A Microprocessor-based Hypercube Supercomputer

John P. Hayes, Trevor Mudge, and Quentin F. Stout
University of Michigan

Stephen Colley and John Palmer
NCUBE Corporation

Each node in the NCUBE/ten parallel processor is organized around a custom, VAX-like, 32-bit CPU chip. With 1024 nodes, the NCUBE/ten provides a throughput of 500 MFLOPS.

The most straightforward and least expensive way to build a supercomputer capable of hundreds of millions of instructions per second is to interconnect a large number of microprocessors. Supercomputers built by corporations such as Cray Research and Control Data do not use this approach, but rely on very fast components and pipelined operations. However, such machines are quite expensive, and each performance improvement of them is increasingly difficult to achieve. In contrast, the performance of machines consisting of interconnected microprocessors can be significantly improved simply by adding more microprocessors, faster microprocessors, and better interconnections among them.

An important consideration for systems made up of interconnected microprocessors is the question of local versus global memory and its effect on the interconnection scheme. In a global memory system, memory is shared by all the processors. Since two or more processors may try to use the same memory location at the same time, a global memory scheme requires the use of hardware or software protocols for arbitrating among processors. Further, since memory references are a very large fraction of any program's execution, the time required to access memory must be kept small. These two requirements make severe demands on the interconnection system between processors and the global memory, and thus they limit the number of processors that can be economically used.

In a distributed memory system each processor has its own memory, and information is exchanged as messages between processors. If each processor has most of the data it will need, the number of messages between processors can be kept relatively small, and the numbers of processors in the system can be larger. This is particularly true in interconnection systems using processor-to-processor connections instead of bus connections. In a processor-to-processor connection scheme, each processor is directly connected to a subset of the other processors (its "neighbors"). Messages between processors not directly connected must be passed through intermediate processors. Because a processor can pass messages more quickly to its neighbors than to processors not directly connected to it, tasks that need extensive intercommunication should be placed on neighboring processors. An interconnection scheme that makes it easier to achieve such placement is the hypercube.

An n-dimensional hypercube computer, also known as a binary n-cube computer, is a multiprocessor characterized by the presence of $N = 2^n$ processors interconnected as an

n-dimensional binary cube. Each processor P forms a node, or vertex, of the cube and has its own CPU and local main memory. P has direct communication paths to n other processors (its neighbors); these paths correspond to the edges of the cube. There are 2^n distinct n-bit binary addresses or labels that may be assigned to the processors. Thus, each processor's address differs from that of each of its n neighbors in exactly one bit position. Figure 1 illustrates the hypercube topology for $n \leq 4$; note that a zero-dimensional hypercube is a conventional single processor. The usual method for constructing a hypercube and assigning binary addresses to its nodes employs the following recursive procedure: Start with a one-dimensional cube (two nodes) and label one of its nodes with a 0 and the other with a 1. In general, an n-dimensional cube is constructed from two $(n-1)$-dimensional cubes. The labels in one of the cubes are prefixed with a 0 (the zero-cube) and those in the other with a 1 (the one-cube); then each node in the zero-cube is connected to its counterpart in the one-cube, i.e., to the node that has the identical address except for the prefix. Thus, node P_{0x} is connected to P_{1x} (subscripts are the binary addresses). All hypercubes of higher dimension and their node addresses can be generated from this procedure. Each dimension of a hypercube has an associated axis that is defined as follows: Node P_{x0y} is connected to P_{x1y} by an edge that is in the direction of the ith axis if there are $i-1$ bits in y.

It has been known for some time that the hypercube structure has a number of features that make it useful for parallel computation. For example, meshes of all dimensions and trees can be embedded in a hypercube so that neighboring nodes are mapped to neighbors in the hypercube. Figure 2 shows how a 3×4 mesh can be embedded in a 4-cube. The communication structures used in the fast Fourier transform and bitonic sort algorithm can be embedded similarly in the hypercube. Since a great many scientific applications use mesh, tree, FFT, or sorting interconnection structures, the hypercube is a good candidate for a general-purpose parallel architecture. Even for problems with less regular communication patterns, the hypercube's maximum internode distance (graph diameter) of $n = \log_2 N$ means any two nodes can communicate fairly rapidly. This diameter is larger than the unit diameter of a complete graph K_N, but is achieved with nodes having only a degree or fanout of $\log_2 N$, as opposed to the $N-1$ degree of nodes in K_N. Other standard architectures with small degree, such as meshes, trees, or bus systems, have either a large diameter (N for a two-dimensional mesh) or a resource that becomes a bottleneck in many applications because too much communication must pass through it (as occurs at the apex of a tree, or at a shared bus). Thus, from general topological arguments we can conclude that hypercube architectures balance node connectivity, communication diameter, algorithm embeddability, and programming ease. This balance makes them suitable for an unusually broad class of computational problems.

Proposals to build large hypercube computers have been made for more than 20 years. In 1962, Squire and Palais at
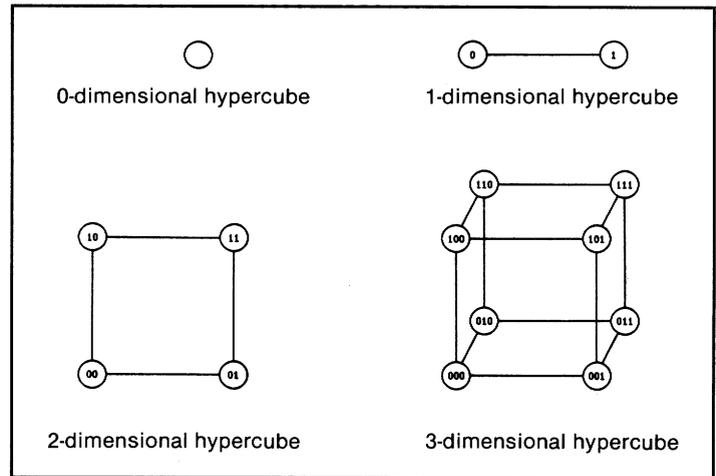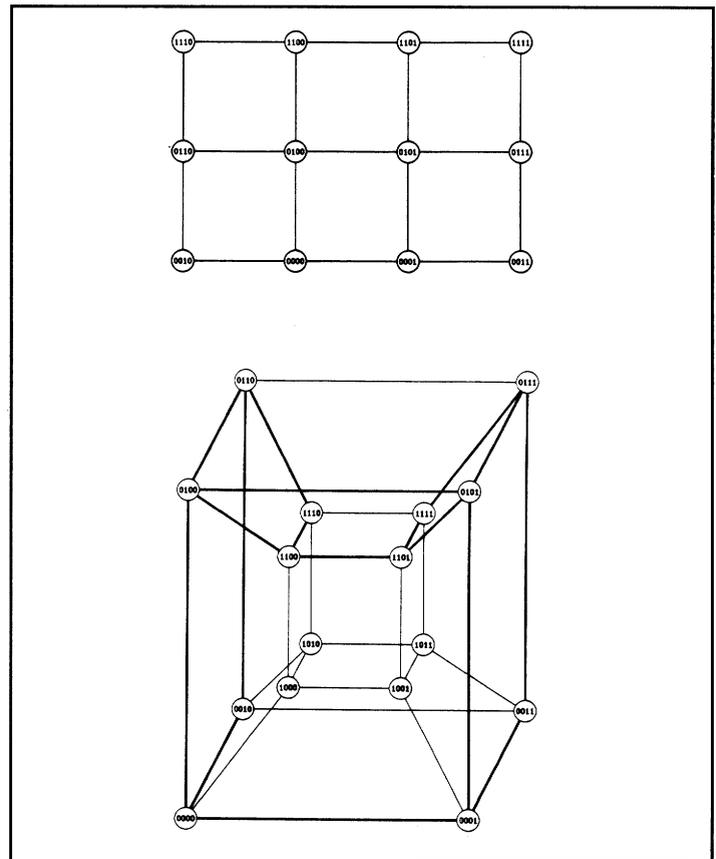

Figure 1. Hypercubes for $n = 0, 1, 2,$ and 3.

Figure 2. Embedding a 3×4 mesh in a 4-cube.

the University of Michigan carried out a detailed paper design of a hypercube computer.[1,2] They estimated that a 4096-node (12-dimensional) version of their machine would require about 20 times as many components as the IBM Stretch, one of the largest and most complex computers

built up to that time. Around 1975 IMS Associates, an early manufacturer of personal computers, announced a 256-node commercial hypercube based on the Intel 8080 microprocessor, but they neither published details of its design nor produced a machine. In 1977, Sullivan and his colleagues at Columbia University presented a proposal for a large hypercube called the Columbia Homogeneous Parallel Processor, or CHOPP, which would have contained up to a million processors.[3,4] In the same year, Pease published a 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.[5] Several other interesting architectures closely related to the hypercube have been proposed—for example, the cube-connected-cycles structure.[6]

It is clear that the early hypercube designs were impractical because of the large number of logic and memory elements they would have required, given the then-available circuit technologies. The situation began to change rapidly in the early 1980's as advances in VLSI technology allowed powerful 16/32-bit microprocessors to be implemented on a single IC chip, and as RAM densities moved into the 100,000- to 1,000,000-bit-per-chip range. The first working hypercube computer—the 64-node Cosmic Cube at Caltech[7]—was demonstrated in 1983. For the hypercube node processor, it uses a single-board microcomputer containing an Intel 8086 microprocessor and an Intel 8087 floating-point coprocessor. Since then, Caltech researchers have built several similar hypercubes and successfully applied them to numerous scientific applications, often obtaining impressive performance improvements over conventional machines of comparable cost.[8]

Influenced primarily by the Caltech work, several companies have developed commercial hypercubes since 1983. In July 1985, Intel delivered the first production hypercube, the Intel Personal Supercomputer, or iPSC, which has a 16-bit 80286/287 CPU as its node processor and up to 128 nodes. If we assume a peak performance of 0.1 MFLOPS per node, the 128-node iPSC has a potential throughput of about 12 MFLOPS, far below that of a traditional vector supercomputer such as the Cray-1, which has a peak throughput of 160 MFLOPS. Other commercial hypercubes introduced in 1985 include Ametek's System/14 and NCUBE Corporation's NCUBE/ten. The System/14 hypercube can have up to 256 nodes, each employing an 80286/287-based CPU similar to that of the iPSC and an 80186 processor 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 0.5 MFLOPS. Thus, a fully configured NCUBE system has a potential throughput of around 500 MFLOPS. This high performance level is supported by extremely fast communication rates (both input/output and node-to-node), making the NCUBE/ten a true supercomputer. NCUBE machines have been installed at several beta-test sites, including the University of Michigan, since early 1985, and have been in general production since December 1985. Other recently introduced hypercube-style machines

with supercomputing potential include the Caltech/JPL Mark III,[9] the Connection Machine,[10] the Intel iPSC-VX, and the Floating Point Systems T Series. The last two machines include a pipelined vector processor at each node. Much faster successors to the current commercial hypercubes can be expected to appear over the next few years. Because of the effort being devoted to the development of hardware and software for these machines, and because of their relatively low cost, hypercube supercomputers seem likely to provide an increasingly attractive alternative to conventional pipelined supercomputers for many applications.

Here, we discuss the architectural and technological issues influencing the design of microprocessor-based supercomputing hypercubes, employing the NCUBE/ten as an example. We pay particular attention to the influence of component packaging, reliability, communication speed, and the operating system environment on the system implementation.

General design issues

To provide supercomputer-level performance, the designer must build a machine with extremely high integer and floating-point execution rates as well as extremely high I/O throughput. He must also provide very large primary (RAM) and secondary (disk) memory spaces. For the principal supercomputer user base of scientific programmers, he must develop a programming environment that includes Fortran and a powerful operating system such as Unix. To achieve low cost and high reliability, he must minimize the component count at all levels, particularly the number of chips and boards. He should also provide some degree of fault tolerance. Since a very large amount of RAM storage is needed, he should consider employing an error-correcting code, or ECC, to detect and correct memory faults, despite the fact that it increases the chip count. To increase reliability and decrease operating cost, the designer should use an air-cooled configuration suitable for a standard office environment. (An examination of existing computer systems shows that air cooling limits machine complexity to 50,000 chips.) The designer should use off-the-shelf parts, since they are cheaper and usually more reliable than custom chips. If he does need custom parts, and if his company relies on outside suppliers, he should use conservative design rules that will be accepted by several silicon foundries. Because large error accumulation can occur in large-scale numerical calculations, he should make individual calculations as accurate as possible. He can do so by adhering to the IEEE 754 floating-point standard and by providing double-precision, floating-point operations.

A key decision in the design of a parallel computer is the choice of interconnection network. Multistage interconnection networks simplify the programming process by providing a global shared memory, but they cannot be built with current technology without significantly delaying the information being passed over them. Since the speed of information passing over the network strongly affects perfor-

mance, direct connection networks, with local memory at every node, provide the most desirable means of achieving supercomputer performance. Many direct interconnection schemes have been analyzed and implemented but, as noted previously, the hypercube structure has a number of inherent advantages. The ease with which efficient application programs were developed for the hypercubes at Caltech has also shown the hypercube to be superior to alternative architectures such as meshes or trees. The neighbor-to-neighbor links of the hypercube provide almost the same communication capabilities as a complete graph while using nodes with only a logarithmic degree. The achievable degree is constrained by a variety of packaging considerations, but with current technology one can build hypercubes with thousands of nodes. In contrast, a complete graph connection of a few tens of nodes may not be possible.

There are additional features of the hypercube that are useful in designing a supercomputer, but that have not been exploited prior to the development of the NCUBE/ten. For example, the hypercube is homogeneous in that all nodes look the same, so an I/O channel can be attached to each node. This provides the potential of extremely high system I/O rates. Also, since there are numerous ways to divide a hypercube into subcubes, the designer, by giving each user a dedicated subcube, should find it easy to support multiprocessing. These dedicated subcubes can be allocated so that all processor-to-processor and I/O communications occur within them and do not use processors or communications lines in other subcubes. Further, the designer can allow the user to define the size of the subcube in the programs he writes; in this way, the user can develop programs in small subcubes and then do production runs in larger ones. This partitionability makes it easier for the system to tolerate faults, since the operating system can allocate subcubes that avoid faulty processors or faulty communication lines.

As we noted previously, technological developments have made it possible to build a reliable hypercube computer with a large number of processors. A fine-grained hypercube architecture, i.e., one with a large number (tens of thousands) of very simple processors, has a high ratio of communication to computation. Thus, the suitability of such an architecture—the Connection Machine is an example—to general scientific computation is uncertain. A very coarse-grained architecture with, say, tens of large and fast processors requires that the nodes achieve extremely high performance. For example, to achieve 10^9 instructions per second with 10 processors, such an architecture requires those processors to be capable of 10^8 instructions per second. The Caltech/JPL Mark III is an example of a coarse-grained hypercube. The designers of the NCUBE machine felt that achieving 10^9 instructions per second was best done with a medium-grained approach—1000 processors running at 10^6 instructions per second.

The Caltech machines and their commercial successors are MIMD (multiple-instruction- and multiple-data-stream) machines, meaning that each processor has its own pro-

gram as well as its own data. These machines are customarily used in an SCMD (single-code, multiple-data) fashion, in which all processors have a copy of a single program, though they may be executing different branches at a given time.

Experience with the Caltech machines has demonstrated that a medium-grained MIMD hypercube architecture can attain high efficiency on a variety of scientific problems without demanding an intolerable amount of revision of serial code and algorithms from its users.[8] This is in contrast to the much greater amount of program and algorithm redesign required of users of fine-grained SIMD (single-instruction- and multiple-data-stream) machines such as the MPP[11] and Connection Machine. In SIMD machines, a controller broadcasts instructions that all processors receive and perform on their own data. The node processors in MIMD machines are more complex than those in SIMD machines since they must fetch instructions rather than just receive broadcasted ones. Further, distributed-memory MIMD machines need additional memory to store the program at each node. In general, one can build more SIMD processors than MIMD processors on a given amount of silicon and have a greater potential system throughput; however, the gain in programming simplicity provided by MIMD machines more than compensates for this, except for a narrow range of applications in which almost any penalty can be tolerated if it yields the required speed. Furthermore, MIMD machines can accommodate multiple independent users, while SIMD machines cannot.

Since there may be hundreds or thousands of nodes in a hypercube supercomputer, their chip count is the most significant component of the total system chip count. Using the densest memory chips available is the most effective way the designer can decrease the total number of chips. The NCUBE/ten, for example, uses 256K DRAM chips to implement the local memories of the hypercube nodes. The next most effective way the designer can reduce the chip count is by putting all node functions onto a single chip. This implies that the processor chip must perform all communication, memory management, and floating-point operations as well as other data processing functions. At present there is no widespread market pressure to produce standard processor chips of this type; consequently, they are not available off the shelf. In 1983, when design of the NCUBE/ten started, the only way to achieve supercomputer performance with a single-chip node processor was to undertake the risky step of custom designing such a chip. INMOS made a similar decision with the Transputer processor chip, with the important difference that the initial version of the Transputer did not provide floating-point operations and has four rather than eleven I/O channels.[12] The performance and functionality demands on the NCUBE processor chip were quite severe, and numerous trade-offs were made to enable it to be built.

System architecture

The overall goal of the NCUBE's designers was to use massive parallelism to build a range of inexpensive and
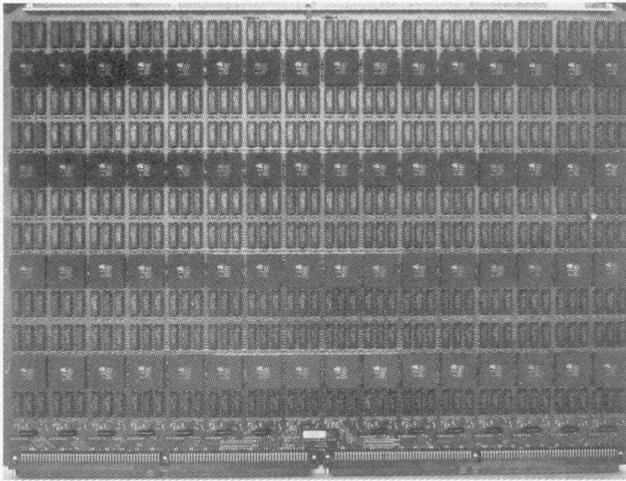
Figure 3. Six-dimensional hypercube with 64 nodes and 8M bytes of memory fits on one 16 × 22-inch board.

reliable software-compatible machines achieving supercomputer performance at the high end of that range. The NCUBE/ten is the largest model in the series; it is a 10-dimensional hypercube containing 1024 custom-designed 32-bit processors, each with a 128K-byte local memory. It uses up to eight front-end host processors to manage I/O operations, with those processors under control of a multiuser Unix-based operating system. It achieves a level of system integration high enough to allow a six-dimensional hypercube with 64 nodes and 8M bytes of memory to be placed on a single 16 × 22-inch board (Figure 3). Its backplane connections are rather formidable—640 connections just for communication channels—since each processor node has off-board bidirectional channels to four more processors of the hypercube plus one bidirectional channel to an I/O board. A maximum-sized NCUBE/ten system is composed of 16 processor boards and eight I/O boards and is housed in a small air-cooled enclosure.

The NCUBE/ten's I/O boards provide the connections between the hypercube and the external world. At least one of the I/O boards must be a host board, and there can be as many as eight. The host board uses an Intel 80286 to run the operating system and has 4M bytes of RAM that is used as a shared memory by the various processors on it. It supports a variety of peripherals, including eight ASCII-standard terminals, four SMD disk drives (which can be as large as 500M bytes), and three Intel iSBX connectors that can accept daughterboards for functions such as graphics control or networking. The host board incorporates a real-time clock and temperature sensors for automatic shutdown on overheating. Besides the host board, other I/O boards available with the NCUBE/ten include a graphics board with a 2K × 1K × 8-bit frame buffer, an intersystem board that connects two NCUBE systems, and an open system board that has about 75 percent of its space left open for custom design.

A distinguishing feature of the I/O boards is that each has 128 bidirectional channels directly connected to a sub-cube of the hypercube (Figure 4). This permits extremely high I/O data transfer rates into the hypercube. To accomplish this, each I/O board contains 16 NCUBE processor chips, each of which serves as an I/O processor and is connected to eight nodes in the main hypercube. Like the hypercube node processors, an I/O processor has a 128K-byte RAM that occupies a fixed slot in the 80286 host's 4M-byte memory space. An I/O processor performs an input operation from the outside world—a disk read, for example—by first transferring the input data to the host's 4M-byte memory. It then transfers the data through its DMA channel directly to the target hypercube nodes. It handles output operations in a similar fashion. In a maximally configured NCUBE/ten system with 16 processor and eight I/O boards, the hypercube nodes do not have to redistribute I/O data to other nodes. This is not always the case with smaller NCUBE systems; it depends on the number and configuration of the I/O and processor boards.

All the I/O and processor boards of a fully configured system, along with their fans and power supplies, fit into a single enclosure that is less than three feet on a side. A maximally configured system dissipates about 8 kW and can be placed in a normal air-conditioned office or lab. An NCUBE/ten peripheral enclosure is about 3 × 2 × 3 feet and contains a 65M-byte cartridge tape drive and up to four disk drives. A minimal stand-alone NCUBE system consists of one host board and one processor board containing a six-dimensional hypercube, and can handle up to eight user terminals. By adding a second processor board, one obtains a seven-dimensional hypercube. Since the operating system can allocate subcubes of arbitrary size, one can have a number of processor boards that do not form a complete hypercube. For example, three boards provide a seven-dimensional and a six-dimensional hypercube, which could also be allocated as three six-dimensional hypercubes or as numerous smaller hypercubes. A maximally configured system (Figure 4) contains a 10-dimensional hypercube. The 1024 processors of such a system have a potential instruction execution rate of about two billion instructions per second, or about 500 MFLOPS, with a 10-MHz clock. The total amount of memory in the nodes is 128M bytes. If all of the I/O boards are host boards, it is possible to support 64 terminals and provide as many as 16 billion bytes of storage. A host board can provide input or output at up to 12M bytes per second, giving a system input or output rate of about 90M bytes per second. In the case in which a single data set is to be broadcast to all nodes, input rates can exceed 90M bytes per second per host board—that is, they can approach 720M bytes per second per system.

The node processor

The NCUBE node processor provides, on a single VLSI chip, the functions of a 32-bit supermini-class CPU, in-
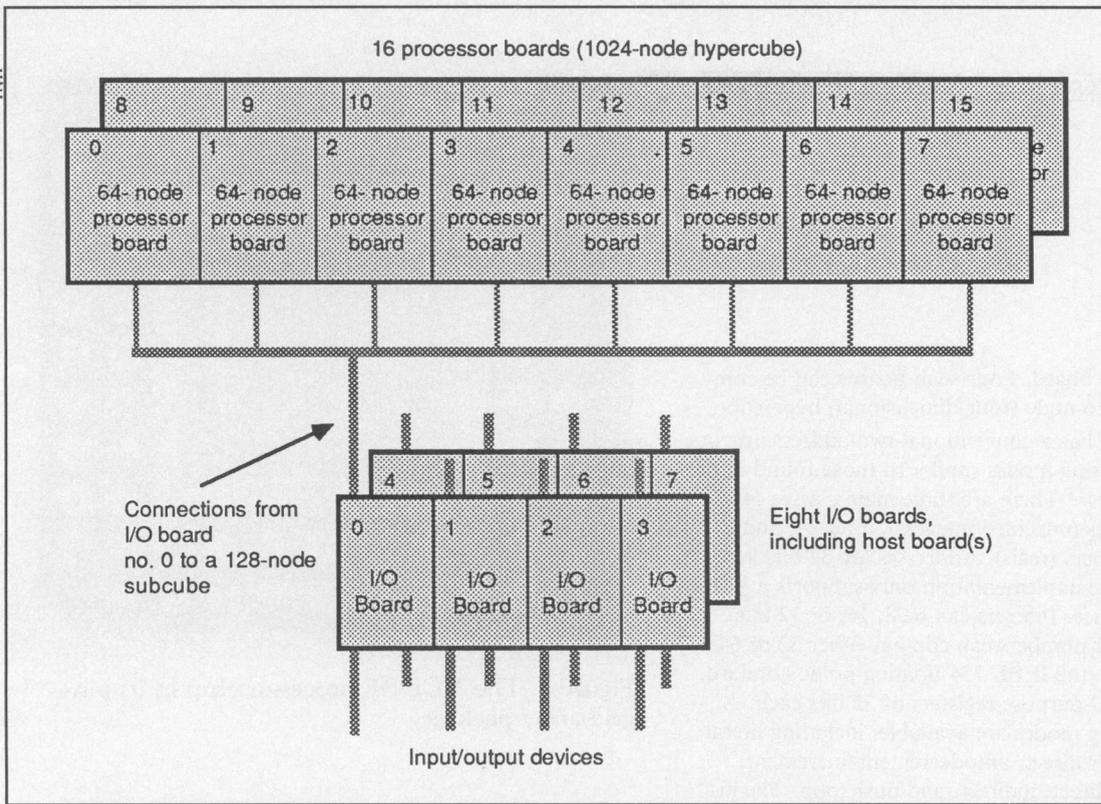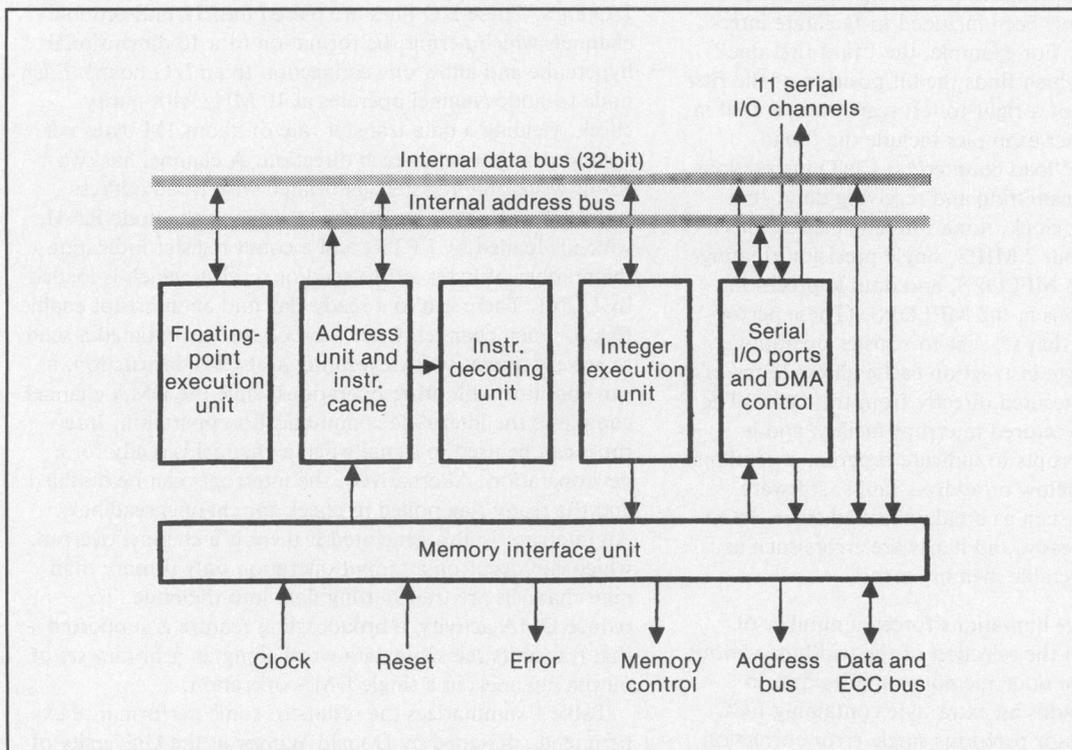
Figure 4. Complete NCUBE/ten system.

16 processor boards (1024-node hypercube)

Connections from I/O board no. 0 to a 128-node subcube

Eight I/O boards, including host board(s)

Input/output devices

Figure 5. Organization of the NCUBE processor chip.

11 serial I/O channels

Internal data bus (32-bit)

Internal address bus

Floating-point execution unit

Address unit and instr. cache

Instr. decoding unit

Integer execution unit

Serial I/O ports and DMA control

Memory interface unit

Clock Reset Error Memory control Address bus Data and ECC bus

cluding a full floating-point instruction set and all the logic needed for memory management and interprocessor communication. Figure 5 shows the major functional blocks of the chip, and Figure 6 is a photograph of the chip in its package. NCUBE began design of the node processor chip in 1983, constraining itself to 2-μm NMOS design rules that were acceptable to several silicon foundries. The chip contains about 160,000 transistors and is housed in a pin-grid-array package having 68 pins. Including six 256K-bit DRAM chips (each organized as 64K $\times$ 4 bits), an entire NCUBE/ten node requires only seven chips. This unusual compactness has prompted the introduction of a four-node

(two-cube) IBM AT board. Four such boards can be combined to provide a 16-node (four-dimensional) hypercube.

The NCUBE/ten has a conventional two-address instruction set with addressing modes similar to those found in the VAX instruction set.[13] There are three main classes of information: addresses (unsigned integers), integers, and floating-point numbers (reals). Addresses are 32 bits long, but the current node implementation only supports a 17-bit physical address space. Integers can be 8, 16, or 32 bits long. Floating-point numbers can contain either 32 or 64 bits and conform to the IEEE 754 floating-point standard. There are 16 general-purpose registers of 32 bits each. A variety of addressing modes are available, including literal (immediate), register direct, autodecrement/increment, autostride, offset, direct, indirect, and push/pop. The instruction set contains a full complement of logical, shift, jump, and arithmetic operations (including square root). Several instructions have been included to facilitate internode communications. For example, the "find first one" instruction, or FFO, which finds the bit position of the first 1 in a word by means of a right-to-left scan, can be used in internode routing. Other examples include the "load pointer" (LPTR) and "load counter" (LCNT) instructions, which are used for transmitting and receiving data. In a system with a 10-MHz clock, nonarithmetic instructions can be executed at about 2 MIPS, single-precision floating-point operations at 0.5 MFLOPS, and double-precision floating-point operations at 0.3 MFLOPS. (These performance figures assume that register-to-register operations predominate.) A 32-byte instruction cache allows loops of up to 16 bytes to be executed directly from the cache. The node processor has a vectored interrupt facility, and it generates various interrupts to indicate program exceptions such as numerical overflow or address faults, software debugging commands such as breakpoint and trace, I/O signals such as input ready, and hardware errors such as correctable or uncorrectable memory errors.

Pin and silicon space limitations forced a number of design compromises in the selection of the width of various system data paths. The node memory supplies data in 16-bit halfwords and adds an extra byte containing ECC check bits. The processor performs single-error correction and double-error detection (SECDED) on all memory words, generating an interrupt in the case of an error. This use of SECDED is an example of a situation in which the pin limitations affect performance, for it requires two memory fetches to obtain a full 32-bit word. It also increases the number of memory chips required, since the SECDED code used for 32-bit data could be supplied by five RAM chips organized as 32K × 8 bits, if such chips were available.
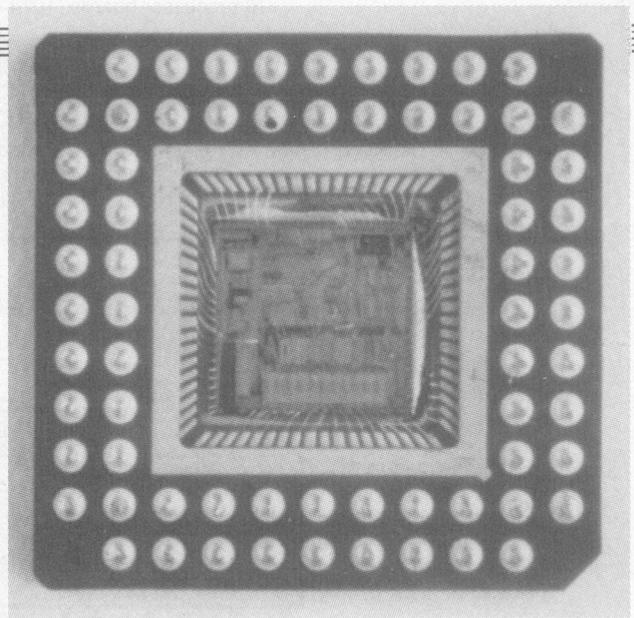

Figure 6. The NCUBE processor chip in its pin-grid-array package.

The node processor communicates with other nodes by means of asynchronous DMA operations over 22 bit-serial I/O lines. These I/O lines are paired into 11 bidirectional channels which permit the formation of a 10-dimensional hypercube and allow one connection to an I/O board. Each node-to-node channel operates at 10 MHz with parity check, yielding a data transfer rate of about 1M bytes per second per channel in each direction. A channel has two 32-bit write-only registers associated with it: an address register for the message buffer location in the node RAM, which is loaded by LPTR; and a count register indicating the number of bytes left to send or receive, which is loaded by LCNT. There is also a ready flag and an interrupt enable flag for each channel. Once a processor has initiated a send or receive operation by executing an LCNT instruction, it can continue with other operations while the DMA channel completes the internode communication operation. Interrupts can be used to signal when a channel is ready for a new operation. Alternatively, the interrupts can be disabled and the ready flag polled to check for channel readiness. An interrupt is also generated if there is a channel overrun, which can occur on an input operation only if more than nine channels are transmitting data into the node. To reduce DMA activity, a broadcasting feature is supported that transmits the same data word along an arbitrary set of output channels in a single DMA operation.

Table 1 summarizes the results of some performance experiments, designed by Donald Winsor at the University of Michigan, that compared the NCUBE node processor to two other CPUs with floating-point hardware: the Intel 80286/80287 (the NCUBE host processor served for this) and the Digital Equipment Corporation's VAX-11/780 with a floating-point accelerator. The measurements were made with the NCUBE node and host processors running at 8 MHz. Extrapolated figures for the 10-MHz version of the NCUBE node processor now nearing production are also given; they assume no wait states. Two widely used syn-

	Table 1. Performance figures.	
Processor	Fortran Dhrystones/s	Fortran Whetstones/s*
NCUBE node processor at 8 MHz	999	381,000
NCUBE node processor at 10 MHz (est.)	1249	476,000
Intel 80286 (NCUBE host) at 8 MHz with 80287 floating-point coprocessor	510	101,000
DEC VAX-11/780 with floating-point accelerator	741	426,000

*Double precision.

thetic benchmark programs were employed in this study: the Dhrystone and the Whetstone codes. [14,15] The Dhrystone benchmark is intended to represent typical system programming applications and contains no floating-point or vectorizable code. The original Dhrystone Ada code [15] was translated into a Fortran 77 version with 32-bit integer arithmetic that attempted to preserve as much of the original program structure as possible. This entailed simulating Ada records with Fortran arrays, and simulating access variables for those records with the array index variables. This "Fortran Dhrystone" produced a substantial performance degradation compared to Dhrystone benchmarks in Ada, Pascal, and C, all of which have pointer or access variables. For example, the C Dhrystone ran two to three times faster on a VAX-11/780 than the Fortran Dhrystone. However, in the study presented here the degradation appears to apply uniformly to all processors considered, since all were given the same Fortran source code and used very similar Fortran compilers. The Whetstone benchmark, which aims to represent scientific programs with many floating-point operations, was used in a double-precision Fortran 77 version that closely resembled the original Algol code. [14] The Dhrystone results in Table 1 are reported in "Dhrystones per second," each of which corresponds roughly to one hundred Fortran statements executed per second. The Whetstone figures represent the number of hypothetical Whetstone instructions executed per second. We can conclude from the data in Table 1 that the NCUBE node processor is quite fast and fully meets its performance targets.

System software

The emergence of several commercial hypercube computers 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, the recent report on the Supercomputing Research Center concludes that the absence of appropriate parallel programming languages and

software tools is the single biggest impediment to the successful use of parallel machines. [16] The operating system is also a major design issue, since memory management and interprocessor communication are critical to the functioning of the programming languages. Three software issues need to be considered. The first is the operating system that is used for developing application programs for the hypercube. The second is the operating system that provides runtime support for application programs running on the hypercube nodes. The third is the set of application languages to be used.

An operating system for application program development that provides the kind of environment associated with a "programmer's workbench" is Unix. Unfortunately, there are two versions of Unix, System V and bsd 4.3, and many lesser-known variants. This leaves the system designer with a dilemma: he can work in a proven, widely known development environment, but he can't exploit the benefits of standardization, since no Unix standard has emerged. The solution chosen by NCUBE was to develop a Unix-like operating system, Axis, [13] that embodies the features common to the major Unix dialects. Changes or additions can be readily made to Axis when a true Unix standard is agreed upon. There are two features of Axis that greatly facilitate program development for a very large hypercube. The first is its ability to share files, and the second is the way it manages the main cube array.

Axis runs on the 80286 host processor that acts as the CPU for each I/O board. (Recall that up to eight I/O subsystems can be accommodated in a 1024-processor NCUBE/ten.) It provides the large number of utilities for editing, debugging, and file management that one has come to expect in a Unix-like operating system. Axis' file system is its most prominent feature, and almost all system resources are treated as files. This is consistent with the Unix philosophy. Massively parallel systems require high I/O bandwidth if they are to be useful for applications that are not simply computation-intensive. The problem of managing high I/O was not foreseen in the earlier generation of massively parallel machines and has proven to be a great limitation. [11] The ability to incorporate up to eight

I/O subsystems in the NCUBE/ten is intended to avoid this problem. However, it introduces the potential for eight separate file systems. To avoid this, Axis provides the capability to organize the eight file systems as one distributed file system; Axis further allows complete systems to be networked through iSBX connections so as to provide a single multisystem file system.

Axis manages a hypercube of node processors as a device, which is simply one type of file. A device can be opened, closed, written to, and read from as if it were a normal file. Axis permits users to allocate subcubes that have the appropriate size for their application. Thus, one or two users with large problems or several users with small problems can share the hypercube. This flexibility greatly increases the system's efficiency and gives a hypercube supercomputer a significant advantage over conventional supercomputers. Partitioning the main hypercube into subcubes is simplified by the fact that each subcube is protected from access by any other subcube.

Vertex, the operating system for the NCUBE/ten node processors, is a small nucleus (less than 4K bytes) resident in each of those nodes. Its primary function is to provide communication between the nodes. It achieves this through, among other facilities, send and receive functions that transfer messages between any two nodes in the hypercube, and through a **whoami** function that allows a program to determine the logical node on which it is executing and the I/O processor to which it is connected. The internode send and receive functions are implemented as subroutine calls **nwrite** and **nread**, respectively; the whoami function is implemented as a subroutine call **whoami**. The messages transferred by **nwrite** and **nread** are arrays of bytes having four attributes: source, destination, length, and type. The first two attributes are numbers in the range 0 to 1023 and indicate the logical nodes being used for the source and destination. The length attribute is the number of bytes in the message; messages as long as 64K bytes are supported. The type attribute can be used to distinguish messages and so permit their selective reception at a destination node.

The subroutine **nwrite** passes the following parameters: **length, message, dest, type, status,** and **error.** They are passed in the general-purpose registers. **Length** is the length of the outgoing message in bytes; **message** is the name of the buffer from which the message is to be taken; **dest** is the logical number of the node in the hypercube that is to receive the message; **type** is the type number of the message; **status** indicates when the message leaves the buffer, i.e., when the buffer is reusable; and **error** is an error code. Message transmission breaks the message into packets of 512 bytes (or some other user-defined size) and sends them to the destination node using the following routing algorithm. Assume that in an n-dimensional cube, the logical number of the source node is $s_n s_{n-1} \ldots s_2 s_1$ and the logical number of the destination is $d_n d_{n-1} \ldots d_2 d_1$. The bit-wise exclusive-OR $x_n x_{n-1} \ldots x_2 x_1$ of the two numbers is formed as follows: $x_i = s_i \oplus d_i$ for $i = 1, \ldots, n$. The values of the x_i's are used to control the routing process.

Those values of i for which $x_i = 1$ indicate the dimensions that must be traversed to transfer a message from source to destination. The FFO instruction mentioned previously can be used to determine the values of i. Since it works by scanning right to left, it will route messages along the lower dimensions first. The routing algorithm was chosen for its simplicity; however, as noted by Valiant,[17] it creates the potential for congestion in some situations. He defines an alternative routing algorithm that avoids congestion by routing 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 discouraged use of it in the initial NCUBE/ten design. Future insights into the behavior of parallel algorithms may change this, however.

In addition to determining the routing path, Vertex must perform the store-and-forward function at each node along the path. At the destination node, it places the message in a queue that is allocated from a heap of 20K bytes. The receive function, **nread**, passes the following parameters: **length, message, source, type, status,** and **error.** It looks for the first message from **source** of type **type** in the input queue, and copies it to buffer **message.** Don't-care conditions are indicated for **type** or **source** by setting these parameters to -1. This allows the next message from a particular source to be received regardless of type, the next message of a particular type to be received from any source, and the next message of any type from any source to be received. Messages with negative types other than -1 are system messages for Vertex and are used for process control at a node, e.g., for node program debugging. In summary, the calls **nwrite** and **nread** provide a fast internode message communication mechanism. The main contributors to this speed are the machine instructions provided explicitly for internode communication and the fact that messages enter nodes through DMA channels.

The current NCUBE/ten application languages, apart from the node and host assembly languages, are Fortran 77 and C. Fortran 77 and C were chosen because the computer is targeted for a user community interested primarily in scientific problems; this group has traditionally programmed in Fortran. Compilers for other languages, including Occam, are presently being developed. The programming model adopted for the initial set of languages, Fortran and C, is a simple extension of the conventional uniprocessor model. Each node is treated as a separate processor. No symbols are shared between nodes—the naming scope is contained within a node. Values of variables are shared by means of calls to the Vertex subroutines **nwrite** and **nread.**

We noted earlier that the hypercube array can be shared by several users if it is partitioned into suitably sized subcubes. When a d-dimensional subcube is allocated to a user, its nodes are given a logical number from 0 to $2^d - 1$. Vertex records the correspondence between the logical

numbers of the nodes and their physical address in the main hypercube array. Along with the **whoami** function, logical numbering makes it possible to write programs that run on subcubes of arbitrary location and size.

The **whoami** function returns four identification parameters: **node, process, host,** and **dim,** where **node** is the logical node number of the calling process, **process** is the process number of the calling process, **host** is the id number for the host communication, and **dim** is the allocated subcube dimension.

We conclude this section with a sample Fortran program for the NCUBE/ten that calculates the sum of squares of the elements of a vector V. The Fortran uses the extensions NWRITE, NREAD, and WHOAMI, which are based on the Vertex functions discussed above. Figure 7 shows the program. We assume that a copy of this program has been loaded into each of the nodes in the subcube allocated for the job. The idea behind the program is to distribute equal numbers of the elements of V among the nodes, form local partial sums of squares, and then accumulate these partial sums along successive dimensions of the hypercube. The internode accumulation collapses the active part of the computation into smaller and smaller cubes. The example computes

$$S = \sum_{i=1}^{K} V(i)^2 ,$$

where $K = N \cdot 2^M$.

Calling WHOAMI on line 008 of the program establishes the caller's logical node number (PN), the node on the host board for I/O communications (HOST), and the order of the allocated subcube (M). Line 012 reads in an N-element slice of V from the host. The parameters of interest in this example have the following meaning: SR is a completion code, V is the address of the message buffer for the vector, N is the length of the vector in single-precision words of four bytes each, and HOST is the node on the host for cube communications. The loop on lines 017 to 018 (loop 1) forms the sum of squares of the slice, putting the result in S. This is done in parallel in each node. For this phase of the computation, all 2^M nodes are doing useful work and the utilization of the allocated cube approaches 100 percent. The loop from lines 020 to 042 (loop 2) accumulates the partial sums. Starting with 2^M partial sums in each of the nodes, it forms 2^{M-1} partial sums by adding pairs of partial results in nodes that are immediate neighbors on the Mth axis. The new partial sums are now confined to an $(M-1)$-dimensional hypercube—the original cube is collapsed to half its initial size. This process of collapsing the cube by half and accumulating the partial sums is repeated until the final sum is accumulated in logical node 0. The internode accumulation procedure is simulating an addition tree. Figure 8 illustrates this for a 3-cube. This phase of the computation is less efficient than the first phase. For an M-dimensional cube, the total processor utilization is given by

$$U = \frac{100}{M} \left(\frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{2^M} \right) \approx \frac{100}{M} \%.$$

Loop 2 counts down through the axes. Line 024 selects the nodes (PNs) in the part of the hypercube that remains active after the collapsing along the (I + 1)th axis. The neighboring nodes (NPNs) of the PNs are those that differ in the Ith position. Their numbers are calculated in line 030 by an exclusive-OR between the PN and $2^{(I-1)}$. The operator .NEQV. (not equivalent) performs this—it is an extension to Fortran 77. Line 031 partitions the active nodes into two sets: those that are to receive partial sums (line 038) and those that send them (line 036). Those nodes that send will not be active in the next iteration of loop 2. Line 047 transmits the result from node 0 to the host.

```
001 *           NODE PROGRAM TO CALCULATE:      SUM ( V(I) * *2 )
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 *M      :   dimension of allocated cube
007
008         CALL WHOAMI   (PN,PROC,HOST,M)
009
010 *receive vector V of length N (4N bytes) from Host
011
012         SR = NREAD   (V,N*4,HOST,TYPEH,FLAG1)
013
014 *compute sum of subset of V that is in this node;
015
016         S = O
017         DO 1  I = 1,N
018 1            S = S + V(I)* *2
019
020         DO 2  I = M,1,-1
021
022 *execute once for each axis of the hypercube;
023
024             IF (PN  .LT.  2* *I) THEN
025
026 *if this node is in the active part of the collapsed cube,
027 *do the computation below, otherwise the node is done
028 * NPN is neighbor of PN on the I-th axis
029
030             NPN = PN  .NEQV.  (2* *(I-1))
031             IF (NPN  .LT.  PN) THEN
032
033 *if neighbor's number is less, send the current accumulation;
034 *otherwise, receive it and update its value
035
036                 SW = NWRITE (S,4,NPN,TYPEN,FLAG2)
037             ELSE
038                 SR = NREAD (A,4,NPN,TYPEN,FLAG3)
039                 S = S + A
040             ENDIF
041         ENDIF
042 2   CONTINUE
043
044 *send final result back to host
045
046         IF (PN  .EQ.  O) THEN
047             SW = NWRITE (S,4,HOST,TYPEH,FLAG4)
048         ENDIF
```

Figure 7. Fortran sum-of-squares program for the NCUBE/ten.
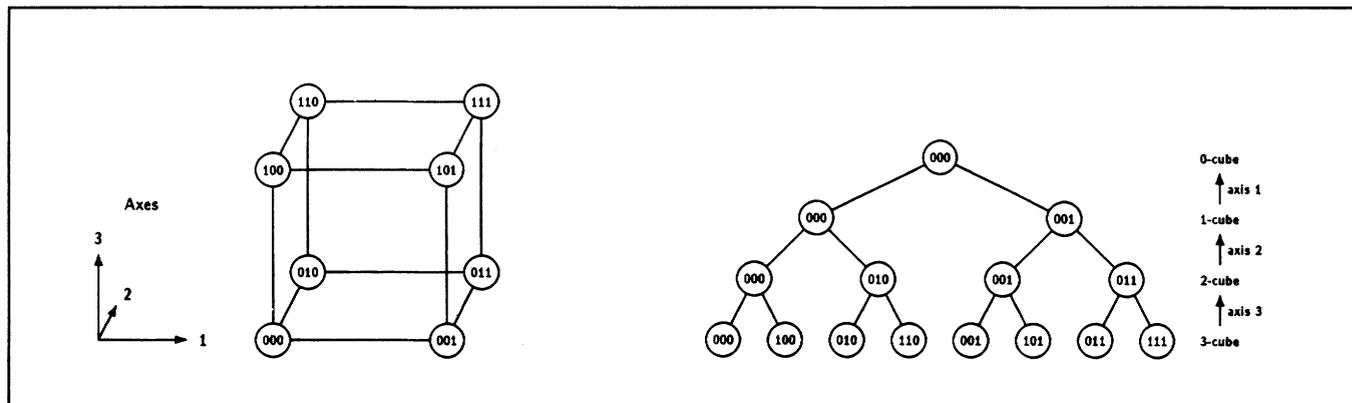
Figure 8. Addition tree for partial sums.

Hypercube architectures are well suited to implementing microprocessor-based massively parallel supercomputers, given the constraints imposed by current technology. They offer an unusually good combination of high node connectivity, software flexibility, and system reliability. The NCUBE/ten is an example of a new generation of low-cost and compact hypercube machines capable of supercomputer performance. Unlike earlier machines, it exploits the inherent homogeneity of the hypercube to provide a Unix-like multiuser programming environment, along with support for extremely high I/O data transmission rates. 🎚️
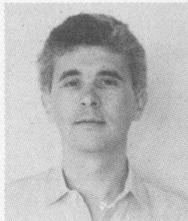
Acknowledgments

References

1. J. S. Squire and S. M. Palais, "Physical and Logical Design of a Highly Parallel Computer," tech. note, Dept. of Electrical Engineering, University of Michigan, Oct. 1962.

2. J. S. Squire and S. M. Palais, "Programming and Design Considerations for a Highly Parallel Computer," *AFIPS Conf. Proc.*, Vol. 23, 1963 SJCC, pp. 395-400.

3. H. Sullivan and T. R. Bashkow, "A Large Scale, Homogeneous, Fully Distributed Parallel Machine, I," *Proc. 4th Ann. Symp. on Computer Architecture*, 1977, pp. 105-117.

4. H. Sullivan, T. R. Bashkow, and D. Klappholz, "A Large Scale, Homogeneous, Fully Distributed Parallel Machine, II," *Proc. 4th Ann. Symp. on Computer Architecture*, 1977, pp. 118-124.

5. M. C. Pease, "The Indirect Binary *n*-cube Microprocessor Array," *IEEE Trans. Computers*, Vol. C-26, No. 5, May 1977, pp. 458-473.

6. F. P. Preparata and J. Vuillemin, "The Cube-connected Cycles: A Versatile Network for Parallel Computation," *Comm. ACM*, Vol. 24, No. 5, May 1981, pp. 300-309.

7. C. L. Seitz, "The Cosmic Cube," *Comm. ACM*, Vol. 28, No. 1, Jan. 1985, pp. 22-33.

8. G. Fox, "The Performance of the Caltech Hypercube in Scientific Calculations," Report CALT-68-1298, California Institute of Technology, Pasadena, Calif., Apr. 1985.

9. J. C. Peterson et al., "The Mark III Hypercube-Ensemble Concurrent Processor," *Proc. Int'l Conf. on Parallel Processing*, Aug. 1985, pp. 71-73.

10. W. D. Hillis, *The Connection Machine*, MIT Press, Cambridge, Mass., 1985.

11. J. P. Potter, ed., *The Massively Parallel Processor*, MIT Press, Cambridge, Mass., 1985.

12. *Transputer Reference Manual*, INMOS Corp., Colorado Springs, Colo., 1985.

13. *NCUBE Handbook*, Version 1.0, NCUBE Corp., Beaverton, Ore., Apr. 1986.

14. H. J. Curnow and B. A. Weichman, "A Synthetic Benchmark," *Computer J.*, Vol. 19, Feb. 1976, pp. 43-49.

15. R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Comm. ACM*, Vol. 27, No. 10, Oct. 1984, pp. 1013-1030.

16. *Report of the Summer Workshop on Parallel Algorithms and Architectures for the Supercomputing Research Center*, Aug. 1985.

17. L. G. Valiant, "A Scheme for Parallel Communication," *SIAM J. Computing*, Vol. 11, May 1982, pp. 350-361.

John P. Hayes has been a professor in the Electrical Engineering and Computer Science Department of the University of Michigan since 1982. He teaches and conducts research in computer architecture, VLSI design, digital system testing, and switching theory, and is director of the department's Advanced Computer Architecture Laboratory. From 1972 to 1982 he served on the faculty of the University of Southern California. Hayes received the BE degree from the National University of Ireland in 1965, and the MS and PhD degrees from the University of Illinois in 1967 and 1970, all in electrical engineering. He is a fellow of the IEEE and a member of the ACM and Sigma Xi.
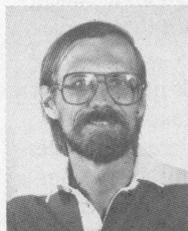
Trevor Mudge is an associate professor in the Department of Electrical Engineering and Computer Science of the University of Michigan. His research interests are computer architecture, programming languages, and computer vision. He received a BSc in cybernetics from the University of Reading, England, in 1969, and an MS and PhD in computer science from the University of Illinois in 1973 and 1977. Mudge is a senior member of the IEEE and a member of the ACM, the Institution of Electrical Engineers, and the British Computer Society.

Quentin F. Stout is an associate professor in the Department of Electrical Engineering and Computer Science of the University of Michigan. His research is in parallel algorithms, programming environments for parallel computers, and image processing. He learned programming in the public schools of Euclid, Ohio, and received a BA from Centre College, Danville, Kentucky, and a PhD from Indiana University. Stout is a member of the IEEE, the ACM, the American Mathematical Society, and the Mathematical Association of America, and serves on the editorial board of the *Journal of Parallel and Distributed Computing*.

Stephen Colley has been president of NCUBE Corporation since its founding in 1983. He spent five years with Intel Corporation working on microprocessor architecture. He also worked as an independent consultant for several years. Colley received a BS in electrical engineering from the California Institute of Technology in 1975.

John Palmer is chairman of the board of NCUBE Corporation, which he helped found in 1983. Prior to that he spent seven years with Intel Corporation, where he was the architect of the 8087 numeric processor. While at Intel, he contributed to the definition of the IEEE standard for floating-point arithmetic. Palmer received a BS in mathematics from Brigham Young University, an MS in mathematics from the University of Michigan, and a PhD in computer science from Stanford University.

Questions about this article can be directed to John P. Hayes at the Advanced Computer Architecture Laboratory, Dept. of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

High   159   Medium   160   Low   161