Hypercube Supercomputers

JOHN P. HAYES, FELLOW, IEEE AND TREVOR MUDGE, SENIOR MEMBER, IEEE

The architecture and applications of the class of highly parallel distributed-memory multiprocessors based on the hypercube interconnection structure are surveyed. The history of hypercube computers from their conceptual origins in the 1960s to the recent introduction of commercial machines is briefly reviewed. The properties of hypercube graphs relevant to their use in supercomputers are examined, including connectivity, routing, and embedding. The hardware and software characteristics of current hypercubes are discussed, emphasizing the unique aspects of their operating systems and programming languages. A sample C program is presented to illustrate the single-code multiple-data programming style typical of distributed-memory machines in general, and hypercubes in particular. Two contrasting hypercube applications are presented and analyzed: image processing and branch-and-bound optimization. The paper concludes with a discussion of current trends.

I. INTRODUCTION

Parallel processing seeks to improve the speed with which a computation can be done by breaking it into subparts and concurrently executing as many of these as possible. The past few years have seen the emergence of commercial computers that employ hundreds of processors working in parallel to achieve the level of performance previously found only in multimillion-dollar supercomputers [1]. In many of these "massively" parallel machines, the processors are connected in a regular pattern called a hypercube. By using hundreds of low-cost microprocessors, the cost of these unconventional multiprocessors can be kept relatively low, putting them within reach of the single user. At the same time, extremely high computing performance can be achieved. To emphasize these points, one manufacturer has called its hypercube product a "personal supercomputer" [2]. This paper explores the origins, architecture, and applications of hypercube-based multiprocessors with performance at the supercomputer level.

The parallel computers of interest here consist of many processors and memory units which communicate via an interconnection network. The latter can range from a single shared bus to a complex multistage interconnection network [3]. Of particular significance in determining system performance is the manner in which the processors communicate with the memory subsystem. Two major approaches are found in contemporary parallel computers. The *shared-memory* approach employs a single central memory unit to which all processors have direct and rapid access. Contention for this shared memory, however, can result in serious performance loss. An alternative approach is to provide each processor with a local memory to which other processors have slow and indirect access. Such a *distributed-memory* scheme simplifies the interconnection of massive numbers of processors, but raises new problems in communication efficiency.

Most recently introduced multiprocessors have a few dozen processors connected to a shared memory over a common high-speed bus. Examples are the Sequent Balance [4] and the Encore Multimax [5]. Another class of shared-memory multiprocessors are massively parallel machines that provide a connection from each processor to a large multiport shared memory. Examples are the BBN Butterfly [6], one of the few commercially available machines in this category, and the RP3, an experimental machine developed at IBM [7]. A key feature of these machines is the omega-type multistage interconnection network that connects the processors to the shared memory [8].

Massively parallel multiprocessors are typically of the distributed-memory variety to avoid the contention problems associated with hundreds or thousands of processors sharing a very large global memory. Communication among the processors, however, requires an efficient interconnection network. Many proposals for such networks have been made, including meshes, pyramids, and multistage networks of the type mentioned above. Given the large variety of these proposals, it is interesting to note that the overwhelming majority of current commercial massively parallel machines are hypercube-connected.

Distributed-memory multiprocessors such as hypercubes eliminate most of the access contention problems associated with a large shared memory. They do so by partitioning the system memory into smaller local memories that are distributed among the available processors. Communication among these local memories then becomes a major design issue, since a relatively slow input-output operation is needed to access shared data assigned to a nonlocal memory. Such accesses take the form of messages passed between the local memories of the two processors. The management of this message-passing has major implications on all aspects of the system design, as well as on

Manuscript received August 5, 1988; revised June 5, 1989. This work was supported by the Office of Naval Research under Contract N00014-85-K-0531, by DoD Contract MDA904-87-C-4136, and by NSF Contract MIP-8802771.

The authors are with the Advanced Computer Architecture Laboratory, Dept. of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122, USA.

IEEE Log Number 8933020.

0018-9219/89/1200-1829$01.00 © 1989 IEEE

applications software design. For example, a high-performance switching network or special interprocessor communication software (which usually is a part of the operating system resident in each node) is necessary. In addition, old algorithms must often be extensively restructured, as we will see later, to execute efficiently in the parallel environment created by these machines. This restructuring has also led to major extensions to traditional programming languages.

A hypercube is a generalization of the 3-dimensional cube graph to arbitrary numbers of dimensions. Just as a 3-dimensional cube has 2^3 nodes (vertices), so an n-dimensional cube has $N = 2^n$ nodes. Similarly, each node of a 3-cube has 3 edges (links) connected to it, and each node of an n-cube has n edges connected to it. Hypercube multiprocessors take this simple topology and use it to define the interconnection pattern among 2^n processors. Processors are placed at the nodes of the cube and are connected by links along the edges. Figure 1 illustrates hypercubes for small values of n.

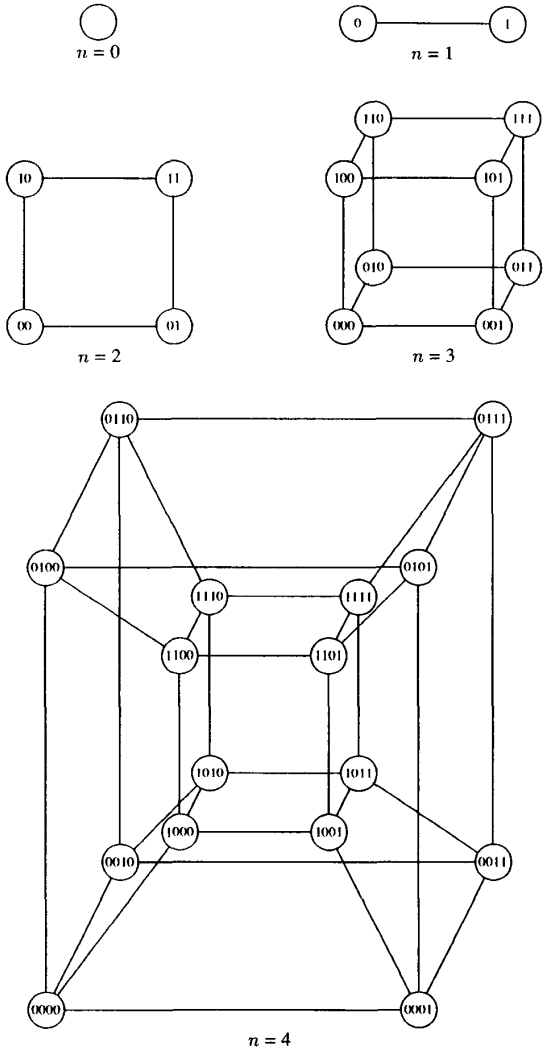

Fig. 1. Hypercubes for $n = 0, 1, 2, 3, 4$.

Hypercube topology has several attractive features. First, it is homogeneous or node-symmetric in the sense that the system appears the same from each node: There are no edges or boundaries where nodes may need to be treated as special cases. The hypercube achieves a good balance between the number of internode links used and their cost. It employs $nN/2$ links to connect $N = 2^n$ nodes, and each node processor has n links to manage. In all but a few recent hypercubes, nodes not directly connected must communicate via messages sent through intermediate nodes in store-and-forward fashion. The hypercube topology guarantees that no two nodes are more than n links apart. In addition, other useful computational structures, most notably meshes of arbitrary dimensions, can be embedded in a hypercube in such a way that adjacent nodes in the original structure are also adjacent in the hypercube. Thus for a wide class of useful applications, especially in scientific computing, the delays associated with interprocessor communication fall within acceptable limits. Another key factor in making hypercube computers practical from a commercial viewpoint is the fact that they can now be built economically with low-cost off-the-shelf microprocessor components [9]. Other proposals for massively parallel machines often require complex, expensive, and specially designed chips if they are to contain a reasonable number of integrated circuits (ICs). This is a more serious restriction than it might first appear, since a component count of more than a few tens of thousands of ICs puts air-cooled systems at the upper limits of acceptable reliability, regardless of the complexity of the subsystem within each IC.

The next section gives a brief history of hypercubes and outlines the main features of hypercube architectures. Section III discusses the structural properties of hypercubes and their influence on system design and application. Software design issues and a representative hypercube program are presented in Section IV. Section V examines two typical applications in depth: image processing and branch-and-bound optimization. Section VI concludes the survey with a brief discussion on current status and trends.

II. Hypercube Computers

The earliest study of hypercube computers was published by Squire and Palais of the University of Michigan in 1963. Their stated goal was to design a computer "where the emphasis is on the programmability of highly parallel numerical computations," with hardware cost a secondary consideration [10], [11]. Among the reasons they cite for selecting the cube organization are the ease with which paths between nonadjacent nodes can be determined, and the fact that all nodes are identical and interchangeable. The proposed 12-dimensional (4096-node) Squire-Palais machine was estimated to require 20 times the hardware of the IBM Stretch, the largest supercomputer of the day, but a speedup of at least 100 was anticipated.

With the advent of the single-chip microprocessor in the early 1970s, several other proposals for microprocessor-based hypercubes were made. In 1975 IMS Associates, a manufacturer of personal computers, announced a 256-node commercial hypercube based on the Intel 8080 microprocessor, but it was never produced [12]. In 1977, Sullivan and his colleagues at Columbia University presented a proposal for a large hypercube called the Columbia Homogeneous Parallel Processor (CHOPP), which would have

contained up to a million processors [13], [14]. Also in that year, Pease published an important study of the "indirect" binary *n*-cube architecture, for which he suggested a multistage interconnection network of the omega type for implementing the hypercube topology [15].

High hardware cost was clearly a major reason why these early hypercube designs were never implemented. The appearance of high-performance 32-bit microprocessor chips and dynamic RAM chips in the 1 M-bit range in the early 1980s made it economically feasible to construct practical hypercube computers of moderate size. The first such machine was the 64-node Cosmic Cube built by Seitz and his colleagues at Caltech, which became operational in 1983 [16]. This pioneering machine employed processor nodes based on the commercial Intel 8086/8087 microprocessor family. The Cosmic Cube was applied successfully to a variety of numerical computation tasks, often yielding significant speedups compared to conventional computers of similar cost. It was also the first of a series of experimental hypercube computers developed at Caltech [17], [18] and provided the main inspiration for the first generation of commercial hypercube computers.

In July 1985, Intel delivered the first production hypercube, the Intel Personal Supercomputer, or iPSC, which has a 80286/80287 CPU as its node processor and up to 128 nodes. Assuming a peak performance of 0.1 million floating-point operations per second (MFLOPS) per node, the 128-node iPSC has a potential or peak throughput of about 12 MFLOPS. (Note that a traditional vector supercomputer such as the Cray-1 has a peak throughput of 160 MFLOPS). Two other commercial hypercubes were introduced in 1985: NCUBE Corporation's NCUBE/ten and System 14/n from Ametek (subsequently Symult Systems). The System 14/n hypercube has up to 256 nodes, each employing an 80286/80287-based CPU similar to that of the iPSC, and an 80186 microprocessor for communication management. The NCUBE/ten can accommodate up to 1024 nodes, each based on a VAX-like 32-bit custom processor with a peak performance of around 0.4 MFLOPS. Thus, a fully configured NCUBE system has a peak throughput of about 400 MFLOPS. This high performance level is supported by extremely fast communication rates (both input/output and node-to-node), making the fully configured NCUBE/ten a true supercomputer. In 1988, researchers at Sandia National Laboratories using a 1024-node NCUBE/ten were the first to meet the widely publicized challenges posed by A. Karp and C. G. Bell to demonstrate the successful application of massive parallelism to large-scale practical problems in scientific computation [19]–[21]. This work, and that of many others, demonstrates convincingly that a properly programmed *N*-node hypercube can provide linear speedup—execution speeds that increase in proportion to *N*—for a wide range of computation problems.

Several new hypercubes with supercomputing performance have been built or announced since 1985, including the Caltech/JPL Mark III [18], the Floating Point Systems T Series [22], and the Intel iPSC/2 [23]. Some of these machines incorporate pipelined vector processors in their nodes, a feature of most earlier supercomputers. In some instances, they also employ special routing circuits to allow direct communication paths to be established between nonadjacent nodes. The second-generation Intel iPSC/2, for instance, has a vector-processing capability, as well as a circuit-switching internode communication network to replace the slower store-and-forward technique [24]. Peak performance figures in excess of 1000 MFLOPS (1 GFLOPS) are cited by the manufacturers of these newer hypercubes. The new communication hardware has reduced the time to pass a message between two nodes from a few milliseconds to a few microseconds. In fact, by effectively reducing the distance between all pairs of nodes to a small constant, a programmer can view the system as a pool of processors with a complete set of node-to-node connections. As a result, the requirement that the application algorithms have a hypercube-like communication structure is diminishing in importance. Furthermore, it raises the possibility of implementing a shared-memory architecture on a hypercube platform.

A few other recent supercomputers employ architectures that have been heavily influenced by the hypercube concept. The Connection Machine series manufactured by Thinking Machines Corporation employs up to 2^{16} or 65 536 simple processing nodes [25]. Sixteen nodes are placed on a chip with switching circuitry that allows any node to be directly connected to any other node on the chip. The 2^{12} chips of the Connection Machine form the nodes of a 12-dimensional hypercube. The second-generation, CM-2 model of the Connection Machine announced in 1987 has a peak performance target of 2.5 GFLOPS [25]. The Symult Systems 2010 introduced in 1988 has up to 256 nodes interconnected as a toroidal mesh which, as we will see later, is closely related to a hypercube [26].

The basic architecture of a hypercube node processor is shown in Fig. 2. It is a self-contained computer with a CPU,

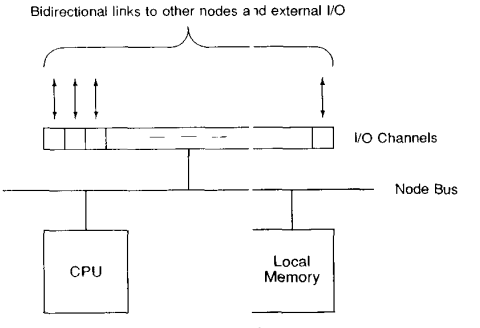

Fig. 2. Architecture of a typical node processor.

local memory for programs and data, and an input/output (I/O) subsystem. Its main distinguishing feature is the set of bidirectional I/O channels that link the node to its *n* immediate neighbors in the hypercube. These channels are used for interprocessor message passing and are typically implemented as bit-serial links with direct memory access (DMA) to the local memories of the nodes being linked. Processors not directly connected to one another by I/O links can communicate via intermediate nodes, which relay messages between the source and destination nodes. Additional links may be provided to connect each processor to a host computer and I/O devices such as secondary (disk) memories; in some instances all access to the I/O system is via the host computer. The host acts as a general system supervisor providing such operating system functions as

I/O management, as well as program editing and compilation facilities.

Communication issues play a central role in the architecture and performance of hypercubes and, indeed, of all distributed-memory computers. Hypercubes are typically configured so that the nodes execute the same application program on different data sets. Each node's program and data are stored in its local memory, so that most computation is within the individual nodes. When two nodes need to share information—for example, to exchange results—the shared data must be inserted in a message which is transmitted via a series of I/O transfers, possibly involving other nodes. The delay required for a node to obtain a data item in this fashion from one of its immediate neighbors is perhaps 1000 times the delay incurred when accessing data in the node's local memory. Thus great pains are taken to design hypercube application programs to minimize the need for message-passing, and to confine unavoidable message-passing to adjacent or nearly adjacent nodes.

The earliest hypercubes used store-and-forward communication schemes, which required of the order of 1 ms to transfer a message between adjacent nodes; k separate transfers or "hops" are necessary when source and destination are k links apart. Improvements in the efficiency of the message-passing software can reduce this delay by a factor of 10 or so [27]. However, to reduce the delay to a level comparable to the local memory access time (1 μs or less), hardware routing circuits have been devised. The Torus routing chip [28] and the Hyperswitch used in the JPL Mark III hypercube are examples of these [29].

The routing circuits proposed for hypercubes resemble crossbar switching networks and provide direct (circuit-switched) connections between arbitrary pairs of I/O channels associated with a single node. This permits a message to pass from source S to destination D without being stored at intermediate nodes. In effect, a direct high-speed circuit is established between S and D. Provided they do not contend for the same links, several separate circuits can pass through the same routing switch. A number of strategies have been devised for dealing with contention when it occurs. The "wormhole" approach of Dally and Seitz [28], versions of which are used in iPSC/2 and the Symult Systems 2010, allows a blocked message to retain control of the routing circuitry up to the blockage point: It is queued in the network until the blockage clears. Alternative "adaptive" strategies to the wormhole approach used in conjunction with the Hyperswitch attempt to find a free path around a blocked node [29]. Finally, there is a class of routing strategies that are adaptive only on their first hop as the message leaves S [30]. These reduce the likelihood of blockage and simplify deadlock avoidance, a concern with adaptive routing [31].

A hypercube computer is managed by an operating system (OS) which resides mainly in the host machine. The OS management functions peculiar to hypercubes include allocation of subcubes of nodes (smaller hypercubes formed by a subset of the available nodes) to multiple users, loading programs into the nodes (often done by a broadcasting operation), and managing processor–processor and processor–I/O communication. Communication functions such as message storing and forwarding may be assigned to the hypercube nodes in the form of a small node-resident OS kernel. It should be noted that node processes are inherently asynchronous, so that any necessary synchronization among processes in different nodes must be taken care of by the OS.

Because of the large numbers of nodes that may be present, packaging considerations are also very important in hypercube design in order to keep physical size, power consumption, and cooling needs within reasonable limits. The Connection Machine employs simple 1-bit processors and is therefore able to accommodate 16 node processors on a single custom IC. Thirty-two of these chips, their memories (4K bits per processor), and internode communication circuitry are placed in a single printed circuit board. Most commercial hypercube machines employ conventional 32-bit processors with much larger and expandable local memories. In a more typical case such as the Intel iPSC, the node consists of one or two small boards with several megabytes of local memory. An intermediate case represented by the NCUBE/ten has a 7-chip node comprising a custom 32-bit microprocessor chip and six memory chips with a combined storage capacity of 0.5 MB. Sixty-four nodes can be placed on a large (16 × 22-in.) board; however, this format does not allow for memory expansion. Figure 3 shows

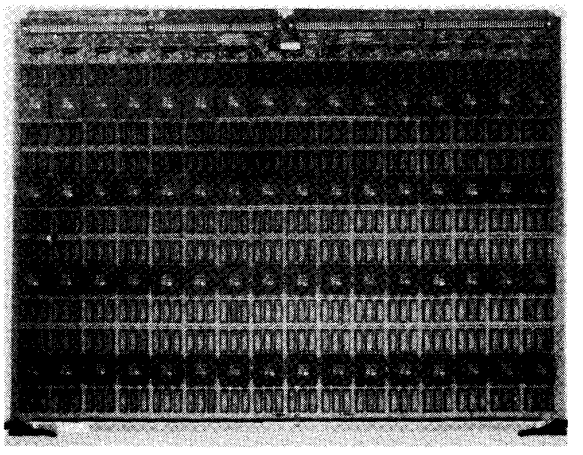

Fig. 3. 64-node NCUBE/ten processor board.

a 64-node NCUBE/ten processor board, which functions as a 6-dimensional hypercube.

III. Structural Properties

The graph properties of hypercubes that are relevant to their use in supercomputers [32], are examined next. An n-dimensional hypercube graph Q_n can be defined recursively as follows:

$$Q_1 = K_2$$

$$Q_n = K_2 \times Q_{n-1}$$

where K_2 is the 2-node complete graph, and $\times$ denotes the cartesian product of graphs. This definition implies that Q_n contains many subcubes of smaller dimensions, a property that may be exploited in several ways. For example, a hypercube Q_n can accommodate multiple users simultaneously by assigning each user a disjoint subcube Q_k within Q_n, where $k \leq n$. Furthermore, a measure of fault tolerance can

be achieved by assigning to users only those subcubes that exclude known faulty nodes.

Q_n contains $N = 2^n$ nodes of degree n, and $n2^{n-1}$ edges. Thus, as the number of nodes N is increased to improve performance, the connection requirements of each node increase at a rate proportional to $n = \log_2 N$. This implies that practical limitations on the number of links per node can be met while allowance is made for the increased communication needs of a larger hypercube. Each node of Q_n is at distance one from n other nodes. The maximum internode distance or diameter of Q_n is n, which therefore defines the worst-case communication delay. The average internode distance is $(n2^{n-1})/(2^n - 1)$, which rapidly approaches $n/2$ as n is increased.

The nodes of Q_n may easily be labeled with n-bit addresses in such a way that nodes adjacent in the ith dimension differ only in the ith address bit (see Fig. 1). The resulting set of 2^n addresses $\{00..0, 00..1, \cdots, 11..1\}$ facilitate the implementation of key communication algorithms, including node-to-node routing and broadcasting from one node to the entire hypercube. Two basic programs for this purpose appear in Fig. 4. A copy of each is assumed to reside in every node of the hypercube as part of its resident OS. The program is executed whenever a message is generated locally or is received from another node for routing or broadcasting. The same basic communication algorithms are also implemented in the circuit-switching hardware discussed in Section II.

Figure 4(a) gives the basic node-to-node routing algorithm ROUTE which always selects a minimum-length path

```
procedure ROUTE;
  begin
    for next message from s_{n-1} s_{n-2}...s_0 to d_{n-1} d_{n-2}...d_0 do
      begin
        x_{n-1} x_{n-2}...x_0 := (s_{n-1} ⊕ d_{n-1})(s_{n-2} ⊕ d_{n-2})...(s_0 ⊕ d_0);
        for i := 0 to n-1 do
          if x_i = 1 then begin
            send message to i-th neighbor; exit;
          end;
      end;
  end.
```

(a)

```
procedure BROADCAST;
  begin
    if the current node is the source then C := 11...1
      else receive message and control word C = c_{n-1}c_{n-2}...c_0;
    for i := 0 to n - 1 do
      if c_i = 1 then begin
        c_i := 0;
        send message and C to i-th neighbor;
      end;
  end.
```

(b)

Fig. 4. Hypercube communication algorithms. (a) Node-to-node routing. (b) Broadcasting.

between the source node $S = s_{n-1}s_{n-2} \cdots s_0$ and the destination node $D = d_{n-1}d_{n-2} \cdots d_0$. ROUTE first computes $X = S \oplus D$ where $\oplus$ denotes the bitwise EXCLUSIVE-OR operation. It then scans X in a fixed direction, say, left to right. If ROUTE encounters some $x_i = 1$, it transmits the current message to its immediate neighbor along the ith dimension of Q_n (its ith neighbor). If $X = 00..0$, then the current node must also be the destination, and the message is retained for processing. By transmitting the message to a node whose ith address bit is 1 whenever it encounters $x_i = 1$, ROUTE ensures that the ROUTE program in all subsequent nodes will find $x_i = 0$. Hence the message is always sent closer to the destination. It therefore travels the minimum possible distance from S to D, which is the number of ones in $S \oplus D$. An alternative routing algorithm developed by Valiant [33] routes each message to a randomly chosen node; from there the message is forwarded to its originally intended destination. The randomization assures that message congestion at nodes will be dispersed. Unfortunately, Valiant's router does not perform as well as the straightforward algorithm in many routine parallel processing tasks, and its more complex implementation requirements have discouraged its use.

A basic broadcasting algorithm, BROADCAST, is presented in Fig. 4(b). Assuming that each node can transmit the message to only one neighbor at a time, and that a single message transmission takes time τ, BROADCAST allows the message to be sent to all nodes in time $n\tau$, which is the minimum possible. A control word C is transmitted along with the message, and serves to tell each receiving node the dimensions along which it should retransmit the message. The first node S transmits the message and (and C) to a neighbor T in the first time period. In the second period, both S and T retransmit the message to two more nodes, and so on. Hence the number of copies of the messages being transmitted in successive time periods is 1, 2, 4, $\cdots$, 2^{n-1} so that all nodes are reached within n periods. Faster broadcasting can be achieved if a node transmits several copies of the message simultaneously.

Hypercubes have a very regular structure, which has several practical implications. As noted in Section I, they are homogeneous in the sense that the system structure looks the same from every node. In graph theoretic terms [34], Q_n is symmetric, meaning that every pair of nodes or lines can be interchanged without altering the graph structure. This property, combined with the fact that Q_n contains many easily identified subcubes of dimensions smaller than n, leads to the following conclusions.

1) A program can readily be designed to run unchanged on a hypercube of any dimension $k \geq 0$ by making k a parameter of the program. Thus program development can be conducted on a small subcube, e.g., one with $n = 2$, while production runs can be executed by a larger hypercube.

2) A large hypercube computer can be efficiently shared by multiple users, each of whom is assigned a disjoint subcube by the OS. Such a scheme is implemented by the AXIS operating system of the NCUBE/ten, which allows a user to specify the dimension k of a desired hypercube. AXIS then allocates a Q_k from among the available free nodes, if it can find one. Several efficient methods for handling arbitrary sets of subcube allocation requests have recently been developed [35]. The subcube Q_k of Q_n can be viewed as a logical entity which can be relocated anywhere in Q_n by EXCLUSIVE-ORing the address of each node in Q_k with the address of the node in Q_n chosen as the logical origin. Broadcasting, and message transfers in general, can be performed using the logical node addresses. This simplifies many message transfer algorithms.

Hypercubes have many attractive and useful embedding properties, some of which have been studied by graph theorists for more than 20 years [36]. An (isomorphic) embedding of G into G' is a one-to-one mapping ϕ of nodes of G onto nodes of G', such that if (u, v) is an edge of G, $(\phi(u),$

$\phi(v))$ is an edge of G'. G is termed cubical if it has an embedding in the hypercube Q_n, for some n. Among the useful graphs that are cubical are trees (cycle-free graphs) and meshes of any dimension. The latter result is especially important since many large-scale numerical problems have data structures defined on d-dimensional meshes. Their solution—for example, by relaxation methods—requires efficient communication between neighboring nodes on the mesh. Thus, to solve such mesh-oriented problems on a massively parallel computer, it is very desirable that meshes be isomorphically embeddable into the structure of the host computer. This is underscored by the fact that at least one hypercube manufacturer has also introduced a product with a mesh rather than a hypercube interconnection structure [26].

Figure 5 illustrates how a 2-dimensional 4×4 mesh can be embedded in Q_4; the labels assigned to the mesh nodes

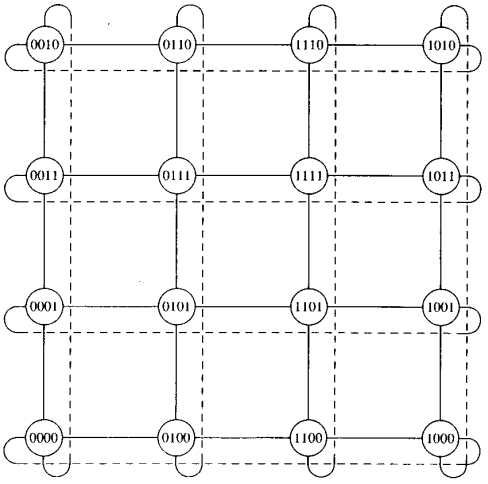

Fig. 5. A 4×4 mesh labeled for isomorphic embedding in Q_4.

correspond to the hypercube node labels in Fig. 1. Note that Q_4 can also accommodate a toroidal 4×4 mesh in which the edge nodes are connected in end-around fashion, as indicated by the dotted lines in Fig. 5. In general, a d-dimensional nontoroidal $k_1 \times k_2 \times \cdots \times k_d$ mesh M can be embedded in a hypercube Q_n of dimension $n \geq \Sigma_{i=1}^{d} \lceil \log_2 k_i \rceil$. If the dimension n of the available hypercube computer is too small, a mesh problem can often be partitioned efficiently into smaller mesh problems, each of which can be solved separately on the hypercube. An embedding of the $k_1 \times k_2 \times \cdots \times k_d$ mesh M into Q_n can easily be obtained by labeling the nodes of M so that the sequence of sublabels assigned to each dimension forms a Gray code [36]. For example, the nodes of the 4×4 mesh in Fig. 5 have labels $a_1a_0b_1b_0$ in which a_1a_0 and b_1b_0 assume the values $G = 00, 01, 11, 10$ along the vertical and horizontal dimensions of the mesh; G is a 2-bit reflected Gray code.

IV. Software

The emergence of the commercial hypercube computer has demonstrated the feasibility of constructing low-cost massively parallel machines. The focus of research can now be expected to shift to the issue of how these machines can be programmed effectively. Indeed, a recent study concludes that the lack of appropriate parallel programming languages and software development tools is the single biggest impediment to the widespread use of parallel machines [37]. The operating system is also a major factor, as memory management, interprocess communication and other OS functions are critical to overall system operation. Three major software issues must be considered: the operating system used for developing application programs; the runtime operating system in the hypercube nodes, and the set of application programming languages to be used.

High-order parallel programming languages and support tools are essential if users of parallel systems are to develop machine-independent concurrent software [38]. Such software will hasten the day when reusable software becomes a reality for parallel machines, as it now is for conventional uniprocessors. The programming of hypercubes is normally done by writing a separate program to run on each processor. The programs communicate by low-level message-passing operations provided by the OS and available to the programmer through extensions to a sequential language such as C or FORTRAN. Typically, these programs are copies of a single program. The distinct copies will execute correctly regardless of their location in the hypercube. This style of programming is referred to as single code multiple data (SCMD) or single program multiple data (SPMD).

Two major problems with the SCMD style of programming are the lack of type checking in internode communications and the machine dependence of the code. These problems can be solved by using a suitable parallel language, that is, one whose units of concurrency are distributed across the processors and executed simultaneously. To be effective, such languages should be able to perform type checking across processor boundaries, provide language constructs for interprocess communications that can also function across processor boundaries, allow data sharing between processes to be specified at the language level, and provide for synchronous creation and termination of processes within a program. Languages that meet these criteria are discussed in [39]-[42].

UNIX provides an attractive OS environment for software development on both sequential and parallel machines. As a result, it is supported (in several different versions) by most of the hypercube manufacturers. The NCUBE/ten, for example, has a UNIX-like operating system called AXIS [9]. It provides the normal UNIX utilities for editing, debugging, and resource management which treat most system resources as files. The NCUBE/ten incorporates up to eight I/O subsystems to meet the high I/O bandwidth requirements of a supercomputer. These are organized, under AXIS, as one distributed file system to avoid having to deal with multiple separate file systems. AXIS manages a hypercube as a device file that can be opened, closed, and so forth, as if it were a normal file. It permits users to allocate subcubes that have the approprite dimension for their application. Thus, one or two users with large problems or several users with small problems may share the hypercube. This flexibility greatly increases the system efficiency and gives a hypercube supercomputer a significant advantage over conventional supercomputers. Partitioning the main hypercube into subcubes is simplified in that each subcube is easily isolated logically from all other subcubes. A small OS nucleus

called VERTEX is resident in each of the NCUBE/ten nodes. Its primary function is to provide communication between the nodes. It achieves this by, among other facilities, send and receive operations that transfer messages between any two nodes in the hypercube, using the ROUTE algorithm of Fig. 4(a), and by a "whoami" operation that allows a program to determine which logical node it is executing on and which I/O processor it is connected to.

We conclude this section with a sample SCMD program that calculates the maximum element of a vector of numbers v. More formally, the program calculates,

$$S = \max_{1 \le i \ge K} v[i]$$

where $K = n \cdot 2^{dim}$, n is the number of elements of v in each node of the cube, and dim is the dimension of the cube. A listing of the program, which is written in a parallel extension of the C language, appears in Fig. 6. This particular programming language is NCUBE's version of C, but other hypercube manufacturers use similar extensions to C (cf. Intel [43]). The extensions to C include internode send and receive operations implemented as function calls nwrite and nread, respectively; and the whoami operation implemented by the function called whoami. These functions are part of the OS resident in the hypercube nodes. We assume that a copy of this program has been loaded into each of the nodes of a particular subcube that has been allo-

```
001     /*  NODE PROGRAM TO CALCULATE THE MAXIMUM OF A VECTOR v
002
003     pn      :   caller's logical processor number in subcube
004     proc    :   process number in node
005     host    :   node on Host for cube communication
006     dim     :   dimension of allocated cube                    */
007
008
009     whoami(&pn,&proc,&host,&dim);
010
011
012     /* Receive vector of length n from the host; VECLEN is the
013       possible length of the vector. (4 bytes per vector element) */
014
015     cf = 0;
016     type = DATA;
017     n = (nread((char *)v,VECLEN*4,&host,&type,&cf)/4);
018
019     /* Find m, the local maximum of v in this node */
020
021     m = MINF;
022     for (i = 0 ; i < n ; ++i) if (v[i] > m) m = v[i];
023
024     for (i = dim ; i > 0 ; --i) {
025
026     /* Execute once for each axis of the hypercube */
027
028         if (pn < power(2,i)) {
029
030         /* If this node is in the active part of the collapsed cube,
031           do the computation below, otherwise the node is done.
032           npn is the neighbor of pn on the i-th axis                */
033
034            type = MAX;
035            npn = pn^power(2,(i-1));
036            if (npn < pn)
037
038            /* If neighbor's number is less, send the local maximum; otherwise
039               receive it and update its value.                                  /*
040
041                   nwrite((char *)&m,4,npn,type,&cf);
042            else {
043                   nread((char *)&rm,4,&npn,&type,&cf);
044                   if (rm > m) m = rm;
045            }
046         }
047     }
048
049     /* send the final result back to the host */
050
051     if (pn == 0) {
052        type = RESULT;
053        nwrite((char *)&m,4,host,type,&cf);
054     }
```

Fig. 6. SCMD program to find the maximum element of a vector.

cated to this particular job. Execution of the program may be summarized as follows: It reads in equal numbers of the elements of v into each node, forms local maxima of these numbers, and then selects, in turn, the largest of these local maxima along successive dimensions of the hypercube. The selection process collapses the active part of the computation into smaller and smaller cubes.

The call to whoami on line 009 returns the calling program's logical node number (pn), the node on the host board used for I/O communications (host), and the order of the allocated subcube (dim). The call to nread on line 017 reads in an n-element slice of the vector v from the host. The for loop on line 022 finds the maximum element of the slice of v in the node (m is initially set to $-\infty$ on line 021). This is done in parallel in each node. During this phase of the computation, all 2^{dim} nodes are doing useful work and the utilization of the allocated cube approaches 100%. $2^{\text{dim}-1}$ new maxima are formed by comparing pairs of local maxima in nodes that are immediate neighbors in the dimth dimension. The new maxima are now confined to a (dim − 1)-dimensional hypercube; in effect, the active cube is collapsed to half its initial size. This process of collapsing the active cube by half and selecting a new smaller set of maxima is repeated until the maximum of all the elements of v appear in logical node 0. The selection of the maximum among the nodes simulates a tree of comparators. Figure 7 illustrates this for a 3-cube.

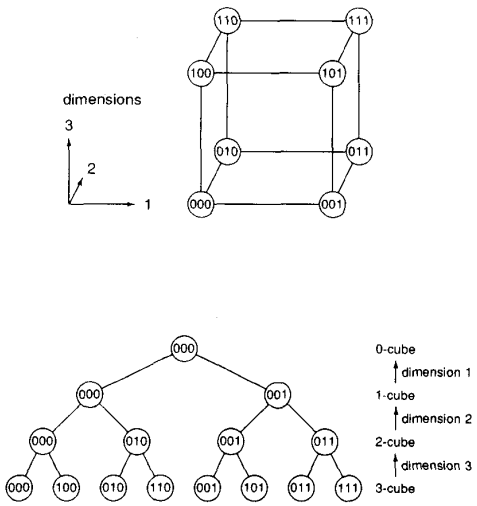

Fig. 7. Comparator tree to find the maximum.

The for loop that begins on line 024 starts at the highest dimension (dim) and finds the larger of each pair of local maxima in nodes that are adjacent in the highest dimension. It repeats this, stepping through all the dimensions. Line 028 selects the nodes (pns) in the part of the hypercube that remains active after collapsing along the (i + 1)st dimension. The neighboring nodes (npns) of the pns are those whose addresses differ in the ith bit position. Their addresses are calculated in the line 035 by performing the EXCLUSIVE-OR operation (^) on pn and $2^{(i-1)}$. Line 036 partitions the active nodes into two sets: those that are to receive local maxima (line 043) and those that send them (line 041). Line 044 finds the larger of the received value and the local maxima already present in the node. Those nodes that send will not be active in the next iteration of the loop. Line 053 transmits the result from node 0 to the host.

V. Applications

Figure 8 shows a representative list of applications of hypercube computers. This is by no means an exhaustive list, but it does illustrate the wide range of applications to

- Mathematics
  - Computational geometry
  - Prime number generation
- Physics
  - Particle transport
  - Lattice gauge theory
  - Molecular dynamics simulation
- Chemistry
  - Polymer simulation
  - Chemical reaction dynamics
- Geology
  - Seismic data processing
- Operations research
  - Resource allocation
- Artificial intelligence
  - Game playing
  - Generalized search
- Computer-aided design for VLSI
  - Placement, layout and routing
  - Circuit simulation
- Image processing and computer vision
  - Image restoration
  - Image encoding/decoding
  - Object recognition
- Other
  - Airfoil simulation
  - Electromagnetic scattering
  - Robot arm control algorithms
  - Databases and file systems
  - Sorting

Fig. 8. List of applications.

which hypercubes are being applied. For a detailed picture of these applications, the reader is referred to [44], [45], the book by Fox *et al.* [46], and the book by Reed and Fujimoto [47]. Although the number of applications has grown rapidly, they are predominantly in the area of scientific computing in which the behavior of a physical system is being analyzed. For the majority of these applications, the parallelism can be determined at compile-time and depends on a simple partitioning of the problem domain which is often a physical space. For example, in the particle transport problems of photons in a fusion plasma [48], the underlying algorithm is a Monte Carlo method which divides physical space into equal subregions, then simulates particle behavior for each space independently, and finally averages the results obtained from the separate Monte Carlo experiments. Similarly, in the case of many image processing algorithms, as we will see below, the image is partitioned into subimages of equal size that are assigned to separate processors.

In contrast to the applications with compile-time parallelism, hypercubes are also beginning to be used in applications where the degree of parallelism cannot be determined before the program is run, and where, consequently, load balancing of work among the processors at run-time becomes an issue. Examples from Fig. 8 are database applications, where addition and deletion of records has the potential to cause some processors to be underutilized, and resource allocation algorithms, which are usually solved

using branch-and-bound algorithms, as will also be examined further. These algorithms are particularly "dynamic" in their behavior because they spawn work in an unpredictable fashion during their execution.

In the remainder of this section we review in detail two applications that typify both ends of the spectrum: obvious compile-time parallelism and dynamic or run-time parallelism. The first is an image processing application which has a large degree of natural parallelism. By making use of the embedding ideas of Section III, it can be implemented on a hypercube in a fairly straightforward fashion. The second is the 0-1 integer linear programming (ILP) problem. This is one of the simplest examples of a large class of algorithms called branch-and-bound methods, which are used in artificial intelligence and operations research. Unlike the majority of applications of hypercubes, in which the work of each node can be scheduled *a priori* by the programmer, branch-and-bound programs schedule processes dynamically at run-time. At first sight, this would seem to make them inappropriate for hypercubes because of the communications overhead associated with dynamic scheduling. Like the implementation of chess on a hypercube reported in [49], the branch-and-bound application shows that many problems hitherto considered unparallelizable have, in fact, a substantial content of exploitable parallelism.

A. Image Processing

The term image processing covers an important class of techniques that include the encoding/decoding of images for transmission, the enhancement and restoration of noisy images, the extraction of features such as edges, and the segmentation of images for the purposes of image understanding [50]. An image is a 2-dimensional mesh of elements (pixels) that can take on a finite number of values (typically 256). These values, or gray-levels, represent the light intensity at each point in the image.

A widely used image-processing technique is to convolve the image with a finite impulse response (FIR) function. Depending on the particular FIR function, this operation can be used for edge detection, template matching, noise removal, and general filtering. The FIR function is defined as an $m \times m$ matrix $K(\alpha, \beta)$ of constant coefficients referred to as the kernel. The kernel is moved across the image in one-pixel steps to implement the convolution function. At each step the pixel $P_{in}(i, j)$ coinciding with the center of the kernel is replaced by $P_{out}(i, j)$, such that,

$$P_{out}(i, j) = \sum_{\alpha=-\lfloor m/2 \rfloor}^{\alpha=\lfloor m/2 \rfloor} \sum_{\beta=-\lfloor m/2 \rfloor}^{\beta=\lfloor m/2 \rfloor} P_{in}(i, j)\, K(\alpha, \beta)$$

A typical example of an image-processing algorithm involving the convolution of the image with FIR functions is the Sobel edge detection algorithm [51]. The image is convolved with the each of the two FIR kernels shown in Fig. 9 (an integer approximation to the exact kernels is shown). The results of the convolution are the two images e_x and e_y where $e_x(i, j)$ is an $M \times M$ array of x-direction edge (gradient) strengths, and $e_y(i, j)$ is an $M \times M$ mesh of y-direction edge (gradient) strengths. These two arrays are then combined to form a combined edge strength array, E, and an edge direction array, Θ where

$$E(i, j) = \sqrt{e_x^2 + e_y^2}$$

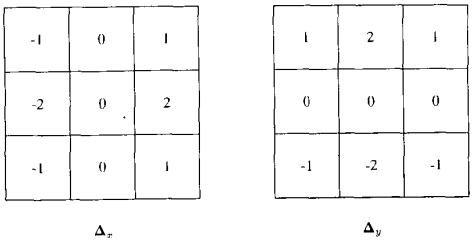

Fig. 9. The Sobel edge detector kernels.

and

$$\Theta(i, j) = \tan^{-1}\left(\frac{e_x}{e_y}\right) - \frac{\pi}{2}.$$

It can be seen from these equations that equal-size areas of the image require equal amounts of processing. Therefore, the natural approach to executing these algorithms on hypercubes is to partition the image into subimages of equal size and assign each subimage to a separate node processor. The subimages can be processed in parallel, using the mesh embedding technique of Section III to map adjacent subimages to adjacent nodes of the hypercube. Figure 10

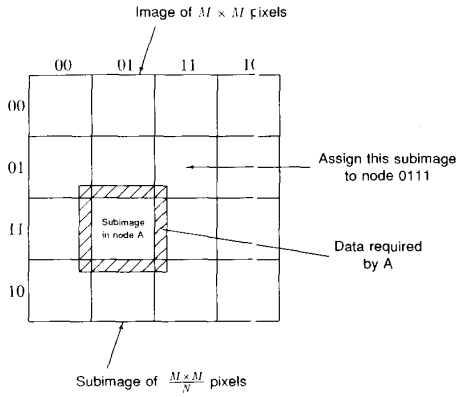

Fig. 10. Partitioning the image.

illustrates this for an image of $M \times M$ pixels and a 4-cube; here the image is partitioned into 16 equal subimages.

In general, convolving with an FIR function is implemented in the SCMD mode of Section IV. The only potential contributor to inefficiency is the communication overhead that results from the need to exchange data around the edges of each subimage to allow the edge pixels to be convolved with the kernel. This is also illustrated in Fig. 10, which shows a subimage in processor A and the data (shaded) that have to be moved from adjacent processors. The number of pixels that has to be transferred is roughly $2Mm/\sqrt{N}$ if n is even, and $3Mm/\sqrt{2N}$ if n is odd. (Recall that $m \times m$ is the size of the kernel, $M \times M/N$ is the size of the subimage, and $N = 2^n$ is the number of processors). The communication time necessary to move the pixels is proportional to their number. However, as we have seen, communications are often performed as DMA operations and can be completely overlapped with processing. Of course, for large kernels and small subimages a point can be reached where overlap is impossible and communication times start

to dominate. This highlights the importance of having sufficiently large problems for a particular size of hypercube if its efficiency is to be maintained [52]. Results reported in [53] show that convolving with a simple FIR kernel can easily be performed at video frame rates (thirty 512 × 512 images per second).

It can be shown that if the discrete Fourier transform and its inverse are implemented by the FFT algorithm and if m is greater than about 10 [54], then it is more efficient to perform convolution in the frequency domain for a 512 × 512 image. The hypercube network is well suited to efficient implementation of the FFT, with communication occurring between pairs of adjacent nodes [55]. The data layout shown in Fig. 10 is also appropriate for calculating the FFTs, and for computing their product in the frequency domain. The FFT and its inverse require data to be communicated between subcubes, that is, between adjacent regions in Fig. 10. These, of course, are in adjacent processors.

B. Branch-and-Bound Algorithms

Our second representative application area for hypercubes concerns problems for which there exists no computationally efficient "direct" solution. A solution is often found via a heuristic search through a large solution space. Unguided search, however, can easily become inefficient as many of these problems are at least NP-complete. Several techniques have been developed to guide the search and improve its average efficiency. The most general of these techniques is the branch-and-bound (B&B) algorithm [56], which has been used to solve some well-known problems, including the traveling salesman problem [57], the knapsack problem [58], and many of the heuristic search algorithms in artificial intelligence such as A*, AO*, and alpha-beta [59].

The branching action of a B&B algorithm is performed by building a search tree, called a B&B tree, over the problem space of interest. The root of the B&B tree represents the complete problem space, and children nodes represent subspaces. The branching process proceeds from the root to the leaves of the tree, systematically partitioning subspaces into smaller ones. The leaf nodes represent subspaces that are small enough to be exhaustively searched for solutions. A subproblem P_i can be characterized by the value of an objective function f, which is defined as the value of the best solution that can be obtained from P_i. This value is not known, however, until the subtree rooted at P_i is completely expanded. Instead another function h, referred to as the lower bound function, is used as an estimate of f. In general, h is a heuristic function that is much easier to compute than f.

A B&B algorithm consists of four major procedures: 1) selection, 2) branching, 3) elimination, and 4) termination test. The selection procedure selects a subproblem from the set of subproblems that have been generated but not yet examined (the active subproblems). The selection is performed according to the heuristic selection function h which determines the order in which the subproblems are selected for expansion. A commonly used heuristic is *best-first* search, in which h is a lower bound estimate of the objective function f. Subproblems with smaller lower bounds are selected first. The branching procedure examines the currently selected subproblem and uses problem-specific methods to break it into smaller-sized subproblems. The elimination step examines these newly created subproblems and deletes the ones that cannot lead to better solutions than those already found. To accomplish this, a special subproblem referred to as the incumbent is used to store the best feasible solution discovered during the search. A subproblem is deleted if its lower bound is greater than or equal to that of the incumbent. Finally, the termination test procedure eliminates a new subproblem that cannot lead to feasible solutions. Again, problem-specific techniques are used to accomplish this.

We now describe a specific problem that uses the B&B algorithm, viz., the 0-1 ILP problem. This is an optimization problem in which it is desired to minimize the value of a linear objective function $f(x_1, x_2, \cdots, x_n)$ subject to a set of constraints. The variables $(x_1, x_2, \cdots, x_n)$, can take only the values 0 or 1. The problem can be more formally stated as follows:

$$\text{Minimize} \quad f = \sum_{j=1}^{n} c_j x_j$$

subject to the constraints

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i \qquad i = 1, 2, \cdots, m$$

$$x_j \in \{0, 1\} \qquad j = 1, 2, \cdots, n$$

It can be assumed, with no loss of generality, that the coefficients c_j, $j = 1, 2, \cdots, n$ are nonnegative. The solution method involves systematically assigning zeros and ones to some of the x_j variables to obtain subproblems. A subproblem which has the smallest lower bound is selected from the list of active subproblems. An unassigned variable is picked and is assigned the values 0 and 1 to create two new subproblems. Each subproblem is evaluated and, if it represents a feasible solution and its lower bound is less than that of the incumbent, then it becomes the new incumbent. Furthermore, all subproblems on the list with lower bounds greater than the new incumbent are deleted from the list. If the subproblem cannot lead to a feasible solution it is deleted. Finally, the subproblem is inserted back on the list if it is not presently feasible and its lower bound is less than that of the incumbent. The algorithm continues by selecting another subproblem from the list. The algorithm terminates when the list becomes empty.

We consider two parallel implementations of the foregoing B&B algorithm on hypercube multiprocessors. The first implementation, referred to as the Central List (CL) algorithm, consists of two major components: a master process which runs on the host and N slave processes which run on the nodes of the hypercube. The master process maintains the list of active subproblems and the incumbent, selects N subproblems from the list, and assigns one subproblem to each slave process. The N subproblems selected have the best bounds among the active subproblems. Each slave process then expands its subproblem, generates children subproblems and calculates their lower bounds. It also performs the lower bound, feasibility, and termination tests on the subproblems it generates. The results are then sent back to the master process, which

inserts them on the list. The algorithm terminates when the list of active subproblems becomes empty and all the slave processes are idle.

The CL algorithm has the advantage of expanding subproblems whose bounds are the best globally. This is advantageous because subproblems that have smaller lower bounds are more likely to lead to solutions than others that have larger lower bounds. The algorithm, however, has some serious disadvantages. It requires two communication messages for each subproblem expansion. The first is required to send the subproblem from the host to the node for expansion. The second is needed to carry the newly created subproblems from the node to the host. Communication with the host becomes a bottleneck that reduces the performance of the algorithm.

The second B&B implementation, known as the Distributed List (DL) algorithm, attempts to put the resources of the hypercube to better use that the CL algorithm by distributing the list of active subproblems and a copy of the incumbent across the processing nodes. It employs $N + 1$ processes, each maintaining its own subset of the list. A supervisor process initiates the computation by generating N subproblems and assigning one to each of the remaining N processes. Each process then expands subproblems from its local list. It also performs the lower bound test, the feasibility and termination tests, and inserts the results back on its local list.

In our implementation on an NCUBE/ten [60], [61] the host runs the supervisor process while each of the N processing nodes run one of the other processes. A mechanism is employed by which the load can be balanced and the subproblems distributed across the processes. When a process becomes idle it requests subproblems from neighboring processes in the system. The process that receives the request examines its own list of active subproblems and either sends a portion of it to the requesting process or denies the request if its own list is too small to divide. In our implementation, a processor requests subproblems from one of its neighbors in the hypercube. It sends one half of its subproblems to an idle processor requesting subproblems.

Because the DL algorithm maintains multiple copies of the incumbent, processes can find feasible solutions independently and update their own incumbents. In the DL algorithm, once an incumbent is updated, its new value is broadcasted to all other processes. Figure 11 shows the speedup measured for the two algorithms for various hypercube sizes. In the CL algorithm, the speedup is reasonable for up to 16 processors. Little is gained by increasing the number of processors beyond that. This can be attributed to host-to-node communication overhead which increases as the cube size increases, and to load imbalance resulting from communication delays. The performance of the DL algorithms shows that a distributed-list approach has better performance than the CL algorithm. This is expected since there is no bottleneck in communication; the communication bandwidth of the hypercube is utilized more efficiently. The performance of the two algorithms on a 64-process hypercube is compared to the performance of the corresponding serial algorithm on the VAX 11/780 and the IBM 3090 (single processor) in Fig. 12.

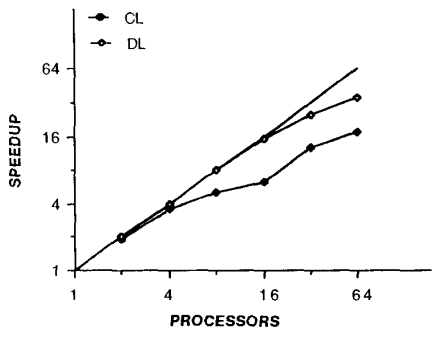

Fig. 11. Speedup of the CL and DL algorithms on the NCUBE/ten.

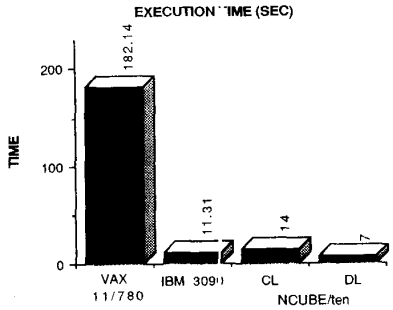

Fig. 12. Execution time for various systems.

VI. Discussion

As we have seen, hypercube multiprocessors are the realization of a concept that has been studied from a theoretical viewpoint for nearly 30 years. They represent one of the first applications of massive parallelism to commercial computers. Most of the current hypercubes can attain peak performance levels approaching those of traditional vector supercomputers. Success in reaching these levels for important practical applications has demonstrated not only the viability of hypercube supercomputers, but also the feasibility of massively parallel distributed-memory computers, in general. In particular, the assumptions underlying Amdahl's law[1] which places severe limits on the achievable speedup due to parallelism, are now seen as not applying to hypercube-class machines as they do to conventional vector architectures [52].

Nevertheless, several factors still make it difficult to achieve supercomputing performance with current hypercubes, including the small memory capacity and I/O bandwidth available in many of these machines. Most important, however, is the different style of programming required for hypercubes and other distributed-memory machines. It is not possible now to take an old sequential program (a so-called "dusty deck") and execute it directly on a hypercube computer. Such programs must be restructured, often

[1]Amdahl's law states that the speedup S of an n-processor system is $n/(1 + (n - 1)f)$, where f denotes the fraction of nonparallelizible operations. Thus, no matter how large n becomes, S can never exceed $1/f$.

extensively, in order to achieve reasonable speedups. There are presently no "parallelizing" compilers or the like for automatic program restructuring, comparable to the vectorizing tools available for pipelined supercomputers. The design of automatic parallelizers for hypercubes, now in the early stages of research, is likely to provide a major impetus to the use of hypercube computers outside the scientific and research community, which accounts for most current hypercube usage. In addition, more user-friendly program development environments, standards for parallel programming languages and operating systems, and shareable software libraries are all likely to have a major positive influence on the use of these machines.

The rapid technological developments in VLSI that made hypercube computers feasible in the first place can be expected to continue to reshape these machines and lead to further improvements in their performance/cost ratio. New IC technologies will undoubtedly allow more powerful processors, larger memories, and more sophisticated interconnection techniques to be incorporated into future hypercubes. The most profound changes in the architecture of these machines seem likely to occur in their interconnection technology. The introduction of fast node-to-node routing circuits makes a hypercube computer seem to a programmer like a completely connected system in which each node is directly connected to all others, i.e., all nodes are neighbors. In such an environment, essentially any application graph can be embedded efficiently into the computer provided a sufficient number of nodes are available. This development is likely to expand the range of applications that can use these machines and to simplify their programming. If this occurs, then the hypercube will appear as merely the internal skeleton of an extremely general and flexible computer of essentially unlimited potential.

References

[1] K. Hwang, "Advanced parallel processing with supercomputer architectures," in *Proc. IEEE*, pp. 1348–1379, Oct. 1987.

[2] Intel Scientific Computers. *iPSC System Overview*. Beaverton, Oregon, 1986.

[3] P. J. Denning, "Parallel computing and its evolution," *Communications of the ACM*, vol. 29, pp. 1163–1167, Dec. 1986.

[4] Sequent Computer Systems, Inc. *Balance Technology Summary*. Beaverton, OR 97006-6063, 1984.

[5] Encore Computer Corporation. *Multimax Technical Summary*, rev. ed. Marlboro, MA, May 1985.

[6] W. Crowther *et al.*, "Performance measurements on a 128-node Butterfly parallel processor," in *Proc. 1985 Int. Conf. on Parallel Processing*, pp. 531–540, Aug. 1985.

[7] G. F. Pfister *et al.*, "The IBM research parallel processor prototype (RP3): introduction and architecture," in *Proc. 1985 Int. Conf. on Parallel Processing*, pp. 764–771, Aug. 1985.

[8] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*. Lexington, MA: Lexington Books, 1985.

[9] J. P. Hayes, T. N. Mudge, Q. F. Stout, S. Colley, and J. Palmer. "A microprocessor-based hypercube supercomputer," *IEEE MICRO*, pp. 6–17, Oct. 1986.

[10] J. S. Squire and S. M. Palais. "Physical and logical design of a highly parallel computer," Department of Electrical Engineering, University of Michigan, Ann Arbor, MI, Technical Report, Oct. 1962.

[11] —, "Programming and design considerations for a highly parallel computer," in *Proc. Spring Joint Computer Conf.*, pp. 395–400, 1963.

[12] W. Millard, "Hyperdimensional μP collection seen functioning as mainframe," *Digital Design*, vol. 5, Nov. 1975.

[13] H. Sullivan and T. R. Bashkow, "A large scale, homogeneous, fully distributed parallel machine I, in *Proc. Computer Architecture Symp.*, pp. 105–117, 1977.

[14] H. Sullivan, T. R. Bashkow, and D. Klappholz, "A large scale, homogeneous, fully distributed parallel machine II, in *Proc. Computer Architecture Symp.*, pp. 118–124, 1977.

[15] M. C. Pease, "The indirect binary *n*-cube microprocessor array," *IEEE Trans. Computers*, vol. C-26, pp. 458–473, May 1977.

[16] C. L. Seitz, "The Cosmic Cube," *Communications ACM*, vol. 28, pp. 22–33, Jan. 1985.

[17] J. Tuazon, J. Peterson, M. Pniel, and D. Liderman, "Caltech/JPL Mark II hypercube concurrent processor," in *Proc. 1985 Int. Conf. on Parallel Processing*, pp. 666–671, Aug. 1985.

[18] J. Peterson, J. Tuazon, D. Liderman, and M. Pniel, "The Mark III hypercube-ensemble concurrent computer," in *Proc. 1985 Int. Conf. on Parallel Processing*, pp. 71–73, Aug. 1985.

[19] A. Karp, "What price multiplicity?" *Communications ACM*, vol. 30, pp. 7–9, Jan. 1986.

[20] G. R. Montry, J. L. Gustafson, and R. E. Benner, "Development of parallel methods for a 1024-processor hypercube," *SIAM J. Scientific and Statistical Computing*, vol. 9, pp. 1–32, July 1988.

[21] J. Dongarra, A. Karp, and K. Kennedy, "Winners achieve speedup of 400," *IEEE Software*, pp. 1–5, May 1988.

[22] J. Gustavson *et al.*, "The architecture of a homogeneous multiprocessor," in *Proc. 1986 Int. Conf. on Parallel Processing*, pp. 649–652, Aug. 1986.

[23] Intel Scientific Computers. *Intel iPSC/2*. Beaverton, OR 97006, 1988.

[24] T. H. Dunigan. *Performance of a second-generation hypercube*. Oak Ridge National Lab., Oak Ridge, TN, Technical Report ORNL/TM-10881, Nov. 1988.

[25] Thinking Machines Corp., *Connection Machine Model CM-2 Technical Summary*. Technical Report HA87-4, April 1987.

[26] Ametek Corp. *Series 2010*. Monrovia, CA, 1988.

[27] T. N. Mudge, G. D. Buzzard, and T. S. Abdel-Rahman, "A high performance operating system for the NCUBE," in *Hypercube Multiprocessors 1987*, M. T. Heath, ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, pp. 90–99, 1987.

[28] W. J. Dally and C. L. Seitz, "The torus routing chip," *Distributed Computing*, vol. 1, pp. 187–196, 1986.

[29] E. Chow *et al.*, "Hyperswitch network for the hypercube computer," in *15th Ann. Int. Symp. on Computer Architecture*, pp. 90–99, May 1988.

[30] G. D. Buzzard, "High Performance Communications for Hypercube Multiprocessors." Ph.D. thesis, University of Michigan, 1988.

[31] W. J. Dally and C. L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Trans. Computers*, vol. C-36, pp. 547–553, May 1987.

[32] F. Harary, J. P. Hayes, and H. J. Wu, "A survey of the theory of hypercube graphs," *Comput. Math. Applic.*, vol. 15, pp. 277–289, 1988.

[33] L. G. Valiant, "A scheme for parallel communication," *SIAM J. Computing*, vol. 11, pp. 350–361, May 1982.

[34] F. Harary. *Graph Theory*. Reading, MA: Addison-Wesley, 1969.

[35] S. Dutt and J. P. Hayes, "On allocating subcubes in a hypercube multiprocessor," in *Proc. 3rd Int. Conf. on Hypercube Concurrent Computers & Applications*, vol. I, pp. 801–810, Jan. 1989.

[36] M. Livingston and Q. F. Stout, "Embeddings in hypercubes," *Math. Comp. Modelling*, vol. 11, pp. 222–227, 1988.

[37] D. A. Buell *et al.*, "Parallel algorithms and architectures: Report of a workshop," *J. Supercomputing*, pp. 301–325, 1988.

[38] ParaSoft Corp. *Programming Parallel Computers Using the EXPRESS System*, Mission Viejo, CA 92692, 1989.

[39] R. M. Clapp and T. N. Mudge, "ADA on a hypercube," in *Proc. 3rd Int. Conf. on Hypercube Concurrent Computers & Applications, vol. I*, pp. 399–408, Jan. 1989.

[40] D. Gelernter, "Generative communication in Linda," *ACM Trans. Prog. Lang. Syst.*, vol. 7, pp. 80–112, Jan. 1985.

[41] R. Pountain, *A Tutorial Introduction to Occam Programming*. Inmos Corp., Colorado Springs, CO, 1983.
[42] R. H. Perrott, *Parallel Programming*. Woking, England: Addison-Wesley, 1987.
[43] C. Moler and D. S. Scott. *Communication Utilities for the iPSC*. iPSC Technical Report 2, Intel Corp., Aug. 1986.
[44] *Hypercube Multiprocessors 1986*, M. T. Heath, ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1986.
[45] *Hypercube Multiprocessors 1987*. M. T. Heath, ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1987.
[46] G. Fox *et al.*, *Solving Problems on Concurrent Processors*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
[47] D. A. Reed and R. M. Fujimoto, *Multicomputer Networks: Message-Based Parallel Processing*. Cambridge, MA: The MIT Press, 1987.
[48] W. R. Martin, T. C. Wan, T. S. Abdel-Rahman, and T. N. Mudge, "Monte Carlo photon transport on shared memory and distributed memory parallel processors," *Int. J. Supercomputer App.*, vol. 1, pp. 57–74, Fall 1987.
[49] E. W. Felten and S. W. Otto, "Chess on a hypercube," in *Proc. 3rd Int. Conf. on Hypercube Concurrent Computers & Applications, Vol. II*, pp. 1329–1341, Jan. 1989.
[50] A. Rosenfeld and A. C. Kak, *Digital Picture Processing*. New York: Academic Press, 1976.
[51] J. M. S. Prewitt, *Object enhancement and extraction*. New York: Academic Press, 1970.
[52] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications ACM*, vol. 31, pp. 532–533, 1988.
[53] T. N. Mudge and T. S. Abdel-Rahman, "Vision algorithms for hypercube machines," *J. Parallel and Dist. Computing*, vol. 4, pp. 79–94, 1987.
[54] ——, *Specialized Computer Architecture for Robotics and Automation*, chapter Architecture for robot vision. New York, NY: Gordon and Breach Science Publishers, 1987, pp. 103–149.
[55] P. N. Swarztrauber, *Multiprocessor FFTs*. National Center for Atmospheric Research, Boulder, CO, 1986.
[56] E. L. Lawler and D. W. Wood, "Branch-and-bound methods: A survey," *Ops. Res.*, vol. 14, pp. 699–719, 1966.
[57] D. W. Sweeney, J. D. C. Little, K. G. Murty, and C. Karel, "An algorithm for the traveling salesman problem," *Ops. Res.*, vol. 11, pp. 972–989, 1963.
[58] G. Ingargiola and J. Korsh, "A general algorithm for one-dimensional knapsack problems," *Ops. Res.*, vol. 25, pp. 752–759, 1977.
[59] V. Kumar and L. Kanal, "A general branch and bound formulation for understanding and synthesizing AND/OR tree search procedures," *Artificial Intelligence*, vol. 21, pp. 179–198, 1983.
[60] T. S. Abdel-Rahman and T. N. Mudge, "Parallel branch and bound algorithms on hypercube multiprocessors," in *Proc. 3rd Conf. on Hypercube Concurrent Computers & Applications, vol. I*, pp. 1492–1499, Jan. 1989.
[61] T. S. Abdel-Rahman, "Parallel Processing of Best-First Branch and Bound Algorithms on Distributed Memory Multiprocessors," Ph.D. thesis, University of Michigan, 1989.

John P. Hayes (Fellow, IEEE) received the B.E. degree from the National University of Ireland, Dublin, in 1965, and the M.S. and Ph.D. degrees from the University of Illinois, Urbana, in 1967 and 1970, respectively, all in electrical engineering.

While at the University of Illinois he participated in the design of the ILLIAC III computer and carried out research in the area of fault diagnosis of digital systems. In 1970 he joined the Operations Research Group at the Shell Benelux Computing Center of the Royal Dutch Shell Company in The Hague, The Netherlands, where he was involved in mathematical programming and software development. From 1972 to 1982 he was a faculty member of the Departments of Electrical Engineering and Computer Science of the University of Southern California, Los Angeles. He is currently a Professor in the Electrical Engineering and Computer Science Department of the University of Michigan, Ann Arbor. His research interests include computer architecture; parallel processing; fault tolerance and reliability; and computer-aided design and testing of VLSI systems.

Dr. Hayes was Technical Program Chairman of the 1977 International Conference on Fault-Tolerant Computing. He is the author of over a hundred technical papers and several books, including *Digital System Design and Microprocessors* (McGraw-Hill, 1984) and *Computer Architecture and Organization*, 2nd Ed. (McGraw-Hill, 1988). He served as Editor of the Computer Architecture and Systems Department of *Communications of the ACM* from 1978 to 1981, and was Guest Editor of the June 1984 Special Issue of *IEEE Transactions on Computers*. He was the founding Director of the Advanced Computer Architecture Laboratory at the University of Michigan from 1985 to 1988. He is a member of the Association for Computing Machinery and Sigma Xi.

Trevor Mudge (Senior Member, IEEE) received the B.Sc. degree in cybernetics from the University of Reading, England, in 1969, and the M.S. and Ph.D. degrees in computer science from the University of Illinois, Urbana, in 1973 and 1977, respectively.

While at the University of Illinois he participated in the design of several special purpose computers and did research in computer architecture. Since 1977, he has been on the faculty of the University of Michigan, Ann Arbor where he has taught classes on logic design, CAD, computer architecture, and programming languages. He is presently an Associate Professor of Electrical Engineering and Computer Science, and Director of the Advanced Computer Architecture Lab. In addition to his position as a faculty member, he is a consultant for several computer companies in the areas of architecture and languages.

Dr. Mudge is the author of more than 80 papers on computer architecture, programming languages, VLSI design, and computer vision, and he holds a patent in computer aided design of VLSI circuits. He is a member of the ACM and of the British Computer Society.