Distributed-Memory MIMD Computing:
An Introduction

E. R. Jessup

September 5, 1995

High Performance Scientific Computing
University of Colorado at Boulder
Copyright ©1995 by the HPSC Group of the University of Colorado

The following are members of
the HPSC Group of the Department of Computer Science
at the University of Colorado at Boulder:

Lloyd D. Fosdick
Elizabeth R. Jessup
Carolyn J. C. Schauble
Gitta O. Domik
# Contents

1 Introduction .............................. 1

2 General architecture ....................... 3
   2.1 Interconnection graphs .................. 3
       2.1.1 Arrays, rings, and tori .......... 3
       2.1.2 Hypercubes ....................... 4
       2.1.3 Trees ................................ 8
       2.1.4 Omega networks .................... 10
   2.2 The evolution of DM-MIMD computers ... 10
   2.3 The communication performance of DM-MIMD computers ... 13
       2.3.1 Message passing between nearest neighbors ........ 14
       2.3.2 Message passing between arbitrary nodes .......... 16
       2.3.3 The passing of large messages ............... 17
       2.3.4 Contention for communication links .......... 20
   2.4 The overall performance of DM-MIMD computers .... 21

3 The hypercube multiprocessor ................. 25
   3.1 The hypercube defined ................ 25
   3.2 Embeddings ........................... 25
   3.3 Interprocessor communication schemes .. 29

4 The mesh multiprocessor ...................... 34
   4.1 The mesh defined ..................... 34
   4.2 Embeddings and interprocessor communication schemes ... 37

5 Some other multiprocessors .................. 41

6 A sample program ............................ 44

7 Acknowledgments ............................ 48

8 Appendix 1: Using the iPSC/2 at UCB ........ 49
   8.1 Writing a node program ................ 50
   8.2 Writing a host program ............... 52
   8.3 Running the host and node programs ... 57
       8.3.1 The makefile ....................... 57
8.3.2 Command line options for controlling the hypercube . . 57
8.3.3 The results ........................................... 59
8.3.4 The hostless node program .......................... 59
8.4 Communication libraries ................................. 61
8.5 Timing a hypercube program ......................... 65
8.6 A happy day on bud .................................... 66
8.7 Debugging tools ....................................... 68

References 69
Trademark Notice

- Ametek S/14, Ametek-2010 are trademarks of Ametek Corporation.

- Cosmic Cube, Mark III Cosmic Cube, are trademarks of the California Institute of Technology.

- Cray-1, Cray X-MP, Cray Y-MP/832, T3D are trademarks of Cray Research, Inc.

- DEC Alpha is a trademark of Digital Equipment Corporation.

- T-series is a trademark of Floating Point Systems.


- IBM RS/6000, IBM SP1, IBM SP2 are trademarks of International Business Machines Corporation.

- MATLAB is a trademark of The MathWorks, Inc.

- Motorola 68020 is a trademark of Motorola Corporation.

- nCUBE, nCUBE/1, nCUBE/2, nCUBE/2S, nCUBE/3, nCUBE/ten are trademarks of Ncub Corporation.

- OSF is a trademark of Open Software Foundation, Inc.

- SPARC is a trademark of Sun Microsystems, Inc.

- Tandem NonStop is a trademark of Tandem Computers.

- Teradata DBC/1012 Data Base Computer is a trademark of Teradata.

- CM-2, CM-5 are trademarks of Thinking Machines Corporation.

- Unix is a trademark of UNIX Systems Laboratories, Inc.

- Weitek is a trademark of Weitek Computers.

- Ethernet is a trademark of Xerox Corporation.
Distributed-Memory MIMD Computing:  
An Introduction*

E. R. Jessup

September 5, 1995

1 Introduction

Computational problems arising in scientific and engineering applications are often very large. Solving them on conventional computers can be a time consuming process demanding extensive storage. In many cases, distributed-memory MIMD (DM-MIMD) multiprocessors can provide both the computational power and the large memory necessary for solving such problems. Recall that MIMD stands for multiple instruction, multiple data. A DM-MIMD multiprocessor is a parallel computer in which each processor has direct access to its own local memory only. The processors are interconnected by communication links, and the processors exchange data by passing messages along those links. Because the multiprocessor is a MIMD machine, completely different programs may run on the individual processors at any one time.

Different models of DM-MIMD multiprocessors are distinguished by such factors as the power of the processors, the size of memory, the speed of

*This work has been supported by the National Science Foundation under an Educational Infrastructure grant, CDA-9017953. It has been produced by the HPSC Group, Department of Computer Science, University of Colorado, Boulder, CO 80309. Please direct comments or queries to Elizabeth Jessup at this address or e-mail jessup@cs.colorado.edu.

Copyright ©1995 by the HPSC Group of the University of Colorado
interprocessor communication, the availability of input/output, and the interconnection pattern of the processors. DM-MIMD multiprocessors have been sold with as few as two and as many as 65,536 processors. The processors have been interconnected as rings, two-dimensional meshes, and tori as well as the more complicated fat-tree, hypercube, and Omega network configurations described in this tutorial.

The very simplest DM-MIMD multiprocessor consists of two interconnected nodes. Each node is a processor with its associated memory. As an example of how we might use this 2-node multiprocessor, suppose that we need to sum a list of twenty numbers and that those numbers are originally split between the two nodes. To begin, both nodes simultaneously sum their ten local numbers. The nodes then send their sums to each other. Each node then adds its received sum to its own local sum to form the sum of all twenty numbers. This parallel implementation requires roughly half the computation time of a sequential implementation of the sum, and its overall speedup is determined by the cost of the communication step relative to the cost of the arithmetic performed.

In general, the design of a DM-MIMD program takes more thought than did this sample program. The purpose of this tutorial is to introduce you to the basics of actual DM-MIMD architectures and to the general techniques needed to write efficient programs for them. In section 2, we first review the graphs defining the interconnection pattern of processors in present-day DM-MIMD multiprocessors. (A graph is defined to be a set of points or nodes interconnected by lines called edges.) We then review the evolution of those machines since their introduction in the 1970’s and provide information on performance of the various architectures. In sections 3 and 4, we focus on the hypercube and mesh-connected multiprocessors, respectively. In both cases, we describe efficient programming techniques based on the interconnection graphs. In section 5, we briefly discuss programming considerations for some other architectures. Finally, in section 6, we work through a sample program for a DM-MIMD multiprocessor with an arbitrary number of processors.

Tutorials on the use of some specific DM-MIMD machines are available by anonymous ftp from cs.colorado.edu in the directory /pub/HPSC.
2 General architecture

2.1 Interconnection graphs

The writing of efficient programs for a DM-MIMD multiprocessor generally requires attention to the interconnection pattern of the nodes in that computer. The nodes of the computer can be thought of as lying on the nodes of a particular graph, and the communication links between processors lie on the edges of that graph. In this section, we review some of the graphs on which commercial DM-MIMD multiprocessors are based and give some examples of architectures based on those graphs. More information is provided on these computers in the subsequent sections of this tutorial.

2.1.1 Arrays, rings, and tori

Figure 1 shows the simplest graphs underlying DM-MIMD multiprocessors. The first diagram is of a linear array of eight processors. In this arrangement, a node has one or two nearest neighbors depending on whether or not it is at the end of the array. Joining the two endpoints converts a linear array into a ring as shown in the second diagram. In this case, all nodes have two neighbors.

Nodes can also be connected to form a $p_1 \times p_2$ two-dimensional array or mesh of processors as shown in third diagram of figure 1. Here, $p_1 = 2$ and $p_2 = 4$, and a node has two or three neighbors depending on its location in the mesh. In a mesh with more rows, a node may have up to four nearest neighbors. These neighbors are identified by their relative positions and are called the north, south, east, and west neighbors.

Connecting corresponding nodes on the left and right sides of the two-dimensional mesh and on the top and bottom of the mesh converts it into a three-dimensional torus as shown in the fourth diagram of the figure. In this case, every node has four nearest neighbors.

Increasing the connectivity of the graph of eight processors reduces the maximum distance between any two processors in the graph. In a multiprocessor, this can translate into a decrease in interprocessor communication time. In a linear array, there are $p - 1$ edges between nodes 0 and $p - 1$, and this is the longest path between any two nodes. This is shown by the arrow in the first diagram of figure 2. Connecting the ends of the array to form a

CU Boulder: HPSC Course Notes
ring cuts this path in half. If communication is permitted in both directions (i.e., the graph is bidirectional), the longest path is between nodes 0 and \( p/2 \), and it follows \( p/2 \) edges as in the second diagram of the figure. In the mesh, the longest path is between nodes at two diagonally opposing corners. It has length \( p_1 + p_2 - 2 \). When \( p = 8 \), the longest path between nodes is four for both the ring and the mesh. The advantage of the mesh is more evident for larger numbers of nodes. For example, the longest path in a \( 16 \times 16 \) mesh traverses thirty edges while the longest path in a ring of \( 16^2 = 256 \) nodes covers 128 edges.

The torus leads to an even better situation. Note that the torus may be viewed as \( p_1 \) horizontal rings of \( p_2 \) nodes each interconnected via \( p_2 \) vertical rings of \( p_1 \) nodes each. Thus, the longest path traveled between node \( A \) and node \( B \) is, for example, the distance from node \( A \)'s vertical ring to node \( B \)'s vertical ring plus the distance around the latter ring to node \( B \). If every ring can be traversed in either direction, the longest path is \( p_1/2 + p_2/2 \). In the \( 2 \times 4 \) torus, the longest path is three. In the \( 16 \times 16 \) torus, it is sixteen.

The Intel Touchstone Delta and Paragon are examples of mesh-connected DM-MIMD multiprocessors. The Japanese PAX-9 uses a torus interconnection of processors. The ring and linear array are not popular multiprocessor architectures on their own, but these graphs are often the basis for efficient programming of mesh and hypercube multiprocessors.

### 2.1.2 Hypercubes

The size of a hypercube is defined by its dimension \( d \). The hypercube graph of dimension \( d \) and the multiprocessor based on it are both called \( d \)-cubes. A cube of dimension \( d = 0 \) consists of one node. A cube of any dimension can be built recursively beginning with 0-cubes. A 1-cube consists of two nodes and is constructed by joining together a pair of 0-cubes. Similarly, a 2-cube (four nodes) is formed by joining corresponding nodes of a pair of 1-cubes. In general, a \( d \)-cube is formed by connecting corresponding nodes in a pair of \((d-1)\)-cubes. Figure 3 shows one way to form a 4-cube from a set of sixteen 0-cubes. Note that a \( d \)-cube always has \( p = 2^d \) nodes.

Every node in a \( d \)-cube is connected to exactly \( d \) others. In particular, if the nodes in a \( d \)-cube are assigned \( d \)-bit binary identifiers (from 0 through \( p - 1 \)), the \( d \) nodes connected to node \( j \) can be assigned identifiers differing from \( j \) in exactly one bit. Figure 4 shows the binary numbering of the nodes.
Figure 1: Linear array, ring, mesh, and torus interconnections of eight processors.
Figure 2: The graphs of figure 1 with a longest path shown. In general, there is more than one path of greatest length in a graph.
Figure 3: The recursive construction of a 4-cube beginning from 0-cubes.
in a 2-cube. A $d$-cube can be constructed by connecting corresponding nodes of two $(d - 1)$-cubes in any of $d$ ways. For example, the familiar 3-cube can be made by linking corresponding processors of the two squares (2-cubes) forming its top and bottom, left and right, or front and back faces. In a $d$-cube, the $d$ neighbors of node $j$ define the $d$ nodes corresponding to node $j$ in the $d$ different $(d - 1)$-cubes.

The Cosmic Cube, the Intel iPSC/2 and iPSC/860, and the nCUBE/1, nCUBE/2, and nCUBE/3 are all examples of hypercube multiprocessors. Section 3 of this tutorial concerns programming techniques for hypercube multiprocessors.

### 2.1.3 Trees

Another type of graph used in DM-MIMD multiprocessors is the tree. Some multiprocessors are based on the binary tree in which each interior node (nodes other than leaves of the tree) has up to two children. Others are based on a $4$-ary or quaternary tree in which each interior node has up to four children. The nearest neighbors of a tree node are its parents and its children.

A *complete* binary tree (i.e., one in which each node has exactly two children) has height $\log_2 p$ when $p$ is a power of two. A complete quaternary tree has height $\log_4 p$ when $p$ is a power of four. Thus, the longest path between any two nodes of a complete $p$-node quaternary tree is shorter than it is in a complete binary tree with the same number of nodes. This maximum path length may also be reduced without changing the height of the tree by
Figure 5: A binary tree and a quaternary tree with more than one parent per child node.

increasing the number of interconnections in a tree so that some nodes have more than two children. In this case, some nodes have more than one parent. Figure 5 shows one way to construct a quaternary tree from a binary tree in this way. (Note that this multi-parent construct is not strictly a tree by most standard definitions even though it is called a tree in the context of machine architectures.)

The Thinking Machines CM-5 is a DM-MIMD multiprocessor based on a quaternary tree in which the nodes have more than one parent. The Teradata DBC/1012 Data Base Computer uses a binary tree interconnection pattern.
2.1.4 Omega networks

An Omega network of dimension $\log K$ has $K(\log K + 1)$ nodes arranged in a $K \times \log K + 1$ two-dimensional array. The nodes, however, are not interconnected as a mesh. Instead, the node at position $(i, j)$ in the array is linked to the node at position $(i, j')$ in the array if and only if the binary representation of $j'$ is formed from a left cyclic shift of the binary representation of $j$ with or without changing the last bit. Thus, the node at position $(3, 2) = (3, (010)_2)$ is linked to the nodes at positions $(3, 4) = (3, (100)_2)$ and $(3, 5) = (3, (101)_2)$, and every node in the Omega network has two nearest neighbors. The Omega network is also sometimes called the butterfly network, the flip network, the baseline, or the reverse baseline network [Leighton 92]. An Omega network is not constructed by using single wire connections but rather by means of $2 \times 2$ switches. For more implementation details, see [Hwang 93]. The IBM SP1 and SP2 use a version of the Omega network.

Although the graphs discussed in this section account for those underlying the majority of existing DM-MIMD multiprocessors, there are other possibilities. One example is the pyramidal structure of the EGPA mentioned in section 2.2. Processors may also be interconnected by means of a common communication channel or data bus. For more information on graphs and their properties in the context of multiprocessor architectures, see [Leighton 92].

2.2 The evolution of DM-MIMD computers

Development of DM-MIMD computers began in the early 1970's. The first example, the Tandem NonStop, was delivered in 1976. In the NonStop, nodes were not connected directly to other nodes. Instead, all nodes were connected to a pair of common data buses. Using the data buses, any one node in the machine could communicate directly with any other node. The NonStop was a computer designed especially for database applications [Almasi & Gottlieb 94].

Other designs followed soon after, beginning with the introduction of the Erlangen General Purpose Architecture (EGPA) in 1977 by W. Haendler, F. Hofman, and H. Schneider at the University of Erlangen in Germany. This machine had nodes in a pyramidal structure in which all nodes at the same level of the pyramid formed a torus [Henning & Volkert 85, Wilson 94].
In 1979, the Processor Array eXperiment (PAX-9) was completed by T. Hoshino at the Institute of Atomic Energy at the University of Kyoto and T. Kawai of Keio University. The PAX-9 was a toroidal array of nine processors. Each node in the array was directly connected to four others. The PAX-9 did not allow the processors to operate fully asynchronously and so permitted a “quasi-MIMD” programming model [Hoshino 86].

The closest ancestor of many of today’s DM-MIMD multiprocessors was the Cosmic Cube hypercube multiprocessor built at the California Institute of Technology. Development of this machine began in 1981 and was carried out by a research team headed by computer scientist Chuck Seitz and physicist Geoffrey Fox. The first prototype model went into operation the following year. The Cosmic Cube had 64 nodes, each directly connected to four others. Each node held Intel 8086/8087 processors and 128 Kbytes of memory. The prototype was released in 1982. The prototype Cosmic Cube was followed by improved versions culminating in the Mark III Cosmic Cube completed in 1987. The Mark III uses Motorola 68020 processors [Almasi & Gottlieb 94, Wilson 94].

The Cosmic Cube inspired the development of at least two successful commercial hypercube multiprocessors. The first of these was the Intel Personal Supercomputer (the iPSC, later called the iPSC/1). The first iPSC/1’s were delivered in 1985 and have up to 128 nodes. The iPSC/1 nodes have Intel 80286/80287 processors and up to 4.5 Mbytes of memory. Models with vector processors were also available. The hypercube is operated via an Intel System 310AP microcomputer connected to the nodes [Intel 86]. This type of controlling computer is termed the host.

In 1985, the first nCUBE hypercube multiprocessor was also released. The nCUBE/1 (also commonly referred to as the nCUBE/ten or nCUBE 3200) had up to 1024 nodes running custom hardware that handled all computation, communication, I/O, and memory management on a single chip. Each node had 64 Kbytes of memory and a 32-bit processor. Its host computer had an Intel 80286 processor and ran a special version of Unix that allowed the nCUBE to emulate a machine with a single distributed file system [Almasi & Gottlieb 94].

The iPSC/1 and the nCUBE/1 have both spawned new generations of hypercube multiprocessors with more powerful processors and more advanced communication. The iPSC/2 hypercube appeared in 1987 using 80386/80387 processors. The iPSC/2 also has separate computation and communication
processors on each node. Thus, a message passing through a node that is not its destination passes through the communication processor without impacting the computation. The iPSC/860, which replaced the iPSC/2 in 1990, employs wormhole routing by which messages can travel from one processor to any other with almost no delay at the intervening processors. The iPSC/860 uses i860 computation processors. The iPSC/860 and later Intel DM-MIMD multiprocessors all retain the separate communication processors [Dunigan 91].

The nCUBE/2 (also called the nCUBE 6400) was released in 1989 and has up to 8192 nodes and up to 64 Mbytes of memory per node. The nCUBE/2 retains the single chip technology that is the hallmark of nCUBE hypercubes but uses a 64-bit processor and wormhole routing. The nCUBE/2S was released in 1991. Patterned after the nCUBE/2, it has substantially improved computational and communication performance and so represents the second generation of nCUBE hypercubes. The third generation hypercube, the nCUBE/3, was released in 1994. It offers even faster communication and computation and can have up to 65,536 nodes and up to 1 Gbyte of memory per node. The second and third generation nCUBE s have been geared toward the database market [Almasi & Gottlieb 94, Wyckoff 94, nCUBE 94].

The mesh multiprocessor has a more scalable design than does the hypercube. In a hypercube multiprocessor with \( p = 2^d \) nodes, a node has \( d \) nearest neighbors and so \( d \) communication wires. In a mesh-connected multiprocessor of the same size, a node has two, three, or four neighbors and wires, depending on its position in the machine. Regardless of the size of the machine, a mesh node never has more than four nearest neighbors or communication links. Thus, a large mesh multiprocessor can be constructed with less communication hardware than can a hypercube multiprocessor with the same number of nodes. While nCUBE has kept the hypercube architecture for its newest machines, Intel has moved to a mesh-connected architecture.

In 1991, Intel delivered the Touchstone Delta. This single machine, located at the California Institute of Technology, has a 16 × 32 array of i860 processors and employs wormhole routing of messages. It served as the prototype for the commercial product, the Intel Paragon. The Paragon also uses a two-dimensional mesh interconnection, but its processor is the faster i860XP. Delivery of the Paragon also began in 1991 [Dunigan 92, Dunigan 94, Wilson 94].

While Intel and nCUBE have been particularly successful suppliers of
DM-MIMD multiprocessors, they are by no means the only ones. In 1986, FPS delivered its T-series hypercube which used a combination of Weitek floating-point chips and Inmos transputers. In 1987, Ametek completed the mesh-connected Ametek-2010, a descendent of its earlier Ametek S/14 hypercube. Commercial DM-MIMD multiprocessors have not been confined to hypercube and mesh architectures. For example, the Thinking Machines CM-5, a MIMD follower of the SIMD CM-2, is based on a fat tree network and uses SPARC processors. The Cray T3D has DEC Alpha chips interconnected as a torus. The IBM SP1 and SP2 use IBM RS/6000 processors interconnected by means of a high-speed Omega switch [Almasi & Gottlieb 94, Fox et al 88, Hwang 93, Wilson 94].

A wholly different type of DM-MIMD multiprocessor is represented by a cluster of workstations. These sets of interconnected workstations are growing in popularity because of their affordability and the substantial computing power of individual workstations. Thanks to such networking software as PVM [Geist et al 95] workstation clusters can be programmed in the same style as a more traditional multiprocessor. Because the nodes do not have to be identical, a workstation cluster is an example of a heterogeneous computing environment. We consider only homogeneous multiprocessors in this tutorial.

2.3 The communication performance of DM-MIMD computers

The evolution of computers has brought with it a great improvement in performance. Like that of any other computer, the performance of a DM-MIMD multiprocessor may be classified according to such measures as its Mflop rating and its LINPACK benchmark (discussed in [Fosdick et al 95]). In addition, the performance of a DM-MIMD computer is determined by the speed of data communication between nodes. In this section, we discuss the communication performance of several machines. In section 2.4, we show how the communication and computation speeds interact to determine the overall performance of a DM-MIMD multiprocessor.
2.3.1 Message passing between nearest neighbors

Messages are passed between two nodes when a command to send a message is issued by the originating node and a command to receive a message is issued by the recipient node. The exact message-passing process that follows these commands depends on the specific computer, but it may involve such preliminary steps as the initialization of message buffers on both nodes and the opening of a communication route between them in addition to the actual transfer of the message along the communication links.

If $\beta$ is the time required to set up the hardware and software for a message transmittal and $\tau$ is the time needed to send one byte of data across the wire linking the two nodes, the time required to send $k$ bytes of data from a node to one of its nearest neighbors is given by

$$T_{\text{comm}} = \beta + k\tau. \quad (1)$$

Typically, the time to send a message is much greater than the time to do a floating-point operation. In particular, $\beta \gg \omega \geq \tau$, where $\omega$ is the time for a flop. Table 1 shows values of $\beta$, $\tau$, $\omega$, and the ratio $\beta/\omega$ for some popular DM-MIMD multiprocessors. All times are given in microseconds. In this table, $\omega$ is defined to be the time for a double precision (8 byte) floating-point multiply. The first column shows the year of release of the listed machine, although some data were gathered on upgraded models. The last column of the table shows the single processor clock rate.

When the message length is less than 100 bytes, a special message passing protocol is used on the iPSC/2 and the iPSC/860. In these cases, the values of $\beta$ are roughly halved. The small message $\beta$ values are listed in parentheses in the table. The communication times for the Paragon are strongly dependent on the operating system, so the operating system name is also listed in parentheses.

This table shows that both computation and nearest neighbor communication performance have increased markedly with the release of each new machine. However, it is still the case that $\beta \gg \tau$. This means that it is better to send many bytes of data in one message than to send the same data in many small messages. Programs for DM-MIMD multiprocessors should be designed with this guideline in mind.

CU Boulder: HPSC Course Notes
<table>
<thead>
<tr>
<th>Year</th>
<th>Computer</th>
<th>$\beta$</th>
<th>$\tau$</th>
<th>$\omega$</th>
<th>$\beta/\omega$</th>
<th>Clock Rate (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1985</td>
<td>iPSC/1</td>
<td>862</td>
<td>1.8</td>
<td>43.0</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>1987</td>
<td>iPSC/2</td>
<td>697</td>
<td>0.4</td>
<td>6.6</td>
<td>106</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>(390)</td>
<td></td>
<td></td>
<td></td>
<td>(59)</td>
<td></td>
</tr>
<tr>
<td>1985</td>
<td>nCUBE/1</td>
<td>384</td>
<td>2.6</td>
<td>13.5</td>
<td>28</td>
<td>20</td>
</tr>
<tr>
<td>1987</td>
<td>nCUBE/2</td>
<td>200</td>
<td>0.6</td>
<td>1.5</td>
<td>133</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>(390)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1988</td>
<td>iPSC/860</td>
<td>136</td>
<td>0.4</td>
<td>0.08</td>
<td>1700</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>(75)</td>
<td></td>
<td></td>
<td></td>
<td>(938)</td>
<td></td>
</tr>
<tr>
<td>1991</td>
<td>Delta</td>
<td>72</td>
<td>0.08</td>
<td>0.08</td>
<td>1028</td>
<td>40</td>
</tr>
<tr>
<td>1991</td>
<td>Paragon XPS35</td>
<td>62</td>
<td>0.01</td>
<td>0.07</td>
<td>886</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>(OSF)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1991</td>
<td>Paragon XPS35</td>
<td>93</td>
<td>0.02</td>
<td>0.07</td>
<td>1328</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>(SUNMOS)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: A comparison of communication and computation times (microseconds) for several hypercube and mesh multiprocessors. Sources: [Dunigan 91, Dunigan 92, Dunigan 94].
2.3.2 Message passing between arbitrary nodes

When the sending and receiving nodes are not nearest neighbors, the cost of data transmission depends completely on how the intermediate nodes along the path of the message handle the message transfer. The early DM-MIMD multiprocessors, such as the iPSC/1, have a single processor per node. This processor handles both computation and communication, so messages coming to a node interrupt any computation in progress even if that node is not the ultimate recipient of the message. Later models, beginning with the iPSC/2 and nCUBE/2, have separate computation and communication hardware that can operate independently. Messages passing through a node are handled by a communication processor without impacting the computation. However, sending a message from one node to a distant node is still more expensive than sending the same message between neighboring nodes as the message is delayed slightly at each intermediate processor. (Throughout this tutorial, we refer to a node somewhat imprecisely as "a processor and its associated memory" as the modern nodes actually do have more than one processor.)

The communication performance has improved further with the advent of wormhole routing. With wormhole routing (as implemented on the Intel computers), a preliminary packet is sent from sender to receiver to set up and reserve a communication channel through the intermediate nodes. The message then passes through that channel without delays. The overhead of this circuit switching mechanism is difficult to measure, but it appears to add only a few percent to the total communication time on the newest machines (the Intel Delta and Paragon). That is, on computers with wormhole routing, the time to send a message between distant nodes is roughly the same as the time to send it between neighboring nodes [Dunigan 94].

The speed of internode communication is reflected in the communication bandwidth of a machine: more bytes of data are communicated per second when message transfer time is fast. Table 2 shows the communication bandwidth of several multiprocessors measured as a function of message size (8, 1024, and 8192 Kbytes) and the distance it must travel in the computer. A "1 hop" message is a message between nearest neighbors, and a "6 hop" message passes through five intermediate nodes between the sender and receiver. The machines that do not have separate communication processors or do not permit wormhole routing (the iPSC/1 and the nCUBE/1) suffer a large...
Table 2: The communication bandwidth of several hypercube and mesh multiprocessors. Bandwidths are shown for messages between nearest neighbors (1 hop) and between nodes 6 hops distant (5 intermediate nodes) for three different message lengths. A symbol $\sim$ means that data were not available. A symbol $\sim$ means that the data are reported to fewer significant figures than are other entries in the table. Sources: [Dunigan 91, Dunigan 92, Dunigan 94].

degradation of bandwidth when the number of hops increases from one to six. The other machines all demonstrate little or no decrease in communication bandwidth for communication between distant nodes.

2.3.3 The passing of large messages

In our expression for $T_{\text{comm}}$ in equation (1) the size of the message appears only in the term $k\tau$. On most computers, however, the startup time $\beta$ is also a function of the message size.

As shown in table 1, the startup time is reduced on the iPSC/2 and the iPSC/860 when $k$ is very small. On some machines $\beta$ is also increased when the message size is very large. In this case, the message may actually be sent as a set of smaller packets rather than as one large message, and the sending of each individual packet adds a small amount to the total communication cost. While most of the startup time is incurred for the first packet, the effect of breaking up the message into packets (packetizing) is often evident in a plot of communication time between two specific nodes versus message
size. Figure 6 shows this plot for two hypothetical computers, one that breaks large messages into packets and one that doesn’t. The solid line shows $T_{\text{comm}} = \beta + k\tau$ plotted as a function of $\tau$ when $\beta = 500$ and $\tau = 1$ for a machine that doesn’t packetize. As expected, it is a linear function of $k$. The slope of the line is $\tau$, and its $y$-intercept is $\beta$.

In contrast, the dotted line in figure 6 shows the characteristic shape of the curve when the message is divided into 250-byte packets. The solid and dotted lines are collinear until $k = 251$. At this point, the packetizing machine splits the message into two packets—one with 250 bytes and one with the remaining one byte. The sudden step up in time reflects the overhead of sending the second packet. The curve then rises linearly as the second packet grows to 250 bytes. At $k = 501$, the curve takes another step up as the third packet is released. While experimentally determined plots are generally not as smooth as those in the figure, the stairstep shape of the dotted curve are often evident in timings of machines with a measurable packetizing overhead. In particular, you may see it in your timings of programs involving large messages. The packet size is generally moderately large: on the iPSC/1 it is 1024 bytes, on the Delta it is 476 bytes, and on the Paragon it is 1792 bytes.

The overhead of packetizing messages is evident in the bandwidths presented in table 2. Specifically, packetizing of messages means that we do not see the linear increase in bandwidth that we might expect from the linear dependence of $T_{\text{comm}}$ on message size. This is apparent in the 1-hop data for the nCUBE/2. When message size is increased from 8 to 1024 bytes (a factor of 128), the bandwidth of nearest neighbor communication jumps from 50 Kb/s to 1289 (a factor of only 26). Increasing message size from 1024 to 8192 (a factor of 8) similarly increases the bandwidth by a only factor of 1.2. However, even though efficiency is lost in packetizing, the overall increase in bandwidth with message size reiterates that it is generally better to pass a few large messages than it is to pass many small ones.
Figure 6: The time to send a message between two specific nodes plotted against the length of the message. The solid line shows the characteristic shape of the curve for a machine that does not packetize messages. The dotted line shows the effects of sending packets.
2.3.4 Contention for communication links

A final and very important concern in communication performance is the number of messages traversing a given communication link at any one time. When more than one message passes on a given wire those messages contend for that wire. The effects of this contention depend on the computer on which it occurs. For example, if neighboring nodes of an iPSC/1 send messages to each other simultaneously, the time for the exchange of data is roughly $2T_{\text{comm}}$: neither the two startups nor the actual transfer of data across the link can be overlapped. Later multiprocessor models permit the overlap of startups for the sent and received messages, and the time for such an exchange (called a head-to-head send) approaches $\beta + 2k\tau$.

The problem of contention becomes more complicated when messages sent between various nodes take unknown paths through the multiprocessor. Suppose that a $k$-byte message is delayed while messages totalling $q$ bytes pass through a link along its path. Then the time it takes for that message to arrive at its destination is increased by time $q\tau$. If $q$ is large, this delay can be substantial. Contention occurs for a communication link whenever the total amount of data attempting to pass through it exceeds the bandwidth of that link [Dunigan 94].

The overall effect of contention on the performance of a parallel program is very difficult to predict in advance, but, in general, it is best to program in a style that limits contention. In sections 3.2–4.2, we discuss methods for programming mesh and hypercube multiprocessors without contention.

Notice that we have included only node-to-node communication in discussion of communication performance even though some DM-MIMD multiprocessors also have hosts. While the host and node may actually be based on the same type of processor, their function is quite different. A node is wholly dedicated to a single user process while the host is a multiuser machine. Thus, the time required for any operations on the host, including communication, is influenced by the number of other machine users as well as by the background processes running on the host. Furthermore, the link between the host and nodes is a much slower communication path than a link between nodes. Therefore, host-to-node and node-to-host communication is generally substantially less efficient than node-to-node communication. It is best to minimize communication between the host and nodes in any DM-MIMD multiprocessor program, and so we do not study host-to-node or node-to-host.
communication here.

2.4 The overall performance of DM-MIMD computers

The overall performance of a DM-MIMD multiprocessor is determined not only by its communication performance but also by its computational performance. Table 1 shows that, like the time required to send a message, the time required for a double precision floating-point multiply has decreased substantially with each new architecture. This is true for the time needed for all other floating-point operations as well. Thus, as both computation speed and communication speed have increased, we can expect that a parallel program will run faster on a new model of a DM-MIMD multiprocessor than on an old one.

However, the faster run time is not the only ingredient of performance. Amdahl’s Law tells us that the speedup of a parallel program is limited by the fraction of time spent in operations that can’t be implemented in parallel. By extension, the speedup is also limited by the time required for data communication. As no communication occurs when a program is implemented on one node, communication is part of the overhead of a parallel implementation.

To see the effect of data communication on speedup, suppose that we have a serial program with perfectly parallel computation. If no internode communication is needed, the time to run the program on $p$ nodes is just the time to run it on one node divided by $p$, i.e., $T_p = T_1/p$. If data communication is needed and computation and communication cannot be overlapped, the $p$-node time increases by the time $T_c$ required for that communication so that

$$T_p = \frac{T_1}{p} + T_c.$$

Recall that the speedup of a parallel program is defined by

$$S = \frac{T_1}{T_p}.$$

To more easily examine the effects of communication on speedup, we first
consider its reciprocal

\[ R = \frac{1}{S} = \frac{T_p}{T_1}. \]

For our perfect program with data communication, this reciprocal is

\[ R = \frac{T_1/p + T_c}{T_1} = \frac{1}{p} + \frac{T_c}{T_1}. \]

If communication is free, \( R = 1/p \) and the program has perfect speedup \( S = p \). Otherwise, the reciprocal of speedup is determined by the ratio \( T_c/T_1 \). The greater the cost of communication compared to the cost of computation, the greater the value of \( R \). That is, the larger the ratio of communication to computation costs, the smaller the speedup \( S \). The last column of table 1 shows the ratio of the message startup time \( \beta \) to the time for a double precision floating-point multiply \( \omega \). These data clearly show that with an increase in computation speed has come a marked growth in the communication to computation cost ratio. Therefore, while we can expect our parallel program to run much faster on the newer machines, we might also expect it to exhibit a smaller speedup.

The increase in the communication to computation ratio, however, is offset by the substantial improvement in communication capabilities of the newest DM-MIMD multiprocessors. Between the iPSC/1 and the Paragon (SUNMOS), the ratio \( \beta/\omega \) has grown by a factor of 66. At the same time, the communication bandwidth for 8192-byte messages has grown by a factor of 164. To see if any gains have actually been made, it is necessary to examine some computational examples. Standard speedup data is not available to us for a wide range of machines, so we instead examine the speedup of a parallel numerical program as compared to the theoretical peak performance of the machine. This is a somewhat unsatisfying efficiency measure as it mixes issues of efficient programming of the single processor (the tightness of the peak performance) with the issues of parallel programming, but it nonetheless gives us a rough indication of speedup trends.

The parallel program is the solution of the largest linear system that will fit on a given DM-MIMD multiprocessor. This is the Highly Parallel Computing benchmark of [Dongarra 91]. Table 3 shows the system size \( n \) (number of equations), the megaflops measured for that system solution, and the theoretical peak performance of that machine. The final column shows


<table>
<thead>
<tr>
<th>$p$</th>
<th>Computer</th>
<th>$n$</th>
<th>Measured (Mflops)</th>
<th>Theor Peak (Mflops)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>nCUBE/2</td>
<td>1280</td>
<td>2</td>
<td>4.4</td>
<td>0.83</td>
</tr>
<tr>
<td></td>
<td>iPSC/860</td>
<td>750</td>
<td>24</td>
<td>40</td>
<td>0.60</td>
</tr>
<tr>
<td></td>
<td>Delta</td>
<td>750</td>
<td>24</td>
<td>40</td>
<td>0.60</td>
</tr>
<tr>
<td>2</td>
<td>nCUBE/2</td>
<td>1280</td>
<td>4</td>
<td>4.7</td>
<td>0.85</td>
</tr>
<tr>
<td></td>
<td>iPSC/860</td>
<td>1500</td>
<td>58</td>
<td>80</td>
<td>0.73</td>
</tr>
<tr>
<td></td>
<td>Delta</td>
<td>1500</td>
<td>60</td>
<td>80</td>
<td>0.75</td>
</tr>
<tr>
<td>8</td>
<td>nCUBE/2</td>
<td>3960</td>
<td>16</td>
<td>19</td>
<td>0.84</td>
</tr>
<tr>
<td></td>
<td>iPSC/860</td>
<td>3000</td>
<td>190</td>
<td>320</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td>Delta</td>
<td>3000</td>
<td>230</td>
<td>320</td>
<td>0.72</td>
</tr>
</tbody>
</table>

Table 3: The Highly Parallel Computing benchmark for machines with $p = 1$, 2, and 8. Source: [Dongarra 94].

The ratio of the measured performance to the theoretical peak. Data are given for machines with $p = 1$, 2, and 8 nodes.

The iPSC/860 was one of the first DM-MIMD multiprocessors with good enough performance to solve realistic scientific problems on a moderate number of processors. Both the iPSC/860 and the Delta use the i860 processor. Between the iPSC/860 and the Delta, the ratio $\beta/\omega$ (for large messages) decreased and the message bandwidth increased. As the table shows, the overall performance of the Delta is even better than that of the iPSC/860, especially as the number of processors is increased.

A comparison of these two machines with the nCUBE/2, however, shows that improvement in communication does not tell the whole story of parallel performance. The data for similarly sized problems when $p = 8$ shows that, while the theoretical peak performance of the Delta is 17 times that of the nCUBE/2, the measured performance increases only by a factor of 14. This is true despite the fact that the communication bandwidth for an 8192-byte message is nearly eight times greater on the Delta than on an nCUBE/2. This is due in part to the almost eight-fold increase in $\beta/\omega$ between the two machines, but it is more greatly influenced by the difficulty of programming the i860 processor. (See [Dewar & Smosna 90] for details.) The data for $p = 1$ show that the theoretical peak performance figures are substantially more realistic for the nCUBE/2 than for the i860-based computers.
Distributed-Memory MIMD Computing

The main advantage of DM-MIMD multiprocessors is in their large distributed memories and cumulative computing power. The Highly Parallel Computing benchmark demonstrates the full potential of a parallel computer for solving linear systems. This benchmark gives the Gflops attained in solving the largest linear system that will fit on the computer by any stable numerical method. Table 4 shows these Gflop ratings and the attainable problem sizes for some of the machines we’ve examined using 128 nodes. For comparison, a 296-node Paragon (OSF) solves a problem of size 29400 at a rate of 12.5 Gflops [Dongarra 94].

In summary, the performance of a DM-MIMD multiprocessor depends on a variety of interacting factors. The theoretical peak performance depends on such standard concerns as the amount of memory and the speed of the processor. When data must be transferred, it is also determined by the communication bandwidth. The attainable speedup depends foremost on the degree to which the serial program can be divided into independent (and parallel) tasks. It depends further on the number and size of the messages passed and on the ratio of communication and computation times. It is difficult to predict the performance of a real program by looking at any of these interacting factors in isolation, but the performance data we’ve examined in this section show that a DM-MIMD multiprocessor can be a very powerful tool for many computations.

Table 4: The Highly Parallel Computing benchmark for some 128-node machines. Sources: [Dunigan 94, Dongarra 94].

<table>
<thead>
<tr>
<th>Computer</th>
<th>Mbytes / Node</th>
<th>Problem Size</th>
<th>Gflops</th>
</tr>
</thead>
<tbody>
<tr>
<td>nCUBE/2</td>
<td>4</td>
<td>7776</td>
<td>2.4 × 10^-4</td>
</tr>
<tr>
<td>iPSC/860</td>
<td>8</td>
<td>12000</td>
<td>2.6</td>
</tr>
<tr>
<td>Delta</td>
<td>16</td>
<td>12500</td>
<td>3.5</td>
</tr>
<tr>
<td>Paragon (OSF)</td>
<td>32</td>
<td>12000</td>
<td>4.0</td>
</tr>
</tbody>
</table>

CUBoulder: HPSC Course Notes
3 The hypercube multiprocessor

3.1 The hypercube defined

A hypercube is a distributed-memory MIMD message-passing parallel computer in which processors are connected according to a hypercube graph. Recall from the discussion in section 2.1 that a hypercube of dimension $d$ is built up recursively from $2^d$ hypercubes of dimension 0. This recursive structure of the hypercube is what makes it a particularly interesting architecture to study. In the next two sections, we show how the hypercube can emulate a variety of other architectures and how those virtual architectures are the basis for efficient data communication algorithms. Recall also that the nodes of a hypercube multiprocessor are numbered so that neighboring nodes have binary identifiers differing in exactly one bit.

3.2 Embeddings

In section 2.3, we noted that when messages contend for communication links, communication performance degrades. The special properties of the hypercube graph make it easy to design contention-free programs for hypercube multiprocessors. In fact, it is easy to carry out many communication operations efficiently using only nearest neighbor communication. In these communication schemes, messages pass along the links of a hypercube in a way that traces out a ring or tree or other simple graph. In this way, the graph is embedded into the hypercube. In this section, we review the basic embeddings that are relevant to this tutorial. A more comprehensive description is provided in [Leighton 92].

The simplest graph that can be embedded into a hypercube using only nearest neighbor connections is a linear array or ring. This is easy to see in the case of the 2-cube. Sending a message around the four nodes in numerical order $(0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0)$ involves communication between non-neighboring nodes (1 and 2), but sending the message along the route $0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 0$ fixes the problem. The messages are passed between processors with binary identifiers differing in only one bit. If node 2 receives the message but does not return it to node 0, the 2-cube models a linear array of four processors. The node numbering shown in figure 4 is precisely the numbering required for embedding the nearest neighbor array or ring by
bit manipulation. If we were to instead number the nodes counterclockwise in numerical order, flipping single bits to determine the route of the message would require us to send a message between two nodes at opposing corners of the cube.

This numbering scheme can be extended to hypercubes of any dimension by numbering the nodes of the ring according to the binary reflected Gray code [Leighton 92, Reingold et al 77]. To construct the Gray code, begin with the binary numbers

\[
\begin{align*}
0 \\
1
\end{align*}
\]

Flip them over the underscore. Preface the numbers above the line with 0’s and below the line with 1’s as follows:

\[
\begin{align*}
00 \\
01 \\
11 \\
10
\end{align*}
\]

At this point, we have constructed the Gray code ordering used in figure 4. Repeating the process a second time produces the Gray code ordering that embeds an 8-node ring into a 3-cube:

\[
\begin{align*}
000 &= (0)_{10} \\
001 &= (1)_{10} \\
011 &= (3)_{10} \\
010 &= (2)_{10} \\
110 &= (6)_{10} \\
111 &= (7)_{10} \\
101 &= (5)_{10} \\
100 &= (4)_{10}.
\end{align*}
\]

This Gray code ordering extends to any cube dimension and so permits the programmer to embed contention-free rings or one-dimensional arrays into any sized hypercube.
Figure 7: A $2 \times 4$ array embedded in a 3-cube.

The Gray code ordering also permits the embedding of arrays or tori into the hypercube. For instance, to embed a $2 \times 4$ array into the 3-cube, we number the nodes in binary as in the following table:

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>001</td>
<td>011</td>
<td>010</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>101</td>
<td>111</td>
<td>110</td>
</tr>
</tbody>
</table>

The top border of the table is the Gray code ordering of a 2-cube, and the left border is the Gray code ordering of a 1-cube. The identifiers of the nodes in the $2 \times 4$ array are constructed by taking the leftmost bits from the left border and the rightmost bits from the top border of the table. This ensures that the north, south, east, or west neighbor of a node in the array is a nearest neighbor of that node in the hypercube. The embedded array is shown in figure 7.

We can apply this same procedure to embed a $4 \times 4$ array into a 4-cube as follows:

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0000</td>
<td>0001</td>
<td>0011</td>
<td>0010</td>
</tr>
<tr>
<td>01</td>
<td>0100</td>
<td>0101</td>
<td>0111</td>
<td>0110</td>
</tr>
<tr>
<td>11</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1110</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>1001</td>
<td>1011</td>
<td>1010</td>
</tr>
</tbody>
</table>

Notice that the nodes in positions $(1, i)$ and $(4, i)$ of this array have binary identifiers differing in the leftmost bit only. Thus, the square array can
be closed into a cylinder by linking the corresponding nodes in its top and bottom rows. Similarly, the corresponding nodes on the left and right sides of the array have identifiers differing in exactly one bit. Linking those nodes closes the cylinder into a torus.

In this way, any array of size $2^{d_1} \times 2^{d_2}$ can be embedded into a cube with $p = 2^d$ processors as long as $d_1 + d_2 = d$. The array is formed into a torus by linking corresponding processors in the top and bottom rows of the array and in the left and right columns of the array. Neighbors in the array or torus are nearest neighbors in the hypercube.

By simple bit manipulation of the node identifiers, a tree of nodes can also be embedded in the hypercube. In particular, we can identify a spanning tree of the hypercube graph by linking only nearest neighbors in the hypercube. A spanning tree of a $d$-cube is simply a tree that includes all $p = 2^d$ nodes of the hypercube. One spanning tree of a 4-cube is shown in figure 8. Note that node 0 has children 1 = (0001)$_2$, 2 = (0010)$_2$, 4 = (0100)$_2$, and 8 = (1000)$_2$ in the tree. The identifiers of node 0’s children thus differ from 0 = (0000)$_2$ in exactly one bit. Similarly, node 1 = (0001)$_2$ has children 3 = (0011)$_2$, 5 = (0101)$_2$, and 9 = (1001)$_2$, and so on for all nodes in the tree.

In general, a spanning tree of height $d$ is embedded into a $d$-cube by placing node 0 at the root of the tree. The remaining levels of the tree are identified in $d$ steps where at step $l$, for $l = 1, \ldots, d$, each node with identifier $j < 2^l$ pairs with the node with identifier differing from $j$ in bit $l$ only. (Bits are numbered from right to left.) For each pair, the node with the smaller identifier is the parent node of its partner.

In the next section, we show how these embeddings can be used in communication routines for a hypercube multiprocessor. Embeddings can also be useful as a program development tool. Some application programs have an obvious underlying data structure. For example, the bisection method for computing eigenvalues, divide and conquer methods, and branch-and-bound methods all are described by trees. One way to implement such methods on a hypercube multiprocessor is to embed the underlying tree data structure into the hypercube architecture. In this way, the tree serves as a virtual architecture for the computational problem. Embeddings can also permit the development of programs for one architecture on a different architecture. It is possible to develop a mesh-oriented program on a hypercube, for instance, by designing the communication scheme to embed a mesh into the hypercube. This practice has proven valuable for computational scientists using
the mesh-connected Intel Touchstone Delta. Access to that machine is very limited, so it is often convenient to carry out program design and debugging for the Delta on a slower, but more accessible, hypercube multiprocessor.

3.3 Interprocessor communication schemes

A processor in a hypercube has direct access to only those data stored in its own local memory. However, all but the most embarrassingly parallel algorithm requires a processor to access data stored at other nodes. Because the cost of data transfer is typically high in comparison to the cost of computation, efficient hypercube programs rely on efficient communication routines. In this section, we discuss several fundamental algorithms for broadcasting data from one node of a hypercube to all others in that hypercube. These same algorithms are applied in reverse to gather data distributed among all nodes into a single node. We also present one more general algorithm to accumulate distributed data in all nodes. More information on data communication in hypercubes is provided in, for example, [Saad & Schultz 89].

The simplest broadcast algorithm is based on a linear array. Node 0 can broadcast data to all others simply by passing it along the links of a
In parallel,
do on all nodes $j$ with binary labels $\beta_j$, $0 \leq j \leq 2^d - 1$:

\begin{verbatim}
MYGRAYPOS = INVGRAY(MYID)
MYNBR = GRAY(MYGRAYPOS + 1)
If (MYID.eq.0) then
  Send data to MYNBR
Else
  Receive data
  If (MYGRAYPOS.lt.2^d - 1) then
    Send data to MYNBR
  Endif
Endif
\end{verbatim}

Figure 9: The linear array broadcast algorithm for a hypercube.

linear array embedded via a Gray code ordering of processors. Any node
receiving a message passes it on to a neighbor unless that node is at the end
of the array. A node identifies its neighbor in the array by finding its own
position in the array, adding one to it, and determining the node at that
position. Figure 9 shows the pseudocode for a node program to perform the
linear array broadcast on a hypercube. In this pseudocode, \texttt{MYID} is the
node identifier of the node executing the algorithm, and \texttt{GRAY}(j) is a function
returning the identifier of the node at position $j$ in a Gray code ordering of
the $p$ nodes. The positions and the node indices both range from 0 through
$p - 1$. \texttt{INVGRAY}(k) is a function returning the position of the node with
identifier $k$ in the Gray code ordering.

To broadcast $k$ bytes of data from node 0 to all others along the links
of the array requires $p - 1$ communication steps. Using the message startup
time $\beta$ and the byte transfer time $\tau$ introduced in section 2.3 gives a total
communication time of

$$T_{array} = (p - 1)(\beta + k\tau).$$

The linear array broadcast can be generalized to a two-dimensional array
Distributed-Memory MIMD Computing

broadcast in an obvious way. Node 0 first broadcasts the data across the linear array of length \( p_1 = 2^{d_1} \) defined by the top row of the array. The nodes of that array then broadcast the data along the linear arrays of length \( p_2 = 2^{d_2} \) defined by the columns of the array. The time for this broadcast is then

\[
T_{2Darray} = ((p_1 - 1) + (p_2 - 1))(\beta + k\tau).
\]

Thus, rearranging a linear array of length 8 into a \( 2 \times 4 \) array cuts the cost of the data broadcast to \( T_{2Darray} = \frac{1}{2}T_{array} \). Rearranging a linear array of length 16 into a \( 4 \times 4 \) array cuts the cost of the data broadcast to \( T_{2Darray} = \frac{3}{4}T_{array} \).

We can combine this approach with the spanning tree broadcast we describe next to increase the parallelism in a two-dimensional array broadcast, but we defer discussion of that hybrid approach to section 4.2.

In section 3.2, we saw how to embed a spanning tree into a hypercube of dimension \( d \). In this tree, the longest path from node 0 to any other node in the tree follows \( d \) links. This suggests that a spanning tree could be used to broadcast data from node 0 to all others in just \( d \) communication steps by distributing the data in parallel along the links of the tree. The pseudocode of figure 10 summarizes the steps in such a spanning tree broadcast (STB). Recall that for \( l \) steps with \( l \) varying from 0 through \( d - 1 \), processor \( j \) pairs with the processor having identifier differing from \( j \) in bit \( l \) only. In the implementation of the algorithm, the partner node’s identifier is easily determined via the exclusive or operator \((\text{ xor}.)\). Taking the exclusive or of \( j \) and \( 2^{l-1} \) returns the integer differing from \( j \) in bit \( l \) only.

The STB gives us a means for sending data from one node to all others in a \( d \)-cube in \( d \) time steps so that

\[
T_{STB} = d(\beta + k\tau).
\]

We can use the same mechanism to gather data distributed among the nodes into node 0. This operation is called a spanning tree gather (STG). Suppose that each processor has computed one element of a vector \( v \) so that \( v_j \) lies on processor \( j \) for \( j = 0, \ldots, p - 1 \). The STG proceeds for steps \( l = 1, \ldots, d \).

At each step, each node at level \( 2^{d-l+1} \) sends its data to its parent in the spanning tree. Thus, the sends originate from the leaves of the tree. When a node receives data, it appends it to the data it already holds and sends the accumulated data to its parent. Node 0 ultimately receives all elements of the vector \( v \).
In parallel, do on all nodes \( j \) with binary labels \( \beta_j, 0 \leq j \leq 2^d - 1 \):

For \( l = 0, \ldots, d - 1 \),

\[
\begin{align*}
\text{MYPARTNER} &= \text{MYID} \oplus 2^l \\
\text{MAXID} &= \max(\text{MYID}, \text{MYPARTNER}) \\
\text{If} \ (\text{MAXID} \lt 2^l + 1) \text{ then} \\
& \quad \text{If} \ (\text{MYID} + \text{MYPARTNER}) \text{ then} \\
& \quad \quad \text{Send data to MYPARTNER} \\
& \quad \text{Else} \\
& \quad \quad \text{Receive data} \\
& \quad \text{Endif} \\
& \text{Endif} \\
& \text{Endfor}
\end{align*}
\]

Figure 10: The STB algorithm for a hypercube.
The STG requires the same number of communication steps as the STB and, hence, the same number of communication startup charges. However, the length of the data doubles at each step, so the total communication cost for the STG is higher than for the STB. If each vector element is $k$ bytes long, the vector $v$ is $pk$ bytes, and the cost of the STG is

$$T_{STG} = d\beta + \sum_{i=1}^{d} 2^{i-1} k\tau = d\beta + (p-1)k\tau.$$ 

The STB and the STG can be combined to accumulate and broadcast a distributed vector to all nodes in the hypercube. The distributed vector is gathered in node 0 using an STG; it is then distributed to all nodes by an STB. The final algorithm we present in this section serves the same purpose but can be carried out in less time than the combination of the STB and the STG. This algorithm is known as the alternate direction exchange (ADE).

The ADE relies on the recursive structure of the hypercube. Note that a $d$-cube can be split into a pair of $(d-1)$-cubes in any of $d$ ways. And these $(d-1)$-cubes can be identified by the binary identifiers of their constituent nodes. As discussed in section 3.1, in a 3-cube, the three pairs of 2-cubes are the top and bottom, the front and back, and the two sides. As figure 11 shows, these pairs of 2-cubes are determined by the rightmost, middle, and leftmost bits, respectively, of the identifiers. That is, all nodes on the left side have zero as their least significant bit while all nodes on the right have that bit equal to one.

The ADE proceeds for $d$ steps, using a different pair of $(d-1)$-cubes at each step. At each step, corresponding processors in the two $(d-1)$-cubes send their data to each other and accumulate the result. Figure 12 depicts the...
communication patterns at the three steps of an ADE on a 3-cube. At step \( l \) of the algorithm, for \( l = 1, \ldots, d \), a node determines its partner by flipping bit \( l \) of its binary identifier. Each node begins with a single vector element \((v_j \text{ in processor } j, j = 0, \ldots, 7)\). It completes step one with two elements, step two with four elements, and ends with the full vector \( v = (v_0, \ldots, v_7)^T \).

The pseudocode for an ADE for a \( d \)-cube is given in figure 13. Again, the partner node at each step is determined via the \( \text{xor} \) operator.

The ADE algorithm appears, like the STB or STG, to require \( d \) communication steps. Thus, it should take about half the time of the STB and STG combination. The actual time for the ADE, however, depends on some characteristics of hypercube on which it is implemented. Note that, at each step of the algorithm, the partner nodes simultaneously send messages to one another. When the startup costs for these head-to-head sends can be overlapped, the cost for such a pair of messages is much less than the cost of two messages sent in turn, and the cost for an ADE approaches

\[
T_{\text{ADE}} = d\beta + 2(p - 1)k\tau.
\]

The ADE thus requires the same data transfer time as the STG and STB combination but only half the number of startups.

Hypercube multiprocessors typically come equipped with communication library routines for basic data communication operations. For example, on an iPSC/2 or iPSC/860, a call to the library routine GCOLX with the data vector argument \( x \) accumulates distributed elements of \( x \) onto all nodes via an ADE. (See [Jessup 95b] which is available via anonymous ftp in the /pub/HPSC directory at the cs.colorado.edu site.) It is not unusual, however, to need a routine that combines communication and computation in a way not provided by the libraries. Thus, it is important to understand the workings of the efficient communication algorithms in order to write efficient hypercube programs.

4 The mesh multiprocessor

4.1 The mesh defined

A distributed-memory MIMD message-passing parallel computer may also be constructed with the processors connected in a two-dimensional array.
Figure 12: The steps of an alternate direction exchange (ADE) of the elements of vector $v = (v_0, v_1, \ldots, v_7)^T$ on a 3-cube.
In parallel, do on all processors \( j \) with binary labels \( \beta_j, 0 \leq j \leq 2^d - 1 \):

For \( l = 1, \ldots, d \):

- MYPARTNER = MYID.xor.2\(^{l-1} \).
- Send vector \( v_{j+1}^{(l)} \) of length \( 2^{l-1}k \) to MYPARTNER.
- Receive vector \( v_{j'}+1^{(l)} \) of length \( 2^{l-1}k \) from MYPARTNER.
- Insert newly received elements to form the vector \( v_{j'+1}^{(l+1)} \) of length \( 2^l k \).

Endfor

Figure 13: The ADE algorithm for a hypercube.

configuration. This type of computer is known as a mesh or mesh-connected multiprocessor. As for the hypercube, each node of the mesh consists of a processor (or separate communication and computation processors) and its own local memory. Transfer of data between nodes is accomplished by message passing across the interprocessor communication links.

The size of a mesh is defined by its number of rows \( p_1 \) and its number of columns \( p_2 \). The total number of processors is \( p = p_1p_2 \). These numbers are not generally confined to powers of two and may be even or odd. A mesh node has two, three, or four nearest neighbors, depending on its position in the machine. If the mesh is connected further to form a torus, all nodes have four nearest neighbors.

The cost of the reduced number of neighbors is reduced versatility. However, many of the same concepts that apply to efficient hypercube programming extend to a mesh multiprocessor. Embeddings of alternate virtual architectures into the mesh and the communication algorithms dependent on them are the subjects of the next section. In describing these algorithms, we number the elements of the mesh according to their row and column positions. For example, the nodes of a \( 3 \times 5 \) mesh would be labeled as in figure 14. In a one-dimensional mesh, the processors are taken to lie in a row so that their labels all begin with the index 1 as in the first row of the figure.
4.2 Embeddings and interprocessor communication schemes

As is the case for the hypercube multiprocessor, the fundamental communication operations on the mesh are the broadcasting of data from one node to all others, the gathering of distributed data into a single node, and the accumulation of distributed data in all nodes.

As is also true for the hypercube, the simplest broadcast algorithms are based on a linear array. When $p_1 = 1$ or $p_2 = 1$, the $p_1 \times p_2$ mesh reduces to a linear array, and data may only be broadcast from one node to all others via a linear array broadcast. When $p_1 = 1$, a linear array broadcast is carried out with node $(1,j)$ receiving data from node $(1,j-1)$ and passing it to node $(1,j+1)$, $j = 2, \ldots, p-1$. The node in position $(1,1)$ initiates the broadcast (and so sends but does not receive), and the node at $(1,p)$ ends it (and so receives but does not send). All communication occurs between nearest neighbors in the computer, meaning that the time for a linear array broadcast stays at

$$T_{array} = (p - 1)(\beta + k\tau).$$

We use the same communication model for the mesh as for the hypercube so that $\beta$ is the communication startup time, and $k\tau$ is the time to transfer $k$ bytes of data between neighboring nodes.

As discussed in section 3.2, when $p_1$ and $p_2$ are both greater than one, the mesh may be viewed as $p_1$ connected horizontal linear arrays of length $p_2$ or as $p_2$ connected vertical linear arrays of length $p_1$. A broadcast from the node at $(1,1)$ to all others can be carried out by passing the data first.
along the top row of the array and then simultaneously along all columns of the array. Equivalently, the data may be passed down the first column and then along the rows. Again, all communication is between nearest neighbors in the mesh, and the communication time is

\[ T_{2D_{\text{array}}} = ((p_1 - 1) + (p_2 - 1))(\beta + k\tau). \]

A comparison of \( T_{\text{array}} \) and \( T_{2D_{\text{array}}} \) shows the advantage of greater connectivity: the time for a linear array broadcast is proportional to the product of \( p_1 \) and \( p_2 \) while the time for a two-dimensional mesh broadcast is proportional to their sum.

The cost for a one-dimensional (linear) array broadcast, and hence a two-dimensional array broadcast, may be reduced further by identifying a greater level of parallelism in those operations. To do this, we assume that both \( p_1 \) and \( p_2 \) are both powers of two. We also abandon the requirement of nearest neighbor communication while maintaining the requirement of a contention-free scheme. The wormhole routing used in modern message-passing architectures ensures that messages travel quickly between distant nodes as long as messages are passed on separate communication wires.

The key to the more efficient algorithm is to involve as many nodes in the broadcast as quickly as possible. In the linear array of eight nodes, we can do this by having the node at \((1,1)\) send the message first to a node in the center position \((1,5)\) of the array. The nodes at \((1,1)\) and \((1,5)\) may then distribute the message independently (and in parallel) in the two halves of the array. Applying this procedure recursively in the two halves increases the parallelism further. In figure 15, we show how the messages are passed in a broadcast from the node at \((1,1)\) to all others in an array of length eight. The routes of the messages are shown at each of the three time steps of the broadcast. Note that no two messages ever travel over the same link simultaneously in this linear array broadcast, so the time required for this contention-free broadcast is

\[ T_{1D_{c.f}} = d(\beta + k\tau), \]

where \( d = \log_2 p \). (This assumes that the time for a multihop message is about the same as the time for a message sent between nearest neighbors.)

This one-dimensional algorithm is applied to a two-dimensional mesh by broadcasting first along the first row of the mesh and then simultaneously
down all of the columns of the mesh. This ensures that increasing the
dimension of the mesh does not introduce contention for wires. Just like the
STB broadcast algorithm introduced for the hypercube in section 3.3, the
contention-free mesh algorithm works efficiently by embedding a spanning
tree into the two-dimensional mesh. Figure 16 shows the spanning tree cre-
ted in a $4 \times 4$ mesh.

The steps of the spanning tree broadcast on a $p_1 \times p_2$ mesh are summa-
ized by the pseudocode in figure 17. The time to carry out this Mesh STB
broadcast for $k$ bytes of data on a mesh with wormhole routing is about

$$T_{\text{MeshSTB}} = (d_1 + d_2)(\beta + k\tau),$$

where $d_1 = \log_2 p_1$ and $d_2 = \log_2 p_2$. A comparison of $T_{\text{2Darray}}$ and
$T_{\text{MeshSTB}}$ shows that using the Mesh STB broadcast algorithm can lead to
a substantial savings in time when compared to the nearest neighbor one-
dimensional or two-dimensional array broadcast.

The Mesh STG is carried out by running the Mesh STB algorithm in
reverse. Data transmission begins from the leaves of the mesh spanning tree
and continues to its root. Data is accumulated along the way so that the root
node (1,1) completes the Mesh STG with a complete copy of all of the data
originally distributed among all nodes. On the hypercube, the alternate
direction exchange (ADE) presented in section 3.3 was the most efficient
means for such a gather operation. The ADE, however, relies completely on
the recursive structure of the hypercube and so does not extend to the mesh
architecture. Thus, a mesh spanning tree gather (MeshSTG) and broadcast
must be combined to accumulate complete copies of distributed data on all
nodes of the mesh. Node (1,1) then begins a Mesh STB to send that data back
to all nodes. The exact cost of this combination depends on how much the
data vector grows or shrinks during the exchange, but it is roughly bounded
by $2T_{\text{MeshSTB}}$ where $k$ is taken to be the length of the accumulated vector.

These mesh operations can also be applied if $p_1$ and $p_2$ are not powers
of 2. For details of the modifications to arbitrary mesh size as well as for
the optimizations available under wormhole routing, see [Barnett et al 91,
Barnett et al 94].

As is true for the hypercube, mesh-connected multiprocessors also are
equipped with library routines that perform the most basic communication
operations. (See, for example, [Jessup & Neves 94] available via anonymous
Figure 15: The messages passed at three communication steps of the contention-free linear array broadcast. Source: [Barnett et al 91].
ftp in the /pub/HPSC directory at the cs.colorado.edu site.) The user should also beware that when only a subset of the mesh processors are used, the subset may not actually represent a rectangular mesh of contiguous processors within the full mesh. On the Paragon, for example, a rectangular submesh is assigned only when specifically requested by the user. When the submesh is not rectangular, the communication algorithms presented in this section are no longer contention-free.

5 Some other multiprocessors

In the case of the tree multiprocessors, the graph and architectural issues are not easily separated. While the nodes in mesh or hypercube multiprocessors are typically identical, in tree multiprocessors, different positions in the tree are often allocated to different types of processors or controllers. For example, in the tree-based Teradata DBC/1012 Data Base Computer, two types of processors at the leaves process queries and access databases. Processors located above the leaves are dedicated to sorting and broadcasting operations.
In parallel, 
do on all nodes \((i, j)\), with \(1 \leq i \leq p_1\) and \(1 \leq j \leq p_2\): 

Get node identifier \(\text{MYID} = (\text{MYROW, MYCOL})\).

If \(\text{MYROW} = 1\) then 

For \(l = 1, \ldots, d_2\), 
\[
\text{PARTNERCOL} = \text{MYCOL} + p_2/2^l \\
\text{MYPARTNER} = (1, \text{PARTNERCOL}) \\
\text{MAXCOL} = \max(1, \text{PARTNERCOL}) \\
\text{If} (\text{MAXCOL} \leq p_2/2^l) \text{ then} \\
\quad \text{If} (\text{MYCOL} < \text{PARTNERCOL}) \text{ then} \\
\quad \quad \text{Send data to MYPARTNER} \\
\quad \text{Else} \\
\quad \quad \text{Receive data} \\
\quad \text{Endif} \\
\text{Endif} \\
\text{Endfor}
\]

For \(l = 1, \ldots, d_1\), 
\[
\text{PARTNERROW} = \text{MYROW} + p_2/2^l \\
\text{MYPARTNER} = (\text{PARTNERROW}, \text{MYCOL}) \\
\text{MAXROW} = \max(\text{MYROW, PARTNERROW}) \\
\text{If} (\text{MAXROW} \leq p_2/2^l) \text{ then} \\
\quad \text{If} (\text{MYROW} < \text{PARTNERROW}) \text{ then} \\
\quad \quad \text{Send data to MYPARTNER} \\
\quad \text{Else} \\
\quad \quad \text{Receive data} \\
\quad \text{Endif} \\
\text{Endif} \\
\text{Endfor} \\
\text{Endif}
\]

Figure 17: The STB algorithm for a mesh.
The general-purpose Thinking Machines CM-5 architecture is based on a quaternary tree with SPARC processors at the leaves and routing chips at the other tree nodes. Each leaf processor has four parent routing chips. Each routing chip has two to four parent routing chips depending on its location in the tree. The idea is to increase the communication bandwidth between neighboring nodes higher up in the tree: these nodes occupy a larger subtree than those lower down in the tree and so are more likely to cause a communication bottleneck if their bandwidth is insufficient. Like a real tree, a tree of this type has “thicker” branches near the root than at the leaves. This architecture is termed a fat tree [Leiserson et al 92].

In an architecture like the CM-5, the concept of nearest neighbor no longer strictly applies. To route a message from one node to another, the message is sent up the tree to the least common ancestor of the two nodes, and then down to the destination node. As a message goes up the tree, it must choose which parent connection to take at each level. This choice is made randomly from among those links unobstructed by other messages. After the message has attained the height of the least common ancestor of the source and destination processors, it travels down the tree to its destination. The random choice at each level is designed to balance the load on the network. (The manufacturers of the CM-5 advertise a minimum of 5 Mbytes per second node to node data transfer regardless of other node communication loads. Near communications can attain up to 20 Mbytes per second because they may not have to be routed as far up the tree.)

The complexity of the routing scheme on the CM-5 means that it is difficult for the programmer to write communication routines. On this machine, all but the most expert programmers use library routines for all communication. For more information on the CM-5, see [Leiserson et al 92].

Other multiprocessor architectures have identical processors at all nodes but use a completely different interconnection mechanism. For example, the nodes of an Omega network are not connected directly by communication wires, but rather they are connected by means of switches [Hwang 93]. The IBM SP1 uses a high speed Omega switch by which a message may be sent quickly between any two nodes of the machine. Thus, the concept of nearest neighbor is not important in programming the IBM SP1.

On the SP1, however, the speed of communication is determined by the transport layer used. The transport layers represent different message pass-
ing protocols. In the layer of lowest bandwidth and highest latency (Ethernet/IP), messages are passed over ethernet connections between nodes without use of the high speed switch. In the layer of greatest bandwidth and lowest latency (EUI-H), messages are passed using a low-overhead interface to the high speed switch. However, only the lowest bandwidth layers support multiple processes per node and multiple parallel jobs per node, so communication bandwidth may be sacrificed in the interest of greater parallelism.

Typically, the applications programmer does not have to be concerned with the details of transport layer programming. These are instead handled by various parallel programming languages. Nevertheless, the programmer may need to specify the layer to be used by the language and so must be aware of the benefits and disadvantages of the layer types. For more details on the SP1, see [Gropp et al 94].

The details of programming a DM-MIMD multiprocessor differ from machine to machine. On some machines, the programmer has much control over such details as the routes traveled by messages, while, on others, such details are hidden in communication libraries. In either case, some knowledge of the inner workings of a particular computer is essential to writing efficient programs for it.

6 A sample program

In this section, we present a rough outline of how to construct a program for a generic DM-MIMD multiprocessor. This example is only intended to present some of the most basic issues one needs to consider in writing this type of program. The details of this process can vary substantially from program to program. They can also vary from machine to machine.

For our example, we develop a parallel program for computing $\|Ax\|_1$ on a DM-MIMD multiprocessor with $p$ processors, given the $p \times p$ matrix $A$ and the vector $x$ of length $p$. Recall that if $y = Ax$, its 1-norm is given by $\|y\|_1 = \sum_{i=1}^{p} |y_i|$.

It is always best to begin with a carefully designed serial algorithm or code because it is hard to think in parallel when there are still errors in the serial ideas. Therefore, we begin by writing a MATLAB program for this computation:
While we can be sure that this program is correct, it does not reveal the independent tasks comprising this computation. The next step in developing the parallel program is to identify those tasks. One way to break up the problem is to recognize that the \( i \)th element of the vector \( y \) comes from multiplying the vector \( x \) by row \( i \) of the matrix \( A \). Therefore, the serial computation may be rewritten as

\[
\begin{align*}
    y & = A \times x; \\
    a & = \text{norm}(y, 1);
\end{align*}
\]

This reformulation of the problem clearly shows that the elements of the vector \( y \) are computed independently of one another. This suggests that the computation of \( y = Ax \) can be implemented in parallel by assigning the computation of the element \( y_i \) to node \( i - 1 \), for \( i = 1, \ldots, p \).

It then remains to implement the norm computation in parallel. Expanding the norm function call to show the operations performed now gives us the serial program

\[
\begin{align*}
    \text{for } i = 1:p \\
    \quad y(i) & = A(i,:) \times x; \\
    \text{end} \\
    a & = \text{norm}(y, 1);
\end{align*}
\]

Thus, computing the norm requires us to sum the absolute values of the distributed components of the vector \( y \). Unlike the matrix-vector product, this sum cannot be implemented without communicating data between nodes.

Most DM-MIMD multiprocessors supply a library call to compute a sum of distributed data (called a *global sum*), but we can also write one ourselves by making simple modifications to a standard communication routine. For
example, on a hypercube or mesh multiprocessor, we could perform a spanning tree gather but replace the vector accumulation with an addition at each step. In this global sum routine, each leaf node of the spanning tree passes the absolute value of its vector element $|y_i|$ to its parent. Each parent node then adds the values received from its children to the absolute value of its own vector element $|y_i|$ and sends the result to its own parent. Continuing the process all the way up the spanning tree leaves the sum $\| y \|_1$ in the root node. If this norm is required by all nodes for subsequent computation, it can be broadcast to them from the root via a spanning tree broadcast.

This organization of the parallel algorithm means that every node $i - 1$, $i = 1, \ldots, p$, of the multiprocessor runs the following program:

$$
y(i) = A(i,:) \times x;
$$

$$
y(i) = \text{abs}(y(i));
$$

call global sum(a,y)

In this program, the routine global sum leaves the sum of the distributed elements of $y$ in $A$ on one or all of the processors, depending on its implementation. To run this parallel program, we must first generate the matrix row $A(i,:)$ and the vector $x$ on each node $i - 1$.

In this case, the computation done in the serial program takes $p$ times as long as the parallel program run on $p$ nodes. Moreover, because only one row of the matrix is used on a node, the serial program also takes $p$ times the storage of the parallel program. The program thus appears to be efficient in terms of both time and storage. However, before we implement it, we need to estimate its overall efficiency. That is, we need to make sure that we have not designed a parallel program in which the cost of data communication outweighs the benefits of parallel computation.

Although the actual performance of a parallel program depends on many interacting factors, we can get a rough estimate of its cost by means of the analytical tools $\omega$, $\beta$, and $\tau$ introduced in section 2.3 to model the costs of a floating-point operation, a message startup, and a byte transfer, respectively.

The serial computation of $\| Ax \|_1$ for a $p \times p$ matrix $A$ requires $p^2$ multiplications and additions. The cost of this is roughly

$$
T_1 = 2p^2 \omega.
$$

In comparison, the cost of executing the instructions $y(i) = A(i,:) \times x$; and $y(i) = \text{abs}(y(i))$; in the parallel program is about $2p\omega$. If the computation
is done in double precision, the numbers are eight bytes long, and the cost of the global sum on a hypercube or mesh is approximately \( d(\omega + \beta + 8\tau) \). The total cost of the parallel algorithm is then

\[
T_p = (2p + d)\omega + d\beta + 8d\tau.
\]

Suppose that we wish to run our program on a DM-MIMD multiprocessor with \( p = 2^d = 128 \) where \( \beta = 500 \), \( \omega = 10 \), and \( \tau = 1 \). On this machine, \( T_1 = 327,680 \) and \( T_p = 6256 \) so that the theoretical speedup is \( S = 52 \). This represents an efficiency of 41\% and so indicates that our algorithm may be worth implementing.

If, on the other hand, we want to run our program on a machine with only 16 nodes, the matrix size drops to \( 16 \times 16 \). For this problem size, \( T_1 = 5120 \), \( T_p = 243 \), and \( S = 2.1 \). The small amount of computation performed is not enough to mask the communication cost, and the efficiency drops to only 13\%. This poor efficiency may signal that we need to reconsider the design of the parallel implementation.

We have created our parallel algorithm by using one obvious division of the computation and communication steps. In general, it is not wise to proceed with the first thing that comes to mind without examining the alternatives. For example, we might want to rewrite the program to use a matrix of arbitrary order \( n \). By using \( p < n \), we could then assign more elements of \( y \) to each node and so increase the amount of computation performed relative to the number of messages sent.

We might even want to reconsider how we split up the matrix operations in the first place. For instance, it is often more efficient to split the matrix into blocks than into rows. (See [Golub & Van Loan 89] for more information on blocked algorithms.) We may also need to alter the program if a preceding computation distributes the matrix \( A \) among the nodes differently than we have required. If the program is to be used repeatedly or if the expected run time is very long, it is particularly important to develop an efficient implementation.

We now summarize the basic steps to remember when writing a program for a DM-MIMD multiprocessor.

1. Develop a good serial algorithm for the problem.

2. Identify the independent computational tasks in that algorithm.
3. Determine how to map those steps to the nodes.

4. Determine the data needs (initial and intermediate) of the parallel tasks.

5. Devise communication schemes to ensure that all data are in the proper place at the proper time and that the final result is easily accessible.

6. Assess the expected performance of the parallel algorithm. If the efficiency appears too small, return to step 2.

7. Write, test, debug, and run the parallel code.

7 Acknowledgments

The author thanks Silvia Crivelli, Jack Dongarra, Dirk Grunwald, and Robert van de Geijn for many helpful discussions, Evi Nemeth and David Kincaid for their comments on the manuscript, and Andre van der Hoek for preparing many of the figures.
8 Appendix 1: Using the iPSC/2 at UCB

The iPSC/2 hypercube multiprocessor at the University of Colorado at Boulder has 32 nodes. Each hypercube node has an 80386/80387 processor and four megabytes of memory. Nodes with this processor type are called CX nodes. The hypercube is connected to a system resource manager or host machine. The host is named bud.cs.colorado.edu and has the IP address 128.138.242.4. Bud is an Intel computer and based on the 80386/80387 processor. Although the iPSC/2 can be purchased with disks attached to the nodes (a Concurrent File System or CFS), the nodes of bud do not have any secondary storage.

The hypercube nodes are accessed via the host. You will be provided a login on bud as part of your course enrollment process. To log on, type the following command:

```
rlogin bud.cs.colorado.edu
```

The host will respond by requesting your password.

Intel hypercubes come in two flavors: those with a local host and those with a remote host. A remote host is one intended to be used as a compile and load engine only. A local host is one that can be used as an active participant in the hypercube program as well. Bud is a local host.

This appendix is organized as follows. In section 8.1, we present an example of a node program. In section 8.2, we present a host program corresponding to the example node program. In section 8.3, we show how to run the host and node programs. In section 8.4, we introduce some of the programming tools provided by Intel for the iPSC/2. In section 8.5, we explain how to time a program on the hypercube. In section 8.6, we give the local rules for use of bud. These rules are the key to maintaining your sanity when using bud. In section 8.7, we mention a couple of options to aid in debugging hypercube programs.

This appendix covers only those operations and techniques most relevant to programs included in the HPSC materials. For more information, see [Intel 91a, Intel 91b].
8.1 Writing a node program

Bud has compilers for two languages. The Fortran 77 compiler is located in /usr/bin/f77, and the C compiler is in /usr/bin/cc. The sample programs we show in this document are written in Fortran 77.

The most important thing to remember when writing a program for a hypercube node is that a node processor has access to the data in its own local memory only. Data on other processors must be accessed by passing messages.

The first step in writing a node program is to identify those parts of the program that can run independently of the others. Any of these parts that can run simultaneously can be implemented in parallel on different processors. In some cases, the implementations lead to wholly different codes on the different processors (a MIMD model). In others, the codes are the same, but different processors work on different data (a SIMD model). Both SIMD and MIMD programs can run on a MIMD multiprocessor.

The sample node program of this section is a generalization of the most basic MIMD program introduced in the first section of [Jessup 95a]. In that case, two nodes shared the work in summing twenty numbers. In the sample program of this section, \( p \leq 16 \) nodes share the work in summing 16 numbers. In this example, all nodes are provided with all 16 numbers but each node only computes a partial sum of \( 16/p \) of them. (If \( p = 16 \), the partial sum is just the node's single element itself.) The total sum is accumulated by a gather operation using an embedded Gray code ring. Node 0 initiates the sum by sending its partial sum to node 1. When node 1 receives node 0's partial sum, it adds it to its own and sends the result to node 3. This process is repeated by all the rest of the nodes in the ring. The last node sends the total value to node 0 which then reports it to the host. The host writes the result into a file.

The node program is shown in figures 18–20. Figure 18 is simply the declaration of all variables used in the program. The list of variables includes those used by the message passing commands. Figure 18 also shows the value of the sixteen elements of the data vector \( a \).

Figure 19 shows the parts of the code where the cube size and the node identifier are determined (\( p = \text{numnodes}() \) and \( \text{menode} = \text{mynode}() \)). The host identifier, the host and node process identifiers, and the other message parameters are also determined in this figure. In the last few lines, the node
determines the next node \((\text{nextnode})\) in the Gray code ring, the node to which it will send the partial sum.

Figure 20 shows how the data is divided up among the nodes of the cube and how each node computes its partial sum. The remainder of the code is the global sum operation. Node 0 calls \text{csend} to begin the summing process. The other nodes receive a message by calling \text{crecv} then forward the message to \text{nextnode} by using \text{csend}. The arguments to these basic message passing commands specify the message location and length and the message destination. The calling sequences are

```c
call csend(msgtype, msgbuf, msglen, destination, destinationpid)
```

- \text{msgtype} - an integer that defines the type of message being sent.
- \text{msgbuf} - the location (variable name) of the message being sent. It may be of any data type (integer, real, etc.) and of any dimension (scalar, vector, etc.).
- \text{msglen} - an integer equal to the message length in bytes.
- \text{destination} - an integer equal to the identifier of the destination node.
- \text{destinationpid} - an integer equal to the recipient process on the destination node. In our applications, we always run only one process per node, so the pid is always 0.
call crecv(msgtype, msgbuf, msglen)

msgtype - an integer that defines the type of message to be received.

msgbuf - the location into which the received message should be written.

msglen - an integer equal to the message length in bytes.

Note that crecv gives no mechanism for identifying which node sent the message being received. However, this information can be encoded in the message type parameter.

The routines csend and crecv are examples of synchronous communication calls. The calling process is blocked until the operation is complete. Asynchronous calls (isend and irecv) are also available. These calls allow execution of the program to continue while messages are in transport.

In the example, the only form of I/O performed by the nodes is in the form of print statements. Because bud does not have a CFS, the nodes cannot write directly into files.

8.2 Writing a host program

A host program may involve computation, node program control, and communication of data to and from the nodes. The host program corresponding to the node program of figures 18–20 is shown in figure 21. In this case, the host program involves only one process, and the first step is to set the identifier of that process to 0. The host then loads the node program onto all of the nodes. Finally, the host waits for the result to be reported by the nodes and writes that result into a file.

The communication routines on the host are identical to those on the nodes. In this program, the host calls crecv. The other important routine in this program is the one (load) to load the node processes onto the nodes. It is called as follows.

call load(filename, destination, destinationpid)
Sample Node Program

Hypercube Variables:

- `p`: number of nodes in hypercube
- `menode`: node id returned by `mynode()`
- `mepid`: process id returned by `mypid()`
- `megray`: position of this node in Gray code ring
- `nextnode`: next node in ring
- `nextpid`: next process in ring
- `hostid`: host processor
- `hostpid`: host process
- `msglen`: length of message (in bytes)
- `msgtype`: message type

```c
program node
integer datasize
parameter (datasize = 16)

integer p, menode, mepid, megray, nextnode, nextpid,
  & hostid, hostpid, msglen, msgtype
integer megray, numelts, offset, j
integer ginv, gray

double precision sumin, sumout
```

Data

```c
double precision a(datasize)
data a /0.3, 1.4, 3.2, 2.1, 6.4, 7.7, 5.8, 4.3, 12.9, 13.7,
  & 15.7, 14.9, 10.0, 11.1, 9.3, 8.2/
```

Figure 18: A sample node program (`node.f`, part 1).
Find number of nodes in hypercube.

```
p = numnodes()
```

Get identifying parameters of this node and of host.

```
menode = mynode()
mepid = mypid()
megray = ginv(menode)
hostid = myhost()
hostpid = 0
```

Determine number of data elements in this node and message length (in bytes) and type.

```
msglen = 8
msgtype = 0
```

Get identifying parameters of next node in the embedded Gray code ring.

```
nextnode = mod(megray+1,p)
nextnode = gray(nextnode)
nextpid = mepid
```

Figure 19: A sample node program (node.f, part 2).
Distributed-Memory MIMD Computing

Compute local sum.

\[
\text{numelts} = \frac{\text{datasize}}{p} \\
\text{offset} = \text{menode} \times \text{numelts} \\
\text{sumout} = 0.0d0 \\
\text{do } j = \text{offset + 1}, \text{offset + numelts} \\
\quad \text{sumout} = \text{sumout} + a(j) \\
\text{enddo}
\]

Perform ring operations. Node 0 initiates ring broadcast and final result to host. Upon receipt of data in message buffer sumin, node adds sumin to message buffer sumout and sends sumout to next node in ring.

\[
\text{if } (\text{menode} .eq. 0) \text{ then} \\
\quad \text{call csend(\text{msgtype}, \text{sumout}, \text{msglen}, \text{nextnode}, \text{nextpid})} \\
\text{endif}
\]

\[
\text{call crecv(\text{msgtype}, \text{sumin}, \text{msglen})}
\]

\[
\text{if } (\text{menode} .ne. 0) \text{ then} \\
\quad \text{sumout} = \text{sumout} + \text{sumin} \\
\quad \text{call csend(\text{msgtype}, \text{sumout}, \text{msglen}, \text{nextnode}, \text{nextpid})} \\
\text{else} \\
\quad \text{call csend(\text{msgtype}, \text{sumin}, \text{msglen}, \text{hostid}, \text{hostpid})} \\
\quad \text{nextnode} = \text{hostid} \\
\text{endif}
\]

\[
\text{print*,'node ',menode,': sent to ',nextnode, ', bye.'}
\]

end

Figure 20: A sample node program (node.f, part3).
Sample host program

program host
  double precision result

set host process id.

call setpid(0)

load node program into all nodes.

print *, 'host: loading node program'
call load('node', -1, 0)

receive result from node and write to file.

call crecv(0, result, 8)
open(3,file='sum.dat',status='unknown')
write(3,*) result
print*, 'host: data is in file sum.dat'

end

Figure 21: A sample host program (host.f).

filename - a character string containing the name of the node program executable.
destination - an integer equal to the identifier of the destination node. A value of -1 means that the program should be loaded into all nodes.
destinationpid - an integer equal to the recipient process on the destination node(s).

While the host and node are both based on the same type of processor, their function is quite different. A node is wholly dedicated to a single user
process while the host is a multiuser machine. Thus, the time required for any operations on the host, including communication, is influenced by the number of other cube users as well as by the background processes running on the host. Furthermore, the link between the host and nodes is a much slower communication path than a link between nodes. Therefore, it is best to minimize communication between the host and nodes in any hypercube program.

In general, the host should be used only for compiling and loading node programs and for communicating initial data to and final results from the nodes. You may find some exceptions among the HPSC software where severe memory limitations on the nodes leads to intermediate host-node communication steps to transfer data.

8.3 Running the host and node programs

Once the programs have been written, it remains to compile, load, and run them. In this section, we discuss these operations.

8.3.1 The makefile

The makefile for compiling the host and node programs is shown in figure 22. You can see that this makefile is just like the makefiles you’ve written for other Unix programs. The only special feature is that that host program must be linked with the -host option, and the the node program must be linked with the -node option. Running this makefile (make both) in a directory with the Fortran 77 programs host.f and node.f produces the executables host and node.

8.3.2 Command line options for controlling the hypercube

The host and node programs are started by commands issued on the host. The first step is to reserve a subcube of bud. This is done with the command getcube -t <p>, where p is the number of processors you wish to use. You must request a number of processors equal to a power of two. The processors assigned to you by getcube remain yours alone to use until you issue the command relicube.
Figure 22: A sample makefile.

The next step is to run the host program by typing the name `host` on the command line. The load instruction in the host program then causes the node executable to be loaded onto the node processors. The host is physically connected to all processors, but it loads the node program onto node 0 which then distributes it to all others. This minimizes the use of the slow host-to-node link.

The final step is to release the cube for others to use once your host and node programs have completed. If you don’t know for sure that all node processes have exited, you should first type the command `killcube`. This will kill all remaining node processes. This is important as surviving node processes can interfere (sometimes in peculiar ways!) with subsequent runs of your host and node programs.

Once all node processes have completed, you must run the command `relcube` to relinquish your claim to the processors you’ve been using. This is very important as other users cannot use the cube until use issue `relcube`. 
To summarize, the host and node programs are run by the following sequence of commands.

1. make both
2. getcube
3. host
4. killcube
5. relcube

You may also find it useful to use the command `cubeinfo -s` to find out who is using which nodes of the hypercube.

### 8.3.3 The results

The results of running the above sequence of commands on bud for our sample host and node programs is shown in figure 23. We compiled both programs successfully by using the makefile. We successfully allocated a subcube with eight nodes via the `getcube` command. We then typed `host` load and execute the node program. The nodes executed their tasks in Gray code order then exited. The host wrote the data file and also exited.

We did not forget to release the cube.

In this test, all the nodes and the host write to the host window. The nodes do not have direct access to this window and so must send their messages to the host for printing. Because these messages must travel over the slow node-to-host links and because the printing of them may be delayed by other host processes, it is not unusual to see the host and node messages displayed in a scrambled order. In some runs of this program, we actually saw the host message precede some of the node messages even though it was generated last.

### 8.3.4 The hostless node program

The sample host program includes a call to the routine `load` that loads the node executable onto the node processors. This instruction can also be issued from the command line on the host. (Similarly, some of the commands
bud>
bud> make both
    f77 -o host host.o -host
    f77 -o node node.o -node
bud>
bud> getcube -t8
getcube successful: cube type 8m4n0 allocated
bud>
bud> host
host: loading node program
node  1: sent to  3, bye.
node  3: sent to  2, bye.
node  2: sent to  6, bye.
node  6: sent to  7, bye.
node  7: sent to  5, bye.
node  5: sent to  4, bye.
node  4: sent to  0, bye.
node  0: sent to  8, bye.
host: data is in file sum.dat
bud>
bud> relcube
relcube released 1 cube
bud>

Figure 23: Output from the sample host and node programs.
introduced here as command line options may also be included in the host program.) Indeed, the host program need contain no node program control whatsoever. Furthermore, if the node programs can generate all the data they need for their computation, there is no need for a host program at all. In that case, the operation of the hypercube proceeds by the following sequence of instructions:

1. make node
2. getcube -t <p>
3. load node
4. killcube
5. relcube

The routine load can take many different arguments to specify such things as the nodes and node processes to be loaded. See [Intel 91b] for details.

Because bud does not have a CFS, output from a hostless program can only be printed to the screen and not written into files.

Sections 8.1–8.3 show only a small fraction of the commands available for carrying out communication steps or controlling the hypercube. For a complete list, see [Intel 91b].

8.4 Communication libraries

Our sample host and node programs show how the most fundamental send and receive commands may be used to carry out global communication and manipulation of data in a hypercube program. In some cases, the programming of internode communication is made even easier by means of the iPSC/2 Fortran routines.

Among other operations, the iPSC/2 library includes routines to compute the elementwise sum or product of two distributed vectors or the maximum or minimum element of a distributed vector. The library also includes routines to perform logical operations such as an “and” and “or” of distributed vector elements. The library routines can be used to manipulate vectors distributed over the nodes by blocks as well as by single elements. Because they operate
on data spread throughout the hypercube, these routines are said to perform global operations.

The global operations are represented in Fortran as subroutine calls. The name of the subroutine depends on the type of global operation. For example, subroutines GDSUM, GSSUM, and GISUM are used to produce the elementwise sum of the elements of two double precision vectors, single precision vectors, and integer vectors, respectively. This pattern of naming, wherein the second letter of the name identifies the data type, applies to the other global operations as well. In this section, we describe only the double precision operations, but these descriptions apply equally well to the commands for the other data types.

The global commands are issued by node processes, and all nodes must execute the command before processing on any node can continue. Thus, the global operations synchronize the nodes of the hypercube. The result of the global command is returned to every node.

The library routines are efficient because they take into account the connectivity and parallelism of the hypercube. For example, the routine GCOLX carries out an alternate direction exchange (ADE) [Jessup 95a] to gather a distributed vector on all nodes. The communication carried out in a GDSUM is based on the ADE, too. (We could use GDSUM to produce a more efficient implementation of the global sum example of section 8.1.) In figures we show how to use GDSUM and GCOLX, respectively, in a node program.
GLOBAL SUM

FORM OF USE: CALL GDSUM(V, LEN, WORK)

V - a vector or scalar of type double precision.
   It is the input for the operation, and receives the result of the operation.

LEN - an integer equal to the number of elements of V.
   If V is a scalar then LEN = 1. The value of LEN must be the same on
   every node.

WORK - a vector of the same type as V with length equal to or greater than LEN.

DESCRIPTION:

Assume that the hypercube has 8 nodes. If V is a scalar and its values on
nodes 0, 1, 2, 3, 4, 5, 6, and 7 are 1, 3, 5, 7, 9, 11, 13, and 15, respectively, then
the result of the nodes executing CALL GDSUM(V, 1, WORK) is to assign the value
64 = 1 + 3 + ... + 15 to V on every node.

If V is a vector, then the global sum is taken element by element. For example, if V0, V1, ..., V7 are vectors of length 4 with vector Vi on processor i, and

\[ V_0 = (0, 1, 2, 3) \quad V_1 = (4, 5, 6, 7) \]
\[ V_2 = (8, 9, 10, 11) \quad V_3 = (12, 13, 14, 15) \]
\[ V_4 = (16, 17, 18, 19) \quad V_5 = (20, 21, 22, 23) \]
\[ V_6 = (24, 25, 26, 27) \quad V_7 = (28, 29, 30, 31) \]

then the result of CALL GDSUM(Vi, 4, WORK) executed on the nodes of the
cube (with i = 0, 1, 2, ..., 7, and vector WORK of length at least 4)
is to produce the result

\[ V_0 = (112, 120, 128, 136) \quad V_1 = (112, 120, 128, 136) \]
\[ V_2 = (112, 120, 128, 136) \quad V_3 = (112, 120, 128, 136) \]
\[ V_4 = (112, 120, 128, 136) \quad V_5 = (112, 120, 128, 136) \]
\[ V_6 = (112, 120, 128, 136) \quad V_7 = (112, 120, 128, 136) \]
GLOBAL CONCATENATE

FORM OF USE:  CALL GCOLX(VSEG,LENS,VCAT)

VSEG - a segment (block) of a vector of type double precision.
       It is the input for the operation. There is one segment of
       the vector on each processor. It is understood that the order
       of the segments is the order of the processors (i.e., segment 0
       on processor 0, segment 1 on processor 1, etc.)

LENS - a vector giving the lengths of the segments in bytes.
       Element i of LENS is the integer value of the length in bytes
       of the segment on processor i. This entire vector must be on every processor
       at the time of executing the command.

VCAT - This is the concatenated vector, the output of this command.

DESCRIPTION:
   Assume that the cube has 4 nodes, and assume that the segments of the vector
   are as follows:

   VSEG0 = (1, 2)
   VSEG1 = (3, 4, 5)
   VSEG2 = (6)
   VSEG3 = (7, 8),

where VSEGi is the segment on node i. Each double precision element is eight
bytes long, so the vector LENS of segment lengths is

   LENS = (16, 24, 8, 16).

If VCATi denotes the vector found on node i, the result of executing the
command CALL GCOLX(VSEGi, LENS, VCATi) on all nodes is

   VCAT0 = (1, 2, 3, 4, 5, 6, 7, 8)
   VCAT1 = (1, 2, 3, 4, 5, 6, 7, 8)
   VCAT2 = (1, 2, 3, 4, 5, 6, 7, 8)
   VCAT3 = (1, 2, 3, 4, 5, 6, 7, 8)

Figure 25: The global operator GCOLX
8.5 Timing a hypercube program

Programs may be timed on the host, on the nodes, or on both the host and nodes. The following two Fortran routines are available for timing programs:

**double precision function dclock**

The call \( t = \text{dclock}() \) assigns to the double precision variable \( t \) the time in seconds since the cube was booted. Dclock is available on the nodes only and has a granularity of 1 ms on CX nodes.

**integer function mclock**

Mclock is available on the host and on the nodes, but it has different functionality on the two different processor types.

On the nodes, the call \( t = \text{mclock}() \) assigns to the integer variable \( t \) the time in milliseconds since the node was booted. It has a granularity of 1 ms.

On the host, the call \( t = \text{mclock}() \) assigns to the integer variable \( t \) the time in milliseconds used by the system to execute the program. It also has a granularity of 1 ms.

When timing a hypercube program, there are a few things you need to consider in order to get a meaningful time. First, the nodes are not started at the same instant. If you try to measure computation time by making calls to dclock or mclock at the beginning and end of the program, the time you measure on node A also includes the time it took to start up the nodes started after node A. To avoid this problem, it is important to synchronize the nodes before beginning the timing. One way to do this is to include the line `call gsync()`

to synchronize the nodes before the first timing call.

Second, timings on the host and nodes tell you different things about a program’s performance. Depending on where the timing calls are placed, the time recorded for the host program can include the time to load the node programs, communicate data to the nodes, as well as the time spent waiting for and receiving results from the nodes. In this way, it is a measure of how much time the programmer has to invest to see the result.

However, a time recorded on the host may not be the best way to evaluate a parallel algorithm. If, for example, you want to find out how the speedup
of an algorithm varies as you change the number of nodes, it may be more informative to time the algorithm on the nodes alone. In this case, the node program should proceed according to the following steps.

1. Receive needed information from host, if any.
2. Synchronize the nodes, e.g. call gsync().
3. \( t_1 = \text{dclock}() \) or \( t_1 = \text{mclock}() \).
4. Compute the results (using no communication with the host).
5. \( t_2 = \text{dclock}() \) or \( t_2 = \text{mclock}() \).
6. Determine elapsed time \( t = t_2 - t_1 \).
7. Report results to host as required.

The node times are almost always more important to performance measurement than are the host times.

### 8.6 A happy day on bud

Bud is old and worn. Following a few local rules will make your time on bud a much happier one. Please don’t forget!

1. If you are not able to rlogin in to bud, try telnetting before you give up.
2. **Do not log on if there are already four users logged in!** (Use finger @bud to check.)
3. Remember that every window you open on bud represents another login. If you have four windows open on bud, you are using up all of the logins.
4. If node processes do not exit, it is important to kill them using the command `killcube` issued on the host.
5. **Issue a relcube command to release any cube you are no longer using.**
6. Bud does not recognize an xterm window type. Work in one window about 24 lines long. Set the terminal type to be vt100 (setenv TERM vt100).

7. Remember that the host (bud) runs System V Unix. Some commands you use in BSD are not available.

8. Not all commands are recognized by bud. In particular,
   
   (a) ^C may or may not be recognized. If not, you need to log in again and kill offending the process (ps -a; kill -9 <PID>). Logout immediately.
   
   (b) If you are logged on through a DEC 5000 workstation, backspace does not work.

9. Do not use Intel global commands (like GDSUM) inside of loops.

10. **Files on bud are not backed up. Do not leave important files there.**
A Sample Satisfying Session

On your home machine:
1. edit your iPSC programs
2. ftp them to bud.

On bud:
1. compile
2. getcube
3. host
4. killcube (if host did not kill all node processes)
5. relcube
6. ftp results to your home machine.

Repeat until done.

8.7 Debugging tools

Running a simple example and printing out the intermediate results is often sufficient to debug hypercube programs. However, Intel also provides a concurrent debugger called DECON. More information on it is provided in [Intel 91b].

In addition, Intel provides a simulator of the iPSC/2 that runs on workstations. This simulator allows you to write and debug hypercube programs without actually using the hypercube multiprocessor. The simulator caused the DEC 5000 workstations in our lab to hang, so we do not support it locally. However, it can be a valuable tool to use on other workstation architectures. See [Intel 89] for more information. The simulator is easy to install and use. The source is available from your instructor.
References


