THE COSMIC CUBE

Sixty-four small computers are connected by a network of point-to-point communication channels in the plan of a binary 6-cube. This "Cosmic Cube" computer is a hardware simulation of a future VLSI implementation that will consist of single-chip nodes. The machine offers high degrees of concurrency in applications and suggests that future machines with thousands of nodes are both feasible and attractive.

CHARLES L. SEITZ

The Cosmic Cube is an experimental computer for exploring the practice of highly concurrent computing. The largest of several Cosmic Cubes currently in use at Caltech consists of 64 small computers that work concurrently on parts of a larger problem and coordinate their computations by sending messages to each other. We refer to these individual small computers as *nodes*. Each node is connected through bidirectional, asynchronous, point-to-point communication channels to six other nodes, to form a communication network that follows the plan of a six-dimensional hypercube, what we call a *binary 6-cube* (see Figure 1). An operating system kernel in each node schedules and runs processes within that node, provides system calls for processes to send and receive messages, and routes the messages that flow through the node.

The excellent performance of the Cosmic Cube on a variety of complex and demanding applications and its modest cost and open-ended expandability suggest that highly concurrent systems of this type are an effective means of achieving faster and less expensive computing in the near future. The Cosmic Cube nodes were designed as a hardware simulation of what we expect to be able to integrate onto one or two chips in about five years. Future machines with thousands of nodes are feasible, and for many demanding computing problems, these machines should be able to outperform the fastest uniprocessor systems. Even with current microelectronic technology, the 64-node machine is quite powerful for its cost and size: It can handle a variety of demanding scientific and engineering computations five to ten times faster than a VAX11/780.

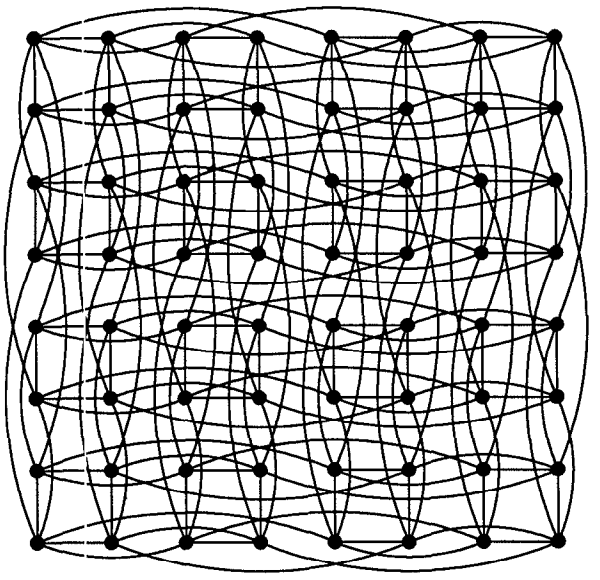
The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, ARPA order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597.

THE MESSAGE-PASSING ARCHITECTURE

A significant difference between the Cosmic Cube and most other parallel processors is that this multiple-instruction multiple-data machine uses *message passing* instead of shared variables for communication between concurrent processes. This computational model is reflected in the hardware structure and operating system, and is also the explicit communication and synchronization primitive seen by the programmer.

The hardware structure of a message-passing machine like the Cosmic Cube differs from a shared-storage multiprocessor in that it employs no switching network between processors and storage (see Figure 2). The advantage of this architecture is in the separation of engineering concerns into processor-storage communication and the interprocess communication. The critical path in the communication between an instruction processor and its random-access storage, the so-called von Neumann bottleneck, can be engineered to exhibit a much smaller latency when the processor and storage are physically localized. The processor and storage might occupy a single chip, hybrid package, or circuit board, depending on the technology and complexity of the node.

It was a premise of the Cosmic Cube experiment that the internode communication should scale well to very large numbers of nodes. A *direct* network like the hypercube satisfies this requirement, with respect to both the aggregate bandwidth achieved across the many concurrent communication channels and the feasibility of the implementation. The hypercube is actually a distributed variant of an *indirect* logarithmic switching network like the Omega or banyan networks: the kind that might be used in shared-storage organizations. With the hypercube, however, communication paths traverse different numbers of channels and so exhibit different latencies. It is possible, therefore, to take advantage of communication locality in placing processes in nodes.



A hypercube connects $N = 2^n$ small computers, called nodes, through point-to-point communication channels in the Cosmic Cube. Shown here is a two-dimensional projection of a six-dimensional hypercube, or binary 6-cube, which corresponds to a 64-node machine.

FIGURE 1. A Hypercube (also known as a binary cube or a Boolean *n*-cube)

Message-passing machines are simpler and more economical than shared-storage machines; the greater the number of processors, the greater this advantage. However, the more tightly coupled shared-storage machine is more versatile, since it is able to support code and data sharing. Indeed, shared-storage machines can easily simulate message-passing primitives, whereas message-passing machines do not efficiently support code and data sharing.

Figure 2 emphasizes the differences between shared-storage and message-passing organizations by representing the extreme cases. We conjecture that shared-storage organizations will be preferred for systems with tens of processors, and message-passing organizations for systems with hundreds or thousands of processing nodes. Hybrid forms employing local or cache storage with each processor, with a message-passing approach to nonlocal storage references and cache coherence, may well prove to be the most attractive option for systems having intermediate numbers of processors.

PROCESS PROGRAMMING

The hardware structure of the Cosmic Cube, when viewed at the level of nodes and channels, is a difficult target for programming any but the most highly regular computing problems. The resident operating system of the Cosmic Cube creates a more flexible and machine-independent environment for concurrent computations. This process model of computation is quite similar to the hardware structure of the Cosmic Cube but is usefully abstracted from it. Instead of formulating a problem to fit on nodes and on the physical communication channels that exist only between certain pairs of nodes, the programmer can formulate problems in terms of processes and "virtual" communication channels between processes.

The basic unit of these computations is the *process*, which for our purposes is a sequential program that sends and receives messages. A single node may contain many processes. All processes execute concurrently, whether by virtue of being in different nodes or by being interleaved in execution within a single node. Each process has a unique (global) ID that serves as an address for messages. All messages have headers containing the destination and the sender ID, and a message type and length. Messages are queued in transit, but message order is preserved between any pair of processes. The semantics of the message-passing operations are independent of the placement of processes in nodes.

Process programming environments with interprocess communication by messages are common to many multiprogramming operating systems. A copy of the resident operating system of the Cosmic Cube, called the "kernel," resides in each node. All of these copies are concurrently executable. The kernel can spawn and

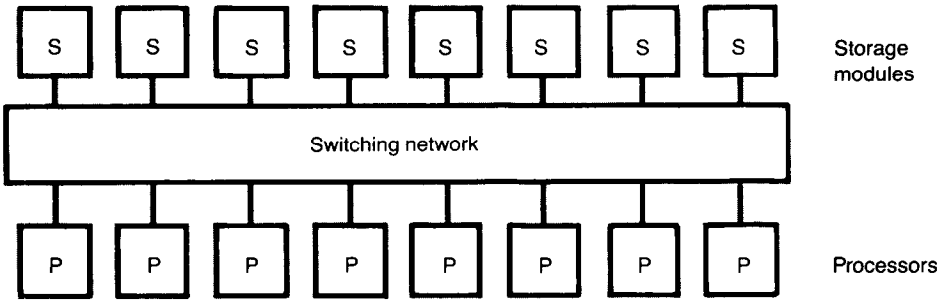
kill processes within its own node, schedule their execution, spy on them through a debug process, manage storage, and deal with error conditions. The kernel also handles the queuing and routing of messages for processes in its node, as well as for messages that may pass through its node. Many of the functions that we would expect to be done in hardware in a future integrated node, such as message routing, are done in the kernel in the Cosmic Cube. We are thus able to experiment with different algorithms and implementations of low-level node functions in the kernel.

The Cosmic Cube has no special programming notation. Process code is written in ordinary sequential programming languages (e.g., Pascal or C) extended with statements or external procedures to control the sending and receiving of messages. These programs are compiled on other computers and loaded into and relocated within a node as binary code, data, and stack segments.

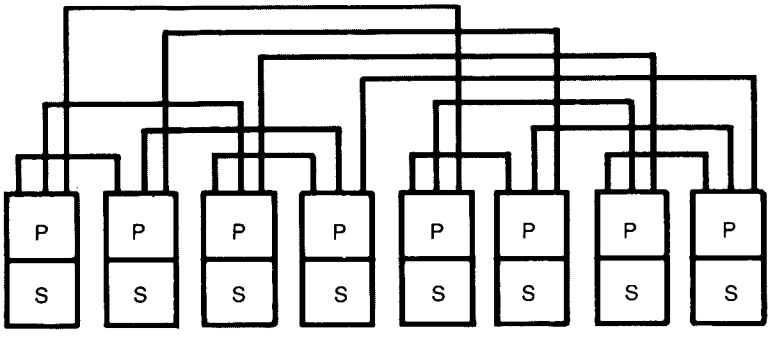
PROCESS DISTRIBUTION

It was a deliberate decision in the design of the kernel that once a process was instantiated in a node, the kernel would not relocate it to another node. One consequence of this restriction is that the physical node number can be included in the ID for a process. This eliminates the awkward way in which a distributed map from processes to nodes would otherwise scale with the number of nodes. Messages are routed according to the physical address part of the destination process ID in the message header.

This decision was also consistent with the notion that programmers should be able to control the distribution of processes onto the nodes on the basis of an understanding of the structure of the concurrent computation being performed. Alternatively, since it is only the efficiency of a multiple-process program that is influenced by process placement, the choice of the node in which



(a) Most multiprocessors are structured with a switching network, either a crossbar connection of buses or a multistage routing network, between the processors and storage. The switching network introduces a latency in the communication between processors and storage, and does not scale well to large sizes. Communication between processes running concurrently in different processors occurs through shared variables and common access to one large address space.



(b) Message-passing multicomputer systems retain a physically close and fast connection between processors and their associated storage. The concurrent computers (nodes) can send messages through a network of communication channels. The network shown here is a three-dimensional cube, which is a small version of the communication plan used in six dimensions in the 64-node Cosmic Cube.

NOTE: Actual machines need not follow one model or the other absolutely: Various hybrids are possible.

FIGURE 2. A Comparison of Shared-Storage Multiprocessors and Message-Passing Machines

a process is to be spawned can be deferred to a library process that makes this assignment after inquiring about processing load and storage utilization in nearby nodes.

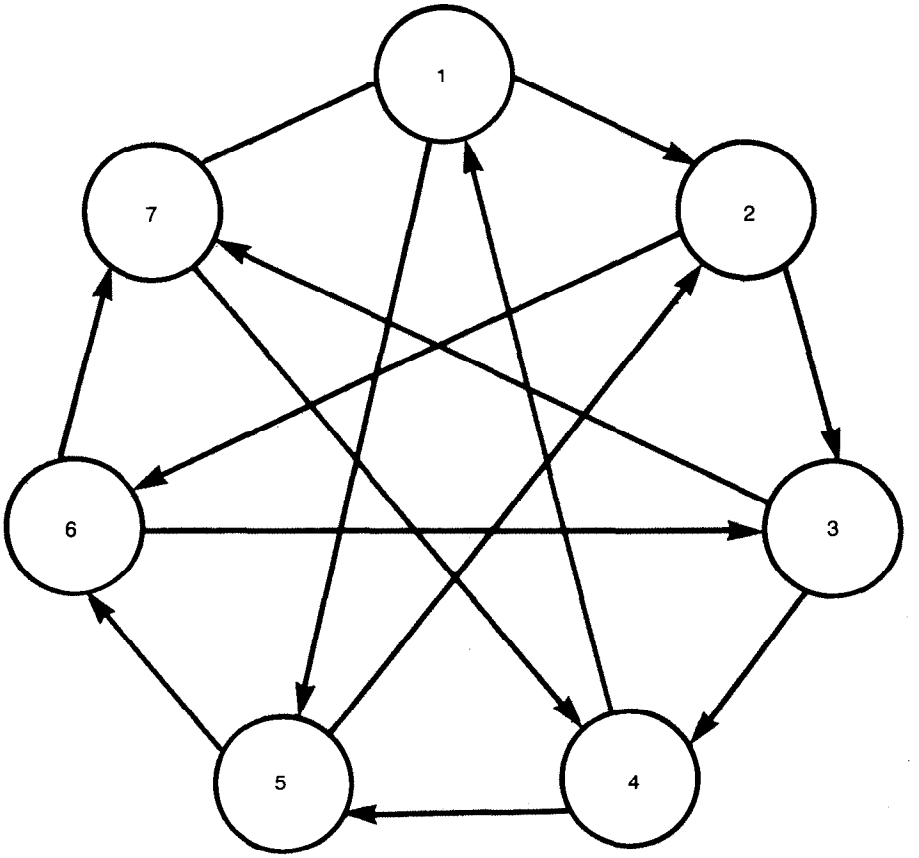
A careful distribution of processes to nodes generally involves some trade-offs between load balancing and message locality. We use the term *process structure* to describe a set of processes and their references to other processes. A static process structure, or a snapshot of a dynamic process structure, can be represented as a graph of process vertices connected by arcs that represent reference (see Figure 3). One can also think of the arcs as virtual communication channels, in that process A having reference to process B is what makes a message from A to B possible.

The hardware communication structure of this class of message-passing machines can be represented similarly as a graph of vertices for nodes and (undirected) edges for the bidirectional communication channels. The mapping of a process structure onto a machine is an embedding of the process structure graph into the machine graph (see Figure 4). In general, the arcs map not only to internal communication and single edges, but also to paths representing the routing of messages in intermediate nodes. It is this embedding that determines both the locality of communication achieved and the load-balancing properties of the mapping.

CONCURRENCY APPROACH

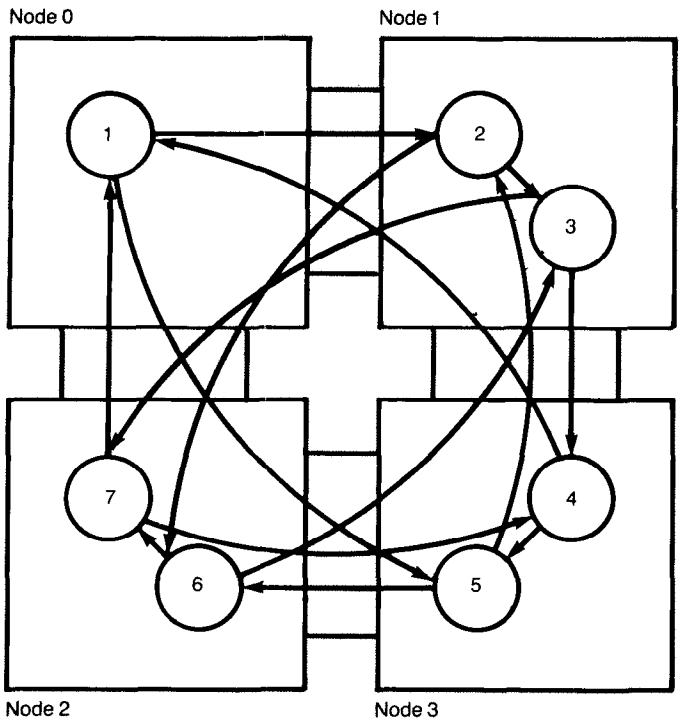
Most sequential processors, including microprocessors like the RISC chips described elsewhere in this issue,[1] are *covertly* concurrent machines that speed up the interpretation of a single instruction stream by techniques such as instruction prefetching and execution pipelining. Compilers can assist this speedup by recovering the concurrency in expression evaluations and

[1] See David A. Patterson's article, "Reduced Instruction Set Computers" (*Commun. ACM 28*, 1 (Jan. 1985)), on pages 8–21 of this issue.



In this example, the processes are computing the time evolution, or orbital positions, of seven bodies interacting by a symmetrical force, such as gravity. Messages containing the position and mass of each particle are sent from each process $(N - 1)/2$ steps around the ring, accumulating the forces due to each interaction along the way, while the process that is host to that body accumulates the other $(N - 1)/2$ forces. The messages are then returned over the chordal paths to the host process, where the forces are summed and the position and velocity of the body are updated. This example is representative of many computations that are demanding simply because of the number of interacting parts, and not because the force law that each part obeys is complex. However, this is not the formulation one would use for very many bodies.

FIGURE 3. The Process Structure for a Concurrent Formulation of the *N*-Body Problem



The distribution of the processes does not influence the computed results, but it does, through load balancing and message locality, influence the speedup achieved by using four computers for this task instead of one.

FIGURE 4. The Process Structure for the 7-Body Example Embedded into a 4-Node Machine

in the innermost iterations of a program, and then generating code that is "vectorized" or in some other way allows the processor to interpret the sequential program with some concurrency. These techniques, together with caching, allow about a ten-fold concurrency and speedup over naive sequential interpretation.

We can use such techniques within nodes, where we are tied to sequential program representations of the processes. In addition, we want to have at least as many concurrent processes as nodes. Where are such large degrees of concurrency to be discovered in a computation? One quick but not quite accurate way of describing the approach used in the Cosmic Cube is that we *overtly* exploit the concurrency found in the outermost, rather than innermost, program constructs of certain demanding computations. It appears that many highly demanding computing problems can be expressed in terms of concurrent processes with either sparse or predictable interaction. Also, the degree of concurrency inherent in such problems tends to grow with the size and computing demands of the problem.

It is important to understand that the compilers used to generate process code for the Cosmic Cube do not "automatically" find a way to run sequential programs concurrently. We do not know how to write a program that translates application programs represented by old, dusty FORTRAN decks into programs that exploit concurrency between nodes. In fact, because efficient concurrent algorithms may be quite different from their sequential counterparts, we regard such a translation as implausible, and instead try to formulate and express a computation explicitly in terms of a collection of communicating concurrent processes.

Dataflow graphs, like those discussed in this issue in the article on the Manchester dataflow machine,[2] also allow an explicit representation of concurrency in a computation. Although we have not yet tried to do so, dataflow computations can be executed on machines like the Cosmic Cube. One of the reasons we have not done so is that many of the computations that show excellent performance on the Cosmic Cube or on other parallel machines, and are very naturally expressed in terms of processes (or objects), are simulations of physical systems. With such simulations, the state of a system is repeatedly evaluated and assigned to state variables. The functional (side-effect free) semantics of dataflow, in pure form, appears to get in the way of a straightforward expression of this type of computation. The process model that we use for programming the Cosmic Cube is relatively less restrictive than dataflow and, in our implementation, is relatively more demanding of attention to process placement and other details.

CONCURRENT FORMULATIONS

The crucial step in developing an application program for the Cosmic Cube is the concurrent formulation: It is here that both the correctness and efficiency of the program are determined. It is often intriguing, and even amusing, to devise strategies for coordinating a myriad of concurrent computing activities in an orderly way.

For many of the demanding computations encountered in science and engineering, this formulation task has not proved to be very much more difficult than it is on sequential machines. These applications are often based on concurrent adaptations of well-known sequential algorithms or are similar to the systolic algorithms that have been developed for regular VLSI computational arrays. The process structure remains static for the duration of a computation.

At the risk of creating the impression that all of the application programs for the Cosmic Cube are as simple, let us offer one concrete example of a formulation and its process code. The problem is to compute the time evolution of a system of N bodies that interact by gravitational attraction or some other symmetrical force. Because each of the N bodies interacts with all of the other $N - 1$ bodies, this problem might not seem to be as appropriate for the Cosmic Cube as matrix, grid-point, finite-difference, and other problems based solely

[2] See J.R. Gurd, C.C. Kirkham, and I. Watson's article, "The Manchester Prototype Dataflow Computer" (*Commun. ACM 28*, 1 (Jan. 1985)), on pages 34–52 of this issue.

on local interaction. Actually, universal interaction is easy because it maps beautifully onto the ring process structure shown for $N = 7$ in Figure 3.

Each of N identical processes is "host" to one body and is responsible for computing the forces due to $(N - 1)/2$ other bodies. With a symmetrical force, it is left to other processes to compute the other $(N - 1)/2$ forces. The process also accumulates the forces and integrates the position of the body it hosts. As can be seen in the C process code in Figure 5, the process that is host to body 1 successively receives guests 7, 6, and 5, and accumulates forces due to these interactions. Meanwhile, a message containing the position, mass, accumulated force, and host process ID of body 1 is con-

```
/* process for an n-body computation, n odd, with symmetrical forces */
#include "cubedef.h"        /* cube definitions */
#include "force.h"          /* procedures for computing forces and positions */

struct body     {   double pos[3];          /* body position x,y,z          */
                    double vel[3];          /* velocity vector x,y,z        */
                    double force[3];        /* to accumulate forces         */
                    double mass;            /* body mass                    */
                    int    home_id;         /* id of body's home process    */
                } host, guest;

struct startup {    int n;                  /* number of bodies             */
                    int next_id;            /* ID of next process on ring   */
                    int steps;              /* number of integration steps  */
                } s;

struct desc my_body_in, my_body_out, startup_in; /* IH channels             */
struct desc body_in, body_out, body_bak;         /* inter-process channels  */

cycle() /* read initial state, compute, and send back final state */
{
    int i; double FORCE[3];

    /* initialize channel descriptors */
    /* init(*desc, id, type, buffer_len, buffer_address); */
    init(&my_body_in ,0,0,sizeof(struct body)/2,&host); recv_wait(&my_body_in);
    init(&startup_in ,0,1,sizeof(struct startup)/2,&s); recv_wait(&startup_in);
    init(&my_body_out,     IH_ID, 2, sizeof(struct body)/2, &host);
    init(&body_in    ,          0, 3, sizeof(struct body)/2, &guest);
    init(&body_out   , s.next_id, 3, sizeof(struct body)/2, &guest);
    init(&body_bak   ,          0, 4, sizeof(struct body)/2, &guest);

    while(s.steps--) /* repeat s.steps computation cycles */
    {
        body_out.buf = &host;               /* first time send out host body */

        for(i = (s.n-1)/2; i--;)            /* repeat (s.n-1)/2 times       */
        {
            send_wait(&body_out);           /* send out the host|guest      */
            recv_wait(&body_in);            /* receive the next guest       */
            COMPUTE_FORCE(&host,&guest,FORCE); /* calculate force           */
            ADD_FORCE_TO_HOST(&host,FORCE); /* may the force be with you    */
            ADD_FORCE_TO_GUEST(&guest,FORCE); /* and with the guest, also   */
            body_out.buf = &guest;          /* prepare to pass the guest    */
        }
        body_bak.id = guest.home_id;                /* send guest back      */
        send_wait(&body_bak); recv_wait(&body_bak); /* the envoy returns    */
        ADD_GUEST_FORCE_TO_HOST(&host,&guest);
        UPDATE(&host);                              /* integrate position   */
    }
    send_wait(&my_body_out);  /* send body back to host, complete one cycle  */
}

main() { while(1) cycle(); }  /* main execute cycle repeatedly */
```

FIGURE 5. Process Code for the *N*-Body Example in the C Language

veyed through the processes that are host to bodies 2, 3, and 4 with the forces due to these interactions accumulated. After $(N-1)/2$ visits, the representations of the bodies are returned in a message to the process that is host to the body, the forces are combined, and the positions are updated.

A detail that is not shown in Figure 5 is the process that runs in the Cosmic Cube intermediate host (IH), or on another network-connected machine. This process spawns the processes in the cube and sends messages to the cube processes that provide the initial state, the ID of the next process in the ring, and an integer specifying the number of integration steps to be performed. The computation in the Cosmic Cube can run autonomously for long periods between interactions with the IH process. If some exceptional condition were to occur in the simulation, such as a collision or close encounter, the procedure that computes the forces could report this event via a message back to the IH process.

This ring of processes can, in turn, be embedded systematically into the machine structure (see Figure 4). In mapping seven identical processes, each with the same amount of work to do, onto 4 nodes, the load obviously cannot be balanced perfectly. Using a simple performance model originally suggested by Willis Ware, "speedup"—S—can be defined as

$$S = \frac{\text{time on 1 node}}{\text{time on } N \text{ nodes}}.$$

For this 7-body example on a 4-node machine, neglecting the time required for the communication between nodes, the speedup is clearly $7/2$. Since computation proceeds 3.5 times faster using 4 nodes than it would on a single node, one can also say that the efficiency $e = S/N$ is 0.875, which is the fraction of the available cycles that are actually used.

More generally, if k is taken as the fraction of the steps in a computation that, because of dependencies, must be sequential, the time on N nodes is $\max(k, 1/N)$, so that the speedup cannot exceed $\min(1/k, N)$. This expression reduces to "Amdahl's argument," that $1/k$, the reciprocal of the fraction of the computation that must be done sequentially, limits the number of nodes that can usefully be put to work concurrently on a given problem. For example, nothing is gained in *this* formulation of an N-body problem by using more than N nodes.

Thus we are primarily interested in computations for which $1/k \gg N$: in effect, in computations in which the concurrency opportunities exceed the concurrent resources. Here the speedup obtained by using N nodes concurrently is limited by (1) the idle time that results from imperfect load balancing, (2) the waiting time caused by communication latencies in the channels and in the message forwarding, and (3) the processor time dedicated to processing and forwarding messages, a consideration that can be effectively eliminated by architectural improvements in the nodes. These factors are rather complex functions of the formulation, its mapping onto N nodes, the communication latency, and the communication and computing speed of the nodes. We lump these factors into an "overhead" measure, σ, defined by the computation exhibiting a speedup of $S = N/(1+\sigma)$. A small σ indicates that the Cosmic Cube is operating with high efficiency, that is, with nodes that are seldom idle, or seldom doing work they would not be doing in the single-node version of the computation.

COSMIC CUBE HARDWARE

Having introduced the architecture, computational model, and concurrent formulations, let us turn now to some experimental results.

Figure 6 is a photograph of the 64-node Cosmic Cube. For such a small machine, only 5 feet long, a one-dimensional projection of the six-dimensional hypercube is satisfactory. The channels are wired on a backplane beneath the long box in a pattern similar to that shown in Figure 2b. Larger machines would have nodes arrayed in two or three dimensions like the two-dimensional projection of the channels shown in Figure 1. The volume of the system is 6 cubic feet, the power consumption is 700 watts, and the manufacturing cost was $80,000. We also operate a 3-cube machine to support software development, since the 6-cube cannot readily be shared.

Most of the choices made in this design are fairly easy to explain. First of all, a binary n-cube communication plan was used because this network was shown by simulation to provide very good message-flow properties in irregular computations. It also contains all meshes of lower dimension, which is useful for regular mesh-connected problems. The binary n-cube can be viewed recursively. As one can see from studying Figure 1, the n-cube that is used to connect $2^n = N$ nodes is assembled from two $(n-1)$-cubes, with corresponding nodes connected by an additional channel. This property simplifies the packaging of machines of varying size. It also explains some of the excellent message-flow properties of the binary n-cube on irregular problems. The number of channels connecting the pairs of subcubes is proportional to the number of nodes and hence on average to the amount of message traffic they can generate.

With this rich connection scheme, simulation showed that we could use channels that are fairly slow (about 2 Mbit/sec) compared with the instruction rate. The communication latency is, in fact, deliberately large to make this node more nearly a hardware simulation of the situation anticipated for a single-chip node. The processor overhead for dealing with each 64-bit packet is comparable to its latency. The communication channels are asynchronous, full duplex, and include queues for a 64-bit "hardware packet" in the sender and in the receiver in each direction. These queues are a basic minimum necessary for decoupling the sending and receiving processes.

The Intel 8086 was selected as the instruction processor because it was the only single-chip instruction proc-

The nodes are packaged as one circuit board per node in the long card frame on the bench top. The six communication channels from each node are wired in a binary 6-cube on the backplane on the underside of the card frame. The separate units on the shelf above the long 6-cube box are the power supply and an "intermediate host" (IH) that connects through a communication channel to node 0 in the cube.

FIGURE 6. The 64-Node Cosmic Cube in Operation

essor available with a floating-point coprocessor, the Intel 8087. Reasonable floating-point performance was necessary for many of the applications that our colleagues at Caltech wished to attempt. The system currently operates at a 5 MHz clock rate, limited by the 8087, although it is designed to be able to run at 8 MHz when faster 8087 chips become available. After our first prototypes, Intel Corporation generously donated chips for the 64-node Cosmic Cube.

The storage size of 128K bytes was decided upon after a great deal of internal discussion about "balance" in the design. It was argued that the cost incurred in doubling the storage size would better be spent on more nodes. In fact, this choice is clearly very dependent on target applications and programming style. The dynamic RAM includes parity checking but not error correction. Each node also includes 8 Kbytes of read-only storage for initialization, a bootstrap loader, dynamic RAM refresh, and diagnostic testing programs.

Since building a machine is not a very common enterprise in a university, an account of the chronology of the hardware phase of the project may be of interest. A prototype 4-node (2-cube) system on wirewrap boards was designed, assembled, and tested in the winter of 1981–1982, and was used for software development and application programs until it was recently disassembled. The homogeneous structure of these machines was nicely exploited in the project when a small hardware prototype, similar to scaled-up machines, was used to accelerate software development. Encouraged by our experience with the 2-cube prototype, we had printed circuit boards designed and went through the other packaging logistics of assembling a machine of useful size. The Cosmic Cube grew from an 8-node to a

64-node machine over the summer of 1983 and has been in use since October 1983.

In its first year of operation (560,000 node-hours), the Cosmic Cube has experienced two hard failures, both quickly repaired; a soft error in the RAM is detected by a parity error on average once every several days.

COSMIC CUBE SOFTWARE

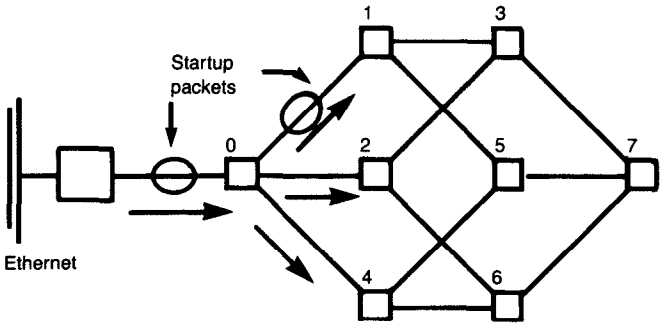
As is the case in many "hardware" projects, most of the work on the Cosmic Cube has been on the software. This effort has been considerably simplified by the availability of cross-compilers for the Intel 8086/8087 chips and because most of the software development is done on conventional computers. Programs are written and compiled in familiar computing environments, and their concurrent execution is then simulated on a small scale. Programs are downloaded into the cube through a connection managed by the intermediate host. In the interest of revealing all of the operational details of this unconventional machine, we begin with the start-up procedures.

The lowest level of software is part of what we call the *machine intrinsic* environment. This includes the instruction set of the node processor, its I/O communication with channels, and a small initialization and bootstrap loader program stored along with diagnostic programs in read-only storage in each processor. A start-up packet specifies the size of the cube to be initialized and may specify that the built-in RAM tests be run (concurrently) in the nodes. As part of the initialization process, each of the identical nodes discovers its position in whatever size cube was specified in the start-up packet sent from the intermediate host by sending messages to the other nodes. The initialization, illustrated in Figure 7, also involves messages that check the function of all of the communication channels to be used. Program loading following initialization typically loads the kernel.

A *crystalline* applications environment is characterized by programs written in C in which there is a single process per node and in which messages are sent by direct I/O operations to a specified channel. This system was developed by physics users for producing very efficient application programs for computations so regular they do not require message routing.

The operating system kernel, already described in outline, supports a *distributed process* environment with a copy of the kernel running in each node. The kernel is 9 Kbytes of code and 4 Kbytes of tables, and is divided into an "inner" and an "outer" kernel. Any storage in a node that is not used for the kernel or for processes is allocated as a kernel message buffer for queuing messages.

The inner kernel, written in 8086 assembly language, sends and receives messages in response to system calls from user processes. These calls pass the address of a message descriptor, which is shared between the kernel and user process. There is one uniform message format that hides all hardware characteristics, such as packet



In the initialization, each of the identical nodes discovers its identity and checks all the communication channels with a message wave that traverses the 3-cube from node 0 to node 7, and then from node 7 to node 0. If node 3, for instance, did not respond to messages, then nodes 1, 2, and 7 would report this failure back to the host over other channels.

FIGURE 7. The Initialization of the Cosmic Cube

size. The kernel performs the construction and interpretation of message headers from the descriptor information. The hardware communication channels allow very fast and efficient "one-trip" message protocols; long messages are automatically fragmented. Messages being sent are queued in the sending process instead of being copied into the kernel message buffer, unless the message is local to the node. Local messages are either copied to the destination if the matching receive call has already been executed, or copied into the message buffer to assure a consistency in the semantics of local and nonlocal send operations.

Processes are often required to manage several concurrent message activities. Thus the send and receive calls do not "block." The calls return after creating a request that remains *pending* until the operation is completed. The completion of the message operation is tested by a lock variable in the message descriptor. Program execution can continue concurrently with many concurrently pending communication activities. A process can also use a *probe* call that determines whether a message of a specified type has been received and is queued in the kernel message buffer. A process that is in a situation where no progress can be made until some set of message areas is filled or emptied may elect to defer execution to another process. The inner kernel schedules user processes by a simple round robin scheme, with processes running for a fixed period of time or until they perform the system call that defers to the next process. The storage management and response to error conditions are conventional.

The outer kernel is structured as a set of privileged processes that user processes communicate with by messages rather than by system calls. One of these outer kernel processes spawns and kills processes: A process can be spawned either as a copy of a process already present in the node, in which case the code

segment is shared, or from a file that is accessed by system messages between the spawn process and the intermediate host. Because process spawning is invoked by messages, it is equally possible to build process structures from processes running in the cube, in the intermediate host, or in network-connected machines. One other essential outer kernel process is known as the *spy* process and permits a process in the intermediate host to examine and modify the kernel's tables, queued messages, and process segments.

Our current efforts are focused on intermediate host software that will allow both time- and space-sharing of the cube.

APPLICATIONS AND BENCHMARKS

Caltech scientists in high-energy physics, astrophysics, quantum chemistry, fluid mechanics, structural mechanics, seismology, and computer science are developing concurrent application programs to run on Cosmic Cubes. Several research papers on scientific results have already been published, and other applications are developing rapidly. Several of us in the Caltech computer science department are involved in this research both as system builders and also through interests in concurrent computing and applications to VLSI analysis tools and graphics.

Application programs on the 64-node Cosmic Cube execute up to 3 million floating-point operations per second. The more interesting and revealing benchmarks are those for problems that utilize the machine at less than peak speeds. A single Cosmic Cube node at a 5 MHz clock rate runs at one-sixth the speed of the same program compiled and run on a VAX11/780. Thus we should expect the 64-node Cosmic Cube to run at best $(1/6)(64) \approx 10$ times faster than the VAX11/780. Quite remarkably, many programs reach this performance, with measured values of σ ranging from about 0.025 to 0.500. For example, a typical computation with $\sigma = 0.2$ exhibits a speedup $S = (64)/(1.2) \approx 50$. One should not conclude that applications with larger σ are unreasonable; indeed, given the economy of these machines, it is still attractive to run production programs with $\sigma > 1$.

A lattice computation programmed by physics postdoc Steve Otto at Caltech has run for an accumulated 2500 hours on the 6-cube. This program is a Monte Carlo simulation on a $12 \times 12 \times 12 \times 16$ lattice, an investigation of the predictions of quantum chromodynamics, which is a theory that explains the substructure of particles such as protons in terms of quarks and the glue field that holds them bound. Otto has shown for the first time in a single computation both the short-range Coulombic force and the constant long-range force between quarks. The communication overhead in this naturally load balanced computation varies from $\sigma = 0.025$ in the phase of computing the gauge field to $\sigma = 0.050$ in computing observables by a contour integration in the lattice.

Among the most interesting and ambitious programs currently in development is a concurrent MOS-VLSI circuit simulator, called CONCISE, formulated and written by computer science graduate student Sven Mattisson. This program has been useful for developing techniques for less regular computations and promises very good performance for a computation that consumes large fractions of the computing cycles on many high-performance computers.

The simulation of an electrical circuit involves repeated solution of a set of simultaneous nonlinear equations. The usual approach, illustrated in Figure 8, is to compute piecewise linear admittances from the circuit models and then to use linear equation solution techniques. CONCISE uses a nodal admittance matrix formulation for the electrical network. The admittance matrix is sparse but, because electrical networks have arbitrary topology, does not have the crystalline regularity of the physics computations. At best the matrix is "clumped" because of the locality properties of the electrical network.

This program is mapped onto the cube by partitioning the admittance matrix by rows into concurrent processes. The linear equation solution phase of the computation, a Jacobi iteration, involves considerable communication, but the linearization that requires about 80 percent of the execution time on sequential computers is completely uncoupled. Integration and output in computing transient solutions are small components of the whole computation. The computation is actually much more complex than we can describe here; for example, the integration step is determined adaptively from the convergence of previous solutions.

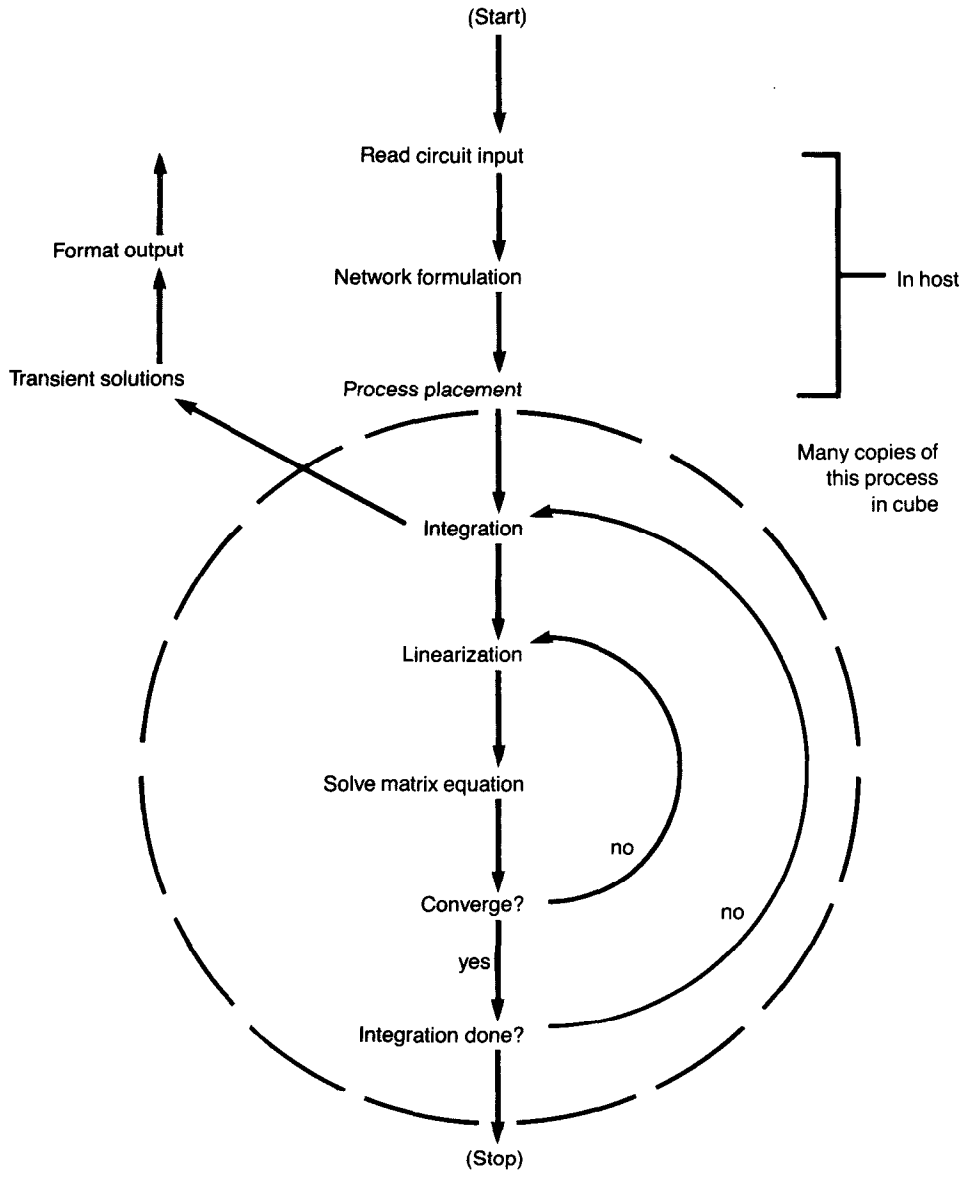
Among the many unknowns in experimenting with circuit simulation is the interaction between communication cost and load balancing in the mapping of processes to nodes. Although "clumping" can be exploited in this mapping to localize communication, it may also concentrate many of the longer iterations occurring during a signal transient into a single node, thus creating a "dynamic" load imbalance in the computation.

FUTURE PERFECT CUBES

Today's system is never as perfect as tomorrow's. Although software can be polished and fixed on a daily basis, the learning cycle on the architecture and hardware is much longer. Let us then summarize briefly what this experiment has taught us so far and speculate about future systems of this same general class.

Although programming has not turned out to be as difficult as we should have expected, we do have a long agenda of possible improvements for the programming tools. Most of the deficiencies are in the representation and compilation of process code. There is nothing in the definition of the message-passing primitives that we would want to change, but because we have tacked these primitives onto programming languages simply as external functions, the process code is unnecessarily baroque.

The way the descriptors for "virtual channels" are



The sequential and concurrent versions of this program differ in that the concurrent program employs many copies of the process depicted inside the dashed circle.

FIGURE 8. The Organization of the CONCISE Circuit Simulator

declared, initialized, and manipulated (see Figure 5), for instance, is not disguised by a pretty syntax. More fundamentally, the attention the programmer must give to blocking on lock variables is tedious and can create incorrect or unnecessary constraints on message and program sequencing. Such tests are better inserted into the process code automatically, on the basis of a data-flow analysis similar to that used by optimizing compilers for register allocation. These improvements may be only aesthetic, but they are a necessary preliminary for making these systems less intimidating for the beginning user.

The cost/performance ratio of this class of architectures is quite good even with today's technologies, and progress in microelectronics will translate into either increased performance or decreased cost. The present Cosmic Cube node is not a large increment in complexity over the million-bit storage chips that are expected in a few years. Systems of 64 single-chip node elements could fit in workstations, and systems of thousands of nodes would make interesting supercomputers. Although this approach to high-performance computation is limited to applications that have highly concurrent formulations, the applications developed on the Cosmic

Cube have shown us that many, perhaps even a majority, of the large and demanding computations in science and engineering are of this type.

It is also reasonable to consider systems with nodes that are either larger or smaller than the present Cosmic Cube nodes. We have developed at Caltech a single-chip "Mosaic" node with the same basic structure as the Cosmic Cube node, but with less storage, for experimenting with the engineering of systems of single-chip nodes and with the programming and application of finer grain machines. Such machines offer a cost/performance ratio superior even to that of the Cosmic Cube. However, we expect them to be useful for a somewhat smaller class of problems. Similarly, the use of better, faster instruction processors, higher capacity storage chips, and integrated communication channels suggests machines with nodes that will be an order of magnitude beyond the Cosmic Cube in performance and storage capacity, but at the same physical size.

The present applications of the Cosmic Cube are all compute- rather than I/O-intensive. It is possible, however, to include I/O channels with each node and so to create sufficient I/O band width for almost any purpose. Such machines could be used, for example, with many sensors, such as the microphone arrays towed behind seismic exploration ships. The computing could be done in real time instead of through hundreds of tapes sent on to a supercomputer. It is also possible to attach disks for secondary storage to a subset of the nodes.

APPENDIX—HISTORY AND ACKNOWLEDGMENTS

The origins of the Cosmic Cube project can be traced to research performed at Caltech during 1978–1980 by graduate students Sally Browning and Bart Locanthi [1, 8]. These ideas were in turn very much influenced by several other researchers. We sometimes refer to the Cosmic Cube as a *homogeneous machine*, from a term used in a 1977 paper by Herbert Sullivan and T.L. Brashkow [13]. They define their homogeneous machine as a machine "of uniform structure." C.A.R. Hoare's communicating sequential processes notation, the actor paradigm developed by Carl Hewitt, the processing surface experiments of Alain Martin, and the systolic algorithms described by H.T. Kung, Charles Leiserson, and Clark Thompson encouraged us to consider message passing as an explicit computational primitive [2, 4, 6, 10].

The Cosmic Cube design is based in largest part on extensive program modeling and simulations carried out during 1980–1982 by Charles R. Lang [7]. It was from this work that the communication plan of a binary n-cube, the bit rates of the communication channels, and the organization of the operating system primitives were chosen. Together with early simulation results, a workshop on "homogeneous machines" organized by Carl Hewitt during the summer of 1981 helped give us the confidence to start building an experimental machine.

The logical design of the Cosmic Cube is the work of computer science graduate students Erik DeBenedictis and Bill Athas. The early crystalline software tools were developed by physics graduate students Eugene Brooks and Mark Johnson. The machine intrinsic and kernel code was written by Bill Athas, Reese Faucette, and Mike Newton, with Alain Martin, Craig Steele, Jan van de Snepscheut, and Wen-King Su contributing valuable critical reviews of the design and implementation of the distributed process environment.

The ongoing work described in part in this article is sponsored through the VLSI program of the Information Processing Techniques Office of DARPA. We thank Bob Kahn, Duane Adams, and Paul Losleben for their support and interest.

REFERENCES

1. Browning, S.A. The tree machine: A highly concurrent computing environment. Tech. Rep. 3760:TR:80, Computer Science Dept., California Institute of Technology, Pasadena, 1980.
2. Clinger, W.D. Foundations of actor semantics. Ph.D. thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, May 1981.
3. Fox, G.C., and Otto, S.W. Algorithms for concurrent processors. *Phys. Today* 37, 5 (May 1984), 50–59.
4. Hoare, C.A.R. Communicating sequential processes. *Commun. ACM* 21, 8 (Aug. 1978), 666–677.
5. Hockney, R.W., and Jesshope, C.R. *Parallel Computers*. Adam Hilger, Bristol, United Kingdom, 1981.
6. Kung, H.T. The Structure of Parallel Algorithms. In *Advances in Computers*, vol 19. Academic Press, New York, 1980.
7. Lang, C.R. The extension of object-oriented languages to a homogeneous, concurrent architecture. Tech. Rep. 5014:TR:82, Computer Science Dept., California Institute of Technology, Pasadena, 1982.
8. Locanthi, B.N. The homogeneous machine, Tech. Rep. 3759:TR:80, Computer Science Dept., California Institute of Technology, Pasadena, 1980.
9. Lutz, C., Rabin, S., Seitz, C., and Speck, D. Design of the Mosaic Element. In *Proceedings of the Conference on Advanced Research in VLSI* (MIT), P. Penfield, Ed, Artech House, Dedham, Mass., 1984, pp. 1–10.
10. Martin, A.J. A distributed implementation method for parallel programming. *Inf. Process. 80* (1980), 309–314.
11. Schwartz, J.T. Ultracomputers. *ACM Trans. Program. Lang. Syst.* 2, 4 (Oct. 1980), 484–521.
12. Seitz, C.L. Experiments with VLSI ensemble machines. *J. VLSI Comput. Syst.* 1, 3 (1984).
13. Sullivan, H., and Brashkow, T.R. A large scale homogeneous machine I & II. In *Proceedings of the 4th Annual Symposium on Computer Architecture*, 1977, pp. 105–124.
14. Ware, W.H. The ultimate computer. *IEEE Spectrum* (Mar. 1972), 84–91.

CR Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors); C.5.4 [**Computer System Implementation**]: VLSI Systems; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.4.1 [**Operating Systems**]: Process Management

General Terms: Algorithms, Design, Experimentation

Additional Key Words and Phrases: highly concurrent computing, message-passing architectures, message-based operating systems, process programming, object-oriented programming, VLSI systems

Author's Present Address: Charles L. Seitz, Computer Science 256-80, California Institute of Technology, Pasadena, CA 91125.