IS-T 1704

Development of a Low-Latency Scalar Communication Routine on Message-Passing Architectures

by

Pai, Rekha

MS Thesis submitted to Iowa State University

Ames Laboratory, U.S. DOE

Iowa State University

Ames, Iowa 50011

Date Transmitted: January 11, 1994

PREPARED FOR THE U.S. DEPARTMENT OF ENERGY

UNDER CONTRACT NO. W-7405-Eng-82.

gtb

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I wish to express my sincere appreciation for Dr. John Gustafson who has helped me throughout the course of my research. He has been friendly, flexible, understanding and extremely motivating. I wish to thank Dr. C.T. Wright for consenting to be my advisor and for his helpful discussions. I wish to thank Dr. James A. Davis and Dr. John Gustafson for agreeing to be on my committee. This research is supported in part by the Applied Mathematical Sciences Program of the Ames Laboratory which is operated for the U.S. Department of Energy under contract No. W-7405-ENG-82.

CHAPTER 1. INTRODUCTION

One of the most significant advances in computer systems over the past decade is parallel processing. Parallel processing has become a common approach for achieving high performance. However, applications continue to demand more computing power. Computational problems in areas such as high-speed aircraft design, medical imaging, and research in advanced structural, electronic and optical materials often require computers that are at least three orders of magnitude faster than the fastest computers presently available.

Although computing demands are increasing rapidly, the performance of conventional, sequential computers is approaching the point of diminishing return. High-performance sequential computers today are already bounded by, among other things, memory speed. Parallel computers, in which a number of processors can work in parallel on a single application, offer the only solution capable of providing orders of magnitude of improvement in computing performance without excessive costs. Accelerated efforts in the area of parallel processing have resulted in the successful development of many parallel processing systems. Examples of such machines are shared memory parallel computers like Alliant, Encore, Sequent, and Cray Y-MP, Distributed machines such as the Transputer, Warp, and hypercubes and SIMD machines such as the Connection Machine and the Maspar. Successful use of these parallel computers has been demonstrated in a number of application areas, including scientific computing, signal and image processing, and logic simulation. For some of these applications, the available parallelism increases as the problem size expands, making it possible to achieve close to linear speedups on parallel machines.

To be scalable to a large number of processing nodes and to be able to support multiple levels and forms of parallelism and its flexible use, new parallel machines have to be *multicomputer* architectures that have general networking support and extremely low internode communication latencies. The performance of a program when ported to a parallel machine is limited mainly by the internode communication latencies of the machine. Therefore, the best parallel applications are those that seldom require communications which must be routed through the nodes. Thus the ratio of *computation time* to that of *communication time* is what determines, to a large extent, the performance metrics of an algorithm. The cost of synchronization and load imbalance appear secondary to that of the time required for internode communication and I/O, for communication intensive applications.

To examine the extent to which this is true, consider a hypercube architecture like the nCUBE 2. All memory is distributed in the nCUBE architecture. Information is shared in the form of messages between processors by explicit communications across I/O channels. A message in the nCUBE-1 requires about 0.35 milliseconds to start and then continues at an effective rate of 2 μ seconds per byte [13]. Suppose an application requires 400 Kbytes for variables on one node (50K 64-bit words). If distributed over 1024 processors, each node will have only 50 variables in its domain. For a typical timestepping problem, each variable might involve ten floating-point operations (120 μseconds) per timestep, for a total of 6 milliseconds before data must be exchanged with neighbors. Data exchange might involve four reads and four writes of 80 bytes each, for a worst-case time of $(4 + 4) \times (350 + 80 \times 2)$ μseconds, or about 4 milliseconds [13]. Therefore when a single-node problem is distributed on the entire 1024-processor ensemble, the parallel overhead on the nCUBE will be about 40 percent.

Since the communication overhead is a significant component of the parallel execution time, it is important to reduce inter-processor communication time. The inter-processor communication time consists of two components: *message startup time*, and the actual DMA *transfer time.* The hardware of the communication channels for each node are, in principle, capable of operating concurrently with the processor itself and with each other, up to the point where memory bus bandwidth is saturated. Therefore, there is a possibility of saving communication time by judicious reorganization of data and computation within the application. However, the DMA channels are managed by software running on the processor. The software creates overhead that limits the extent to which the communications can be overlapped. In particular, the *message startup time* dominates the parallel overhead, and limits the fine-grained parallel capability of multiprocessor ensemble. Reducing the startup time is the main strategy used in the developing the scalar communication call. To send and receive a message, the actual DMA transfers require 1.2 *μseconds* per byte [13]. Before a message can be sent over the DMA channel, however, it is first copied to a location in system buffer memory where messages are stored in a linked list format. Similarly after the message is received over a DMA channel, it is copied into a system buffer. The time for the copy and startup are cut down in the scalar call, and the major part of the latency of the scalar call is the time for the actual DMA transfer.

The following thesis is organized in chapters. The first chapter deals with the communication strategies in various message-passing computers. A taxonomy of inter-node communication strategies is presented in the second chapter and a comparison of the strategies in some existing machines is done. The implementation of communication in nCUBE Vertex O.S is explained in the third chapter. The fourth chapter deals with the communication routines in the Vertex O.S, and the last chapter explains the development and implementation of the scalar communication call. Finally some conclusions are presented.

CHAPTER 2. COMMUNICATION STRATEGIES IN MESSAGE-PASSING COMPUTERS

Inter-processor Communication in iWARP Systems

iWARP, jointly developed by Carnegie Mellon and Intel Corporation, is a private-memory architecture. An iWARP system can include a large number of building blocks or cells. Each iWARP cell is a custom VLSI single chip processor, called iWarp, which contains both a powerful computation processor (20 MFLOPS) and a high throughput (320 Mbytes/sec), low latency (100-150 ns) communication engine. Explicit data transfer from one processor to another is the principal form of inter-processor communication. There are two possible sources for the data when one processor wants to transfer data to another processor. Either the data have been computed earlier, stored into memory, and then transferred directly out of memory, or the data are computed "on-the-fly", that is the data are sent directly from the computation engine of the processor. Based on these two sources of data, there are two styles of communication in iWARP [14].

- **Memory Communication**

  In message-passing or memory-to-memory communication, messages are first built in the local memory of the sending cell and then delivered (as a unit) to the local memory of the receiving cell. The user program running on the cell is insulated from communication. In the case of the sending cell, the network software handles the delivery of the message over the network only after the complete message has been built in the local memory of the sending cell. Similarly in the case of the receiving cell, the user program will operate on the data in the message only after the entire message has been delivered to the local memory by the network software. The advantages of memory communication are

1. Communication is decoupled from computation. While the message is being delivered and buffered through memory, the program at the sending or receiving cell can operate autonomously on its local data.
2. Communication protocols can be developed independently from the program to handle communication-specific issues such as deadlock avoidance and recovery from transmission lines.
3. Applications need not have detailed knowledge about intercell communication.

- **Systolic Communication**

All data sent along each direction in a can be viewed as belonging to one message. However, instead of waiting until all the data in the message have arrived, each cell operates on the data items within a message, as they arrive individually. It then sends the results of the computation to other cells on-the-fly of outgoing messages.

The advantages of systolic communication are

1. The message routing and header information overheads are not paid with each unit of synchronization. This makes it possible for the cells to cooperate in fine-grain parallel processing.
2. Incoming and outgoing data need not be buffered in the cell's local memory unless required by the computation. Extra transfers to and from memory are avoided, thus reducing the latency of communication.
3. Systolic inputs and outputs provide additional parallel sources of operands for instruction-level parallelism.
4. Avoiding the buffering of data in the local memory also reduces the memory size requirement for some applications.

However, systolic communication is harder to use than memory communication, because the local memory of a cell can be accessed *randomly*, while message queues in the communication agent can only be accessed *sequentially*. Therefore the burden of making sure that the reads and writes of message queues are properly sequenced falls on the programmer at user level.

iWARP Networks

The networks supported by the iWARP communication agent can either be *public* or *private* [15]. A public network allows possibly unrelated processes to transmit data over this same network, and the issues of network access, resource conflicts and contention have to be addressed by the software support for the network implementation. A private network is accessible only by processes participating in the same application. Public networks are suited for general-purpose computations where little or nothing is known *a priori* about the communication patterns. Private networks provide a chance for optimizations since they can be custom-tailored to a specific application to result in lower overheads and/or higher throughput.

Pathways A connection between two nodes is realized by a *pathway*, which establishes a link between these two nodes. Each pathway is unidirectional and data is transmitted from the source node to the destination node over this pathway. Pathways consists of one or more *pathway segments*, with each pathway segment connecting two adjacent nodes. Pathway segments are implemented by the communication agent by multiplexing the busses that connect adjacent nodes to create *logical channels*. The nodes on the pathway cannot access any data that is transmitted over the pathway (nor can they use this pathway to send data).

Sending and Receiving Messages At the User level, the program sets up networks and then gets access to those networks via a *port*; a port is a local name that identifies the network and the type of network access (input or output). To transmit data, the node must specify a *destination* and possibly a *route*. Ports are mapped into *gates*, an iWARP hardware resource that affects the actual transmission of data. Gates can be thought of as registers and can be addressed like registers by the instruction set, and are multiplexed between different ports.

One dimensional networks are built by first creating pathways between the nodes and then connecting the pathways to form a network. Each node starts by initializing a variable of type *network* to allow the local runtime system on the node to initialize appropriate tables. Then, to create a pathway between Node1 (the source node) and

Node2 (the destination node), Node1 invokes the function *create_pathway_out*, and Node2 invokes the function *create_pathway_in*. Both functions return a port that is then used to send or retrieve data.

In numerous applications, the design of the network is fixed at the time an algorithm is turned into a program for an iWARP array. In such cases, it is wasteful to build up the network at runtime, while it could easily be initialized at load time. Furthermore, in this case a static check is sufficient to determine if any node has insufficient resources (pathway segment records) to support the network(s) defined for the application.

The iWARP environment therefore allows the programmer to specify directions that are interpreted at load time and are turned into network information for each node, by calling the function *create_pathway*.

Data transfers are indicated by using variants of *send* and *receive*. There are functions available to either handle a single word or a block of data.

All data is encapsulated in messages, the basic operation is sending or receiving a message that is stored in memory. The user interface to the data transfer is as follows [15]

```
send_msg(port, destination, address, number_of_bytes);
receive_msg(port, address, number_of_bytes);
```

There also exist more primitive operations to create and receive the message header as well as the message trailer. These operations are needed for Systolic communication so that the data can be directly generated (consumed) by the computation unit. The following example illustrates how a program sends and receives a sequence of individual words [15].

```
port Yout, Yin;
int i, foo, var;
        send_msg_open(Xout, destination);
        receive_msg_open(Xin);
        for (i=0; i<N; i++) {
            foo = receive_int(Yin) + var;
            send_word(Yout, foo);
            }
        send_msg_close(Xout);
        receive_msg_close(Xin);
```

Data transferred over the network in encapsulated into units called *messages*. Each encapsulation unit consists of a header, the data to be transmitted, and a closing trailer to terminate the message. Data is encapsulated regardless of whether it is systolic or memory-memory communication.

The four busses (XR, YU, XL, YD) define four directions. For each of the input busses, there exists a default output bus. If the message header arrives and does not match the address of the current node, then the header (and the message) is forwarded to the default output bus. These defaults are $XRin \rightarrow XLout, XLin \rightarrow XRout, YUin \rightarrow YDout, YDin \rightarrow YUout$, and they are chosen so that messages keep traveling in the same direction, e.g from left to right.

If a message wants to change direction (that is, switch to a pathway segment that is multiplexed over a bus different from the default bus), the first word of the header is specially marked as "change_direction" [15]. This first word contains the address of the node where the change of direction is supposed to happen. The next word contains additional routing information (either the address of the destination node or the address of another node for a subsequent change of directions). When this message header reaches the node where it has to change (this is the node with an address that matches the address in the in the first header word), then this header word is removed, and the next word becomes the leading word that determines the destination (or node for the next direction change). There is no limit on the length of the header, but each change of direction removes one word from the header.

iWARP uses a variant of wormhole routing; as soon as the first word of the message header has been assembled in a node, this node can determine if the message is destined for this node or another node. If the message is destined for another node, and no change of direction is required, then the header is forwarded immediately. The testing and forwarding is done in 100 ns. If a change of direction is required, then the next word is examined to determine the new direction. This operation adds 50 ns. In both cases, further delay is possible if the outgoing pathway segment is already in use for the transmission of another message.

Thus iWARP achieves low-overhead communication by separating the set-up of the communication network from the actual transmission of data. There is a choice between a general two-dimensional network, that includes all nodes in the torus and a specialized one-dimensional (or two-dimensional) network. The application programmer (or program generator) is provided with an opportunity to customize the communication system to the needs of the application.

Communication in INMOS Transputer Networks

The INMOS T800 Transputer

The INMOS T800 Transputer contains a CPU, 4 Kbytes of on-chip memory, 1 MFLOPS floating-point unit, and four high-speed communication links, each with its own DMA Controller [12].

To run concurrent processes efficiently, the transputer maintains one queue each, for low-priority and high-priority processes. Low-priority processes are time-sliced automatically; when one is interrupted, it is placed at the tail of the low-priority queue and the next one is scheduled. High-priority processes are not time-sliced, and cannot be interrupted, even by other high-priority processes. A high-priority process runs till it is deschedules itself or is blocked by an I/O operation.

The transputer's CPU contains only three general-purpose registers, called **Areg**, **Breg** and **Creg**, arranged as a stack. The transputer also contains a workspace pointer register called **Wptr** which holds the base address of the active process's workspace. All local variable accesses are offset relative to this register. When a low-priority process is descheduled the values of **Wptr** and the instruction pointer **Iptr** are saved, but the contents of the register stack are not.

The instructions *in* and *out* are used in implementation of both internal and external communications in routing software. *in* takes the address of a data block, the address of a communications channel, and a byte count as register arguments. When *in* is executed to carry out internal communications, the transputer checks the value in the communications channel word. If this contains *mint* (the least integer the transputer can represent), then the other process taking part in the communication has not yet rendezvoused. The process doing the *in* puts its workspace pointer in the channel word and stores the base address of the data to be transferred in a reserved location in its workspace. When the other process involved executes an *out*, it finds a valid address in the channel word, so it copies its data into the first process's workspace and reschedules that process. (If the sending process reaches the communication first, the reciprocal steps are carried out).

In the case of external communication, the values of the register stack are given to the controller responsible for the link over which communication is taking place. When both the link controllers involved have been given their orders, they arrange for data to be transferred without further CPU intervention, rescheduling the communicating processes when the transfer completes. While external communication is slower than internal communication (as link bandwidth is lower than memory bandwidth), it retards other CPU activities slightly by stealing memory cycles for DMA.

Tiny: An Example Routing Kernel

An example processor-resident router called **Tiny** is explained in the following pages [12]. To users, Tiny is a kernel running on each processor, connected to one or more clients by channels. Each process in the network has a unique user-assigned client identifier (CID).

Since the networks using transputers have low number of interprocessor connections, and these connections are the ones with the least bandwidth, each link controller is given its own pair of processes (one each for input and output). Each client process (i.e. user process doing the communication) is connected to the router through its own input and output handlers, to decouple the interface procedures from

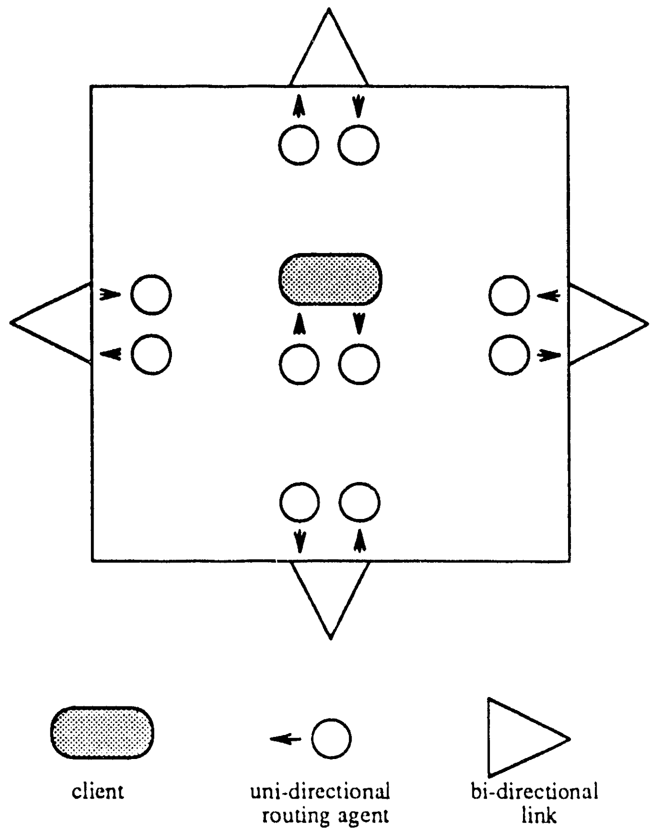


Figure 1: Structure of processes in a router of Tiny

the underlying router. These various output link handlers are buffered with FIFO queues to avoid head-of-line blocking problem. Each routing process is either a multiplexer or a demultiplexer, and is referred to as an *agent*. Through routing of messages has greater priority over the user's calculations. The demultiplexing agents have the routing logic which decides where to send messages. Thus the process structure in Tiny is as shown in Figure 1.

Internal Structure of Tiny Tiny is made up of several interacting agents on each processor. Each handles the input or output half of a link, or a single input or output channel connecting Tiny to a client user process. Each agent's workspace, or *A-page*, contains space for certain transputer instructions and an *agent handle* used directly by Tiny. Agents manipulate *buffers*, represented by *B-pages* which contain routing and ownership information and a pointer to the message's data. The part of B-page manipulated by Tiny is called its handle.

Typing of Messages In Tiny, messages are *typed* by sending them on different channels. The channels connecting the process to the router have different types and thus the process has a multi-channel interface. When there are multi-channel interfaces, agents handling traffic from clients to Tiny are the same as those in single-channel interfaces. An intermediate agent called *agent_multi_man* is used. All messages to a client C are sent to its *agent_multi_man*, which forwards the message to the subsidiary agent responsible for the channel through which that message should be delivered.

Routing Strategies In the routing of point-to-point messages, two strategies are used [12] which are given as follows.

- **sequential**

  This strategy guarantees that messages arrive in the order in which they were sent, which is useful for implementing fragmentation of large messages. The sequential strategy uses the same shortest path from any through-routing node towards the destination.

- **adaptive**

  This strategy decides locally at each through-routing node which of the shortest paths from that node to the destination seems likely to be the quickest. Using this strategy, Tiny examines the output queues of each of the through-routing node's appropriate links, and enqueues the message for the links with the shortest queue.

Interface The interface to Tiny is called PKT interface, and it copies data from the user's process into one of Tiny's internal buffers during a send, and copies back during a receive. The interface provides the user with four functions:

pktRead, pktWrite, pktSeqWrite and pktBroadcast.

The *pktRead* is a blocking read, and *pktWrite* and *pktSeqWrite* are non-blocking writes. The input parameters are the channel #, CID of the source or destination, a pointer to message buffer, and the size of the message. The functions do not return any value.

Structure of Agent Processes in Tiny The five different agents used within Tiny to implement the router are

- *agent_multi_man* This process demultiplexes messages arriving at a client with a multi-channel interface, passing those messages to the client's input agents. Its main loop dequeues a buffer, then enqueues it for a subsidiary agent.

- *agent_router_input*

  This process accepts messages on the input half of a hard link and routes them. Its main loop dequeues a buffer, gets a message into the buffer, and routes the buffer, putting it in another agent's queue.

- *agent_router_output*

  This process handles the output queue for a hard link. Its main loop dequeues a buffer, sends its contents down the link, then checks if the client has to send any more messages through any other link, and if not, returns the buffer to its owner.

- *agent_usrrpkt_input*

  This process accepts messages from a client process and routes them. Its main loop dequeues a buffer, gets a message into the buffer, makes a header for the message, then routes the buffer, putting it in another agent's queue.

- *agent_usrpkt_output*

  This process gives a message to a client process and frees the buffer. Its main loop dequeues a buffer, gives the buffer contents to the user, then checks if any more messages for that client on any other link, and if not returns the buffer to the owner.

Routing of Messages During Initialization, each node builds routing tables whose structure is as shown below:

free_ptr	link indices	routing segments	tables	free space

where

- free_ptr is an index into the *free space* used during initialization;

- the *link indices* are five indices (one for each physical link and another for a "pseudo-link" representing the user processes) indicating where in the *routing segments* the routing table for the messages entering the processor on that link is located;
- the *routing segments* contain a status word and a pair of indices into the *tables* for each process in the entire network;
- *tables* are lists of links to which messages are sent.
- *free space* is that part of the table which has not been used.

Each input link has a pointer to a routing table associated with it. These pointers may indicate the same routing table, or different routing tables. A fifth routing table is required for routing messages which originate in the user processes on the transputer. Each link's index indicates a segment of the form

status	$(txmt_0, brdcst_0)$	...	$(txmt_n, brdcst_n)$

where *status* indicates whether the table is bad (i.e. Tiny has not built the table yet), or in an array/index state (before initialization), or has been converted into pointers. The remainder of the segment contains pairs of indices into *tables* area. The table indicated by the first index is a negative-terminated list of the links through which messages for a particular processor may be sent; if there are several equally short paths to a processor, the list will contain more than one entry. The table indicated by the second index is a negative-terminated list of the links through which broadcast messages from other processors through this processor must be sent.

Messages are routed as soon as they arrive in a processor based on a 3-word *message header* which Tiny keeps in each buffer page. The three entries in this header are the *message source* and *message destination* CIDs, and the *message length* (in bytes).

Tiny's first action when routing is to check which routing strategy is being used by testing the two least significant bits in the destination field of the header. If the strategy is sequential, Tiny adds the CID of the message's destination to the base address of the agent's routing segment to obtain the address at which the pointer to the routing table entry is stored. The entry found at the routing table segment corresponding to the pointer is the address of the workspace of the agent responsible for the link through which the message is to be routed. The current agent puts the buffer in that agent's queue.

If the adaptive strategy is to be used, Tiny accesses the routing table segment as above, but then examines the lengths of the queues of the agents responsible for the links down which it could send the message. The message is enqueued for an agent with a shortest queue length.

To analyze the performance of inter-node communication in Transputer networks, two strategies of point-to-point communication are tested [12].

- messages which read and sent messages on the link directly
- messages sent from user processes using Tiny

If the $T(l,b)$ is the time taken for a message of length b to pass over l links, then

$$T(l,b) = \alpha + \beta l + \gamma b + \delta b l$$

where α is a constant set-up time, β is the overhead per link, γ is the time taken to move message data from the user to the router and back, and δ is the time to transfer a single byte through a link;

Tests gave

Method	α	β	γ	δ
Raw	1.0	16.7	0.0	1.3
Tiny	62.4	41.0	0.1	1.3

To summarize the results, the significant performance differences between Tiny and raw hardware are the time taken to communicate header information and the once-per-message cost of protocol generation, which affect only the source and destination processors. This is shown by parameter α and is independent of message length. The differences in β values between Tiny and raw communications shows the time taken to enqueue and dequeue buffers, and to make routing decisions. The additional CPU impact of Tiny on intermediate nodes, which is the difference between β_{raw} and β_{Tiny}, is 24.4 μs/node. γ, the multiplier on message size, is very small.

The time is in microseconds and the tests made were for medium-sized messages (256 bytes).

Communication in iPSC/2 and iPSC/860

iPSC/860, also known as the Intel Touchstone Gamma Prototype is a MIMD supercomputer in which numerous computational nodes are connected in a hypercube network. Each computational node in the iPSC/860 consists of an Intel i860 processor plus memory and communication components. The maximum configuration available is a dimension 7 hypercube. Each computational node has 8 Mbytes of memory and each i860 processor features an integer core unit, pipelined floating point units for addition and multiplication, a graphics unit, memory-management support, a large register set, separate instruction and data caches, and 64-bit data paths. With a clock rate of 40 Mhz, each i860 processor has a peak execution rate of 80 Mflops (32-bit) and 60 Mflops (64-bit).

The processors in the iPSC/860 communicate to each other via "wormhole" routing. Hardware is employed to provide efficient message routing between non-adjacent processors. The network essentially provides circuit-switching (as opposed to packet-switched, store-and-forward message routing), thus resulting in very little penalty for nonlocal communication. The peak data transfer rate across hypercube interconnection network between any two nodes is 2.8 Mbytes per second.

In addition to the 128 computational nodes, the maximum configuration iPSC/860 has four I/O nodes each of which has an Intel 80386 processor to access the disks across the interconnection network. The iPSC/860 requires a host machine to serve as its interface to the outside world for program development, resource management, and external network access. The host machine or *System Resource Manager* (SRM) is an Intel 301 microcomputer which has an Intel 80386/387 pair running at 16 Mhz. The link between the SRM and the hypercube network provides a peak transfer rate of over 1 Mbyte per second.

The computational nodes in the iPSC/860 system run a simple operating system kernel called NX that supervises process execution and provides buffered, queued message passing. Like other MIMD hyercubes, the programming model for the iPSC/860 is based on adding explicit communication calls (send/recv) to serial code written in conventional programming language (C or Fortran).

Intel Paragon/Touchstone Delta

In conjunction with a consortium of institutions led by Caltech, Intel has developed a massively parallel, distributed memory parallel processor called the Touchstone DELTA system. The Intel Touchstone DELTA system is a mesh-connected parallel processor, consisting of 528 i860 compute nodes, 32 80386 I/O nodes, two 80386 network interface nodes, six service nodes, and two tape nodes. The Delta Mesh is shown in Figure 2. Each compute node has 16 Mbytes of memory and is connected to a Mesh Routing chip (MRC) through a Mesh Interface Module (MIM). Each MRC channel is 8-bits wide and the peak data rate of the communication system is 22 Mbytes per second. The largest mesh available to an application is 16×32 [16].

Communication in Intel Touchstone DELTA

Information moves over the interconnection network in the form of messages contained in fixed length packets. Each packet contains routing information, data, and an end-of-packet flag. If a message is too long for one packet, it is divided up among a number of packets and is transmitted over the network interleaved with messages from other nodes.

Mesh Interface Module

The MIM connects the node processor to the associated MRC. It includes 2048 byte transmit and receive FIFOs and associated logic that provide an 8-bit interface to the MRC and a 32-bit interface to the i860 compute node.

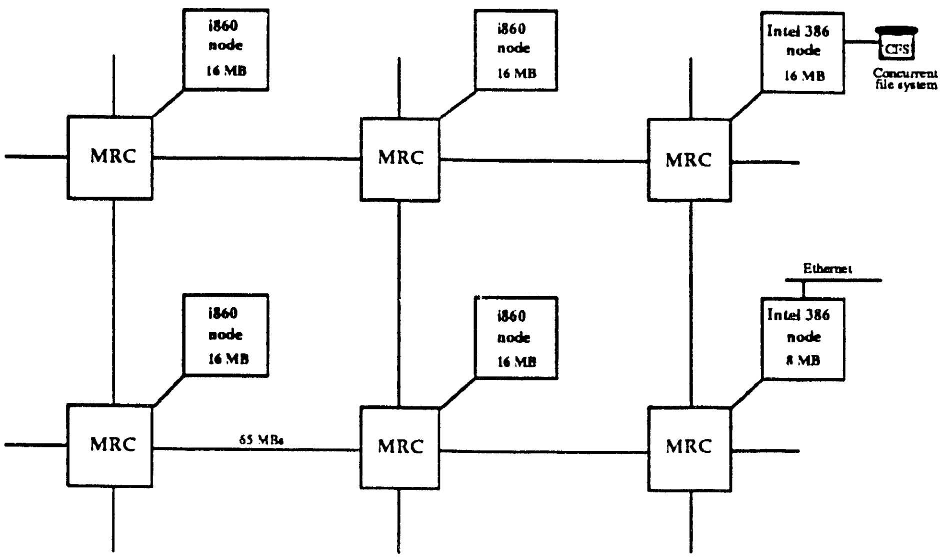


Figure 2: Intel DELTA mesh and nodes

The MIM can transfer data to the compute node i860 at a rate of 26.7 Mbytes per second. The MIM also provides 32-bit CRC generation and error detection.

Mesh Routing Chip

The Mesh Routing Chip routes messages moving through the mesh at high speed. There are five ports on each MRC (see Figure 3). Four of them route messages between routers in the X and Y directions of the mesh. The fifth port attaches to the node. Each MRC port consists of two byte-wide communication channels. One channel is for communication into the MRC, and one is for communication out. Logically, the MRC is self-timed, to avoid distributing a high-speed synchronous clock over a physically large area.

A sending node transmits one or several packets that contain a message by

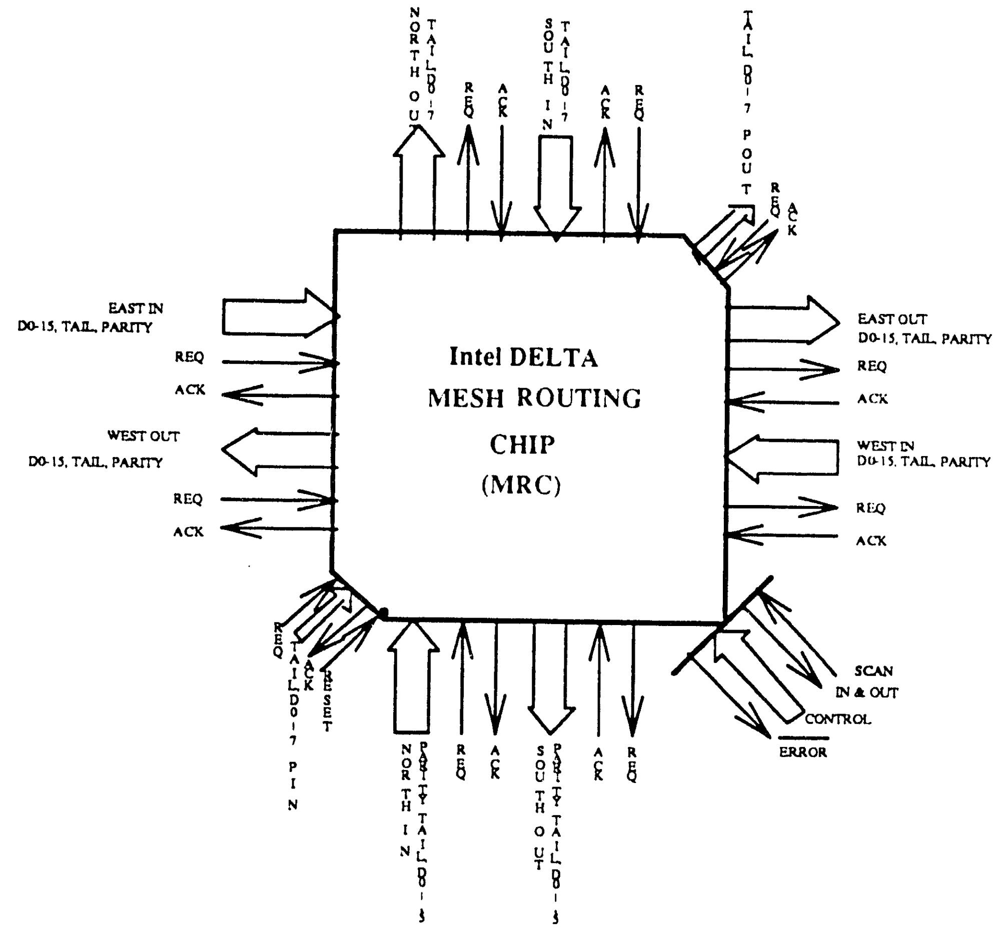


Figure 3: MRC block diagram

moving the data for each packet from the node's memory to the MRC. Address information at the head of each packet is examined to determine how to route the packet through the network depending on the direction bit in the routing information. The MRC tries to route the packet in the X-direction (east-west). If the X displacement specified in the first byte of routing information is zero, the MRC strips off that byte and examines the second byte for Y displacement. If the Y displacement is also zero, the MRC strips off the second byte of routing information and transfers the packet to the node connected to the MRC.

If the MRC finds the X displacement specified in a packet to be nonzero, it routes the packet to the next MRC in the X direction. Routing in the X direction continues until the X displacement goes to zero, then Y routing begins. Routing in the Y direction continues until the Y displacement goes to zero, then the MRC routes the packet to the node.

If for any reason the destination node or any of the intermediate MRCs cannot take a packet, its progress stops. The packet remains queued within the mesh until the blockage or the condition at the destination clears and the packet can continue.

By the time the packet reaches its destination, the MRCs have stripped off its first two bytes. As the end of packet indicator passes through an MRC, it releases the channel that was reserved for the packet.

Each MRC channel has a bandwidth of 65 Mbytes per second. The latency of the MRC is 75 nanoseconds for messages moving in the same direction and 150 nanoseconds for messages that change direction.

CHAPTER 3. TAXONOMY OF COMMUNICATION METHODS

Every message-passing multicomputer supports a set of inter-node communication methods. It is basically the set of supported inter-processor communication methods that make one multicomputer differ from another. A classification of inter-node communication is based on five dimensions [11]. The major alternatives with respect to each dimension are defined below and shown in Figure 4.

1. *Communicating nodes.* In an application program, a given node may communicate directly with a number of other nodes. The node may communicate with either a *restricted* set of nodes (such as its neighboring nodes) or any *arbitrary* node in the system.

2. *Connection setup.* A connection is a physical routing path on the network allocated to carry out one or more communications required by the application during program execution. A connection may be set up either *statically* before program execution or *dynamically* during program execution.

3. *Routing path selection.* To build a connection for a given communication between two nodes, there could be multiple choices for the physical routing path. The routing path selection can be either *deterministic* or *adaptive.* In the deterministic case, the route is totally determined by the source and destination

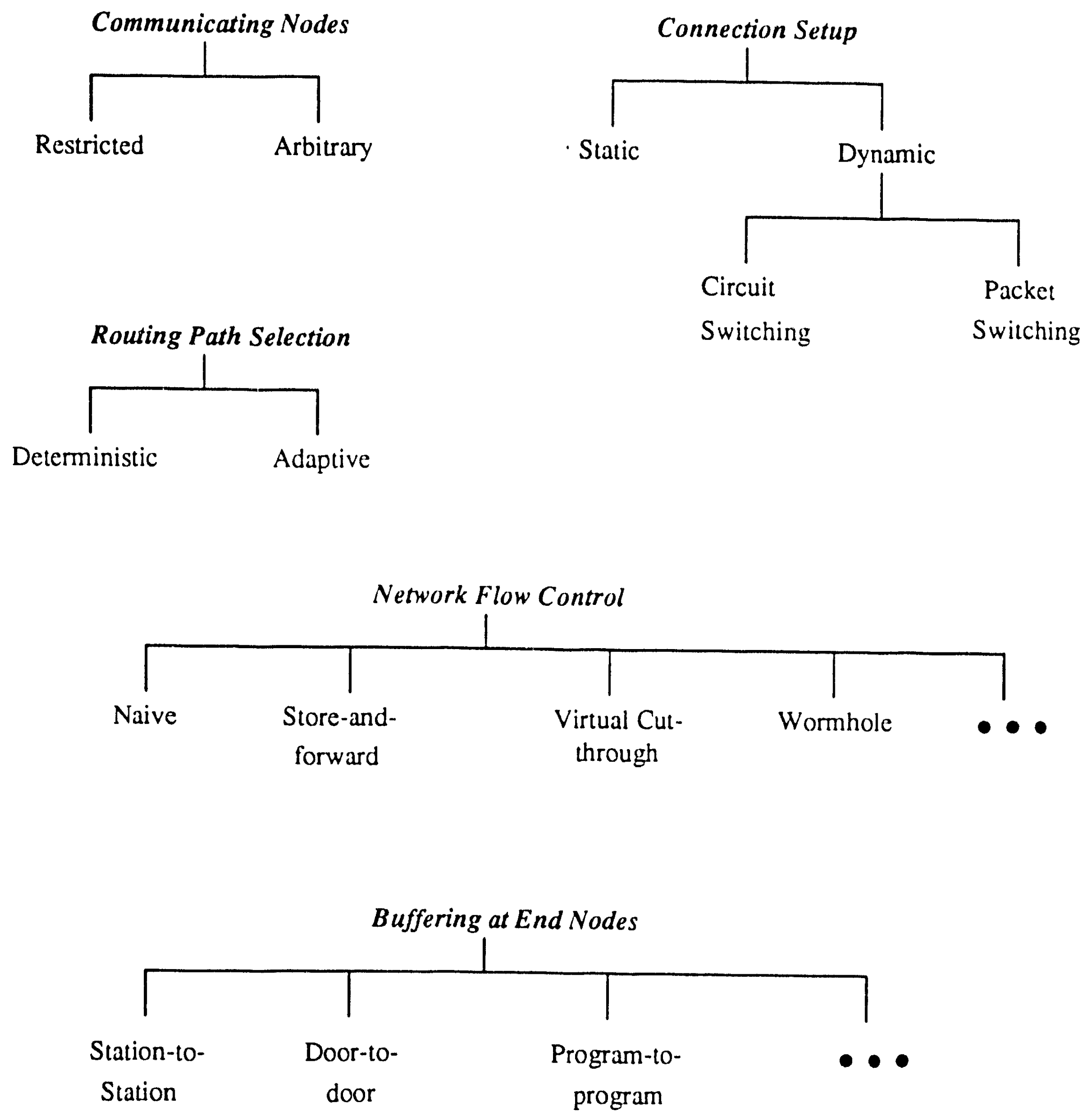


Figure 4: Taxonomy of Inter-processor Communication Strategies

address of the connection. In the adaptive case, the route can be selected by the *source node*, which originates the communication, or by the *network* to avoid congested nodes on-the-fly.

4. *Network flow control.* During transmission a message packet may not be able to proceed to the next node while the next node is busy. Network flow control is concerned with methods to avoid network queue overflows and underflows in the presence of the possibility that messages may be blocked inside the network.

   There are at present four known methods to avoid messages being blocked

   - **Naive** Message blocking can be totally avoided by starting transmission only after the entire connection has been set up and the destination node has acknowledged its readiness to accept a packet. This method has a relatively long communication latency (to set up the connection and to wait for acknowledgment from destination).

   - **Store-and-forward** When arriving at each intermediate node, the complete packet is first stored in the system memory of the node, and then sent forward to a selected neighboring node when the neighbor is not busy. Implementation of this method is relatively easy since at any time, at most two nodes are involved in transmitting a packet. However, the buffering of the packet consumes memory space and bandwidth of the intermediate nodes and introduces communication delays.

   - **Virtual cut-through** When the header of a message arrives at an intermediate node whose selected output channel is free, the packet will be directed to the next node immediately. Therefore a message packet is buffered at an intermediate node only when its selected output channel is busy.

- **Wormhole** If the blocked messages remain in the network while waiting for the selected output channel to become available, then the method is *Wormhole* method. The channels along the route of the blocked message do not become free for transmission of other messages, but in Virtual cut-through, they do. Thus wormhole method totally avoids the overhead of buffering packets in system memories of intermediate nodes. In Wormhole routing, a message is serialized into a sequence of parallel data units that we refer to as flow control digits or (flits). A *flit* is the smallest unit of information that a queue can accept or refuse [2]. As soon as a node examines the header flit(s) of a message, it selects the next channel on the route and begins forwarding flits down that channel. As the flits are forwarded, the flits of one message may be spread out among several nodes as the message moves. The communications in a node consists of a set of queues-one for each channel. The queue of an input channel is connected to an output channel, and the message flits flow through the connection, in a pipeline fashion until the entire message has passed. Then the connection can be broken and other connections made. Since most flits have no routing information, the flits in a message must remain in contiguous channels of the network and cannot be interleaved with the flits of other messages.

Network Latency The network latency for both virtual cut-through and wormhole routing [4] is

$$T_f D + L/B = (L_f/B)D + L/B,$$

where T_f is the delay of the individual routing nodes on the path, D is the number of nodes on the path (distance), and L/B is the time required for the message of length L to pass through the channels of bandwidth B, and L_f is the length of each flit. If $L_f << L$, th distance D has a negligible effect on the network latency. Thus cut-through and wormhole routing have network latencies which result in a *sum* of two terms, one of which depends on the message length L, and the other of which depends on the number of communication channels traversed, D. Store-and-forward routing, on the other hand, gives a latency that depends on the *product* of L and D.

Wormhole routing has low network latency and requires small amount of dedicated buffers at each node, and therefore is adopted in Symult 2010, nCUBE-2, iWARP, and Intel's Touchstone project. It is also being used in some fine-grained multicomputers, such as MIT's J-machine and Caltech's MOSAIC [4].

5. **Buffering at end nodes.** For some communication methods, message packets need to be buffered in the system memory at the sending or receiving node, or at both ends. Some buffering schemes in use are given as follows:

   - When packets can be sent and received only from system buffers, messages have to be buffered at both ends. At the sending end, packets built in the user space have to be copied to the system buffer before they can be sent. Correspondingly, at the receiving end, packets received into the system buffer have to be copied to the user space before they can be read. This type of communication is called *station-to-station* communication.

- When packets are sent and received from user spaces directly, the communication is called *door-to-door* communication. This has smaller communication latency than station-to-station communication.

- When the application program is allowed to send or consume individual data items as they are computed or received respectively, the communication is called *program-to-program* or *systolic* communication. Thus the sending program does not have to build a complete packet before sending out a single data item, and the receiving program does not have to receive the complete packet before reading a single item in the packet. This method has the minimum possible latency since individual data items can be sent out as soon as they become available at the sending end, and can be used as soon as they are received at the receiving end. The drawback of this method is that it requires careful synchronization between the sending and receiving programs.

Classification of the iWARP, transputer networks, iPSC/860, Intel Touchstone DELTA and the nCUBE 2 on the basis of the above taxonomy is given in Table 3.1. Since iWARP uses two different types of inter-node communication modes, we have classified iWARP in two different architectures for the purpose of comparing the communication methods used. It is interesting to note that every architecture other than transputer networks uses wormhole routing as a strategy for network flow control.

Table 3.1: Communication-based classification of some architectures

Architecture	*Commu–nicating nodes*	*Connection setup*	*Routing path Selection*	*Network flow control*	*Buffering atend nodes*
iWARP, Systolic	Restricted	Static	Deterministic	Wormhole	Program-to -Program
iWARP, Message-passing	Arbitrary	Dynamic	Deterministic or Adaptive	Wormhole	Station-to -Station
Transputer N/ws (with Tiny)	Arbitrary	Static	Deterministic	Store-and -forward	Door-to -Door
iPSC/2, iPSC/860	Arbitrary	Dynamic (Circuit-SW)	Deterministic	Wormhole	Station-to -Station
Intel DELTA	Arbitrary	Dynamic (Circuit-SW)	Deterministic	Wormhole	Station-to -Station
nCUBE 2	Arbitrary	Dynamic (Circuit-SW)	Deterministic	Wormhole	Station-to Station

The transputer networks use conventional store-and-forward routing.

CHAPTER 4. IMPLEMENTATION OF COMMUNICATION IN nCUBE VERTEX O.S

System Architecture

The topology of the nCUBE 2 supercomputer is an n-dimensional cube in which the vertices are node processors and the neighboring nodes are linked along the edges by message-passing communication channels. The nCUBE 2 supercomputer consists of the following basic parts:

- **Nodes** (Processing Elements)- The nCUBE 2 supercomputer is a network of independent CPUs, each with its integrated floating-point unit, local memory, and communication hardware [10]. Each node runs a complete copy of the nCUBE 2 Vertex operating system. When executing an application, each node runs a program element, or a local program; the parallel program comprises all the program elements running on all the nodes allocated for the program. The nCUBE 2 computer can run several parallel programs simultaneously, with each program allocated a different collection of nodes. Each node is a sequential computer operating at 2.5 MFLOPS. The number of nodes is scalable.
- **Communication Channels-** Each node is linked to the other nodes by hardware-routed, DMA communication channels. Direct hardware exists only between

nearest neighbors in the network, but the communication between distant nodes is done by wormhole routing technique which makes it possible for the routing to continue without affecting the performance of the intermediate CPUs.

- **I/O Processors-** Separate DMA I/O Channels are provided to each node for transfers to and from I/O processors.

Communication Hardware of nCUBE Processor

The Network Communication Unit (NCU) built into the nCUBE processor's hardware implements automatic cut-through message routing. Messages from one processor node to another non-neighboring processor node pass through the intermediate nodes without interrupting the intermediate node CPUs.

The Network Communication Unit or NCU consists of 28 unidirectional, independent, direct memory access (DMA) I/O channels. Each I/O channel can be directly connected to a corresponding I/O channel in another processor.

The maximum number of processors in an nCUBE 2 hypercube is 8192, which is a dimension 13 hypercube. This implies that a processor in the highest dimension hypercube is directly connected to I/O channels (ports) of 13 other processors.

The 28 DMA channels are arranged in pairs. Fourteen channels handle input and fourteen channels handle output, thus providing fourteen full-duplex I/O ports. Thirteen I/O ports handle communication with other processors in a hypercube of maximum dimension 13, and one I/O port (called SI port) handles communication with other hypercube spaces or foreign (non-hypercube) systems.

Each DMA I/O channel has two registers. The **DMA address register** and the **DMA Count register**. The DMA Address and Count Registers are loaded by specific instructions in routines in the Vertex O.S. servicing the DMA channel.

Interface between the User and Vertex O.S.

Communication in the nCUBE Architecture is based on message passing. This is accomplished in a User program by making calls to the nCUBE communication library.

The nwrite() call is used to send a message, and the nread() to receive a message [9]. The parameters with which nwrite() and nread() are called are processed by the library call for error and validity. The library call then executes a trap instruction which directs the program to execute the corresponding trap handling routine within Vertex O.S. The trap routine psend() buffers the message, creates a header with the available information from user and the system, and determines the channel through which the data has to be sent. It then places the message into the queue of messages for the corresponding channel. The trap routine for the receive checks for a valid message, if any, in the "received messages" queue for that process, and if the message has arrived, copies the message into the buffer specified in the parameter of the call. The actual sending and receiving of the data occur asynchronously by the input and output message ready interrupts for that channel when generated. These channel interrupts are generated asynchronously and independent to the CPU.

As soon as the DMA Output Channel has nothing more to send, at the end of a transmission, the Output Message Ready Interrupt for that channel is generated, indicating to the CPU that the DMA channel is ready to transmit more messages. The Interrupt service routine handling the Output message ready interrupt corresponding to that channel then checks if there are any more queued messages to be sent along that channel, and if there are any, sends them along that channel.

As soon as the DMA Input Channel has nothing more to receive, when it receives an EOT packet, the Input Message Ready Interrupt for that channel is generated, indicating to the CPU that the DMA channel is ready to receive further messages. The Interrupt handler then initializes the DMA Input channel to receive the header of the next transmission along that channel.

A message at the user level can be divided into more than one message packet in a transmission. The Vertex O.S. handles each user message as two message packets in one transmission: a header message packet and a data message packet.

The first 32 bits of the header contain the destination address and destination mask which are encapsulated by hardware into an address packet. The rest of the header message packet and the data message packet is encapsulated by hardware as data packets containing 32 bits of data information in one packet. A communication path is established by sending an address packet over a channel. The address packet is then routed from node to node along the channels determined by the following process. Each processor compares the masked node address with its own processor ID and sends the address packet out the port number corresponding to the bit position of the first difference, starting at bit $n + 1$, where n is the number of the port on which the message was received. Thus messages are always sent out on a port which has a number higher than the port on which the message was received. An example of routing a message through the hypercube is shown in Figure 5.

Once a communication path (consisting of a sequence of channels along a path) is established, it is reserved for further data or command packets until an EOT packet clears the path for other messages. The Vertex O.S. takes care of sending

For a message with :
Destination = 101
Source = 010,
the Route is shown in bold arrows

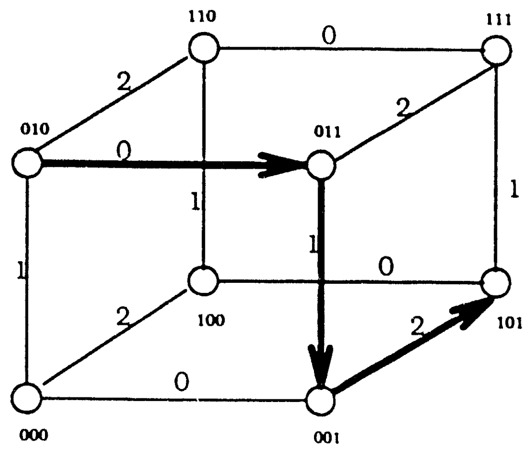


Figure 5: Routing in a 3-Dimensional Hypercube

EOM packets at the end of each message packet, and sends an EOT packet once a transmission is over, by using different instructions to initialize the DMA registers in each case.

The sending and receiving of data by DMA channels is done independent of the CPU. The DMA channel starts functioning independently of the CPU execution as soon as its Count Register is loaded with a non-zero value. The Address register of the sending DMA channel is loaded with the pointer to the least significant byte of the next word to be transferred. In the case of a receiving DMA channel, the address register contains the pointer to the location at which the next word received on the channel will be stored. The DMA Count register in the output channel is loaded with the number of bytes to be sent. The instruction used for loading the DMA Count register determines whether an EOM packet (LPCNT), or an EOT packet

(LCNT), will be sent after the message packet is sent. As the data is sent, the output channel DMA Address Register is incremented by the number of bytes transferred, and the corresponding DMA Count register is decremented by the number of bytes transferred. When the count reaches zero, an End of Message (EOM) packet or an End of Transmission (EOT) packet is sent to the receiver, and an Output message ready interrupt is generated, if enabled. As data is received, the input channel DMA Address register is incremented by the number of bytes transferred. When an EOM packet or EOT packet is received, the Input Message Ready interrupt is generated, if enabled.

CHAPTER 5. COMMUNICATION ROUTINES IN VERTEX O.S.

The fundamental message-passing system provided with nCUBE hypercubes. Vertex, handles process loading, execution, file system access, debugging, and statistics on the nCUBE 2 processor. It also handles all interrupts, system calls, and interprocessor and I/O communication.

The basic message-passing implementation in the nCUBE 2 system consists of two trap routines, one for send and the other for receive. The actual asynchronous sending and receiving of the data is implemented in two interrupt service routines handling the Output Message Ready and Input Message Ready Interrupts.

Trap Routines for Send and Receive

The nCUBE 2 Processor uses automatic cut-through message routing to achieve communication across hypercube without interrupting intermediate nodes CPUs. This usage of vectored Interrupts for communication reduce the load on the CPU.

The User program calls a communication library routine which executes a trap instruction *psend* primarily to enqueue the message into the "send queue" of the channel along which it has to be transmitted. The Output Message Ready Interrupt corresponding to that channel, when generated, sends the next message on the queue.

Similarly, the trap instruction for the receive *precv* is executed primarily to read the message, if it has arrived, into the user-specified buffer. The Input Message Ready Interrupt for a channel when generated, enqueues the incoming message into the process object communication buffer, which the trap routine scans for the message specified by the user.

When the above trap instructions are executed by the user program, the current Program Status Word (PSW) and the Program Counter (PC) are pushed on the stack (in the given order). The physical vector address (VPC) of the trap handling routine occupies the first four bytes at the physical address equal to eight times the argument of the trap instruction, and the next four bytes are occupied by the new PSW (VPSW) for the trap handling routine. The PSW register is loaded with the PSW and the VPC is written into the PC. The new PSW has all interrupts enabled and the mode bit set to Supervisor mode. Thus the instructions in the trap handler are executed in the supervisor mode.

psend()

When the sending trap instruction is executed, the PC jumps to the corresponding VPC which is the address of *psend* routine. The input parameters for *psend* routine at entry are user-specified and are present in the registers R0 through R4. They are in the following order:

- R0: length of message in bytes
- R1: pointer to message data
- R2: pointer to message status byte
- R3: message type

- R4: destination

The trap handler does the following for a single node-to-node message:

- The user-specified pointer of the message data which is really an offset into the data space of the user process is converted to absolute physical address.
- A user buffer is allocated for the message and a header is constructed (in Vertex Header Format). The header and the message data are copied into the allocated buffer.
- The correct channel over which the message has to be sent is calculated, and all interrupts are disabled.
- The pointer to the allocated buffer is queued at address corresponding to *sendq(channel)*, where *channel* corresponds to the channel number along which the message packet is to be physically sent.
- If the channel is idle, then the trap sends the header and sets the state of the Interrupt handler routine to where it should be after it has sent the header of a message and has to send the data.
- Interrupts are enabled and the process returns to the user program.

Vertex Queue for Send

The Vertex Communication Area includes a linked list for send messages corresponding to each of the 14 channels. Each linked list entry consists of two words. The first word stores the pointer to the most recent message put on the queue. The second word contains the pointer to the next message on the list. Both are initialized during Vertex initialization to the address of the first word in the linked list. The Vertex communication area also provides one word per input DMA channel for storage of the pointer to the buffered received message.

precv()

The messages arriving over any channel are queued by the Input Message Ready Interrupt service routine into the process object communication buffer. The process object communication buffer is a linked list of pointers of the "received but not yet read" messages for that process. The trap routine for the receive basically does the following for a single node-to-node message.

- First the entire linked list of pointers is checked for a valid message (by a call to *plook*()),
- If the call to the routine *plook*() does not find a message of valid type and source id, then process state is set to "msgwait" that is, the process *blocks* till a valid message is found.
- When the valid message is found, the data is copied into the user-specified buffer, and the source id of the sending node and the type (of message) are returned in registers.
- The message is then removed from the "receive queue" in the process communication buffer and is deallocated from memory and becomes free for other incoming messages.

Input and Output Message Ready Interrupt handlers

Output Message Ready Interrupt (Outrdy)

The Output Message Ready Interrupt corresponding to a port is generated only under the following conditions:

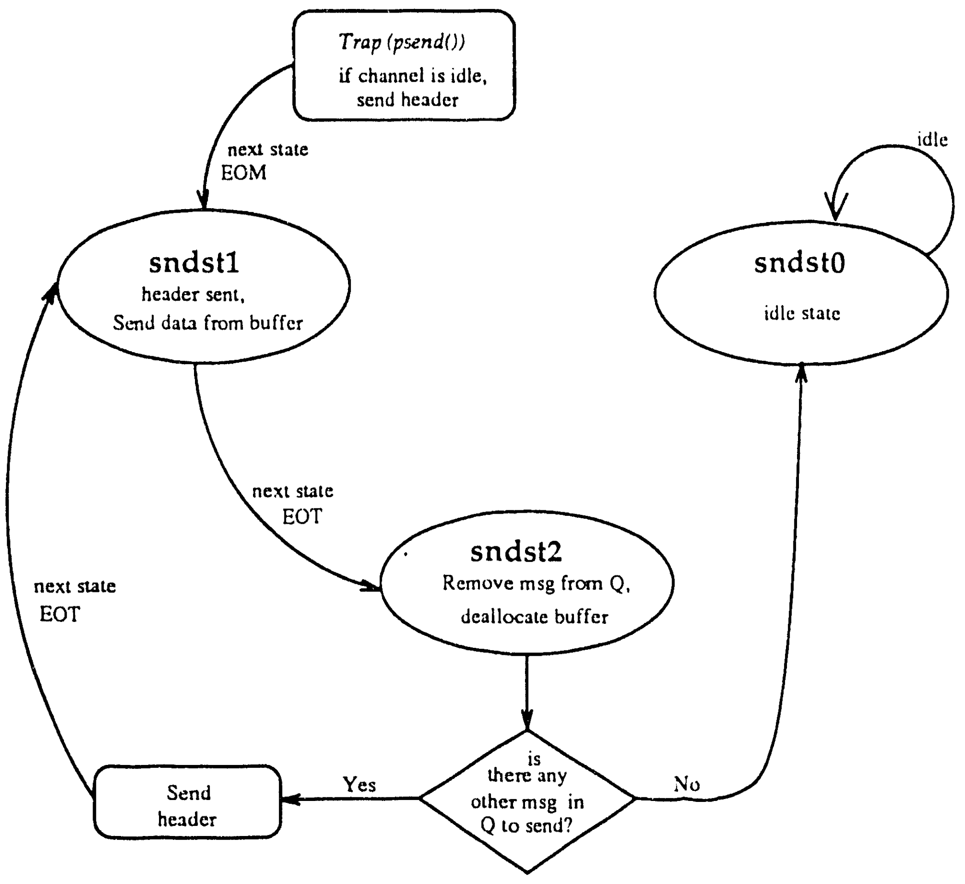
- The Global Interrupt Enable bit (IE) in the PSW must be 1. (This bit is always true except in the case when the process executes the Interrupt handlers for Inprdy, Outrdy and Badecc interrupts)

- The Enable Output Interrupt bit (IO) in the PSW must be 1. (This bit is always true except when the process executes the Inprdy, Outrdy and Badecc interrupt handlers)

- Output Interrupt Enable Register bit corresponding to the port in use must be 1. (The trap routine *psend*() or the interrupt handler for Outrdy interrupt for that port executes an LCNT or an LPCNT instruction and then sets the corresponding bit to 1, if it isn't already set)

- The Output Message Ready flag corresponding to the port in use must be 1. The bit in the Output Message Ready Register corresponding to this port is set to 1 on processor reset, at the end of a message (EOM), or at the end of a transmission (EOT). It is cleared on an LCNT or LPCNT instruction corresponding to the port number.

The handler for the Output Message Ready Interrupt for a particular Output port can be in one of the three following states as shown in Figure 6.



Trap (psend())
if channel is idle, send header
next state EOM
sndst1
header sent, Send data from buffer
idle
sndst0
idle state
next state EOT
sndst2
Remove msg from Q, deallocate buffer
next state EOT
is there any other msg in Q to send?
Yes
No
Send header

Figure 6: Outrdy Interrupt States

- **sndst0** This state is an idle state, which is the state of the interrupt handler when it has no more messages to send. The interrupt handler thus resets interrupts, and sets new state to *sndst*0. When the state of the interrupt handler is *sndst*0, the only way the state can be changed to any other state is when the user tries to send another message which has to be sent over this channel. In such a case, the trap routine *psend*() sends the header of the message over the channel (because the channel was idle anyway), and sets the new state to *sndst*1, so that at the end of transmission of the header, when an Outrdy interrupt is generated, then the interrupt executes in state *sndst*1.

- **sndst1** If the header of a message is already sent, (either by the trap *psend*() or by the interrupt handler in state *sndst*2), then the generation of the next Outrdy interrupt on that channel will direct the Interrupt handler to this state. In this state, the data of a message is sent from the buffer which is at the top of the queue of the messages to be sent, pointed by sendq(channel #). The new state of the interrupt handler is set to *sndst*2.

- **sndst2** The Service routine for Outrdy Interrupt on a particular channel is entered in state *sndst*2 when a complete message is sent over the channel. Therefore, the interrupt handler in this state dequeues the message from the sendq(channel #). The buffer used for queueing the message is also deallocated, so that it becomes free for buffering other communication messages. The interrupt handler then checks for any more messages to be sent from the sendq(channel #). If there is a message to be sent, then the header of that message is sent along that channel, and the new state is set to *sndst*1. Otherwise, if there are no more messages to be sent, then the new state is set to *sndst*0, which is an idle state.

Input Message Ready Interrupt (Inprdy)

The Input Message Ready Interrupt corresponding to a port is generated only under the following conditions:

- The Global Interrupt Enable bit (IE) in the PSW must be 1. (This bit is always true except in the case when the process executes the Interrupt handlers for Inprdy, Outrdy and Badecc interrupts)
- The Enable Input Interrupt bit (II) in the PSW must be 1. (This bit is always true except when the process executes the Inprdy, Outrdy and Badecc interrupt handlers)
- Input Interrupt Enable Register bit corresponding to the port in question must be 1. (The trap routine *precv*() or the interrupt handler for Inprdy interrupt for that port executes an LCNT or an LPCNT instruction and then sets the corresponding bit to 1, if it isn't already set)
- The Input Message Ready flag corresponding to the port in question must be 1. The bit in the Input Message Ready Register corresponding to this port is set to 1 on processor reset, at the end of a message (EOM), or at the end of a transmission (EOT). It is cleared on an LCNT or LPCNT instruction corresponding to the port number.

The message header when received over a particular channel is automatically transferred to a Vertex communication space called "recvbuf" which provides a buffer space of 16 bytes of header per input channel. The interrupt handler for the Input Message Ready Interrupt then buffers this header in a user buffer and all data packets arriving over the channel henceforth until the arrival of the EOT packet are directly transferred to this user buffer. The pointer to this user buffer then is queued into the process object communication buffer. The Interrupt handler also prepares the DMA input channel to receive the next header transmitted over the channel, by loading the address of the "recvbuf" corresponding to that channel into the DMA Address register.

The handler for the Input Message Ready Interrupt for a particular input port can be in one of the two following states as shown in Figure 7.

- **rcvst0** When the header of a message is completely received over a channel, ie. when the EOM packet at the end of a header triggers an Input Message Ready Interrupt corresponding to that channel, the interrupt handler is entered in *rcvst*0. Since the header of a message is already received, the interrupt handler first determines whether the message is a Vertex command message, and if it isn't, it gets a buffer from user space. It then buffers the header, and sets the DMA input channel to receive the message data directly into the allocated buffer, and the new state for the interrupt handler is set to *rcvst*1.

- **rcvst1** When the complete message is received over a channel, the EOT packet at the end of the message transmission generates an Inprdy Interrupt for that channel, and the interrupt service routine is entered in the *rcvst*1 state. The handler first initializes the corresponding DMA input channel to receive the next header which arrives over the channel, then sets the next state to *rcvst*0. If the recently received message was a Vertex control message, then appropriate

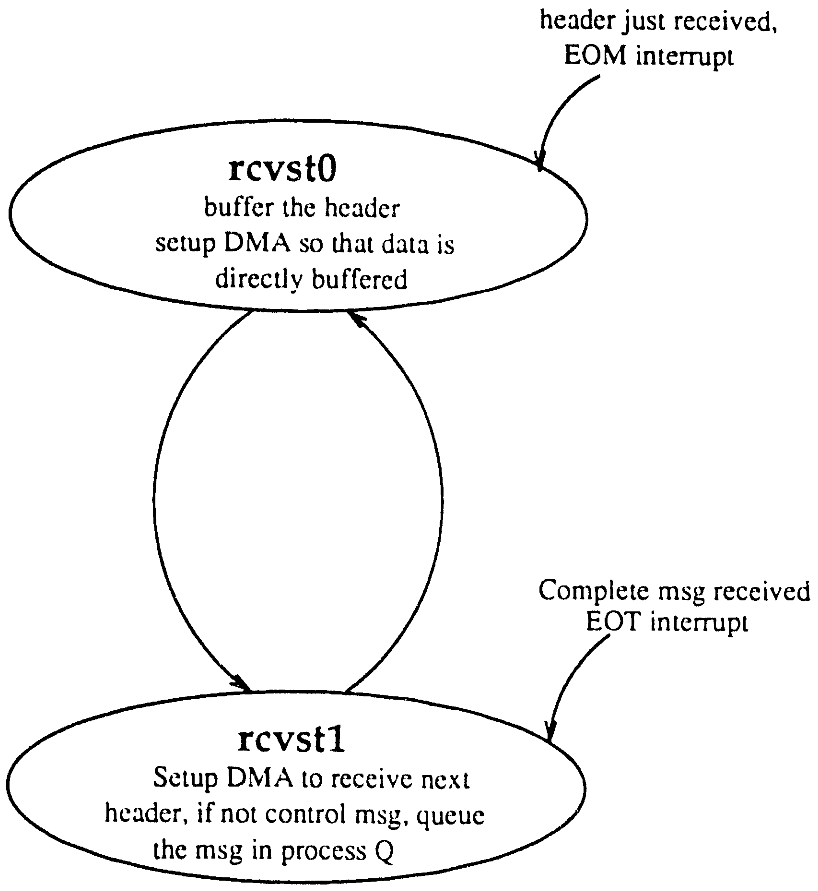


Figure 7: Inprdy Interrupt States

action is taken, otherwise the received message is queued into a "received but not read" queue whose head is at offset "pojcomb" in the process object table of the current process.

CHAPTER 6. THE SCALAR COMMUNICATION CALL

User Interface to the Scalar Call

The User program executes a call to *scsend*() to send a scalar data to a destination node. In a C program, the call would look like

scsend(&*send_struct*);

where

send_struct is a structure of the type

```
struct{
        shortint    sendid;         /* destination nodeid */
        shortint    sendmask;       /* destination mask */
        shortint    my_id;          /* source nodeid */
        shortint    type_send;      /* type of message */
        int         data_scalar;    /* scalar data for send */
        }send_struct;
```

The User program executes a call to *screcv*() to receive a scalar data from a source node. In a C program, the call would look like *screcv(&recv_struct)*;

where

recv_struct is a structure of the type

```
struct{
        shortint    source_id;     /* source node of expected message */
        shortint    type_recv;     /* expected type of the message */
        int         data_scalar;   /* the scalar data for the receive */
        }recv_struct;
```

scsend()

The algorithm equivalent of *scsend()* is given below.

```
scsend()
    convert input parameter pointer into real address;
    adjust destination in subcube to destination in hypercube;
    calculate channel # over which message is to be sent;
    if (channel is idle)
        send complete message structure of 12 bytes;
        set next state for Outrdy interrupt handler of corresponding channel to sndst0;
        enable corresponding interrupt bit in Output Interrupt Enable Register;
        restore registers;
        return from interrupt;
    else
        restore registers;
        return from interrupt;
```

screcv()

The pseudocode for *screcv()* is given below.

```
screcv()
    convert input parameter pointer into real address;
    go to step 5;
    step 4: decrease sc_count; (count of "received but not read" scalar messages)
    if (sc_count == 0)
        indicate message not found;
        return from trap;
    else
        step 5: if (source specified = −1)
                    go to step 2;
                else
                    go to step 6;
        step 2: if (type = −1)
                    read next data in scalar buffer into user-provided buffer;
                else
                    go to step 3;
        step 3: if (type of next message in buffer == specified type)
                    copy next data in scalar buffer into user-provided buffer;
                else
```

```
            go to step 4;


step 6: if (source of next message in buffer == specified source)
            go to step 2;
        else
            go to step 4;
```

The receiving of data incoming over the channel into a buffer reserved for each channel in the Vertex communication area is done by the Inprdy Interrupt Service Routine for that channel. Since

- the actual arrival of the data over a channel is asynchronous with the execution of the user program reading the incoming message.
- the channel number over which a particular message is scheduled to arrive is often not known prior to its arrival.

The actual reception of scalar data over the channel is done by modifying the Inprdy Interrupt Service Routine to take care of the scalar message when it comes over the channel.

Modifications to Inprdy Interrupt Service Routine

The Inprdy Interrupt Handler for an input DMA port in the state *rcvst1* sets up the channel to receive the header of a message directly into a 16-byte per channel buffer reserved in the Vertex communication data space. This header buffer is at an offset *recvbuf*(*channel*# $\times$ 16) from the start of the Vertex communication data area. On the arrival of an EOM packet over the channel after the address and data packets of the header of a message, the Input Message Ready Interrupt (Inprdy) corresponding to the channel if enabled, is generated. On generation of the Interrupt, the process executes the Service handler of the Inprdy interrupt corresponding to that channel in the state *rcvst0*. In this state, the modification to the Inprdy Interrupt Service Routine takes care of the case of the arrival of a scalar message. It accomplishes this by looking for a specific sequence of bits in the halfword at an offset of 4 from the start of any header arriving on the channel.

In the standard communication header format, the halfword at an offset of 4 from the beginning of the header contains the destination process id for the message. Since there is only one process on every nCUBE processor node in the existing communication call on the nCUBE, the halfword at an offset of 4 from the start of the header always has 0. Therefore a bit sequence of 1's in the highest 14 bits of the halfword will not have any effect on the normal handling of the Inprdy interrupt.

The Inprdy Interrupt Service Routine for a channel looks for the specific bit sequence in the halfword at an offset of 4 from the beginning of any header arriving on the channel. If the bit sequence is found, a branch is taken to code at *recv_sc_int* which specifically takes care of reading in the scalar data into a scalar communication buffer area reserved in the Vertex communication data area. This is done so that the channel can be prepared to receive the next header arriving on the channel. The scalar communication buffer area consists of two words per scalar message. A total of 64 scalar messages can be buffered before the user program actually executes a trap to read in the scalar message into the user-specified buffer. The two-word buffer per message consists of a word for the scalar data, a halfword for the type of message, and a halfword for the source nodeid of the message. A halfword in the scalar communication buffer (*sc_count*) keeps count of the "received-but-not-yet-read" scalar messages.

The actions taken by *recv_sc_int* are listed as the following steps.

recv_sc_int:

1. load the pointer to the recently received header into register R0
2. calculate the offset of first empty buffer using *sc_count*
3. copy scalar data into the empty buffer corresponding to count
4. copy the type and source of the scalar message into the buffer
5. increment *sc_count*
6. set up the DMA input channel to receive the next header arriving on the channel
7. set next state of Inprdy Interrupt handler for that channel to *rcvst*0
8. Set the corresponding bit in the Output Interrupt Enable Register
9. restore the registers
10. return from interrupt

CHAPTER 7. RESULTS AND CONCLUSIONS

In the preceding chapters, we have described various communication methods in message-passing architectures and their implementation. The main concern during the development of the low-latency scalar communication call is the time taken for execution by the call. In order to minimize the latency of the scalar communication call, first we tried to identify the components of the call and analyze the time taken in each of the components.

The user code invokes the functions to send and receive a scalar data by calling *sc_send* and *sc_recv.* Both functions take an address parameter which is passed through the stack. The Vertex scalar send trap handler *scsend()* takes the address parameter in register R0. Therefore, it is necessary to execute an instruction to copy the parameter from the stack to the register R0 before executing the appropriate trap instruction.

Some experimental execution statistics showed a call and a return on an nCUBE processor takes 1 to 3 μseconds. A move instruction from stack to register R0 takes 0 to 1 μseconds. An empty trap instruction takes 2 to 3 μseconds. Thus the non-Vertex overhead for the scalar call (both send and receive) is 6 μseconds.

The actual execution of the scalar send and receive from user code takes 27 μseconds and 20 μseconds respectively. Therefore the time taken by Vertex to execute

Table 7.1: Break-up of latency of *scsend()*

Component	*Numberofcycles*	*WorstCaseCycles*
to save registers in stack	36	44
to convert user address to real address	107	122
to find actual destination id in hypercube	40	46
to find the channel number to send on	15	15
to check if channel is active or not	17	17
set up for send on DMA channel	20	26
to restore interrupts	37	50
to restore registers	36	44
TOTAL	308 (15.4 μseconds	364 (18.2 μseconds)

the trap routines *scsend()* and *screcv()* is 21 μseconds and 14 μseconds respectively. The component-wise breakup of the latency for the trap handler *scsend()* is shown in Table 7.1. The component-wise breakup of the latency for the trap handler *screcv()* is shown in Table 7.2.

The discrepancy of the average timings for the *scsend* may be because of the following reason. The DMA channel and CPU can both be masters of the bus connecting to the memory. When the CPU sends some data through the DMA channel, the DMA is in control of the bus. During that time, if the CPU has to do any data transfer, it has to arbitrate for the control of the bus. This arbitration is asyn-

Table 7.2: Break-up of latency of *scsend()*

Component	*Numberofcycles*	*WorstCaseCycles*
to save registers in stack	40	50
to convert user address to real address	107	122
to check type and source and find which buffer data goes	105	125
to restore registers	40	50
TOTAL	292 (14.6 μseconds	347 (17.3 μseconds)

chronous and may take as much as 3 to 5 μseconds. This discrepancy is not seen in the timings for *screcv* because *screcv* does not execute any instructions which give bus control to the DMA.

The temporal overhead involved within Vertex for using *scsend()* and *screcv()* is about 1 μsecond (4 instructions including a branch which is taken) in the Inprdy Interrupt service routine. The Spatial overhead within Vertex is about 514 bytes of Vertex data area for buffering the scalar data when it arrives and before it is read by the user.

Thus,

- by reducing the header size, so that overhead time for processing the extra header information is eliminated
- by eliminating the queuing of buffers in the sending processor
- by avoiding the passing of parameters on the stack, and passing parameters

through registers instead,

- by making the trap routine specifically for point-to-point communication
- by not checking for the validity of the user-specified parameters, thus making the call dangerous

we achieve low latencies in sending and receiving scalar data between any two nodes in a hypercube with a message-passing architecture.

BIBLIOGRAPHY

[1] Kermani, P., Kleinrock, L., Virtual cut-through: a new computer communication switching technique, *Computing Networks* Vol 3, 1979, pp 267-286

[2] Dally, W.J., Seitz, C.L., Deadlock-free message routing in multiprocessor interconnection networks, *IEEE Transactions on Computers* Vol C-36, No.5, May 1987, pp 547-553

[3] Seitz, C.L., The Torus Routing Chip, *Distributed Computing*, Vol 1, No.3, 1986

[4] Lin, X., Ni, L.M., Deadlock-free Multicast Wormhole Routing in Multicomputer Networks, *Proceedings of Supercomputing '91*, pp 116-125

[5] Athas, W.C., Seitz, C.L., Multicomputers: message-passing concurrent computers, *IEEE Computer*, Aug 1988, pp 9-25

[6] Ni, L.M., Communication issues in multicomputers, *Proceedings of the First Workshop on Parallel Processing*, (Taiwan, ROC), Dec 1990, pp 52-64

[7] Linder, D.H., Harden, J.C., An Adaptive and Fault Tolerant Wormhole Routing Strategy for k-ary n-cubes, *IEEE Transactions on Computers*, Vol 40, No.1, Jan 1991

[8] Palmer, J.F., The NCUBE family of high-performance parallel computer systems, *Proceedings of the Third conference on Hypercube Concurrent Computers and Applications*, ACM press, Pasadena, CA, Vol 1 of 2, Jan 1988, pp 847-851

[9] NCUBE Corp, *nCUBE 2 Programmer's Reference Manual*, Foster City, CA, Feb 1992

[10] NCUBE Corp, *nCUBE 2 Processor Manual*, Foster City, CA, Feb 1992

[11] Kung, H.T., Network-Based Multicomputers: Redefining High Performance Computing in the 1990s, *Proceedings of Decennial Caltech Conference on VLSI*, MIT Press, Pasadena, California, March 1989, pp 49-66

[12] Clarke, L., Wilson, G., Tiny: an efficient routing harness for the Inmos transputer, *Concurrency: Practise and Experience*, Vol 3, No.3, June 1991, pp 221-245

[13] Gustafson, J.L., Montry, G.R., Benner, R.E., Development of Parallel Methods for a 1024-Processor Hypercube, *SIAM Journal of Scientific and Statistical Computing Reprint*, Vol 9, No.4, July 1988, pp 609-638

[14] Borkar, S., Cohn, R., Cox, G., et al, iWarp: An Integrated Solution to High-Speed Parallel Computing, *Proceedings of Supercomputing '88*, IEEE Computer Society and ACM SIGARCH, Orlando, Florida, November, 1988, pp 330-339

[15] Gross, T., Communication in iWarp Systems, *Proceedings of Supercomputing '89*, November 1989, pp 436-445

[16] Intel Corporation, *A Touchstone DELTA System Description*, Intel Supercomputer Systems Division, Portland, Oregon, Feb 1991

END

DATE
FILMED
3/30/94