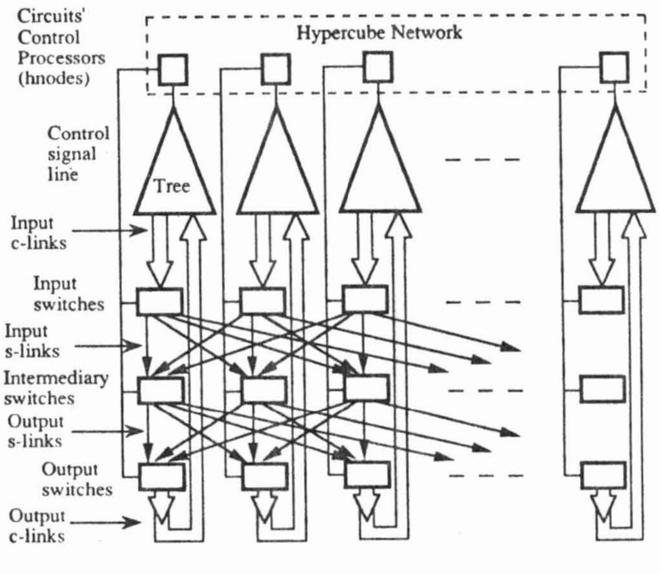


Figure 2: The connections between the *hnodes* and other parts of the network

must be high enough to handle the flow of the requests. To alleviate the problem, a group of control processors are employed for the task. The circuit setup requests will arrive at one of the control processors depending on where the requests have originated. A control processor receiving a request will process the request with cooperation of the other control processors. In this way, the requests arrive through many independent channels and are processed in parallel by many processors. Thus the controller bottleneck can be significantly relaxed.

As mentioned above, the control processors are linked into a hypercube network. Each node of the hypercube is called an *hnode*. Figure 2 shows the relationship between the *hnodes* and the other components of the network.

Each triangle in the figure represents a group of data processors and they are so connected that the circuit setup requests are converged to the top of the triangle. We call each triangle a tree. The switches in the figure are logical switches, i.e., each switch can be composed of a group of switching components. For simplicity, we treat each switch in the figure as one component. Each switch is connected to a set of data processors. We say that a switch is connected to a tree if the switch is connected to the data processors in the tree. Each *hnode* controls the switch connected to its tree and one or more intermediary switches. For convenience, an intermediary switch was drawn together with each pair of input and output switches. In practice, the number of input and output switch pairs and the number of intermediary switches are often not the same. Further, only a few input s-links and output s-links were drawn.

Each *hnode* keeps the following information about every switch under its control:

- Input Connection Status:
  A vector, called "input vector", indicating which in-

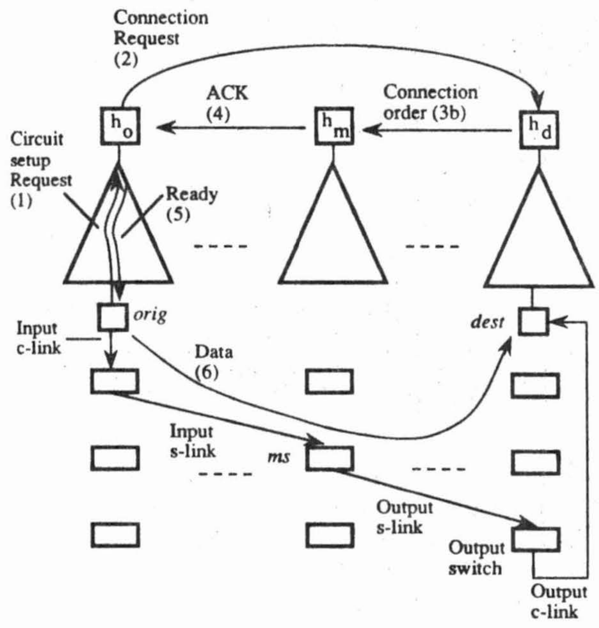


Figure 3: A process of setting-up a circuit

put s-links connected to the input switch are available.

A list of the current connections between the input c-links and the input s-links connected to the input switch.

- Output Connection Status:
  A vector, called the "output vector", indicating which output s-links connected to the output switch are available.

  A list of the current connections between the output c-links and the output s-links connected to the output switch.

- Intermediary Connection Status:
  A matrix, called the "relay matrix", indicating the current connections between the input s-links and the output s-links on the intermediary switches.

We now describe the process of setting-up a circuit. To simplify our discussion, let us assume the circuit-switching network is a non-blocking network, i.e., as long as two processors to be connected are not previously connected to someone else, there is a path available between them.

Suppose the circuit-setup request is originated by a data processor, *orig*, and the destination data processor is *dest*. Let h_o be the root of the tree that *orig* is in, and h_d be the root of the tree that *dest* is in. Note that *orig*, h_o, h_d and *dest* are all defined in terms of a particular circuit setup request. Figure 3 shows a flow diagram of the process of setting-up a circuit. The messages flow between the *hnodes* for the purpose of setting-up circuit have the general form:

Message-type
Message Destination (*hnode* number)
Message Originator (*hnode* number)
. . . . (other information)

The content of the message after the third field depends on the type of messages and will be described individually. The algorithm is described by the following steps.

1. Starting from *orig*, a circuit-setup request message will be sent up the tree until it reaches h_o.
2. Upon receiving a circuit-setup request, h_o will first determine the position of h_d in the hypercube; then send a "connection request" to h_d. The connection request has the following format:

Message-type (CR)
Message Destination (h_o)
Message Originator (h_d)
Destination data processor number (*dest*)
Input vector of h_o

3. Upon receiving the connection request, the destination *hnode* h_d will first determine if the c-link to the destination data processor is already occupied, i.e., the destination data processor is talking to someone else.
   (a) If the c-link is busy, a busy message is sent to h_o, and a waiting record holding the number of h_o will be inserted into the waiting list of the desired c-link. Upon receiving a busy message, h_o will put the circuit setup request into a waiting list until h_d calls back.
   (b) If the c-link is free, h_d will check the input vector of h_o against its own output vector to find an intermediary switch which has free links to connect h_o to h_d. Since we assume that the network is a non-blocking network, a intermediary switch will be found. Let *ms* indicate the intermediary switch selected and h_m indicate the *hnode* which controls *ms*. A "connection order" is sent to h_m. The connection order has the following format:

Message-type (CO)
Message Destination (h_m)
Message Originator (h_d)
Switch Number (*ms*)
Request Originator h_o

   In the meantime, hardware signals are sent to the output switch to connect the output c-link and the output s-link connected to *ms*.

4. Upon receiving the connection order, h_m will send hardware signals to *ms* to connect the input s-link and the output s-link, and send an "ack message" to h_o indicating that the requested circuit is setup. The ack message has the format:

Message-type (ACK)
Message Destination h_o
Message Originator h_m
Switch Number (ms)
Request Destination (h_d)

5. Having received the ack message, h_o will send a signal to the input switch to connect the input c-link to the input s-link which links to the intermediary switch indicated by ms. A "ready message" is then sent to the originating data processor *orig*.

6. Upon receiving the acknowledgement from the control processor network, the data processor will send its data through the circuit, which effectively initiates the communications. Thereafter, the two connected data processors can communicate in the way they choose.

When the data processor *orig* finishes using the circuit, a release circuit control message is sent to h_o, and h_o will in turn send a release message to h_d and h_m. After all the involved *hnodes* updated their connection status, the circuit use cycle is completed.

When a circuit is being released, the *hnode* which is in charge of the output c-link of the circuit will check the waiting list on the c-link. If there are requests waiting, the first request will be removed and a "call back" will be sent to the *hnode* which originated the request. The *hnode* will start the above process from step 2 as the h_o.

In some situations, a problem can occur in the above algorithm. When an *hnode* sends a connection request to h_d before a previous connection request is acknowledged, its input vector sent with the connection request could be outdated since the previous connection request could have already selected an intermediary switch but has not informed the originating *hnode* yet. This can result in a situation where an *hnode* receives a connection order to connect an input s-link to an output s-link and finds that the input s-link has already been connected to someone else. We call this a "race condition".

Although there are several approaches to avoid the race conditions, we think the following is the best in terms of maximizing parallelism, preserving first-comes-first-served order, etc. In this approach, a connection request will be sent to the $h_{d'}$ of the last outstanding connection request (connection request which has not yet been acknowledged) unless there is no outstanding connection request, in which case, the connection request will be sent directly to the correct h_d. Figure 4 shows an example of the process. Connection request 1 is sent when there is no outstanding connection request from h_o. Connection request 2 is sent before connection request 1 is acknowledged. The input vector of h_o held in connection request 2 is modified by h_{d1} to indicate that a intermediary switch has been selected. If connection request 3 is sent before connection request 2 is acknowledged, the same modification will be made to the request as that made to connection request 2. In this way, the connection requests will always have an up-to-date in-

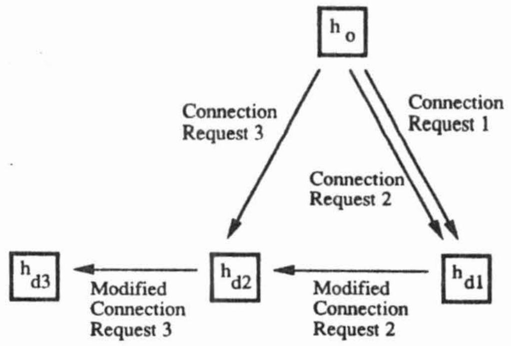


Figure 4: Illustration of the "race condition" elimination algorithm

put vector of h_o. In addition, the connection requests sent by each *hnode* needs a sequence number in order to avoid confusion when connection request 3 arrives at h_{d2} before the modified connection request 2 arrives. Although the traffic in the network is slightly increased since some connection requests do not travel in their shortest path, the overhead is under control and is statistically small.

If the circuit-switching network is not a non-blocking network, the situation will be more complicated. In step 3(b) of the algorithm, there is a possibility that there is no available path found. If that is the case, a "blocking message" will be sent to h_o and a record indicating that h_o is blocked will be inserted into the block-list of h_d. Whenever a circuit is released, h_d will send a message containing its output vector, and the message will travel according to the sequence of the *hnodes* in the block-list so that the *hnodes* will send their request again.

The experimental system that we are currently building consists of a number of INMOS transputers[3]. The configuration under investigation has 1024 processors. The circuit-switching network is a three-stage Clos network, $N(32, 32, 32)$, consisting of 96 INMOS C004 dynamic reconfigurable switches (32×32).

Acknowledgement

The authors gratefully acknowledge the advice of David Barton, Nagarajan Prabhakaran and Doron Tal.

References

[1] V.E. Benes. On rearrangeable three-stage connecting networks. *The Bell System Technical Journal*, 1481–1492, Aug 1962.

[2] C. Clos. A study of nonblocking switching networks. *The Bell System Technical Journal*, 406–424, Mar 1953.

[3] INMOS Corporation. *Transputer Architecture Reference Manual*. INMOS Corporation, Bristol, U.K., 1986.

[4] N. Rishe, D. Tal, and Q. Li. Architecture for a massively parallel database machine. *Microprocessing and Microprogramming. The Euromicro Journal*, 1988. In press.