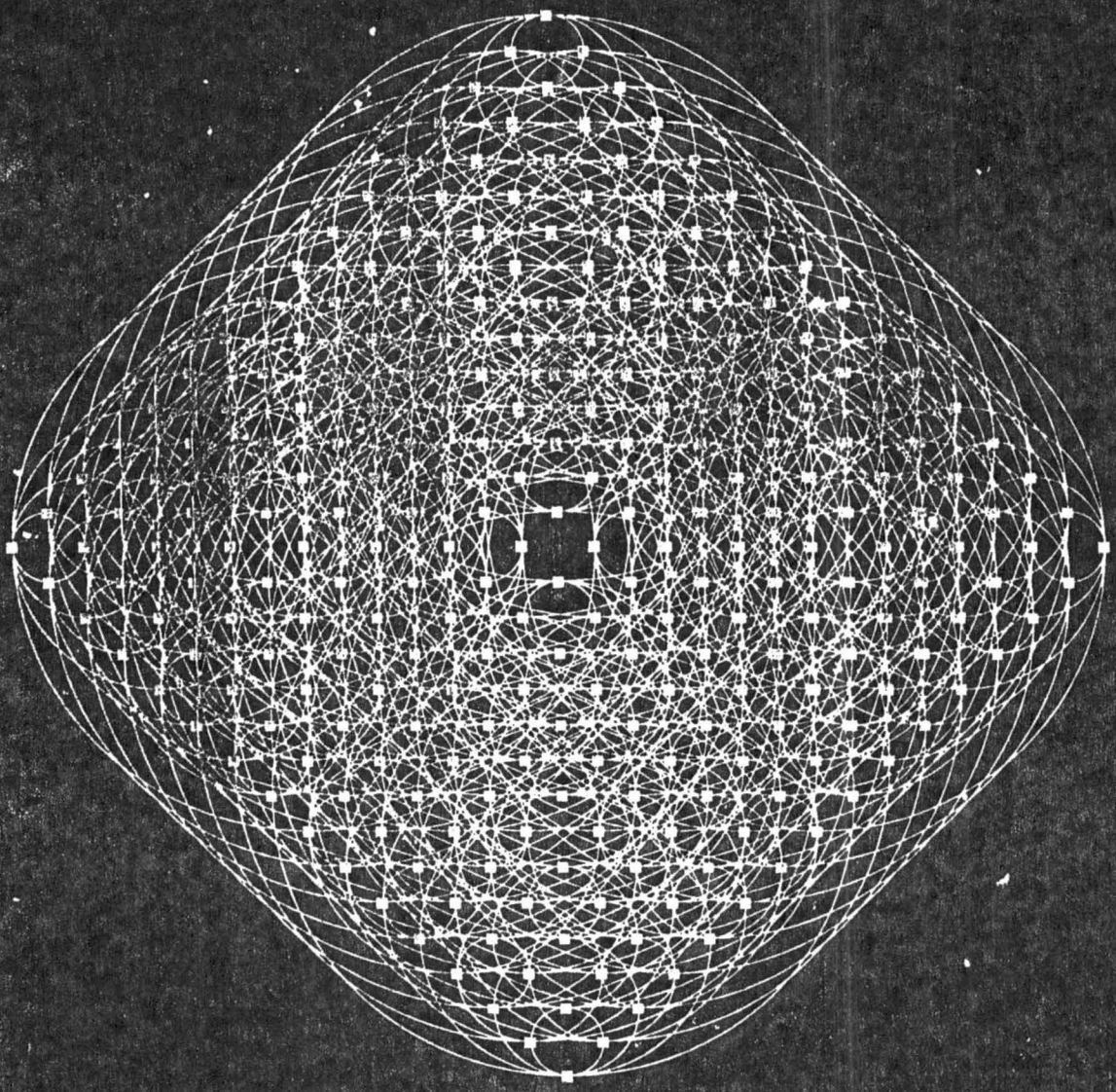
90-PI

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

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 Proof of Impossibility of Deadlock in a Fixed-routing Hypercube Network

Naphtali Rishe and Qiang Li

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

Abstract

This paper presents a proof of the fact that a store-and-forward packet-switching binary hypercube network with the fixed routing algorithm is deadlock free. A graph model of the network's status is used for the proof.

Keywords: Hypercube, Deadlock, Packet switching, State graph.

1. Definition of the system

Although it is assumed that the readers of this paper are familiar with the hypercube networks, it is important to have a clear definition of the network and the routing algorithm in order to guarantee the correctness of the proof. In a binary hypercube, each node is assigned a unique binary number as its identification number (id-number) or address in the network. Two nodes are directly connected in a hypercube iff their id-numbers differ in exactly one bit. In a D–dimensional binary hypercube, each node is connected to exactly D nodes. A connection between two nodes consists of two channels, one for each direction. Therefore, each node has D input channels and D output channels. An input (output) channel of node N is numbered as the i-th input (output) channel of N if it is connected to node M such that N and M differ in the i-th bit. Bit positions are counted from 1 to D.

The hypercube networks can be used in different fashions: pattern mapping or random store-and-forward packet switching. When used in the pattern mapping fashion, the avoidance of deadlocks purely depends on the correctness of the algorithm. On the other hand, when a hypercube network is used in a random store-and-forward packet switching fashion, there are different methods to route the messages in the network. We are interested in the most straightforward one, i.e., the fixed routing algorithm as follows.

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

Algorithm:

Let *dest_id* be the id-number of the destination node of a message being received by node *my_id*.

1. Receive a message
2. If *my_id* = *dest_id* then
   - Consume the message
3. Else
   - i := bit position of the lowest bit that *my_id* and *dest_id* differ in;
   - Send the message to the i-th output channel.
4. Repeat steps 1–3 forever. #

Like in other store-and-forward packet switching networks, a hypercube working in this fashion could potentially have a deadlock situation since many cyclic graphs exist in a hypercube. Effort has been made to solve the deadlock problem [4,6,2,1,3]. However, additional overhead had to be introduced to prevent the deadlock. The overhead can be very heavy sometimes. This paper will prove that the fixed routing algorithm of a hypercube is deadlock free, i.e., one can safely use the routing algorithm without any deadlock–prevention overhead.

The system in question is assumed to work as follows. Figure 1 shows a simplified node in a hypercube. For each input or output channel, there is an independent process to handle the incoming or outgoing messages. The data transfer between the processes is through buffers. There is a buffer for each of the output channels of a node. The buffer is big enough to hold the longest message. A message

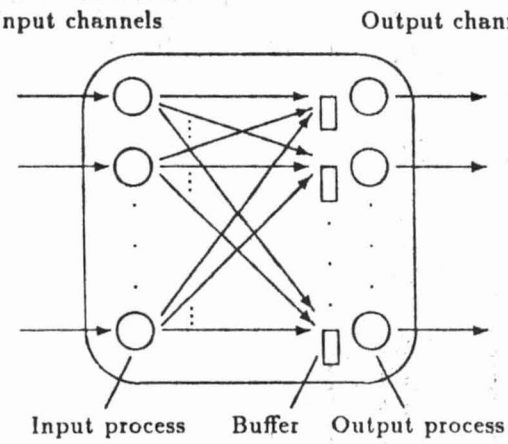


Figure 1: A sample node of a hypercube

packet has a header which contains its destination address and some other information. When node A wants to send a message m to node B, A will send the header to B first and B will decode the header. If B happens to be the destination of m, B will inform A to send the entire message and m will be taken off the network after the receiving. Otherwise, B will examine the buffer for the output channel where the message m would go. If the buffer is empty, B will inform node A to send the entire message and put the message into the buffer; a separate mechanism will try to send the message in the buffer through the output channel. If the buffer is occupied, the message will have to wait in A's buffer until B is able to accept the message.

Although the buffer and process setup can vary from implementation to implementation, they are equivalent to the setup we laid out here as long as the channels are handled asynchronously.

Since the availability of a buffer is indistinguishable from the availability of the channel to which the buffer is attached, we identify the buffers as the resources which are requested and used by messages. Let $\overline{M}_i$ denote the number obtained by flipping the i-th bit of M. We define the following:

- A buffer is denoted by $B(M, N)$ if it is a buffer of node M and is exclusively associated with the output channel of M connecting to node N.
- A buffer $B(M, N)$ is called an i-th level buffer if M and N differ in the i-th bit, i.e., $N = \overline{M}_i$.
- A message in an i-th level buffer is called an i-th level message at the time it is in the buffer.

We further assume that the buffers are non-preemptive and exclusive, i.e., a message in the buffer of an output channel must be sent out before another message can occupy the buffer and there can not be two messages in a buffer at the same time.

It is important to understand that when we normally say that a message is waiting for a channel, the message is actually waiting for the buffer selected by the algorithm on the other end of the channel. It is the *waiting for a buffer* that plays a basic role in causing deadlocks. To expose this fact, we need a modified algorithm which is equivalent to the original algorithm but expressed in terms of buffers so that the buffer waiting can be made explicit. The equivalent algorithm is the following.

Algorithm:

Let *dest_id* be the id-number of the destination node of the message being put into a buffer $B(M, N)$.

1. Wait until a message is put into buffer $B(L, M)$
2. If $M =$ *dest id* then
   - Take the message out of the network
3. Else
   - $i :=$ bit position of the lowest bit that M and *dest id* differ in;
   - Wait for $B(M, \overline{M}_i)$ to be available
   - Put the message into the buffer $B(M, \overline{M}_i)$.
4. Repeat steps 1–3 forever. #

It is not difficult to justify the equivalence of this algorithm and the previous one. We will leave it without further elaboration. Notice that this is a virtual algorithm: although it is functionally equivalent to the previous one, it can not be implemented directly.

2. Proof of the Impossibility of Deadlock

A state graph G of the hypercube network describes the network's status at a given moment in time. G is defined as a labeled directed graph as follows:

- A vertex $V(M, N)$ is in the state graph G iff $B(M, N)$ is a buffer of the network.
- The graph contains a directed arc from vertex $V(L, M)$ to $V(M, N)$ if there is a message in buffer $B(L, M)$ requesting the buffer $B(M, N)$.
- A vertex $V(M, N)$ of G is called an i-th level vertex if $B(M, N)$ is an i-th level buffer in the hypercube.

At different moments in time, the state graphs are normally different. Their sets of arcs are different, though their sets of vertices are the same. Notice that there is a one-to-one correspondence between the buffers of the network and the vertices of the state graph G. Introduction of the notation V is merely to help to build a clear mathematical model.

Lemma 1:

Let G be a state graph of a hypercube with the fixed routing algorithm, and $V(L,M)$ and $V(M,N)$ be two vertices of G. If the level of $V(M,N)$ is less than that of $V(L,M)$, then there is no arc from $V(L,M)$ to $V(M,N)$ in the state graph G.

Proof:

Since the existence of an arc from $V(L,M)$ to $V(M,N)$ indicates that a message in $B(L,M)$ is requesting buffer $B(M,N)$, it is sufficient to prove that a message can only request buffers with a level higher than its own. We shall prove this by induction on the number of buffers that a message has requested.

Let *Dest* denote the id-number of the destination of the message.

Basis:

A newborn message has level 0, and every buffer has level greater than 0. Therefore, when a new message requests the first buffer on its path, the buffer has a higher level number than that of the message.

Hypothesis:

Assume that for the first n buffers on its path, a message always requests a buffer of higher level than its own.

Induction:

The message is in its n-th buffer $B(L,M)$ and requesting its $(n+1)$-th buffer $B(M,N)$. Suppose $B(L,M)$ has level i, then $M = \overline{L}_i$. Thus, by definition, the message has currently level i. We need to show that $B(M,N)$ has a level of at least $i+1$.

Since the buffer selected always has a level equal to position of the lowest different bit of M and *Dest*, it is sufficient to show that M and *Dest* have at least the lowest i bits the same.

Since $B(L,M)$ has level i, L and *Dest* have the same lowest $i-1$ bits, because otherwise the message would not have been put into $B(L,M)$ according to the algorithm. But L and M have the same lowest $i-1$ bits since $M = \overline{L}_i$. Thus, M and *Dest* have the same lowest $i-1$ bits. Since L and *Dest* differ in the i-th bit and L and M differ in the i-th bit, M and *Dest* must have same i-th bit. Thus, M and *Dest* have at least the lowest i bits the same, i.e. $N = \overline{M}_j$ where $j > i$. Therefore, the buffer $B(M,N)$ has level greater than i which is the level of the message. #

Definition:

A directed loop of a state graph of a hypercube network is a sequence of vertices $V(N_1,N_2)$, $V(N_2,N_3)$, .., $V(N_{k-1},N_k)$, $V(N_k,N_1)$ such that there is a directed arc from the first vertex to the second vertex of every two consecutive vertices in the sequence, and from the last vertex to the first vertex of the sequence.

Lemma 2:

Every directed loop of the state graph G has vertices of at least two different levels.

Proof:

Let $V(L,M)$ and $V(M,N)$ be two consecutive vertices of a directed loop. Then, $B(L,M)$ and $B(M,N)$ must be two buffers of the hypercube. By the definition of the hypercube, L and M can not differ in the same bit as M and N do, i.e., $M = \overline{L}_i$ and $N = \overline{M}_j$, where $i \neq j$. Therefore, $B(L,M)$ and $B(M,N)$ have different levels, i.e., $V(L,M)$ and $V(M,N)$ have different levels. #

Having proven the above properties of hypercube with fixed routing algorithm, we now turn to the deadlock problem. We have assumed that the buffers as resources are non-preemptive and exclusive and a message can hold a buffer and request another. It has been shown that, under our assumptions, a directed loop in the state graph is necessary and sufficient condition for a deadlock[5]. We use this result as our definition of deadlock.

Definition:

The hypercube network is deadlocked if and only if there is a directed loop in the state graph of the network.

The intuitive interpretation of the definition is the following. The hypercube network is deadlocked if and only if there is a circle of buffers in the network such that there is a message in each of the buffers requesting the next buffer in the circle.

We now prove the main theorem.

Theorem:

Any hypercube network with the fixed routing algorithm is deadlock free.

Proof:

If there were a deadlock in the hypercube, there would be a directed loop in the state graph of the

network. By Lemma 2, the vertices of the loop have at least two different levels. Therefore, there must be two consecutive vertices, $V(L,M)$ and $V(M,N)$ in the loop such that $V(L,M)$ has a higher level than that $V(M,N)$ does. But, by Lemma 1, there is never a directed arc from $V(L,M)$ to $V(M,N)$. Therefore, there can not be a directed loop in the state graph. Thus, there is no deadlock in any hypercube with the fixed routing algorithm. #

Acknowledgement

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

References

[1] D. Gelernter. A DAG-based algorithm for prevention of store-and-forward deadlock in packet networks. *IEEE Transactions on Computers*, C-30:709–715, Oct. 1981.

[2] I.S. Gopal. Prevention of store-and-forward deadlock in computer networks. *IEEE Transactions on Communications*, COM-33(12):1258-1264, 1985.

[3] K.D. Gunther. Prevetion of deadlocks in packet-switched data transport system. *IEEE Transactions on Communications*, COM-29:512–524, April 1981.

[4] J.W. Havender. Avoiding deadlock in multitasking systems. *IBM Systems Journal*, 74–84, 2 1968.

[5] E.G. Coffman Jr., M.J. Elphick, and A. Shoshani. System deadlocks. *Computer Survey*, 3(2):67–78, June 1971.

[6] A. Shoshani and E.G. Coffman. Prevention, detection, and recovery from system deadlocks. In *Proc. of 4th Annual Princeton Conference on Information Science and Systems*, March 1970.