SUBMICRON SYSTEMS ARCHITECTURE
SEMIANNUAL TECHNICAL REPORT

Sponsored by
Defense Advanced Research Projects Agency
ARPA Order Number 3771

Monitored by the
Office of Naval Research
Contract Number N00014-79-C-0597

5202:TR:85

Computer Science Department
California Institute of Technology

September 1985

Submicron Systems Architecture

Semiannual Technical Report

Department of Computer Science
California Institute of Technology

5202:TR:85

September 1985

Reporting Period:	16 March 1985 to 15 September 1985
Principal Investigator:	Charles L Seitz
Faculty Investigators:	James T Kajiya
	Alain J Martin
	Robert J McEliece
	Martin Rem
	Charles L Seitz

Sponsored by the
Defense Advanced Research Projects Agency
ARPA Order Number 3771

Monitored by the
Office of Naval Research
Contract Number N00014-79-0597

SUBMICRON SYSTEMS ARCHITECTURE

Department of Computer Science
California Institute of Technology

1. Overview and Summary

1.1 Scope of this Report

This document is a summary of the research activities and results for the six month period 16 March 1985 to 15 September 1985 under the Defense Advanced Research Project Agency (ARPA) Submicron Systems Architecture Project. Technical reports covering parts of the project in detail are listed following these summaries, and can be ordered from the Caltech Computer Science Library.

1.2 Objectives

The central theme of this research is the architecture and design of VLSI systems appropriate to a microcircuit technology scaled to submicron feature sizes, and includes related efforts in concurrent computation and VLSI design. Additional background information can be found in previous semiannual technical reports [5052:TR:82], [5078:TR:83], [5103:TR:83], [5122:TR:84], [5160:TR:84], [5178:TR:85].

1.3 Highlights

Some highlights of the previous 6 months are:

- Cosmic cube host software completed (2.1.2) and available for distribution (2.1.3).
- Robust Mosaic RAMs (2.2).
- Experimental Concurrent Language developments (3.1).
- New method for deadlock-free routing on arbitrary networks (3.5).
- Torus Routing Chip completed (4.1).

1.4 Acknowledgement

Please let us take this opportunity — the semiannual report for what Paul Losleben claims is his last VLSI contractor's meeting before leaving DARPA — to thank and to congratulate Paul Losleben for his efforts as manager of the DARPA VLSI program.

2. Architectural Experiments

2.1 Cosmic Cube Project

W C Athas, Reese Faucette, Michael Lichter, Wen-King Su, Chuck Seitz

The Cosmic Cube's host environment was substantially refined in the past six months. Features such as "space-sharing" that were mentioned as plans in our previous report are now operational and in routine use. The entire programming system is stable, and most of our current work is involved with using the system. Typical applications include (1) concurrent algorithm experiments, (2) concurrent programming language experiments, (3) computer-aided design and analysis tools, and (4) simulations of other concurrent architectures.

In addition, the project group has recently received an Intel iPSC d6, scheduled to be upgraded to an iPSC d7 in October. The iPSC d7, with 128 nodes and a total of 64M bytes of primary storage, is a very high performance system for the applications listed above. The system software of the iPSC is not as mature as that of the Cosmic Cube, however we are in the process of improving this software.

In the interests of encouraging our colleagues in the DARPA community who may wish to use these systems, a copy of the latest "C Programmer's Guide to the Cosmic Cube" [5203:TR:85] is reproduced at the end of this report. Our 3-cube and 6-cube machines, and Intel iPSC are ARPAnet hosts. Also, the "cosmic environment" software – a runtime system and library that supports the same multiple-process message-passing environment on Unix 4.2bsd computers – is available for distribution. This system is described in Chapter 7 of the "C Programmer's Guide to the Cosmic Cube".

2.1.1 Node Operating System

The work on the Cosmic Cube's node operating system, the Cosmic Kernel, has progressed from development to maintainance and tuning. Very few changes have been made in the inner kernel. Several refinements have been accomplished in the two outer kernel processes, SPY and SPAWN. For example, the necessity for the `NEVERRUN` process state was eliminated, and a Unix influenced "fork" operation has been included that creates new copies of running processes was added.

A logic analyser was used to monitor the different subsystems of the Cosmic Kernel. The logic analyzer showed that the low level send/receive subsystem was not set-up for maximum overlap. Minor improvements that we are planning are to improve the synchronization between read and write operations and a more robust deadlock avoidance mechanism. However, a recent benchmark on the 6-cube showed message passing speed to be close to the 2Mbit/s hardware and interrupt routine limit.

We have a plan to create an alternate inner kernel by converting the present inner kernel code from 8086 assembly language into C. This effort is a first step in producing a portable kernel for possible future machines of this type. It is also intended to make it easier to do experiments in which we can run user programs under kernels that implement the message system differently.

2.1.2 Host Software

The host programming system of the Cosmic Cube, the "cosmic environment", has been substantially rewritten over the past six months. Although this version 6 is a major internal

revision, it has involved very few changes in user programs from the version 3 system of 6 months ago. What changes the user has seen have been in the nature of "cleaning up" and simplifying some of the definitions and conventions, and have required only recompiling source programs.

The version 3 system had a number of restrictions based on the total number of socket connections that could be opened to the cube dæmon. It was possible for a user program to run out of socket connections depending on the number of others who were using the same cube dæmon. Also, the user was at the mercy of the limited queueing supplied by the 4.2bsd socket implementation. The versions 5 and 6 implementations have moved the socket connections and flow control into the **getcube** process of each process group. The version 6 system also includes a network-based update mechanism, described in the following section.

The C compiler used for cube process code was also rewritten in this period.

2.1.3 "Cosmic Environment" software distribution

The ***cosmic environment*** software described in the "C Programmer's Guide to the Cosmic Cube" is available for internal use to most university, government, and non-profit research organizations under the arrangements outlined below. This software may be of interest to other contractors both for concurrent computing experiments and for use in courses. Software distributions are available to other organizations only through a license with Caltech. It is expected that another version of this software will be available through Intel Scientific Computers.

The cosmic environment is package of programs, C library routines, and dæmon processes that supports on one or on multiple network-connected Unix systems the same message-passing concurrent process programming environment that runs on the Cosmic Cube. Process code is compiled with a set of definitions and library routines and run as Unix processes. The interprocess communication mechanism used by this system is internet sockets. Processes may accordingly be run all on one host or distributed across a number of network-connected hosts. This software can also be used as the host environment for Cosmic Cube programs, or in the future, for Intel iPSC programs. (See the "C Guide" for additional details.)

These programs were written to run under Berkeley Unix 4.2bsd, and have been installed and used on SUNs, Berkeley Unix VAXes, and Ultrix VAXes. We do not know whether the system runs correctly or can be made to work on other Unix systems. A minimal necessity to be able to adapt these programs to other Unix systems is a socket implementation.

This software also comes with a network-based update mechanism that is recommended for ARPA internet hosts.

In order to get a copy, send a letter to Professor Charles L Seitz, Computer Science 256-80, California Institute of Technology, Pasadena CA 91125, that includes statements:

1. of your intended use of this software (brief).
2. that you agree not to redistribute this software outside of your organization, and in no case outside of the United States, and will keep the files protected on your systems.
3. that you agree that this software is for your internal use, and that you will not sell access to or otherwise permit use of these programs to people outside your own organization.

4. that you understand that this is experimental, unsupported software that may or may not work as described, and that you hold Caltech harmless from any loss or damage that may result from its installation or use.

We are able to answer questions about this software as our time allows, and are happy to have bug reports, but only by electronic mail to "cube@cit-sol.arpa". It is most helpful if a bug report includes the smallest example of code that you can devise that demonstrates the bug.

If you are installing this software on a machine that is not an ARPAnet host, please include with your letter a blank 600' 1/2 inch tape labeled with your postage address. We will mail the C sources to you in Unix tar format. The files can be placed in any source directory (we recommend a cube login directory). The installation instructions are included in the make file.

If you are installing this software on an ARPAnet host, you do not need to send a tape. We hope you don't. Just send us your ARPAnet mail address and the name of the host on which you intend to maintain your source directory. We will then enter the name of the host on which you maintain the sources into a permission table, and will send you a message that includes a C program that you can compile and run to bring the sources over the ARPAnet to your machine. This program makes a socket connection to the cube dæmon that runs on cit-sol.arpa, and brings the files across the network. This same program can be used later for automatic updates, which are selective by timestamp in order to minimize the volume of messages sent through the network.

Requests to use the Cosmic Cubes or Intel iPSC d7 operated by the Caltech Computer Science project group should be addressed to "chuck@cit-vlsi.arpa".

2.1.4 Hardware Status

Our 6-cube system finally exhibited a hard failure after running continuously without a failure for 14 months. The two previous hard failures since the machine was put into service 23 months ago were both in dRAM chips. The failure observed this time was in an SSI communication channel driver chip. The bad board was swapped into the 3-cube for testing and diagnosis, and downtime was only a few minutes after the problem was first noticed. We also took advantage of this opportunity to experiment informally with using space sharing to keep the system in service in spite of bad or absent node boards. Users have been able to utilize subcubes while a portion of the cube has not been functioning or was running test programs.

The 6-cube alone now has logged 16,500 hours of operation. Total node-hours accumulated on all machines are now over 1,500,000, with 3 hard failures. The calculated MTBF of the nodes of 100,000 hours reported before this machine was constructed can now be regarded as conservative at the 99% confidence level, and an MTBF of 200,000 hours can be stated at a 95% confidence level.

Both machines continue to be heavily used.

2.1.5 Application programs

We continue to make the Cosmic Cube available to selected guest researchers for problems that appear to be particularly interesting or difficult. For example, several researchers from the Aeronautics group at Caltech report very good results on a very difficult jet plume

computation. In addition, the following small sample of programs from the research group are suggestive of our more interesting applications of the Cosmic Cube.

The concurrent circuit simulator CONCISE was translated last May from Pascal to C, using the HCR Pascal to C translator. It now runs correctly as a concurrent program under the "cosmic environment" system. It continues to be developed and refined by Sven Mattisson in a joint project between Caltech and the University of Lund, Sweden.

A concurrent circuit extraction algorithm and program were developed as a student project, and will be developed into a useable form.

Reese Faucette, author of the cosmic kernel, wrote a program that uses the 6-cube to produce a high resolution graphic image based on a simple mathematical model taken from Mandelbrot's fractal geometry. Another of Reese's programs, virus, starts up two-sided games of 3-dimensional tic-tac-toe throughout a cube, with "mutations" occuring in the playing parameters of losing players, so that the overall playing level of a cube of processes rises with time.

2.2 Mosaic Project

Steve Rabin, Don Speck, Chuck Seitz

A 16-inch stack of SPICE simulation listings and several layout modifications to the Mosaic dynamic RAM module have finally converged to a version that is reasonably tolerant of process, timing, and power supply variations. The most recent chips received from MOSIS had all bits working, could tolerate clocks of 3.5 to 7.5 volts, and ran as fast as the tester. The yield on these recent runs was poor in contrast to our previous experience with chips with a similar active area, number of transistors and contacts, etc. Checking the defective chips under a microscope revealed missing contacts, shorts, and a few large-area poly defects. Thus we attribute the poor yield to the fabrication rather than to the design.

The Mosaic RAM seems to prefer a less "hot" clock than we had envisioned. This may allow us to eliminate the 7-volt power supply and remove some of our power dissipation worries.

Now that the RAM design looks usable, we can safely return to full Mosaic element assembly. The floorplan and schematics are done, leaving us with the layout of the ROM and routing still to do.

The floorplan is designed to keep the die *width* under five millimeters to allow packaging the chips in the same type of 18-pin large well DIPs that are used for commercial dRAMs. Far from being a hindrance, this constraint resulted in an elegant structure, with the data bus running the length of the chip and processor, ROM, and RAM modules lined up on both sides, and pads at each end. We chose to limit the length to 7 millimeters, to avoid pushing our luck on the yield. This 5mm×7mm allows us 4 RAM modules (1K words total) per node.

A preliminary design of the top-assembly of the Mosaic has been completed. The Mosaic elements will by this design be packaged with 64 elements on each of 16 fairly small PCBs that can be plugged into different backplanes for either (Ultracomputer style) shuffle or mesh networks. Some recent results on routing on meshes (see section 3.5) have made the mesh connection more attractive than it formerly appeared to be, and we could very much simplify the top assembly by using a simple toroidal mesh structure.

A simulator and an assembler for the version B Mosaic architecture have been completed, and the floating point arithmetic code, and other short programs have been extensively simulated using this simulator and assembler. The datapath artwork for the version B architecture has been completed; the control and I/O sections have not.

2.3 Future Architecture Experiments

The research group has decided to defer building a fine grain SIMD message-passing machine along the lines of the Supermesh design previously reported. Having been set back in our schedule on the Mosaic project by problems with the Mosaic RAM, we expect to see the first working Mosaic systems early in 1986. We appear to be able to build new experimental systems at intervals of about 18 months. Our work with fine- and medium-grain MIMD message-passing machines such as the Cosmic Cube and Mosaic currently has a great deal of momentum, and we have decided that we would do better to concentrate in that area for the immediate future.

In our experience with distributed operating systems, in "pushing the envelope" of applications of these machines, in some of the recent results described in the following section, and in the appearance of much higher performance single-chip processors, we see an opportunity to contribute to a second generation of these MIMD message-passing machines. These machines will exhibit about 10 times better processor performance and 100 times better communication performance – particularly with short messages –, with the principal objective of extending the application span and programmability. Some of our ideas in these areas are described more completely in the following section.

What we are planning, as described in previous reports under the heading of "advanced technology homogeneous machines", is to start construction in the summer 1986 of an experimental system based on the DEC microVAX II processor and floating point coprocessor, 1MB of primary storage, and a custom communication chip. The first version of this communication chip, which was designed in the new MOSIS SCMOS technology, is described in sections 3.5 and 4.1, and it has already been submitted for fabrication. The system will employ a toroidal mesh connection rather than a binary n-cube. The size of the system is expected initially to be 256 nodes, a 16 by 16 mesh, and of course will be expandable. DEC has agreed verbally to contribute the microVAX II chips and Ultrix source licenses. The operating system kernel will be derived from the Ultrix (Berkeley Unix) kernel (VMUNIX). All aspects of this plan are subject to change depending on new developments.

3. Concurrent Computation

3.1 Experimental Concurrent Language

Bill Athas, Chuck Seitz

A highly concurrent object-oriented language, tentatively named ECL for "Experimental Concurrent Language", has been designed and exists now as a compiler, software engine, and technical report [TR:85:5196]. The unit of computation for ECL is the "concurrent object" which can interact with other such objects only by message passing. ECL is based on the computational model of the Cosmic Cube experiment, including the weak synchronization properties of the message system, but abstracts away many of the details currently necessary to program the Cosmic Cube. These details, such as the management of message locks and the automatic setting-up of message descriptors are handled by the compiler through extensive program flow analysis.

The "process abstraction" used in existing Cosmic Cube programs is captured by ECL's concurrent objects. To soldify both the semantics and syntax of the language, a test suite of four programming examples were chosen as a basis for resolving design issues about the language. These examples were a Hamming decoder, solution of the 8-queens problem, Gaussian elimination, and the generation of primes by sieving. The concurrent formulations and their solutions written in ECL have been included in the technical report. The solutions represent a wide range of concurrent formulations and have been compiled and the results checked using the first software engine.

The software engine is a microcosm of the cosmic environment and compiled ECL programs could readily be ported to such an environment, provided that a special resource allocator or "real estate" process be included. The current plan, however, is to perform static analysis of the code generated by the compiler so that resource allocation may be conducted at load time rather than at runtime, whenever possible.

3.2 The Sneptree

Pey-yun Peggy Li, Alain J Martin

Compared with other augmented binary trees, the Sneptree is shown to be the only network that can simulate an arbitrary size binary tree well, and also has regular and symmetric topology, thus is suitable for VLSI implementation.

An improved routing algorithm for the leaf nodes of the Cyclic Sneptree has been developed. The routing algorithm always finds the shortest path within the minimal subtree containing the source and the destination nodes. The routing result is a good approximation to the optimal result, yet still can be found in $O(n)$ time, where n is the height of the Sneptree. In some specific communication patterns, such as shifting by 2^k, the average routing result is almost optimal. Besides, the traffic at the upper-level nodes is reduced to about a half of the traffic in a pure binary tree.

Many interesting results of the Sneptree can be found in the technical report [5194:TR:85].

3.3 The Sync Model for Parallel Prolog Processing

Pey-yun Peggy Li, Alain J Martin

The research of the machine model for parallel Prolog processing has continued in last six months. Our Sync Model combines data flow concepts with a special synchronization

mechanism to realize AND parallelism. The dynamic data flow diagram of sibling AND processes is constructed during the time the AND processes are invoked. The change of the data flow diagram caused by binding a variable to a partially instantiated term is detected by a simple type checking, and is realized by adding extra dynamic links between AND processes. The above method is more efficient than the previous data flow models. An efficient merge algorithm has been designed to merge the multiple input streams in an AND process or the partial solution streams in an OR process. The merge algorithm is capable of handling a set of finite input streams with arbitrary combinations of Sync signals. The correctness proof of the merge algorithm as well as the Sync Model is being developed.

Different mapping algorithms for mapping our Sync Model to the Sneptree have been investigated. The one we selected is to map our Sync Model to a binary tree by adding extra padding nodes, and then map the binary tree onto the Sneptree. This algorithm turns out to have the maximal parallelism among other mapping algorithms.

3.4 Concurrent Computational Geometry

John Ngai, Chuck Seitz

The appearance of parallel computers has motivated the development of parallel algorithms and distributed data structures for solving problems by exploiting the concurrency of these machines. Until recently, very little work has been done in developing parallel algorithms and distributed data structures to solve geometric problems. Computational Geometry is the systematic study of algorithms and complexity of fundamental problems such as inclusion, intersection, hull finding, and proximity searching, all of which are geometric in nature. Such problems arise in many conventional applications, such as CAD, Robotics, Data Base Retrival, and Operations Research, as well as in many military systems. Traditionally, such problems are solved by uniprocessor systems employing complicated data structures.

Our research focuses on the investigation of intrinsic geometric properties of such problems to reveal concurrencies that can be exploited in the concurrent formulation of their solutions. Proceeding in this direction, we have been able to develop fast sublinear parallel algorithms for the determination of the convex hull of point sets in the Euclidean two- and three-space. The complexity of our algorithm for the two-dimensional case matches the best known in published literature, for example, $O(\log^2 N)$ on the binary n-cube. To the best of our knowledge, no corresponding parallel algorithm for the three-dimensional convex hull has been published. By exploiting the criteria derived from the normal vector representations, we are able to generalize and obtain sublinear algorithm for spatial point sets. A detail report of these algorithms is being prepared and will be submitted for publication.

Current work on concurrent computational geometry is concentrating on convex decomposition and geometric searching. Parallel algorithms for multiple point searching in planar subdivisions have been developed. Investigations to generalize them to other objects and to higher dimensions are in progress.

3.5 Deadlock Free Routing

William Dally, Chuck Seitz

We have developed a method for constructing deadlock-free routing algorithms in arbitrary strongly-connected interconnection networks. To construct a deadlock-free routing algorithm, we consider the routing function to be of the form $C \times N \rightarrow C$ (the current channel and the destination node determine the next channel) rather than the conventional

$N \times N \rightarrow C$. A routing function from channel to channel describes a channel dependency graph, D. We have proved that a routing function is deadlock free if and only if D is acyclic.

To make a routing function that describes a cyclic D deadlock-free, we have developed the concept of virtual channels. Several virtual channels, each with its own queue, can be multiplexed over a single physical channel. It is the interaction of queues that determines the channel dependencies and thus the deadlock properties of the network. A routing function that has a dependency graph with a single cycle can by made deadlock-free by splitting each physical channel around the cycle into two virtual channels and specifying the routing on the virtual channels so that their dependency graph describes a helix. Routing functions with several cycles can be treated by splitting each cycle in turn.

This technique, described in full in a preprint reproduced at the end of this report, has proved to be a key idea for our plans for advanced technology homogeneous machines. The Torus Routing Chip described in section 4.1 is a demonstration of this technique.

3.6 Concurrent Graph Algorithms

William Dally, Chuck Seitz

We have developed two new concurrent algorithms for the max-flow problem. The concurrent augmenting digraph (CAD) algorithm works by propagating an augmenting directed graph rather than a single augmenting path as in the Ford-Fulkerson algorithm. The concurrent vertex flow (CVF) algorithm, on the other hand, works by pushing flow from one vertex to another. It is similar in spirit to Karzanov's algorithm and achieves the same worst case bound of $O(V^3)$.

We have developed a new concurrent heuristic algorithm for graph partitioning. Based on the Kernighan and Lin algorithm, our algorithm works by iterative improvement. Unlike K & L, however, the new algorithm may move several vertices at once. A two-pass attraction algorithm is used to prevent multiple moves from interfering with one another. An embedded tree is used to assure that the partition remains balanced.

3.7 Generalizations of Message Routing Mechanisms

Craig Steele, Chuck Seitz

This investigation is seeking generalizations and extensions of the current message routing mechanisms for message-passing concurrent computers. Communications deadlock must be avoided in such systems, but can occur when a message system cannot deliver any message already in the system due to competition from other messages in the system. An efficient method – "distance queues" – for constructing deadlock-free store-and-forward message systems has been found for networks of arbitrary connectivity. We can easily find minimal cost routings for powerful but complex interconnections such as the shuffle-exchange-shift network. In addition, the ability to automatically generate deadlock-free routes for irregular networks allows a distributed processor to tolerate failure of some nodes or communications links. If process placement is performed by the previously reported technique of simulated annealing, automatic reconfiguration and restart of a computation is possible even in the event of serious hardware faults.

For toroidal mesh networks it is possible to generalize the e-cube deadlock-free routing scheme used in the Cosmic Cube. These networks allow a trade-off to be made between communications performance and the cost of implementing the connections. With dedicated virtual circuits, it is possible to emulate toroidal meshes on an arbitrary physical

network. The cost of such emulation is minimized by using simulated annealing to find a good embedding. This approach to the problem of deadlock-free routing allows us to decrease undesired competitive interaction of messages by increasing the buffering resources allocated.

Even a deadlock-free message system can be overwhelmed and deadlocked by unconstrained use of its buffering capabilities. Currently, Cosmic Cube programs written without explicit synchronization depend on process placement and the particular hardware characteristics for correct operation. Current research is examining extensions of the present Cosmic Kernel mechanisms to allow such programs, written in a convenient "pipelined" or "data flow" style, to run reliably on arbitrary networks.

3.8 Semantics of Infinite Lists

Young-il Choo, Jim Kajiya

To define rigorously the meaning of higher order infinite lists (lists which can contain infinite lists as elements recursively), we were led to topological idea of limits. The domain of infinite lists constructed by using the inverse limit method provides a formal basis for dealing with infinite objects in an implicit manner with lazy evaluation as the essential programming language construct. Full discussion of the technique and its applications in programming language sementics can be found in the Technical Report *An Inverse Limit Construction of a Domain of Infinite Lists* [5188:TR:85].

3.9 On Representing Petri Net Behavior

Young-il Choo, Jim Kajiya

Petri nets are mathematical constructs for modeling concurrent activity. Their structure is given by a bipartite directed graph; their behavior is determined by the movement of tokens around the graph. The behavior is usually represented as the set of strings generated by the firings of the transitions.

Concurrency Algebra is an algebraic framework for reasoning about the dynamic behavior of Petri nets. Based on a simple algebra containing concatenation, choice, and shuffle of strings, concurrency algebra uses a substitution schema determined by the structure of the given Petri net to generate new terms that denote its behavior. For non-terminating Petri nets, the behavior becomes the smallest set of strings satisfying a recursive equation over sets of strings.

Full details can be found in the Technical Report *Concurrency Algebra and Petri Nets* [5190:TR:85].

4. VLSI Design

4.1 The Torus Routing Chip

William Dally, Chuck Seitz

As a demonstration of the use of virtual channels to perform deadlock-free routing we have designed a custom VLSI integrated circuit: the torus routing chip (TRC). The TRC performs packet routing in arbitrary k-ary n-cube or torus networks. Each chip provides a single node's communications needs in two dimensions. The chips are designed so they can be cascaded to construct networks of higher dimension. The TRC uses virtual channels to break the channel dependency cycles inherent in this class of networks.

In addition to virtual channels, the chip incorporates several other innovative features. First, it is completely self-timed. Each of the two virtual channels in each dimension and the processor channel, as well as the internals of the chip, operate completely asynchronously. Arbiters in the chip's crossbar switch perform all required synchronization. Second, the chip uses incremental routing. A header byte specifies the distance to be routed in each dimension. After routing in one dimension is complete, the byte is stripped off and the message is passed to the next dimension. Third, the chip uses through-node or "wormhole" routing. Rather than storing the entire message in a node and then forwarding it, the message is routed directly to its destination. This technique makes message latency almost independent of distance, and avoids consuming storage bandwidth in routing nodes. Finally, the chip is intended to be fast. It uses byte-wide communication channels, and τ-model calculations indicate that each channel should operate in excess of 20MHz (160Mbits/s).

The torus routing chip has been laid out in SCMOS technology using the Magic system, simulated using MOSSIM, and submitted to MOSIS for fabrication.

4.2 Compiling Programs into Self-timed VLSI Circuits

A. J. Martin

We have developed a method for "compiling" a high-level description of a computation (a set of communicating processes) into a self-timed VLSI circuit. Self-timed (or delay-insensitive) circuits are sequential circuits in which the sequencing is entirely enforced by communication. There is no clock, and no assumption is made on the propagation delays in operators and wires. The advantages are many: First, with the increasing size of circuits, it becomes more and more difficult to distribute safely a clock signal across a chip. Second, clocked circuits rely on worst-case assumptions on the timing behavior of the components, which decreases their performances. Third, with no restriction on the length of wires, layout is facilitated.

The compilation is systematic and essentially relies on the four-phase handshaking implementations of communication actions. The program of each process is compiled into a set of "production rules" from which all explicit sequencing has been removed. By matching these production rules to those describing the semantics of the VLSI-operators (and-gate, or-gate, C-element, arbiter, flip-flop, etc), the programs are identified with networks of operators, *ie*, self-timed circuits.

The method has been applied to a whole spectrum of problems, some of them quite difficult, like distributed mutual exclusion and fair arbitration. The results are far beyond our expectations. For most circuits, especially complex ones, the compiled circuits are superior to their "hand-designed" counterparts.

California Institute of Technology
Computer Science, 256-80
Pasadena CA 91125

Technical Reports
September 1985
Available from the Computer Science Department Library
Prices include postage and help to defray our printing and mailing costs.

Publication Order Form

If you wish to order any of the reports listed, complete this form and return it with your check or international money order (i U.S. dollars) payable to CALTECH. Prepayment is required for all materials.

____5203:TR:85 $9.00 *C Programmer's Guide to the Cosmic Cube*
Su, Wen-King, Reese Faucette and Chuck Seitz

____5202:TR:85 $15.00 *Submicron Systems Architecture*
ARPA Semiannual Technical Report

____5200:TR:85 $18.00 *ANIMAC: A Multiprocessor Architecture for Real-Time Computer Animation*
Whelan, Dan

____5198:TR:85 $8.00 *Neural Networks, Pattern Recognition and Fingerprint Hallucination*
Mjolsness, Eric

____5197:TR:85 $7.00 *Sequential Threshold Circuits*
Platt, John

____5196:TR:85 $5.00 *ECL: An Experimental Concurrent Language*
Athas, Bill

____5195:TR:85 $3.00 *New Generalization of Dekker's Algorithm for Mutual Exclusion*
Martin, Alain J

____5194:TR:85 $5.00 *Sneptree - A Versatile Interconnection Network*
Li, Pey-yun Peggy and Alain J Martin

____5193:TR:85 $4.00 *Delay-Insensitive Fair Arbiter*
Alain J Martin

____5190:TR:85 $3.00 *Concurrency Algebra and Petri Nets*
Choo, Young-il

____5189:TR:85 $10.00 *Hierarchical Composition of VLSI Circuits*
Whitney, Telle

____5188:TR:85 $3.00 *Inverse Limit Construction of a Domain of Infinite Lists*
Choo, Young-il

____5185:TR:85 $11.00 *Combining Computation with Geometry*
Lien, Sheue-Ling

____5184:TR:85 $7.00 *Placement of Communicating Processes on Multiprocessor Networks*
Steele, Craig

____5178:TR:85 $9.00 *Submicron Systems Architecture*
ARPA Semiannual Technical Report

____5177:TR:85 $4.00 *Hot-Clock nMOS*
Seitz, Charles, A H Frey, S Mattisson, S D Rabin, D A Speck, and J L A van de Snepscheut

____5174:TR:85 $7.00 *Balanced Cube: A Concurrent Data Structure*
Dally, William J and Charles L Seitz

____5172:TR:85 $6.00 *Combined Logical and Functional Programming Language*
Newton, Michael

____5168:TR:84 $4.00 *Object Oriented Architecture*
Dally, Bill and Jim Kajiya

____5165:TR:84 $4.00 *Customizing One's Own Interface Using English as Primary Language*
Thompson, B H and Frederick B Thompson

____5164:TR:84 $13.00 *ASK French - A French Natural Language Syntax*
Sanouillet, Remy

____5160:TR:84 $7.00 *Submicron Systems Architecture*
ARPA Semiannual Technical Report

____5158:TR:84 $6.00 *VLSI Architecture for Sound Synthesis*
Wawrzynek, John and Carver Mead

____5157:TR:84 $15.00 *Bit-Serial Reed-Solomon Decoders in VLSI*
Whiting, Douglas

____5148:TR:84 $4.00 *Fair Mutual Exclusion with Unfair P and V Operations*
Martin, Alain and Jerry Burch

____5147:TR:84 $4.00 *Networks of Machines for Distributed Recursive Computations*
Martin, Alain and Jan van de Snepscheut

____5143:TR:84 $5.00 *General Interconnect Problem*
Ngai, John

____5140:TR:84 $5.00 *Hierarchy of Graph Isomorphism Testing*
Chen, Wen-Chi

____5139:TR:84 $4.00 *HEX: A Hierarchical Circuit Extractor*
Oyang, Yen-Jen

____5137:TR:84 $7.00 *Dialogue Designing Dialogue System*
Ho, Tai-Ping

____5136:TR:84 $5.00 *Heterogeneous Data Base Access*
Papachristidis, Alex

____5135:TR:84 $7.00 *Toward Concurrent Arithmetic*
Chiang, Chao-Lin

____5134:TR:84 $2.00 *Using Logic Programming for Compiling APL*
Derby, Howard

____5133:TR:84 $13.00 *Hierarchical Timing Simulation Model for Digital Integrated Circuits and Systems*
Lin, Tzu-mu

____5132:TR:84 $10.00 *Switch Level Fault Simulation of MOS Digital Circuits*
Schuster, Mike

____5130:TR:84 $3.00 *LOG The Chipmunk Logic Simulator User's Guide*
Gillespie, Dave

____5129:TR:84 $5.00 *Design of the MOSAIC Processor*
Lutz, Chris

____5128:TM:84 $3.00 *Linguistic Analysis of Natural Language Communication with Computers*
Thompson, Bozena H

____5125:TR:84 $6.00 *Supermesh*
Su, Wen-king

____5124:TR:84 $4.00 *Probe: An Addition to Communication Primitives*
Martin, Alain

____5123:TR:84 $14.00 *Mossim Simulation Engine Architecture and Design*
Dally, Bill

____5122:TR:84 $8.00 *Submicron Systems Architecture*
ARPA Semiannual Technical Report

____5120:TM:84 $1.00 *Mathematical Approach to Modeling the Flow*
Johnsson, Lennart and Danny Cohen

____5119:TM:84 $1.00 *Integrative Approach to Engineering Data and Automatic Project Coordination*
Segal, Richard

____5118:TR:84 $2.00 *SMART User's Guide*
Ngai, John

____5114:TM:84 $3.00 *ASK As Window to the World*
Thompson, Bozena, and Fred Thompson

____5113:TR:84 $4.00 *WoLery*
Mead, Carver A

____5112:TR:83 $22.00 *Parallel Machines for Computer Graphics*
Ulner, Michael

____5106:TM:83 $1.00 *Ray Tracing Parametric Patches*
Kajiya, James T

____5105:TR:83 $2.00 *Memory Management in the Programming Language ICL*
Wawrzynek, John

____5104:TR:83 $9.00 *Graph Model and the Embedding of MOS Circuits*
Ng, Tak-Kwong

____5103:TR:83 $7.00 *Submicron Systems Architecture*
ARPA Semiannual Technical Report

____5102:TR:83 $2.00 *Experiments with VLSI Ensemble Machines*
Seitz, Charles L

____5101:TM:83 $1.00 *Concurrent Fault Simulation of MOS Digital Circuits*
Bryant, Randal E

____5099:TM:83 $1.00 *VLSI and the Foundations of Computation*
Mead, Carver

____5098:TM:83 $2.00 *New Techniques for Ray Tracing Procedurally Defined Objects*
Kajiya, James T

____5097:TR:83 $4.00 *Design of a Self-timed Circuit for Distributed Mutual Exclusion*
Martin, Alain J

____5094:TR:83 $2.00 *Stochastic Estimation of Channel Routing Track Demand*
Ngai, John

____5093:TR:83 $1.00 *Design of the MOSAIC Element*
Lutz, Chris, Steve Rabin, Chuck Seitz and Don Speck

____5092:TM:83 $2.00 *Residue Arithmetic and VLSI*
Chiang, Chao-Lin and Lennart Johnsson

____5091:TR:83 $2.00 *Race Detection in MOS Circuits by Ternary Simulation*
Bryant, Randal E

____5090:TR:83 $9.00 *Space-Time Algorithms: Semantics and Methodology*
Chen, Marina Chien-mei

____5089:TR:83 $10.00 *Signal Delay in General RC Networks with Application to Timing Simulation of Digital Integrated Circ*
Lin, Tzu-Mu and Carver A Mead

____5086:TR:83 $4.00 *VLSI Combinator Reduction Engine*
Athas, William C Jr

____5084:TM:83 $3.00 *Tree Machine: An Evaluation of Strategies for Reducing Program Loading Time*
Li, Pey-yun Peggy, and Lennart Johnsson

____5082:TR:83 $10.00 *Hardware Support for Advanced Data Management Systems*
Neches, Philip

____5081:TR:83 $4.00 *RTsim - A Register Transfer Simulator*
Lam, Jimmy

____5080:TR:83 $4.00 *Distributed Mutual Exclusion on a Ring of Processes*
Martin, Alain

____5079:TR:83 $2.00 *Highly Concurrent Algorithms for Solving Linear Systems of Equations*
Johnsson, Lennart

____5078:TR:83 $5.00 *Submicron Systems Architecture*
ARPA Semiannual Technical Report

____5075:TR:83 $2.00 *General Proof Rule for Procedures in Predicate Transformer Semantics*
Martin, Alain

____5074:TR:83 $10.00 *Robust Sentence Analysis and Habitability*
Trawick, David

____5073:TR:83 $12.00 *Automated Performance Optimization of Custom Integrated Circuits*
Trimberger, Steve

____5068:TM:83 $1.00 *Hierarchical Simulator Based on Formal Semantics*
Chen, Marina and Carver Mead

____5065:TR:82 $3.00 *Switch Level Model and Simulator for MOS Digital Systems*
Bryant, Randal E

____5055:TR:82 $5.00 *FIFO Buffering Transceiver: A Communication Chip Set for Multiprocessor Systems*
Ng, Charles H

____5054:TM:82 $3.00 *Introducing ASK, A Simple Knowledgeable System*
Thompson, Bozena H and Frederick B Thompson

____5052:TR:82 $8.00 *Submicron Systems Architecture*
ARPA Semiannual Technical Report

____5051:TM:82 $2.00 *Knowledgeable Contexts for User Interaction*
Thompson, Bozena, Frederick B Thompson, and Tai-Ping Ho

____5047:TR:82 $3.00 *Torus: An Exercise in Constructing a Processing Surface*
Martin, Alain

____5046:TR:82 $3.00 *Axiomatic Definition of Synchronization Primitives*
Martin, Alain

____5045:TM:82 $3.00 *Distributed Implementation Method for Parallel Programming*
Martin, Alain

____5044:TR:82 $10.00 *Hierarchical Nets: A Structured Petri Net Approach to Concurrency*
Choo, Young-Il

____5038:TM:82 $4.00 *New Channel Routing Algorithm*
Chan, Wan S

____5035:TR:82 $9.00 *Type Inference in a Declarationless, Object-Oriented Language*
Holstege, Eric

____5034:TR:82 $12.00 *Hybrid Processing*
Carroll, Chris

____5033:TR:82 $4.00 *MOSSIM II: A Switch-Level Simulator for MOS LSI User's Manual*
Schuster, Mike, Randal Bryant and Doug Whiting

____5029:TM:82 $4.00 *POOH User's Manual*
Whitney, Telle

____5021:TR:82 $5.00 *Earl: An Integrated Circuit Design Language*
Kingsley, Chris

____5018:TM:82 $2.00 *Filtering High Quality Text for Display on Raster Scan Devices*
Kajiya, Jim and Mike Ullner

____5017:TM:82 $2.00 *Ray Tracing Parametric Patches*
Kajiya, Jim

____5016:TR:82 $4.00 *Bristle Blocks - Scrutinized and Analyzed*
McNair, Richard and Monroe Miller

____5015:TR:82 $15.00 *VLSI Computational Structures Applied to Fingerprint Image Analysis*
Megdal, Barry

____5014:TR:82 $15.00 *Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture*
Lang, Charles R Jr

____5012:TM:82 $2.00 *Switch-Level Modeling of MOS Digital Circuits*
Bryant, Randal

____5001:TR:82 $2.00 *Minimum Propagation Delays in VLSI*
Mead, Carver, and Martin Rem

____5000:TR:82 $6.00 *Self-Timed Chip Set for Multiprocessor Communication*
Whiting, Douglas

____4777:TR:82 $7.00 *Techniques for Testing Integrated Circuits*
DeBenedictis, Erik P

____4724:TR:82 $2.00 *Concurrent, Asynchronous Garbage Collection Among Cooperating Processors*
Lang, Charles R

____4716:TM:82 $4.00 *Rectangular Area Filling Display System Architecture*
Whelan, Dan

____4684:TR:82 $3.00 *Characterization of Deadlock Free Resource Contentions*
Chen, Marina, Martin Rem, and Ronald Graham

____4675:TR:81 $7.00 *Switching Dynamics*
Lewis, Robert K

____4655:TR:81 $20.00 *Proc Second Caltech Conf on VLSI*
Seitz, Charles, ed.

____4654:TR:81 $12.00 *Versatile Ethernet Interface*
Whelan, Dan

____4653:TR:81 $10.00 *Toward A Theorem Proving Architecture*
Lien, Sheue-Ling

____4618:TM:81 $5.00 *Tree Machine Operating System*
Li, Peggy

____4600:TM:81 $3.00 *Notation for Designing Restoring Logic Circuitry*
Rem, Martin, and Carver Mead

____4530:TR:81 $20.00 *Silicon Compilation*
Johannsen, Dave

____4527:TR:81 $11.00 *Communicative Databases*
Yu, Kwang-I

____4521:TR:81 $8.00 *Lambda Logic*
Rudin, Leonid

____4517:TR:81 $7.00 *Serial Log Machine*
Li, Peggy

____4407:TM:82 $3.00 *Experimental Composition Tool*
Mosteller, Richard C

____4332:TR:81 $3.00 *RLAP, Version 1.0, A Chip Assembly Tool*
Mosteller, R

____4320:TR:81 $7.00 *Hierarchical Design Rule Checker*
Whitney, Telle

____4317:TR:81 $10.00 *REST - A Leaf Cell Design System*
Mosteller, Richard C

____4298:TR:81 $7.00 *From Geometry to Logic*
Lin, Tzu-mu

____4204:TR:78 $8.00 *16-Bit LSI Digital Multiplier*
Masumoto, R T

____4191:TR:81 $4.00 *Towards A Formal Treatment of VLSI Arrays*
Johnsson, Lennart S, Uri Weiser, D Cohen, and Alan L Davis

____4128:TM:81 $2.00 *Shifting to a Higher Gear in a Natural Language System*
Thompson, Fred and B Thompson

____4116:TR:79 $25.00 *Toward A Mathematical Theory of Perception*
Kajiya, Jim

____3999:TM:76 $3.00 *REL System and REL English, REL Report no.22*
Thompson, Bozena H and Frederick Thompson

____3975:TM:80 $3.00 *Rapidly Extendable Natural Language*
Thompson, B H and Fred B Thompson

____3762:TR:80 $8.00 *Software Design System*
Hess, Gideon

____3761:TR:80 $7.00 *Fault Tolerant Integrated Circuit Memory*
Barton, Tony

____3760:TR:80 $10.00 *Tree Machine: A Highly Concurrent Computing Environment*
Browning, Sally

____3759:TR:80 $10.00 *Homogeneous Machine*
Locanthi, Bart

____3710:TR:80 $10.00 *Understanding Hierarchical Design*
Rowson, James

____3364:TR:79 $8.00 *Stack Data Engine*
Efland, G and R C Mosteller

____2276:TM:78 $12.00 *Language Processor and a Sample Language*
Ayres, Ron

Please fill in your name, address and amount enclosed below:

Name _____

Address _____

City _____ State _____ Zip _____ Country _____

Amount enclosed $ _____

_____ Please check here if you wish to be included on our mailing list

_____ Please check here for any change of address

Return this form to: Computer Science Library, 256-80, Caltech, Pasadena CA 91125

C Programmer's Guide to the COSMIC CUBE

Wen-King Su, Reese Faucette, Chuck Seitz

Department of Computer Science
California Institute of Technology

Technical Report 5203:TR:85

20 September 1985
(corresponds to system version 6)

This technical report replaces a periodically revised internal working document [5150:DF:84] of the same title. It describes accurately – as far as we know – the operation of the cosmic cubes as of the date above. This version (6) of the cosmic cube system is expected to remain stable over the next year, and will be changed only by the addition of new functions, such as those required to support compatibility with the Intel iPSC. This programming guide is expected to be revised at about 6 month intervals. Please send corrections and suggestions about this document to `chuck@cit-vlsi.arpa`, and bug reports to `cube@cit-sol.arpa`.

A version of this programmer's guide is to be included in chapters 3 & 4 of a book, *Message-Passing Concurrent Computers: their architecture and programming*, by Charles L Seitz, William C Athas, William J Dally, Reese Faucette, Alain J Martin, Sven Mattisson, Craig S Steele, and Wen-King Su, to be published by Addison-Wesley Publishing Company.

The cosmic cube project is sponsored in part by the Defense Advanced Research Projects Agency, ARPA Order number 3771, monitored by the Office of Naval Research under contract number N00014-79-C-0597, and in part by a grant from Intel Corporation.

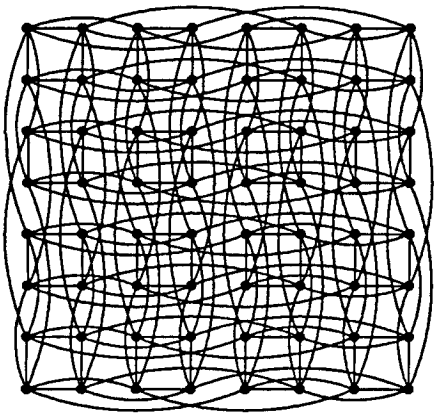
C Programmer's Guide to the COSMIC CUBE

Contents:

Changes from the previous version

This 20 September 1985 version of the "C Programmer's Guide to the Cosmic Cube" differs from the 24 May 1985 version due to a number of changes and "cleanups" in the system. However, the version of the system described here is not expected to change significantly in the next year except for the addition of iPSC compatibility functions and a few new features. Some of the changes incorporated since the previous version are an attempt to converge or not to conflict with some iPSC conventions.

- `typedef int BOOLEAN` and definitions of `FALSE` and `TRUE` are removed; likewise the `typedef unsigned short word` is removed. These changes will require some editing of existing programs.
- host utilities are now in `/usr/cube` instead of `/usr/local/cube`. Search paths should be changed accordingly.
- The old `cube/cubedef.h` is divided into those definitions that are common to cube and host processes, and remain in `cube/cubedef.h`, and those that are special to the cosmic cube, and would only rarely be used even in cosmic cube programs, and are in `cube/cosmic/cubedef.h`.
- `fix86` now produces a single `.cos` file instead of 3 separate files. Mention of the `-n` option is removed from the guide, although it will continue to exist until this optimization is made automatic.
- The `print` function now includes the `%s` format (section 5.2), and the `execute` function was added.
- The `malloc` family of functions is moved from experimental to operational category, and their description is found in section 5.3.
- The `bspawn` functions were replaced with a generalization of the `spawnf` and `spawnp` functions (section 6.3), which also suffered a change in the order and format of their parameters. Also, the host `load` utility is replaced with utilities called `spawnf` and `spawnp`
- The `cfork` family of functions was added (section 6.4). Associated with this change is the equivalence of the `NEVERRUN` and `SUSPEND` run states (section 6.1), and a reduced emphasis on the direct use of the SPAWN messages (much of the detail formerly appearing in section 6.2 is relegated to the synopsis).
- Chapters 1, 2, and 7 were revised to reflect changes in the host environment between the version 4 and version 5 systems.
- We have tried to contract this guide wherever possible in order to keep it of an appropriate length for continued use in courses that use the cosmic cube for laboratory exercises.



Chapter 1: Introduction

This programmer's guide describes the essentials of writing C programs for the *cosmic cube*, an experimental message-passing concurrent computer. It is meant to serve as a reference manual for the operation of the cosmic cube, for the services of the cosmic cube's operating system and host runtime system, and for the programming tools.

Our intention is that a person already familiar with C programming and with Unix would find that a single reading of this guide would be sufficient preparation to start writing cosmic cube programs. Chapter 8 is a synopsis that can be used to recall format details. People who program in Pascal but not in C tell us that they have little difficulty following the examples. However, for writing cosmic cube programs beyond the elementary category, a systematic introduction to C programming, such as that provided in *The C Programming Language* [Kern 78] by Brian Kernighan and Dennis Ritchie, is necessary.

We trust that the reader will excuse the occasional digressions into the reasoning behind the design of the cosmic cube and its operating system. An appreciation of why the functions work as they do is often useful in understanding what they do. For a more general introduction to the architecture and programming of the cosmic cube, see the January 1985 *CACM* article "The Cosmic Cube" [Seitz 85].

Certain details of the operating system functions and the responses to certain rare error conditions are not included here. Conditions such as parity errors detected in the primary storage produce self-explanatory error messages on the user's terminal. We are truly sorry that this guide still has so many details.

Users are hereby *warned* that the cosmic cubes are *experimental* computers. Their hardware is stable and reliable, but their system software and programming tools, being a part of the experiment, are changed from time to time. Changes to the system software that we can anticipate as of the date shown on the cover page, and that may have an impact on user programs, are mentioned in footnotes tagged with †. Sections of this guide shown in small type are notes about system functions not yet implemented, or about intended revisions to this guide.

Design and patent rights for the cosmic cube architecture and message-passing mechanisms were licensed to Intel Corporation in 1984, together with a resale license for the cosmic cube's operating system and host runtime system. The Intel Personal Supercomputer, or iPSC, manufactured by Intel Scientific Computers, Beaverton, Oregon, is accordingly very similar to the cosmic cube. However, a few of the C functions of the same name differ between the cosmic cube and the iPSC – for example, in the order of the parameters of the functions. Also, some iPSC functions are absent in the cosmic cube, and some cosmic cube

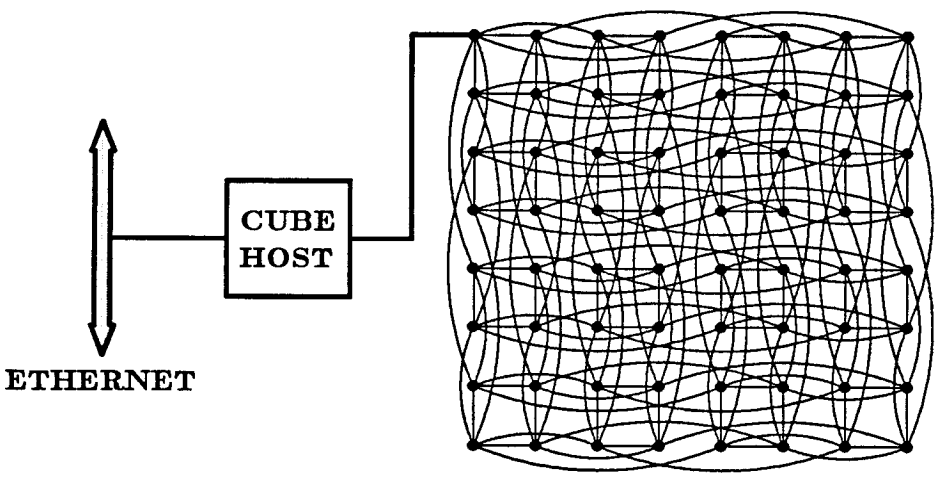
functions are absent in the iPSC. Thus, most of the programs shown here cannot be run "as is" on an iPSC. However, the process model and message semantics are identical for the two machines, and the concurrent formulations and programming approaches presented here are applicable to both machines.

1.1 Hardware structure

The hardware details of the cosmic cube are largely invisible to the C programmer. We will describe here only the overall structure of this message-passing concurrent computer, and introduce some of the terms and notational conventions used to describe its parts.

The cosmic cube is a collection of $N = 2^n$ small computers that operate concurrently and communicate by messages. The small computers, called *nodes*, are numbered $0, 1, \ldots, N-1$. Cosmic cube nodes are connected by bidirectional communication channels structured as a binary n-cube; that is, node i is connected to nodes $i \oplus 2^k$ by channel k, where $k = 0, \ldots, n-1$, and $\oplus$ is the bitwise mod-2 sum (or "XOR"). The two machines currently operated by the Caltech Computer Science Department are 3-cube (8-node) and 6-cube (64-node) machines.

Another computer called the *cube host* connects to the cosmic cube by a communication channel to node 0. The cube host provides program loading and input/output functions, and access to the cosmic cube via local networks. Here is a block diagram of the 6-cube system:



Each cosmic cube node, pictured above as a dot, has 128K bytes of primary storage, uses an Intel 8086 instruction processor and 8087 floating point arithmetic coprocessor, and includes 6 bidirectional communication channels. Node 0 has an extra channel for communication with the host. Each node also includes read-only storage for initialization and loading the operating system on a cold-start, and for diagnostic programs.

The cosmic cube hosts are SUN Microsystems workstations and file servers with a hardware interface that provides control signals and a channel to node 0. These SUN workstations use a Motorola 68010 instruction processor and run Berkeley Unix 4.2.

1.2 Process Model

The basic unit of computation in the cosmic cube is a *process*. A process can be regarded as an *instance* of a C program that includes statements that cause messages to be sent and received. Computations are performed by the concurrent execution of processes distributed

through the cube nodes, and one or more *host processes.*

A single cosmic cube node may contain many processes; hence the number of processes involved in a single computation may greatly exceed the number of nodes. All processes execute concurrently, either by virtue of being in different physical nodes or by being interleaved in execution within a single node. Processes coordinate their activities and exchange data only by sending and receiving messages. There are no variables shared between processes.

Processes located outside the cube nodes, either in the cube host or in other network-connected machines, are all referred to as *host processes.* These host processes are Unix processes, and may use Unix and language processor utilities in addition to communicating with the cube processes and with other host processes through messages. Because the cube processes do not have access to input/output devices, that part of the computation that involves input/output and user interaction must be consolidated into the processes that run in the hosts.

This concurrent process model of computation with process interaction by messages has some similarities to the programming environment in other multiprogramming systems. In fact, exactly the same functions provided for cube processes are also supported by a library of C functions that can be used in the host processes. These functions interact with a runtime system that accomplishes interprocess communication with Internet sockets, a mechanism that works both within a single host computer or between multiple computers on the same Internet network. Thus host processes may be run on the cube host and on other hosts on the same network.

This host programming environment – called the *cosmic environment* – provides uniform communication between processes independent of the cube node, cube host, or network host on which they happen to be located. It is a principle and formal property of the entire system that – within limits of the computation being determinant and not exceeding available storage sizes – the results of a computation do not depend on the way in which the processes are distributed.

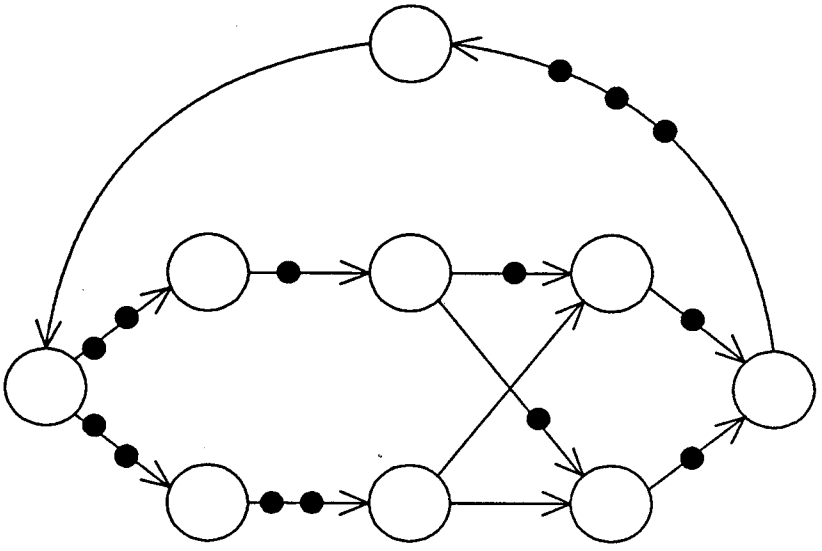
One can run computations formulated for this multiple-process message-passing environment with or without a cube. You could run all the processes on a single host, or distributed across a number of network-connected hosts. The package of functions and programs that support the cosmic environment – useful both as the host system and as a tool for the early development of programs for the cube – is described in chapter 7.

The process model is also quite easy to relate to the hardware structure of the cosmic cube, but is usefully abstracted from it. Instead of formulating a computation to fit on nodes and on the physical communication channels that exist only between certain pairs of nodes, one formulates the computation in terms of processes that are not bound to particular nodes, and in terms of communications that may occur between any pair of processes.

What is necessary for one process to send a message to another process is that the sending process have *reference* to the receiving process – that is, must know its "name and address". The "address" is just the node number, or `node`. For processes inside the cube the `node` corresponds to the numbering of nodes described in section 1.1. For processes in the hosts the `node` is defined by the constant `HOST`. The "name" is the process identifier or `pid` within that node, or a `pid` maintained by the host runtime system. The same `pid` may be used in different nodes. The `(node, pid)` pair that uniquely identifies a process is called its *ID.*

A message addressed by the `(node, pid)` pair is routed by the cosmic cube's resident operating system, or by the host runtime system, from the sending process to the destination process. Messages are queued as necessary both in nodes that the message may be routed through and at the destination. Messages can be distinguished by a message *type.*

The entire set of cube and host processes involved in a single computation is called a *process group*. By the term *process structure* we mean the set of processes that exist at a given moment together with each process's references to other processes. The process structure is naturally represented as a directed graph with vertices representing processes and arcs representing reference. The arcs can also be visualized as virtual communication channels, with messages traveling along the arcs, such as:



Given a process structure for a computation, the *placement* of processes is an embedding of the process structure graph into the n-cube graph and host machines, with each process assigned either to a cube node or to the hosts. An arc of the process structure graph that connects two cube processes may involve communication within a single node, between adjacent nodes, or along a path in the n-cube graph. The process placement is constrained by the storage requirements of processes and the total storage available in each node. Beyond this absolute constraint, process placement influences the speed of the computation through load balancing and message routing and delay considerations.

Process placement in the cosmic cube is controlled by the programmer, and may be decided either in advance by a static analysis of the problem or dynamically in execution. The way in which one causes an instance of a program to be loaded or *spawned* as a process in a node i, or to kill a process in this node, is to send a message to the SPAWN process in node i, that is, to process (*i*, `SPAWN`). Once a process is spawned it will not migrate to another node. The functions that are generally used to interact with the SPAWN process are described in chapter 6. For simple applications one may use utility programs available on the hosts to handle program loading.

Cube process code may be written in any sequential programming language — *eg*, C, Pascal, Fortran, assembly, etc — for which one has a suitable compiler or assembler to 8086/8087 code. One can mix processes written in different source languages in a single computation. The object code must adhere to the 8086 "small" addressing model. A process consists of a code segment and a combined data and stack segment, each limited to a maximum of 64K bytes. The compilation of C process code is discussed in chapter 2.

1.3 The kernel

The resident operating system of the cosmic cube is called the *cosmic kernel*. The kernel is what supports the process model and hides from the programmer the machine-dependent details of the hardware operation of the cosmic cube. One copy of the kernel resides in each node of the cosmic cube, and all of these copies are concurrently executable. Each copy of the kernel provides services for the processes within its own node as well as supporting distributed functions such as message routing.

All those parts of the kernel with which user processes communicate directly by system calls – or by the C functions that contain these system calls – are in the *inner kernel* (IK). The inner kernel supports message sending and receiving, and behind the scenes deals with message routing, process scheduling, storage management, and error conditions. These system calls and corresponding C functions are called *primitive* functions to indicate that they correspond directly to kernel primitives. The primitive `send`, `recv`, `probe`, `flick`, and `block` functions described in chapter 3 operate in a way that allows a process to manage many concurrent message activities.

The concurrency, queueing, and message order properties of the message system are discussed and illustrated in chapter 4. Here we show that the weakly synchronized form of message passing provided in the cosmic cube is well suited to typical computations, and is also sufficient to express the most tightly synchronized forms of message passing. For example, it is easy to define functions that provide the same tight synchronization between sender and receiver as is used in Hoare's Communicating Sequential Processes (CSP) notation [Hoare 78] or in Occam [InMOS 84].

An extended library of C functions that can assist the programmer are described in chapter 5.

All other kernel functions are invoked by sending messages to a set of processes called the *outer kernel* (OK). The outer kernel processes are just like user processes, except that they are loaded along with the kernel and use privileged system calls. Messages sent to the SPAWN process in a node cause this process to spawn or to change the *run state* of a specified process in its node. Messages sent to the SPY process provide means for monitoring what is going on in a cube node. These outer kernel processes are described in chapter 6.

1.4 Concurrent formulations

The design objectives of the programming system described in this guide were to support a highly portable low-level programming environment for message-passing systems. For example, there is only one message format visible to the programmer. Only the kernel "knows about" the hardware channels of the cosmic cube. The kernel handles messages of different lengths by different protocols, but these hardware dependencies and protocols are internal to the kernel and are completely invisible to the programmer. Future machines with different hardware communication structures will, with their own version of the cosmic kernel, be able to run the same user programs.

The semantics of these computations are independent of process placement, just as the semantics of the message-passing operations have been made independent of the hardware structure of the physical message channels. Thus the expression of a concurrent computation as a cosmic cube program may be regarded as reasonably portable between different machines (concurrent or sequential) that support this same multiple process message-passing environment. For example, the host runtime system allows exactly the same programs that run on the cube to run on individual computers or on a number of computers that communicate through a conventional network. Other operating systems or runtime systems could

be developed to allow these same programs to be run on other architectures, such as on shared-storage multiprocessors.

Although the functions provided in this low-level portable environment can be used as a compilation target for higher-level concurrent programming notations [Athas 5196:TR:85], the low-level environment built on the C programming language has proved to be adequate in itself for many purposes. It has been used extensively for experiments with concurrent algorithms and for writing useful programs based on explicit concurrent formulations of computing problems typical of those found in science and engineering. These application programs are often based on concurrent adaptations of well-known sequential algorithms [Seitz 84], or on the systolic algorithms that have been developed for regular VLSI computational arrays [Kung 80]. Systolic algorithms are particularly easily implemented as cosmic cube programs, since these algorithms are already expressed in terms of processes and message-passing.

The multiple process message-passing model of computation and the primitives of the cosmic kernel and the cosmic host environment are based on a principle of a "separation of concerns" between (1) the expression, in a collection of processes, of an explicit concurrent formulation of a computing problem, and (2) the distribution or *placement* of processes into the nodes and hosts. This system thus encourages "late binding" of processes to nodes, so that this decision can deferred until the interdependence and computational demands of different processes is known. The performance of a particular program depends on how well the concurrent formulation and process placement is able to keep the nodes busy. The objective is to assure that:

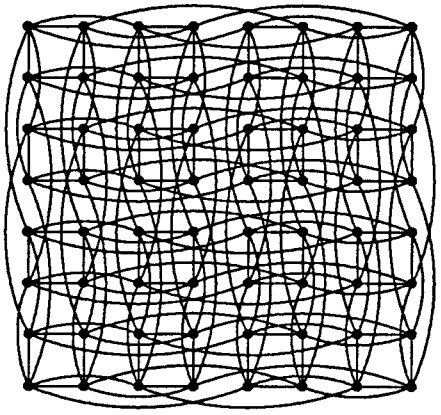
1. The number of concurrent processes that are able to make progress is on average comparable to or larger than the number of nodes.
2. There is a way to distribute the processes so that the average computational load on the nodes is satisfactory.

It is possible (although difficult) to formulate a concurrent computation that satisfies the first but not the second requirement. One should accordingly consider the "load balancing" properties of the computation from the earliest stages of program design.

In addition to these primary considerations, which are fundamental to the architecture, there is another performance consideration that is more important to the cosmic cube than it will be to future machines of this type. Messages should be reasonably infrequent compared with computation, for the reason that there is an appreciable operating system overhead for each message. In addition, a single long message is somewhat more efficient than a series of short messages that convey the same data.

Finally, although the placement of processes to minimize communication in the cube structure does influence the performance of the computation, the programmer should not be overly concerned in the formulation phase about the communication implications of the physical placement of processes. Analytical results and statistical studies [Steele 5184:TR:85] show that even after the constraints above are met, it is nearly always possible to find a process placement in a binary n-cube that reasonably minimizes the distance a message must travel. If worse comes to worse and many messages must travel to distant nodes, the system routes messages quite efficiently from any process to any other process, and future systems are expected to be even more efficient in this respect.

The examples in the Appendices are intended to provide an introduction to some of the issues in the design of concurrent formulations and programs for the cosmic cube.



Chapter 2: Processes

For the purposes of describing the compilation and execution of programs for the cosmic cube, let us assume that the programmer is has an account on one of the Caltech Computer Science VAXes or SUNs. These machines all run Berkeley Unix (4.2bsd) and communicate with each other by Internet protocols over an ethernet. Two of the SUNs are cube hosts, although the programmer doesn't need to know which machines these are.

The programs used for compiling and running programs on the cosmic cube are in directory `/usr/cube`, which should be in the user's search path. The directory `/usr/include/cube` contains `#include` files, and file `/usr/lib/libcube.a` contains a version of the cube library for that host. The definitions needed either for cube or host processes are provided by:

```
#include <cube/cubedef.h>
```

at the beginning of a C program for a process. A broader set of definitions containing both the basic cube definitions and a set of rarely used definitions, such as the definitions of the SPAWN process message formats, can be included from `cube/cosmic/cubedef.h`. The features accessed by these extra definitions are usually invoked by library functions. In accordance with the usual C conventions, constants and `typedefs` are in all capital letters; for example `HOST` is the constant that defines the node value to use for host processes.

The C functions described in this guide are linked with the program by the `-lcube` option of the particular C compiler being used. A synopsis of all of the definitions and functions is given in chapter 8.

2.1 Cube process representation

A process is represented inside the cube as a code segment and a combined data and stack segment. Facilities provided for sharing code segments, as may occur when multiple processes that are instances of the same program are located in one node, are described in section 6.2.

Process segments are dynamically relocated by the kernel, using segmentation features of the 8086 that require the code to conform to the 8086 "small" addressing model. The system calls that invoke kernel functions are implemented as interrupts that are vectored through the interrupt table, and hence do not reference out of the current segment. The 8086 does not have facilities that allow the system to be protected, so the compiler must be relied on to produce code that does not use absolute jumps or calls, does not load segment registers, does not perform I/O instructions, and does not tamper with the interrupts. As far as we know, our C compiler can be depended on in these respects.

The only runtime system support for a cube process is the set of functions described in this guide. A cube process must otherwise be self-contained. A process created in a node may not, for example, perform file or terminal I/O except through communication with a cooperating host process. It may use language runtime facilities and library routines so long as they can be linked in a way that conforms with the process representation described above. For example, a cube process may use the C `math` library but cannot use the `stdio` library.

The code for a process is compiled independently of the code for other processes that may be a part of the same computation. Because of the independence in the construction of process code and the standardization of kernel functions, there is a general compatibility of processes independent of source language. For example, a process written in Fortran or assembly code can be used in a computation in which other processes were written in C. The precautions that the user should be aware of are questions of the compatibility in data representations. These precautions are the same as those required when calling subroutines written in a different programming notation.

2.2 Compilation

The C compiler currently used for cube process code, `cc86`, is a version of the "portable C compiler". It was adapted at Caltech for the 8086/8087 originally by Eugene Brooks, and has since been modified extensively by Mike Newton and Wen-King Su. The compiler runs both on VAXes and on SUNs. Typical use is:

`cc86 -o objfile` *[other options] files* `-lcube` *[-lm]*

The standard `cc` options such as `-c` for compile-only and `-O` for optimization work as expected. The default library is not quite complete, and the only other library currently available is the `math` library, which is linked by `-lm`. The *files* may be any combination of `.c` and `.o86` files. One object file produced will be `objfile.o86`, and if the object code is completely linked, an "executable" file `objfile` is also produced.

The file `/usr/cube/bugs` describes any currently known bugs in `cc86`, as well as known bugs in the kernel or host systems.

Once the program is compiled the object code must be converted to the format required by the SPAWN process. Run `fix86` to perform the conversion:

`fix86` *[-s stacksize]* `objfile`

If the `-s` option is omitted, `fix86` will default to a stack size of 1K bytes. The stack is used for automatic variables while the rest of the data segment is used for constants and for the external variables. If the program requires a larger stack size than the default, or if you know that the program can get by with a smaller stack and you need to be conservative in your use of storage, the `-s` option should be used. Its argument is a decimal number giving the maximum number of bytes of stack that will be required by the program at any time. `fix86` allocates this much stack for the program plus additional stack space for saving the state of the process.

The `fix86` program produces a `.cos` file that contains, in addition to the program's code and data segment, a summary of the lengths of the program's code and data segments and various time stamp and version control information. The `.cos` file is an ASCII file that can be displayed to see this information. When you spawn a `.cos` file as a process, the system will tell you if it is out of date with the current library, so you must recompile, and will even remind you if you have recompiled a program but forgotten to run `fix86`.

2.3 Confidence-building first program

If you are already a C programmer, you know almost enough to write a program for a cube process. Let us now illustrate how programs are compiled, loaded, and run on the cube. This informal description of the runtime system and utilities is intended to remove some mysteries and to make the later detailed descriptions more comprehensible.

Following the excellent example of Kernighan and Ritchie, you might enter your favorite text editor and compose a simple program such as:

```
#include <cube/cubedef.h>

main() {
    print("Hello, world, from (%2d, %2d)", mynode(), mypid());
    print("Goodbye, cruel world!");
}
```

The cube's `print` function is much like the Unix `printf`, and `mynode` and `mypid` are functions that return values representing the process's own node and pid. Then we proceed (this is a session on one of the SUNs, called venus) to compile the program:

```
(venus:2) cc86 -o hello hello.c -lcube
hello.c:
(venus:3) fix86 hello
Code  size: 848
Data  size: 60
Stack size: 1024
```

Now we are left with the puzzle of how to load and run this program as a cube process. This task does not really involve much more to do than it would for a sequential machine program: just a sequence of (1) allocating a cube, (2) spawning the program as a process, and (3) freeing the cube that was allocated. However, because the object of this exercise is to understand how the host runtime system works, we shall discuss each step in some detail.

Each cube and host process exists within a *process group*. The way in which a process group is created is to run the utility program `getcube`. Here we first set an environment variable `cube` so that the group name will formed as `demo` combined with our Unix username, `vivian` in this case, and then run `getcube`:

```
(venus:4) setenv cube "group demo"
(venus:5) getcube 3 cosmic
3-cube allocated
```

In the `getcube` arguments we requested a cosmic cube of dimension 3. When `getcube` is successful in allocating a cube, it forks out a process that enters the host environment with the ID = `(HOST,SERVER)`. The `getcube` process also puts itself into the background and remains in the host environment doing various invisible chores such as queueing messages. You could observe the `(HOST,SERVER)` process by running the `peek` utility:

```
(venus:6) peek

CUBE DAEMON version 6, up 1 day 19 hours on host sol

   {              } 3d cosmic cube, b:0000 [    sol 6-cube] 15.0h
   { demo vivian  } 3d cosmic cube, b:0008 [  venus 6-cube] 23.2s
   {              } 4d cosmic cube, b:0010 [    sol 6-cube] 15.0h
   {group kaw31638} 5d cosmic cube, b:0020 [cit-vax 6-cube] 39.5m

GROUP {demo vivian}:

   ( -1  -1)    SERVER   0s   0r   0q   [venus 1368]  20.7s
   (--- ---)   CUBEIFC   0s   0r   0q   [  sol  454]  20.7s
```

The peek display reveals two other curiosities. First, the 3-cube that was allocated is actually a piece of a 6-cube, and another user has a process group assigned to another subcube of this machine. Not to worry. "Space-sharing" of cosmic cubes is much like "time-sharing" of conventional computers. The behavior of the logical 3-cube that was allocated is indistinguishable from that of a physical 3-cube. For example, the node values in our program actually refer to the node numbering in a logical 3-cube based at physical node 0x0008 rather than the absolute node numbering in the 6-cube. Second, there is another host process, CUBEIFC participating in the process group from host sol. This is the cube interface dæmon process for the 6-cube hosted on sol, and it handles the communication to and from the cube.

Error messages and the output of the print function are directed to stdout of the (HOST,SERVER) process. This process also performs the file accesses for process spawning. When a SPAWN process receives a message asking to create a process from a file, the SPAWN process interacts through a series of messages with (HOST,SERVER). Thus all pathnames for .cos files to be spawned as processes are interpreted relative to the current working directory of the (HOST,SERVER) process.

Although we could load our hello program by writing a host program that sends a suitable message to a SPAWN process, there is already a spawnf utility that will do this task for us. The spawnf program has the usage:

spawnf *filename [node [pid [runstate]]]*

The *node* and *pid* numbers are specified in decimal. The *runstate* is usually left blank to default to a running process.

```
(venus:7) spawnf hello 7 3
hello spawned successfully in node 7, pid 3
7,3: Hello, world, from ( 7,  3)
7,3: Goodbye, cruel world!
```

You can see that the cube's print function is slightly different from Unix's printf, in that it starts the line with the node,pid pair of the process executing the print, and adds its own carriage-return (\n).

If the *node* parameter given to the spawnf program is -1, the process is spawned in all nodes. This operation is performed with a broadcast spawn process (section 6.3) that is faster than loading processes individually. To get a feeling of power, we might try the broadcast spawning option:

```
(venus:8) spawnf hello -1 55
hello loaded in all nodes, pid 55
2,55: Hello, world, from ( 2, 55)
1,55: Hello, world, from ( 1, 55)
1,55: Goodbye, cruel world!
3,55: Hello, world, from ( 3, 55)
6,55: Hello, world, from ( 6, 55)
3,55: Goodbye, cruel world!
2,55: Goodbye, cruel world!
5,55: Hello, world, from ( 5, 55)
7,55: Hello, world, from ( 7, 55)
5,55: Goodbye, cruel world!
7,55: Goodbye, cruel world!
6,55: Goodbye, cruel world!
4,55: Hello, world, from ( 4, 55)
4,55: Goodbye, cruel world!
0,55: Hello, world, from ( 0, 55)
0,55: Goodbye, cruel world!

(venus:9) freecube
Cube space deallocated
```

Grateful now that we didn't get a 6-cube, we run the `freecube` utility in order to release the cube for another user.

All of the mechanisms illustrated in this extended example can also be invoked from host programs. The person using a cube program written by someone else does not need to know any of the manual steps illustrated above. Thus running a cube program can be made to look just like running any other program. For example, the long-winded illustration above can be duplicated in the following host program:

```
#include <cube/cubedef.h>

main() {

    if (system("getcube 3 cosmic"))   /* create process group */
        exit();
    cosmic_init(HOST,5);          /*Enter the environment with ID */
    spawnf("hello",3,7,"r");      /* spawn hello with ID = (3,7) */
    spawnf("hello",-1,55,"r"); /* broadcast spawn hello with pid = 55 */
    system("freecube");           /* release the cube */
    cosmic_exit();                /* and exit */
}
```

2.4 Process IDs and debugging

The ID of a cube process is determined when it is spawned. Its node value will be the same as that of the SPAWN process that created it. Its pid is determined by a value passed in the message to the SPAWN process. The ID of a host process is determined by the process itself when it enters a process group with the `cosmic_init` function.

The flexibility of process IDs being determined at run time, a set of rerouting conventions built into the kernel, and the principle of the results of a computation being independent of process placement, work together to provide a convenient mechanism for developing and debugging programs.

A message sent by a cube process to a process whose node is outside the range $0 \ldots 2^n - 1$, where n is the dimension of the cube in use, is treated as message for a host process. For example, suppose you were running a program on a 3-cube, for which the possible cube node values are $0, 1, \ldots, 7$. If a cube process sent a message to process `(12,15)`, the message would be routed out of the cube.

The decision to route a message out of the cube because the destination node is out of range is a special case – detectable at the sending node – of a process that is not present in the cube. If a message arrives at a destination node at which the specified pid does not exist, the message is again rerouted out of the cube. The cube host's `CUBEIFC` process then sends this message on to the group's `getcube` process, which in turn sends it to the corresponding host process if it exists, or produces an error message if the host process does not exist.

The programmer can exploit these rerouting features in developing and debugging programs, at least if the processes intended to be run on the cube are written to be able to run also on the hosts. The trick here is that *any* such cube process can be relocated to the host, up to and including all of the processes.

One can, for example, omit a particular process from the process structure spawned in the cube and run it instead as a host process with the same `(node,pid)` it would have in the cube, and under a debugger. Messages originating in the cube and destined for this process are rerouted from its usual node to the host environment. Messages sent from a host process to a process whose node is in the range of the allocated cube are forwarded through `CUBEIFC` to the cube only if the process does not exist in the host environment, hence go directly to this process. It is possible, but is obviously not a good practice, to have two processes with the same ID, one in the cube and one in the host environment.

In addition to this technique of relocating selected processes from cube to host for debugging, you can – if you do not have to have regular access to a cube while developing programs – run programs entirely on one or more hosts. Programs with very sparse interactions can even be run usefully on a collection of workstations on a local network.

Processes written to run only on the hosts, for example, processes that use Unix functions, should be assigned a node equal to `HOST`, a constant specified in `cubedef.h`.

2.5 Range restrictions

The node, pid, and message type are represented as `short` integers*. There are, however, restrictions on the range of values of these variables beyond those implied by their type.

---

* In the cube nodes an `int` is a 16-bit quantity, the same as a `short`; and in the SUN hosts an `int` is a 32-bit quantity, the same as a `long`. The reason that the node, pid, and type are represented to the programmer as `short` rather than `long` integers is because shorts are sufficient to represent the ranges of these variables for the cosmic cube. The

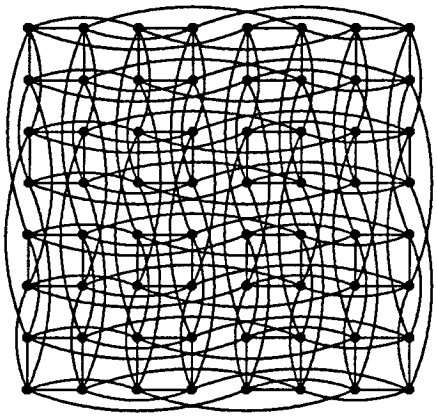
Negative values of node, pid, and type are permitted, but are reserved for special purposes in the system. For example, the node constant HOST is defined as -1. Otherwise, node values in the cosmic cube should be in the range 0...63, since the largest cosmic cube is a 6-cube. Larger node values cannot necessarily be represented at certain points in the system, such as in message headers, or may be used for other purposes in the message system protocols. Programs run entirely in the hosts do not have this same restriction. You could have a process in the host environment in which the node value was larger than 63, but then there would be no way for a cube process to send it a message, nor could the process be relocated to a cosmic cube node.

The number of processes per node is currently limited (by a kernel table size parameter) to 20. The pids of user processes must be in the range from 0 to MAXUPID (currently equal to 127). Negative pids are used in the cube nodes for pids of the outer kernel processes and utility processes such as the broadcast spawn process, and in the hosts for the pids of processes such as (HOST,SERVER).

Negative values of the message type are reserved by convention for library and system functions. The reply messages used by the library functions print, execute, cspsend, spawnf, and spawnp, which do their work through messages and which do not return until the reply message is received, use the type -1. User programs should not use this type, at least not concurrently with any of these functions, lest a message for the user process be interpreted as a reply message, or vice versa.

---

reason they are represented as shorts rather than as ints is because node, pid, and type values are often passed in messages between the cube and host, and must therefore be of compatible representations. Nevertheless, the functions that return node or pid values return int values, which of course will be converted to short values if assigned to one of these elements. In fact, all of the functions described in this guide return the usual C default int type except for malloc, which returns a pointer.



Chapter 3:
Inner Kernel Functions

The cosmic kernel provides five functions: `send`, `recv`, `probe`, `flick`, and `block`, to control and monitor message passing, and a number of other functions. The principal properties of the cosmic cube message-passing system are the following:

- All messages include headers that contain the destination ID, sender ID, a message type, and the message length.
- There is one format for messages of any length from 0 to the maximum message length (64K-1 bytes). Messages are sent and received whole.
- Messages are queued in transit, but between any pair of processes, message order is preserved.
- Process execution proceeds concurrently with message activities.

The descriptions of the primitive message functions later in this chapter concentrate on the interaction of a process with the message system. The next chapter describes the properties of the message system itself. Let us start, however, with an informal introduction to the message system functions and properties.

The two basic message system calls, and corresponding primitive C message functions, are `send` and `recv`. It is very important to understand from the outset that the return from these functions does *not* imply that the specified action has been completed. It isn't possible to receive a message that has not yet arrived. Similarly, it might not be possible to send a message immediately due to the outgoing channels being occupied. So, if the action specified in a `send` or `recv` cannot be performed immediately, a *request* is created that remains *pending* until it is satisfied, and the function returns anyway. This mode of operation is essential to sophisticated cube programs in which program execution may continue concurrently with many concurrently pending communication requests. However, it is also a dangerous source of bugs if one does not appreciate the unusual semantics of these functions.

It is correct to think of `send` or `recv` as functions that create (fork) another concurrent process, a descendant or "coroutine" of the process executing the `send` or `recv` function. This "coroutine" will persist for as long as the specified message action remains pending. The communication between a process and its message coroutines is through shared variables, both the message areas and a structure called the *message descriptor*.

When data dependence dictates that a process wait for a pending operation to complete, it need only execute the `block` function. The `block` function can also be thought of as a *join* between the process and a particular message coroutine. The common combinations

of send, recv, and block, such as:

```
recv(&d);     /* receive message described in message descriptor d */
block(&d);    /* wait for message to arrive */
```

also exist as single functions in the library, in this case recvb(&d).

In addition to message and process concurrency, another property of the cosmic cube message system that should be understood from the outset is that messages are queued. This queueing occurs both in routing the message and at the destination. It is possible for a process to have many incoming messages queued for it. The kernel uses that part of the primary storage of a node that is not used for processes as a *kernel message buffer.*

When a process dispatches a message with the send function, there is no automatic acknowledgment when the message has arrived at the node of the destination process, nor when it has been received by the destination process. A sending process can get as far ahead of the receiving process as its own dependence on results from the receiving process may allow, with messages simply being queued between them. However, message order is preserved between any pair of processes. This property of message order preservation between pairs within a weak synchronization between sender and receiver allows processes to get "out of step" as far as their dependence on each other may allow, but is also sufficient to allow the tightest forms of synchronization to be expressed in terms of the primitive kernel functions.

3.1 The Message Descriptor

Message headers are generated by the kernel as part of the message sending activity, and are separated from the message on receipt, so the exact format of the header is of no concern to the programmer. For specifying message length the header is not considered to be part of the message, but can be thought of as the envelope in which a message is sent. The header contains: (1) the ID of the destination process, (2) the ID of the sending process, (3) a message type, and (4) the length of the message.

The destination ID is what routes the message to another process. The sender ID is always included in the message and is available to the process that receives the message, or for discovering in debugging the origin of a rogue message. The type is significant in the way in which messages sent are matched against messages expected by the destination process. The message length is specified in the number of bytes, not including the header, and may be zero. A message of length zero is called a *synchronization message.*

The information in the message header is specified through a data structure that is called the *message descriptor.* The primitive send, recv, and probe functions take as a single parameter a pointer to an instance of this structure. The message descriptor is defined as:

```
typedef struct{
                short          node;
                short          pid;
                short          type;
                short          seg;
                char           *buf;
                unsigned short msglen;
                unsigned short buflen;
                short          lock;
              } MSGDESC;
```

and these 8 elements have the meaning:

node part of the destination (send), or sender (recv and probe) ID.

`pid`	part of the destination (send), or sender (recv and probe) ID.
`type`	of message.
`seg`	is irrelevant to user processes, and will not be mentioned again.
`buf`	is a pointer to the message buffer.
`msglen`	is the actual length of a message, in bytes.
`buflen`	is the length in bytes of the message buffer (recv), or unused (send and probe).
`lock`	contains a variable that indicates whether the message operation is pending, 0 (false) for not pending, and non-0 (true) for pending.

The same descriptor may be used over and over again. A declaration of a MSGDESC as an external variable can include its initialization:

```
MSGDESC results =
    {HOST, RPROC, RTYPE, 0, (char*) &x, sizeof(x), sizeof(x), 0};
```

or, one may use the idesc macro:

```
#define idesc(desc,node,pid,type,ptr,len)\
    MSGDESC desc = {node,pid,type,0,(char *)ptr,len,len,0}
```

in order to be able to express the declaration of an external MSGDESC with less to type and to remember about the order of MSGDESC elements. For example, the MSGDESC declaration and initialization above could be expressed:

```
idesc(results, HOST, RPROC, RTYPE, &x, sizeof(x));
```

This example can be thought of as the declaration of a *channel* named results to a host process (HOST,RPROC) to which results are sent in messages of type RTYPE from a structure x. This declaration is typical of that of a descriptor for sending messages to a process whose ID is known when writing the process code, rather than being passed in a message. Similarly:

```
idesc(startup, 0, 0, SUTYPE, xarray, sizeof(xarray));
```

is typical of an initialized message descriptor for receiving a "startup" message into array xarray. Of course, descriptors can also be built dynamically, and can be used for different purposes in different phases of a process.

The lock element of a message descriptor is the basic synchronization mechanism for sending and receiving messages, and being simply a variable that can be tested by the process code, allows concurrent message activities to be monitored during process execution.

The send, recv, and probe functions are indifferent to the state of the lock at the time the function is called. The lock is written by the kernel in the send and recv functions so that when these functions return the lock will be false if the message operation is complete and true if the message operation is still pending. If the function returns with the lock true, the kernel – concurrently with process execution – sets the lock to false after the message operation is completed.

It is rarely necessary or useful to assign to the MSGDESC lock in process code, and then only to initialize it to false immediately after it is declared so that it will be in the correct logical state for an iteration that tests the lock.

After a send or recv, the descriptor may be read but should not be written while the lock is true; similarly for the message area it refers to. This is the usual and basic rule to which concurrent processes that interact through shared storage – the process and its message coroutines – must adhere. Only one process may write at once. When the lock is false, only the process may write; when the lock is true, only the kernel – in its role as "message coroutine" – may write. The lock gets its name from the *intent* that the message

descriptor and buffer are locked against the process writing to these structures while the lock is true. However, there are no runtime mechanisms to enforce this rule.

Whether these rules are adhered to in process code can be checked in source programs. We expect eventually to have available programs similar to the Unix `lint` program to perform rigorous tests that the process code cannot violate a lock. However, in the meanwhile, the cosmic cube programmer needs to be particularly careful about locked descriptors and message buffers.

3.2 The send function

The statement:

```
send(&d);
```

causes the message specified by the message descriptor `d` to be sent or to become pending. In a `send`, `d.lock` is the only message descriptor element written by the kernel. Otherwise, the descriptor is specified by the process as follows:

- `d.node` is the node part of the ID of the process to which the message will be sent.
- `d.pid` is the pid part of the ID of the process to which the message will be sent.
- `d.type` is the message type to be included in the message header.
- `d.buf` is the starting address of the variable, structure, or array from which the message is to be sent.
- `d.msglen` is the length of the message to be sent, in bytes. It is included in the message header, and also determines how many sequential bytes will be sent starting from the address specified in `d.buf`.
- `d.buflen` is unused.
- `d.lock` is the lock variable.

As discussed in section 2.4, a message sent to a process that does not exist within the cube, either by `d.node` being out of range of the logical cube or by the absence of a process matching `d.pid` in the destination, causes the message to be routed to the cube host. If the destination process does not exist in the host environment either, an error message will be directed to `stdout` of the `(HOST,SERVER)` process.

It is not an error for a process to send a message to itself.

After the `send` call, the lock variable becomes false only after the message is completely sent. By "completely sent" is meant that subsequent writes to the message array do not change the message. Hence the message data may be read while the lock is true, but should not be written.

3.3 The recv function

The statement:

```
recv(&d);
```

requests that the message specified by message descriptor `d`, whether it is already queued, or when it is received by the node at some time in the future, be placed in the specified variable, structure, or array.

- `d.node` is irrelevant when the `recv` is called, but when `d.lock` later becomes false contains the node part of the ID of the process from which the message was sent.
- `d.pid` is as above, except applying to the pid part of the sender ID.
- `d.type` is the type of the message to be received.

`d.buf` is the starting address of the variable, structure, or array into which the incoming message is to be written.

`d.msglen` is irrelevant when the `recv` is called, but when the lock later becomes false will contain the length of the message, in bytes, taken from the message header. If `d.msglen <= d.buflen`, the entire message was received. If `d.msglen > d.buflen`, the tail of the message is lost.

`d.buflen` specifies the size of the variable, structure, or array, in bytes, in order to assure that a longer message will not overrun the allowed size.

`d.lock` is the lock variable.

There are no detectable error conditions created by `recv`.

If there was a previously queued message of the specified type in the kernel message buffer when `recv` is called, the oldest such message will be copied into the structure pointed to by `d.buf`, the descriptor information is filled in, and the lock will be false on the return from `recv`. If there is no such queued message, the lock is set to true in the call. The first incoming message that matches the type specification will be placed in the buffer as specified, the descriptor information is filled in, and the lock then becomes false.

Only the type is used to match incoming or queued messages against `recv` descriptors. Incoming messages cannot be matched by sender ID, nor by maximum length, but only by type.

The type may be used to filter messages into different buffers according to any criteria the programmer may care to devise. For example, a point process in a grid point computation may need to send and receive messages of several types from neighbors in four directions. If the direction is coded in part of the type and the distinguished types of messages in another part, one can filter these messages readily into different message areas for appropriate action. The type may also be used to invoke in the object-oriented programming sense a particular attribute or method of a process (object), including the ability of the process to select [Lang 5014:TR:82] amongst a number of concurrently available messages. Of course, the message contents may also be used to discriminate different interpretations of the data. The type mechanism was accordingly not made very general, but it is adequate for the purpose usually associated with data types – making sure that the structure received is conformal to the structure sent. The type may also be used for filtering messages according to other criteria depending on the application and programming style.

There may be concurrently pending `recv`s with descriptors whose type specifications are the same. In this case an incoming message will be matched to the `recv` that was pending first. In the dual situation a `recv` might be executed with more than one message of that type queued in the kernel message buffer. This situation is just the usual case of message order being preserved, and the first queued message (from whatever sender) will go into the message space immediately.

3.4 The probe function

The *probe* developed by Alain Martin [Mart 85] has been adapted for the queued communications in the cosmic cube with a function that allows a process to determine whether a message of a given type is queued for the process. The statement:

```
present = probe(&d);
```

assigns to `present` a value true if a message of this type is present, and also leaves the sender ID and message length in the descriptor. The `probe` function returns the value false and leaves the descriptor unchanged if the message is absent.

Just as with `recv`, with the possibility of more than one message of the specified type being queued, `probe` returns information in the descriptor about the *oldest* such message.

Thus if probe returns true the next recv call with the same type (it would usually be the same descriptor) causes the same message that was probed to be copied into the designated message area.

3.5 Scheduling and the flick and block functions

The scheduler runs all processes in a node in a round robin fashion, including the outer kernel processes. The time each user process is allowed to run is a constant determined by a kernel parameter, or until the process calls the flick* or block functions.

The flick function should be used when a process cannot make progress until one or more locks become false, so that the process might as well defer to another process. The flick function can be thought of as a null operation that incidentally delays execution for a period. You can make a process run at low priority by sprinkling flicks in it.

There is an important point to be understood about flick, namely that it is not analogous to the usual operating system "sleep" call. For example,

```
if (d.lock) flick();        /* flick if d is pending          */
d.type = 3;                 /* WRONG -- may still be pending */
```

is an *incorrect* use of flick, since flick may return before the lock becomes false, and the change of type should not occur while the message action is pending. A correct use of flick is:

```
while (d.lock) flick();     /* wait while d is pending */
d.type = 3;                 /* now you can change d    */
```

There is a good reason why flick works this way, and why flick, rather than a "sleep until a lock changes" call, is the fundamental waiting mechanism provided by the kernel. The conditions for a process to proceed may be more complex than can be specified in a single lock, for example:

```
while ((ch1.lock || ch2.lock) && (urgent.lock)) flick();
```

Here the process is waiting to proceed either until both of the ch1 and ch2 locks are cleared, or until the urgent lock is cleared. There is no critical section mechanism in user processes, and between the time in which the locks are sampled in order to make a decision and the execution of a flick call, one or more of the locks just tested may have changed. However, even if the process decided to flick just before the state of a lock changed, it will be run again next time its turn arrives. Compound "guards" or waiting conditions are thus implemented by busy-wait loops that are made reasonably efficient by the inclusion of the flick no-op.

In cases in which progress is blocked on a single lock, one should use the function:

```
block(&d);
```

which is equivalent to `while (d.lock) flick();`, but which is somewhat more efficient in that the lock testing is performed in the kernel. However, flick is available for the general case of dealing with more than one lock, or for cases in which a process may flick for other reasons, such as to lower its own priority.

3.6 Other IK functions

There are several other functions of the inner kernel that are accessed through C functions as follows:

---

* The expression to "flick it in" is Caltech slang meaning roughly to "give it up".

`mynode()` is a function that returns the number of the node in the logical cube in which the process is running.

`mypid()` is a function that returns the process's own `pid`.

`cubedim()` is a function that returns the dimension of the logical cube in which the program is running.

`clock()` is a function that provides a sampling of a real time value, the same for all processes. It returns a value that should be interpreted as an `unsigned short` that counts mod 2^{16} at approximately 2 msec intervals. A full cycle is just over two minutes. Use of the clock function to synchronize different parts of a computation is strongly discouraged. The function is intended only for making performance measurements.

`led(x)` turns the light-emitting diode for that node off or on for x false or true, respectively.

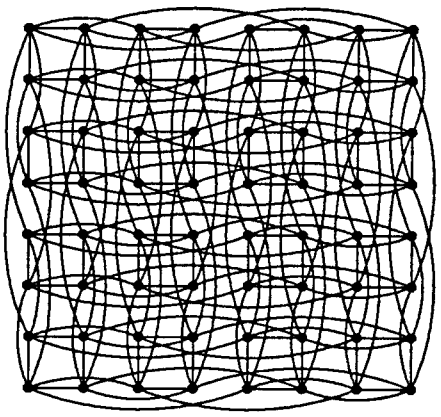
`stop()` places the process into the "suspend" run state (see section 6.1), in which execution is suspended.

`exit()` terminates the process; a process kill (see sections 6.1–6.3). An `exit()` need not be included as the last statement in `main()`. The "return" from `main()` includes an `exit()`.

There are other system calls with corresponding C functions, but they are either privileged to the outer kernel processes or are experimental and not necessarily recommended or supported.

Besides routing messages and scheduling, the inner kernel's other behind-the-scenes activity is storage management. Storage is used both for processes and for the kernel message buffer. Processes are packed into high memory and message buffering into low memory, with a movable partition between them. The queueing of messages of variable size causes the free space in the kernel message buffer to become fragmented. In computations in which processes are killed the process space also becomes fragmented. The partition is moved on demand with repacking occurring only if necessary.

These storage management mechanisms may be of interest to the application programmer in assuring that the total storage demand in a node does not exceed the storage available. After deducting the space used by the kernel, the user process and message buffer space is approximately 107K bytes.



Chapter 4:
Message System Properties

4.1 Message concurrency

The message descriptor is laid out in such a way that if repetitive messages of the same type are expected, `recv` can be called again using the same descriptor after the last reference to a previous message. Similarly for `send`. In fact, the same descriptor could be used for both a `send` and a `recv`, as follows:

```
/* Process that echoes a type 0 message back to the sender */
#include <cube/cubedef.h>
main()
{
    MSGDESC d;                          /* the message descriptor  */
    int a[40];                          /* the message array       */

    d.buf = a;                          /* set buffer address      */
    d.type = 0;                         /* message type = 0        */
    d.buflen = sizeof(a);               /* max length, in bytes    */

    while(1)
    {
        recv(&d); block(&d);            /* recv, wait for lock = 0 */
        send(&d); block(&d);            /* send, wait for lock = 0 */
    }
}
```

This process awaits a message of type 0 and then sends it back as a message of the same type and length to whatever process sent the message. In this example there is no checking whether the incoming message may have been longer than the `sizeof(a)`. If the incoming message were longer, the reply message would be truncated to 40 integers. This is certainly a contrived example, since a process could produce the same result without this "echo" process just by sending a message to itself.

The following slightly more realistic example is a process in which a `send` and `recv` are pending concurrently. The process waits for a message of type 0 consisting of an array of

short integers of length equal to or less than 100, reverses the order of the array, and sends it back to the process from which it received the message. For example, if you sent this process the message $\{1,2,3\}$, it would send you a reply message $\{3,2,1\}$.

The communication structure of a process that performed a more complex computation, such as computing a Fourier transform, would be the same. Whenever a process requires a separate array for its input and output data, one might as well provide two descriptors so that the process can specify message sending to take place concurrently with receiving the next message.

```
/* Process that reverses a message and returns it to the sender */
#include <cube/cubedef.h>

short ra[100], sa[100];                /* the message arrays      */
idesc(rd, 0, 0, 0, ra, sizeof(ra));    /* recv message descriptor */
idesc(sd, 0, 0, 0, sa, sizeof(sa));    /* send message descriptor */

main()
{
    int i,j;
    for(recv(&rd);;)                   /* call 1st recv, then forever  */
    {
        while(rd.lock||sd.lock) flick(); /* wait if send or recv pending */
        if (rd.msglen > rd.buflen)     /* check for too long message   */
           print("Msg too long");      /* report error but keep going  */
        for(i = rd.msglen, j = 0; i--;) /* reverse the list            */
             sa[i] = ra[j++]);
        sd.node = rd.node;             /* setup node part of return ID */
        sd.pid = rd.pid;               /* setup pid part of return ID  */
        sd.msglen = rd.msglen;         /* setup length for send        */
        recv(&rd);                     /* call recv again              */
        send(&sd);                     /* call send again              */
    }
}
```

This program exhibits a typical symmetry between `recv` and `send`. The locks for the `recv` and `send` are tested together, and the last `recv` and `send` could be in either order. Here we show the `recv` first to indicate the preference for a process to `recv` a message at the earliest possible opportunity; that is, as soon as the descriptor it uses is set up and the space is free. Also illustrated here is the `print` function, which is described in section 5.2.

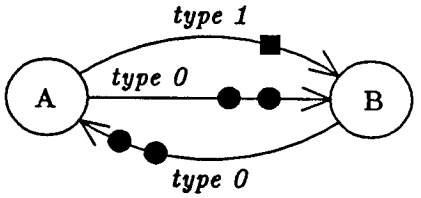
In this second example we show the message descriptors and message data arrays as externals; hence the declaration and initialization of the message descriptors is accomplished using the `idesc` macro defined in section 3.1. In the first example the message descriptor and buffer were automatic variables. Because the initialization of the external variables is free in time and code space, and is easy to read and to type, external message descriptors and buffers are used unless there is some advantage in storage utilization in making the variables automatic.

There is one obvious precaution about message buffers and message descriptors that are automatic variables. Since these variables are allocated dynamically on the stack, a function

should not return while any message referring to an automatic descriptor or message buffer is still pending. Otherwise, a pending send or recv may read from or write into stack area that was reallocated.

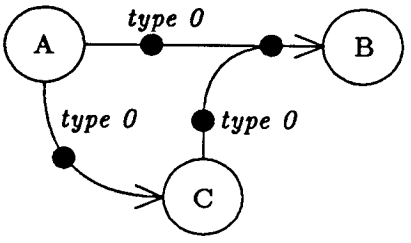
4.2 Message queueing and message order

But for a crucial property of maintaining message order between pairs of processes, the cosmic cube message system works much like a postal system. Messages are stored as necessary in transit and at the destination node. One can visualize this queueing of messages in the process structure graph as a set of message tokens on the arcs:



Messages of different types can be thought of as conveyed on parallel arcs. The oldest message *of a given type* queued at the destination is the one that is read with a recv or sensed with a probe. However, this picture is a bit deceptive. One message is not only unable to pass another on the same arc, but cannot pass another message on a parallel arc. If process A always sent to process B pairs of messages, the first of type 0 and the second of type 1, the type 0 message is assured to arrive at process B before the type 1 message. Because process B might not be running during the interval between message arrivals, it might not be able to "observe" the sequential arrival. However, the presence of the type 1 message is sufficient for the process to know that the type 0 message must also be present. It is not uncommon in cosmic cube programs to depend on this extended property of preserving message order between pairs of processes independent of type, length, or other differences between the messages. The message order preservation properties of the system are in part a consequence of the route of a message being determined only by its source and destination node, and not by the other traffic in the network.

Messages of the same type that arrive at a process from different processes are combined in one queue in the order in which they happen to arrive. We can picture this situation informally as:



where the merging arcs indicate the common queue for incoming messages of the same type. In this process structure suppose that:

step 1: Process A sends a message of type 0 to process B.

step 2: After the message in step 1 is sent, and the lock is false, A sends a message of type 0 to process C.

step 3: After process C receives the message sent by A in step 2, it sends a message of type 0 to B.

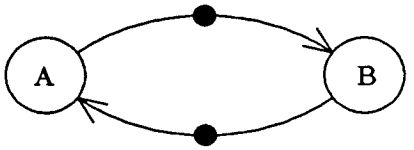
Will the message A sent in step 1 necessarily appear in the input queue of process B before

the message C sent in step 3? The answer is no, for this reason: Although messages always reach their destination *eventually*, nothing can be said about the relative times they spend in transit. If the message $A \rightarrow B$ is delayed in transit, the activity involving messages $A \rightarrow C$ and then $C \rightarrow B$ may be completed first. While message order is always retained between pairs, it is not necessarily maintained in situations involving intermediate processes.

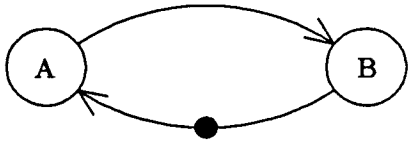
These queued communications include no automatic acknowledgment to the sender when the receiving process has taken the message. This scheme is — but for the message order being preserved — the most weakly synchronized, or most "loosely coupled", form of reliable message communication. This "loosely coupled" mode of communication is motivated both by physical design considerations for this class of concurrent computers, and also by the favorable effects in concurrent formulations of the resulting asynchrony between processes and messages. When the most weakly synchronized message communication is strengthened by the property of preserving message order between pairs of processes, the result is a set of message primitives that are (1) ideal for a very broad class of applications, and (2) sufficient to allow the strongest forms of message synchronization to be expressed simply. Let us now illustrate these points in the two following sections.

4.3 Message queueing example

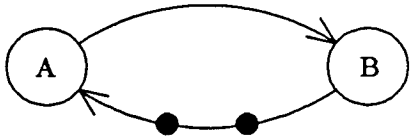
Weakly synchronized communication allows a process to proceed as far as the data dependences in a computation allow. Consider, for example, two processes in a "universal interaction" (see Appendix 1) or "grid point" (see Appendix 3) computation. Such processes start their computation with each sending its state (or boundary) to the other:



Suppose now that process B reads its message and is busy computing its next state, while process A has fallen behind or been blocked awaiting some other message:



Process B may even complete a full state-changing cycle and emit another message before A gets around to receiving the first message:



Process B is now blocked awaiting input, while process A has its inputs queued for it. It is obviously crucial to the computation that the order in which A reads the messages is the same as the order in which B sent them.

Although the programmer may think about point processes in grid point computations as operating in lockstep, they may in execution get as far out of step as the data dependencies allow. One underlying reason that no automatic reply mechanisms were included in the cosmic cube message system is that they would be redundant in a great majority of cases. In the example above, the messages that convey state information are also, in effect, messages

that acknowledge receipt of a previous message. This interdependence serves to bound the number of messages that might be queued on a given channel.

There are other programming situations, such as a process sending a very long stream of results back to a host process, in which the "virtual channel" may queue a volume of messages that eventually becomes limited by the total storage available in the system. This situation is discussed in section 4.5.

4.4 CSP send and receive

Concurrent formulations such as the grid point example are expressed somewhat differently when the message-passing primitives are tightly synchronized. For example, the send and receive primitives in Hoare's *Communicating Sequential Processes* (CSP) notation [Hoare 78] are defined such that the number of completed send and receive operations at the processes at either end of a channel are exactly the same. CSP thus employs message primitives that could not be more tightly synchronized. The same communication primitives are used in Occam [InMOS 84], however in processes that may employ parallel constructs.

If one were to use the CSP primitives in the grid point problem, one could not start the computation simply by each process sending a message to the other, and then receiving a message from the other. Neither CSP send operation could complete until the following receive completed; an "after you, after you" deadlock. If there is a red-black coloring of the processes, the red processes could send while the black receive, and then reverse roles. However, this scheme keeps the whole grid in unnecessarily tight synchrony. The Occam language allows the send and receive in different segments of a parallel construct, which would serve similarly. However, the usual solution is to define "buffer" or "queue" processes between the processes A and B in the example above, where a buffer of two messages will suffice to allow processes to run with maximal asynchrony. In the cosmic cube these queues are implicit, and their size is bounded only by their use.

In situations requiring the tight CSP synchronization in passing messages, one can implement the CSP send and receive operations in terms of the cube's `send` and `recv` by the receiving process sending a reply message. In the cube library versions of `cspsend` and `csprecv` shown below, the reply message is of length 0 – a *synchronization message* – and is of a reserved negative type:

```
#include <cube/cubedef.h>
#define REPLYTYPE -1

    cspsend(d)
    MSGDESC *d;
    {
        MSGDESC replymsg;                   /* message desc for reply */
        send(d);                            /* dispatch the message   */
        replymsg.type = REPLYTYPE;          /* setup type for reply   */
        recv(&replymsg);                    /* call recv for reply    */
        block(&replymsg);                   /* wait for reply         */
    }

    csprecv(d)
    MSGDESC *d;
    {
        MSGDESC replymsg;                   /* message desc for reply */
```

```
    recv(d);                          /* call recv for msg      */
    replymsg.msglen = 0;              /* setup length of reply  */
    replymsg.type = REPLYTYPE;        /* setup type for reply   */
    block(d);                         /* wait for message       */
    replymsg.node = d->node;          /* setup node for reply   */
    replymsg.pid = d->pid;            /* setup pid for reply    */
    send(&replymsg);                  /* send reply             */
    block(&replymsg);                 /* wait for reply on way  */
}
```

These cspsend and csprecv functions are called in exactly the same way as send and recv, and can be used in programs together with the send, recv, and probe functions. The reason that one can say that the number of completed sends and receives over this channel is identical, even though csprecv is obviously able to return before cspsend has received the reply message that allows it to return, is this: Any effect the process executing the csprecv may have on the process executing the matching cspsend will be *seen* later than the reply message, and this is true only because message order is preserved.

One may also implement these CSP primitives with the CSP receive function first producing a synchronization message to which the CSP send replies with the data. The form shown above is preferred if one wishes that the probe function be used as a "receive probe", as it was defined previously. The more general symmetrical probe [Mart 85] in conjunction with CSP send and receive functions requires that both the send and receive functions initiate a communication, and that both reply.

4.5 Congestion and Deadlock

Deadlock is the permanent failure of a concurrent computation to make progress, and is implied in a strongly connected process structure by all processes being blocked, that is, waiting for a lock to change. It is perfectly possible to write programs for the cosmic cube that, because of incorrect sequential dependences on the messages, will deadlock. This is simply a programming error.

The descriptions of the message system properties presented above are adequate only for writing programs that do not encounter the limits of available queueing space in a kernel message buffer. A node that has reached the limits of its message buffer is in a condition that we refer to as *congested.*

As long as all nodes remain uncongested the message system does not impose any sequencing or synchronization of messages and processes that is not inherent in the definition of the computation. When a node becomes congested the kernel must make decisions on the sequence in which it queues messages. These extra sequencing decisions – spurious synchronizations – between messages may introduce a deadlock into a computation that would be free of deadlock if the queues were larger. These spurious synchronizations may also reduce the performance of the message system due to decreased concurrency in the message flow.

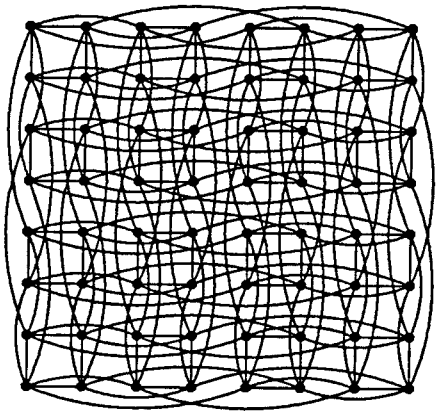
One technique to assure that the message system never introduces a deadlock into a computation is to assure in the concurrent formulation of the problem that the queue space used is strictly less than that available. Queue space is easily bounded in most scientific and engineering applications by the data dependencies in the algorithms. For example, in the grid point computation discussed in section 4.3, the states of neighboring points cannot get out of step by more than one time unit, and the number of messages queued on the pair of message channels cannot exceed two.

The more general and powerful technique to assure that the message system does not introduce a deadlock — in spite of any number of nodes being congested — is to assure that any message that reaches a process is eventually consumed by the process independent of the arrival of or ability to send some other message. The kernel uses a message routing and channel priority algorithm that, in addition to preserving message order, assures that deadlock cannot be introduced into a computation whose processes adhere to the "eventual consumption rule".

For example, consider an event-driven simulation involving the interaction of a large number of objects, each represented by a process. Suppose that the simulation itself did not involve enough messages to cause congestion, but that the reporting of the results of the simulation back to a slow host process could, over a period of time, create more than enough messages to congest every node. The *eventual* consumption of messages by the host process is sufficient to assure that the simulation will continue to run, although not necessarily with the same performance as in the uncongested case.

An example of sequential dependence is a process that cannot consume messages of type 0 until it receives a message of type 1. The queue space necessary to accept the type 1 message may be filled by type 0 messages. Another kind of sequential dependence can be on the completion of **send** operations. A process unable to execute a `recv` while waiting for a blocked **send** to complete also exhibits a sequential dependence.

A point related to the eventual consumption rule is that it is generally most efficient if the `recv` for a message is executed before the message arrives at a node. The message is then delivered directly from the message channel to the process rather than being kept in the kernel message buffer and then moved again. Some computations can be organized so that a `recv` is always executed before the matching message arrives. In this case the kernel message buffer is used strictly for routing messages and not for queueing them for processes in the node. This programming style necessarily satisfies the eventual consumption rule.



Chapter 5:
Library Functions

The way in which the C functions described in chapter 3 work is exactly the way in which the kernel functions work. These functions do nothing more than to set up the parameter passing and execute the system call. The format of the message descriptor is also built into the kernel.

This chapter describes a set of library routines that provide several additional functions for message passing, a `print` function, a function for executing host programs, and functions for dynamic storage allocation.

5.1 Functions that use and manipulate the message descriptor

When a message descriptor is an automatic variable, one may initialize the message descriptor using the "setup descriptor" function:

```
sdesc(&d, node, pid, type, buf, length);
```

The `sdesc` function has the same format as the `idesc` macro defined in section 3.1, but please remember that it is a function and does not declare the message descriptor. The `length` parameter initializes both `d.msglen` and `d.buflen`. The lock variable is initialized to false to be compatible with the `idesc` macro, and so that the lock will be correctly initialized for a first use inside an iteration that tests the lock.

The programmer who may wish to specify all the parameters of a `send` or `recv` as parameters of the function can use the "setup and `send`" function:

```
ssend(&d, node, pid, type, buf, msglen);
```

with the same general format as `sdesc`, or the "setup and `recv`" function:

```
srecv(&d, type, buf, buflen);
```

which is similar except that the `node` and `pid` parameters are omitted. *Caution:* Unlike the regular `send` and `recv`, `ssend` and `srecv` are *not* indifferent to the prior state of `d.lock`. Both of these functions wait for `d.lock` to be false before they copy their arguments to the message descriptor, and then initiate the `send` or `recv`. Hence they may refer to a message descriptor whose lock is true, and do the extra duty of blocking until the lock becomes false. Thus, for example, the following statement:

```
for(i=0; i<ndest; i++)
    ssend(&d1, destnode[i], destpid[i], QTYPE, stv, sizeof(stv));
```

is a good way to send an array stv as a message of type QTYPE to each of a list of ndest destination processes. The necessary waiting on d1.lock is absorbed into the ssend function. The same action would be accomplished using the primitive functions by:

```
block(&d1);
d1.type = QTYPE;
d1.buf = stv;
d1.msglen = sizeof(stv);
for(i=0; i<ndest; i++)
{
    block(&d1);
    d1.node = destnode[i];
    d1.pid = destpid[i];
    send(&d1);
}
```

When one needs to block on a lock immediately after a send, ssend, recv, or srecv, one may use the following "block" variants on these functions:

```
    sendb(&d); = send(&d); block(&d);
    recvb(&d); = recv(&d); block(&d);
ssendb(&d,...); = ssend(&d,...); block(&d);
srecvb(&d,...); = srecv(&d,...); block(&d);
```

5.2 The print and execute functions

The print function displays formated messages on stdout of the (HOST,SERVER) process. The cube's print function is identical to the printf function described in the Unix manual pages except for the addition of a binary format code, %b:

%b	prints the lower 8 bits of the argument.
%nb	prints the lower n bits of the argument.
%n.mb	prints the lower n bits of the argument after first shifting it down by m bits.

The text displayed is preceded by the node,pid of the process in which the print function was executed. It is not necessary to terminate the format string with \n. Since the print interpreter routine in the CUBEIFC process uses Unix's printf to display the messages it receives, all format code variations and prefixes are accepted. The print function avoids congesting the cube with messages by returning only after a reply message is received from the (HOST,SERVER) process.

You might think of print as running printf in the (HOST,SERVER) process. It is actually the CUBEIFC process that expands the message produced by the print function into a character string. CUBEIFC then it passes it on to the (HOST,SERVER) process to print.

The execute function has exactly the same format as print, but the resulting character string is "executed" by the (HOST,SERVER) process as a Unix command. That is, the string is passed as the argument of the Unix system function. The execute function returns whatever value is returned by the Unix system function.

One might use the execute function for mundane purposes, such as running standard Unix utilities, for example:

```
execute("echo Finished >> logfile.%d.%d", mynode(), mypid());
execute("date          >> logfile.%d.%d", mynode(), mypid());
execute("echo I am done | mail vivian");
execute("/usr/lib/sendmail -q");
```

Whether you might like getting mail announcing that a certain process is finished depends, perhaps, on your state of mind.

The primary use of the `execute` function is in initiating host processing activities that need to be coordinated with different phases of a computation in the cube. One can, for example, start up from a cube process new processes in the host, such as a checkpointing program, when the cube computation reaches certain states. One can also execute cube utilities such as `freecube` in the event of a fatal error.

5.3 The `malloc` function and its associates

The usual `malloc` and `free` functions may be used in cube processes for runtime storage allocation. The functions `calloc` and `realloc` available in most other C programming systems are not supported. The functions provided are:

`malloc(n)` returns a pointer to a block of memory n characters long. It prints an error message and returns zero on error.

`free(p)` frees a previously allocated memory block pointed to by p. It prints an error message when errors are detected.

`mallchk()` returns zero and prints error messages when errors are detected in an integrity check; otherwise it returns a true value.

The cube's `malloc` function obtains blocks of unused memory from the gap between the stack pointer and the top of static data. The size of the gap is controlled by the stack size parameter given to `fix86`. The `malloc` function extends its block list from the top of the static data toward and against the stack pointer. If the stack segment size specified in running `fix86` is too small to accommodate both the stack and the allocated blocks, the stack and the data may overwrite each other, with obvious unhappy consequences.

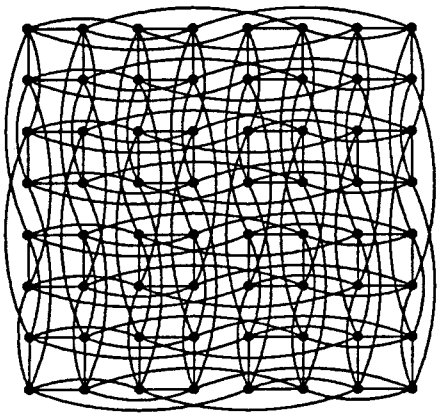
To guard against this and other potential troubles, an integrity check function is provided. The `mallchk` function returns zero when errors are detected. The integrity of the block list is also checked every time `malloc` or `free` is called. When an error is detected by any of these functions, a warning is printed with the `print` function.

The `malloc` and `free` functions are not present in the cube library for host processes; thus the usual Unix functions will be used there. However for compatibility between cube and host processes the `mallchk` function is defined to return a true value in the host's cube library.

5.4 Intel iPSC compatibility functions

This section is reserved for a collection of functions to be inserted for compatibility with the Intel iPSC. The iPSC kernel is expected to support both the `send`, `recv`, and `probe` functions described previously and an alternative form in which the information that is passed by reference to a descriptor is instead passed as arguments. The descriptor is then kept in the operating system, and there are some additional functions provided to access descriptor information. The motivation for this form was apparently to provide better compatibility with FORTRAN.

The descriptions of these functions will eventually be integrated with Chapter 3.



Chapter 6:
Outer Kernel Processes

The outer kernel (OK) is a privileged set of processes whose functions are invoked by messages rather than system calls. We refer to these processes as being part of the system "kernel" because they are always loaded with the system, and are essential. Without the SPAWN process, there would be no way to load programs. Without the SPY process, there would be no way to monitor what is going on in a node.

A user process may equally communicate with the OK processes in its own node, or with the OK processes in any other node. Messages that invoke different operations of an OK process are distinguished by message type, and may produce *reply* messages. In order that these reply messages be distinguishable from each other and from other messages, the type of the reply message is specified in the request message.

Interactions with the SPAWN and SPY processes are most often performed using either host utilities or by functions that do their work with messages. The "raw" form of the message interaction with these process is accordingly not described in this chapter. The message formats for interactions with the SPAWN process are included in section 8.3 in the synopsis. The definitions of constants and types used can be included from `cube/cosmic/cubedef.h`. The message formats for the SPY process can be found in the file `/usr/include/cube/cosmic/spy.h`.

6.1 Process run states

A process exists in one of the following *run states*:

`LOADING` — In this state the `pid` exists and it is possible for messages to be queued for the process, but one or more of its segments is still being loaded.

`READY` — This is the only run state of a process that allows the scheduler to run the process. When this run state is specified as a string, `"r"` or any string beginning with this character can be used.

`SUSPEND` — Execution of a process in this state has been suspended either by the process being created in this state, by the process being set into this state by the SPAWN process, or by the process executing the `stop` function. The `SUSPEND` state is independent of a process being "frozen", a condition that can be created by the SPY process or by certain error conditions. When this run state is specified as a string, `"s"` or `"n"` or any string beginning with these characters can be used.

`DEAD` — A process in this state does not really exist from the standpoint of interaction with other processes. When this run state is specified as a string,

one can use any string beginning with the character "d".

All processes start in the run state LOADING and then, depending on information provided in the spawn request, enter either the READY or SUSPEND state. A run state of a process can be changed from SUSPEND to READY by a message to the SPAWN process, and can be changed from READY to SUSPEND either by a message to the SPAWN process or by the process executing the stop function. Once a process has been put into DEAD state, either by a message to the SPAWN process or by the process executing the exit function, it cannot be changed to any other state, and will soon be gone.

6.2 SPAWN process messages

This section presents only an abbreviated description of the message interaction with the SPAWN process. The formats of the messages to and from the SPAWN process are given in section 8.3 in the synopsis. Skip to the next section to learn about the functions that use these mechanisms.

A transaction to create a process in node i starts with a message from process (j,k) to process $(i,$SPAWN$)$. We have already mentioned that because spawning is accomplished in response to messages, host processes may use this mechanism as well, and this is how programs are initially loaded.

A message of type FSPTYPE requests that a .cos *file* be instantiated as a new process in node i. The message conforms to a defined type FSPMSG, and conveys the file name, desired pid, run state, and a type for the reply message. If the desired pid is specified as -1, the SPAWN process will select an unused pid.

What actually transpires is that $(i,$SPAWN$)$ sends a message to (HOST,SERVER) inquiring about a .cos file corresponding to the root file name conveyed in the file spawn message. Thus the file name is relative to the current working directory of the (HOST,SERVER) process. The (HOST,SERVER) process sends information back to $(i,$SPAWN$)$ indicating the existence and size of the segments of this process. If the message indicates that the file exists, the SPAWN process attempts to allocate storage for it, and if successful sends a message to (HOST,SERVER) to retrieve the code and data segments. Various errors may be detected in the SPAWN sequence, and cause the sequence to terminate and the reply message to be returned with an error code, which is always negative. The meanings of the various error codes can be found in the include file cube/cosmic/cubedef.h. The reply message to a successful process spawn conveys the pid of the new process, which is always positive.

The first 10 characters of the base file name from which a process is spawned is retained in a process descriptor block in the kernel. If the process is later replicated this name goes with it. The name is not used for anything in the computation, but is accessed by the SPY process and is displayed by the "cube process status" (cps) utility.

For a *process spawn*, the message to $(i,$SPAWN$)$ is of type PSPTYPE and conforms to the defined type PSPMSG. This message specifies the node and pid of a process to be replicated, the desired pid and run state for the new process, and a reply type. As with the file spawn, the desired pid may be specified as -1 to cause the SPAWN process to choose an unused pid. The process to be replicated may be in the same or in a different node than the SPAWN process. What transpires is an interaction with the SPAWN process in the source node that is much like the interaction with (HOST,SERVER) for a file spawn. If the process to be replicated is located in the same node, the code segments are shared.

The process created contains no reference to the process that caused its creation.

A message of type RUNTYPE to a SPAWN process causes it to change the run state of a specified process in its node. The message is of defined type RUNMSG, and conveys the pid

of the process effected and a state code defined by the constants READY, SUSPEND, or DEAD. No reply is produced to this message. A message to change the run state of a non-existent process causes an error message to be sent to the (HOST,SERVER) process.

A process that is made DEAD will not disappear until all pending send actions are completed, or, if the process were in the middle of being copied, this copying action would be completed before the space and pid for the process is reclaimed. When a process is to change its own state to SUSPEND or DEAD, it should use the stop() or exit() functions, respectively, in that they take effect immediately, not after some delay in the message system and SPAWN process.

A process should kill itself upon termination, so that its space and pid may be reused. The programmer should avoid the error of retaining references to processes that have terminated, as well as the error of leaving unreferenced processes lying around. For example, if a process has created a number of processes to which only it has reference, it could either send them messages to kill themselves before killing itself with the exit function, or if there were some assurance that there were no messages enroute to the process, could kill these processes directly.

Messages queued for a process are deleted along with the process, but messages that may arrive after a process has disappeared will generate a rash of annoying "message for non-existent process" error messages back to the host.

6.3 SPAWN convenience functions

The functions of the SPAWN process are generally invoked by the following functions instead of by messages.

The statement:

```
pid_or_error = spawnf(filename, node, pid, state);
```

causes the *file* filename.cos to be spawned as process (node,pid) in the specified run state. The statement:

```
pid_or_error = spawnp(snode, spid, node, pid, state);
```

causes the *process* (snode,spid) to be replicated as process (node,pid) in the specified run state. The state specified in the spawn functions is a string, either "r" for READY, or "s" or "n" for SUSPEND. The pid may be set to -1 to cause the SPAWN process to select an unused pid. The value returned by these functions is either the pid assigned, which is always positive, or an error code that is always negative.

As an illustration of the use of these functions, here is how spawnp might be used in code that allowed for it to fail:

```
i = mynode();
j = mypid();
if ((code=spawnp(i, j, i ^ 1, j, "r")) < 0)
{
    print("failed spawn, code %d", code);
    stop();             /* suspend myself */
};
```

Here we suppose that the process attempts to create a new copy of itself in the node adjacent to its own node along dimension 0 (node $i \oplus 2^0$). The new process is to have the same pid. If this operation fails the process prints an error message including the code returned, and suspends itself with the stop function.

The statement:

```
ckill(node, pid, state);
```

causes process (node,pid) to be placed into the specified run state. The allowed run states are "r", "s" or "n" – as described above –, or "d" for DEAD.

Broadcast spawn

With the node argument set to -1, the spawnf and spawnp functions create an instance of a specified process in every node. The pid must be specified; it may not be set to the -1 wildcard.

The way the functions accomplish this broadcast spawn operation is first to spawn a copy of the utility broadcast spawn process, which distributes itself to every node in a "time-on-target" tree (similar to the program illustrated in Appendix A2). It then distributes copies of the specified process to every node by the same algorithm. Because the broadcast spawn process distributes copies of itself or of other processes in this concurrent scheme, it requires time proportional only to the dimension (rather than the number of nodes) of the cube. The broadcast spawn process uses a reserved negative pid.

The Real Estate Agent

This section to be expanded with a description of the *real estate agent process* that acts as an intermediary in spawning a process in a node that is unspecified, but in a relatively low-rent area. This process uses the SPY process.

6.4 The cfork family of functions

The cfork function replicates the calling process, creating a child process whose node and pid are given in the call. The return from the cfork function occurs only after the child process is successfully spawned or after the operation has failed. For the cfork function and its variants to follow, a successful call returns 1 for the parent, 0 for the child, or a negative error code if the child process could not be created. The following is an example of the use of cfork:

```
switch(cfork(node,pid))
{
    case 1:     parent();               break;
    case 0:     child();                break;
    default:    print("cfork error");   break;
}
```

Notice the differences between the cfork and spawnp functions. The spawnp function makes a copy of an arbitrary existing process, and starts the new process with the program counter and stack pointer at their initial values. The cfork family of functions makes a copy of the calling process, and starts the child executing with the program counter and stack pointer at the same position as in the parent.

The gfork (global fork) function uses the cfork function to create a replica of the calling process in all nodes except the node of the calling process, and with the same pid as the parent. The gfork function returns only after all child processes are spawned, or after the operation has failed. If the operation fails there are no children created. The following is an example for the use of gfork:

```
main()
{
    if((Im_root = gfork()) < 0) {
        print("unable to gfork");
```

```
        exit();
    }
    .
    .
}
```

The ccore function uses the cfork function and a host utility to create a restartable core image file of the calling process. The function ccore forks the calling process out of the cube by specifying that the child process have an ID of (HOST,SPAWN). An associated host utility with this ID intercepts the various program segments.

The host utility coreman can be run to store these program segments in a .cos file†. The .cos file stored by the coreman utility can later be spawned as a process which picks up execution immediately following the ccore function call. The ccore function can thus be used for backing up a long running computation when it has reached a certain point in execution. For example:

```
if(ccore() < 0) print("checkpoint error");
```

is sufficient to create the restartable backup copy if coreman is running anywhere in the host environment. This use of ccore does not make use of the return information that tells which is parent and which is child. A use of ccore that requires this information is saving a branch of a tree-structured computation for future use, for example:

```
switch(ccore())
{
    case 1:     this_way();          break;
    case 0:     that_way();          break;
    default:    print("ccore error"); break;
}
```

In this case the return value from ccore, 1 for the parent that keeps running, 0 for the child that is stored as a file, determines the future course of execution both for the parent that continues running and for the child that is backed up and may be spawned later.

An important property of this family of functions is that the child inherits only the code, data, and stack segments from the parent. Pending receives, pending sends, and queued messages are not replicated. The construct below is a fatal mistake:

```
recv(&input);
cfork();
block(&input);
```

Because the kernel of the child process was never notified of the receive action, the process will never return from the block call, even though a message of the type specified by the descriptor may have arrived. Thus the functions in this family should be called only when the process is in a state in which there are no pending message operations. Of course, queued messages remain for the parent process to deal with.

The cfork and gfork functions are also available for host processes; however, new processes are created only in the host of the calling process. The ccore function exists for host processes only for compatibility, and prints a message when it is called.

---

† Other host utilities may be developed in the future to perform other functions on the program segments, such as to display them or to perform process swapping.

6.5 SPY process

A variety of messages, distinguished by type, may be sent to the SPY process in order to check the condition of processes and messages, and to determine and change process segments in its node. The details of the SPY process will not be described here as they are unlikely to be pertinent to users. The SPY process is used by the "cube process status" (cps) utility, the "cube load average" (cla) utility, and the real estate agent process.

A display of the cube process status, similar to the Unix ps, can be obtained with the cps utility. The cps utility defaults to showing only the user processes in node 0. However, it contains numerous options that can be displayed with the -u option:

```
(venus:33) cps -u
usage: cps [-aomlL] [-nnn] [-pnn] [-Nxx] [name]
          -o: show outer kernel processes
          -m: show memory usage
          -l: give long description of each process
          -L: give very long description
          -a: show all nodes
          -s: show nodes running each process
        -nnn: show only node nn, or all if nn == 'a'
        -pnn: show only process nn
        -Nxx: list only processes whose name is "xx"
        name: sets group to "name"
          -u: print usage message
switches may be grouped together whenever it makes sense.
```

For example, let us display the process status including the outer kernel processes (-o), in node 33 (-n33):

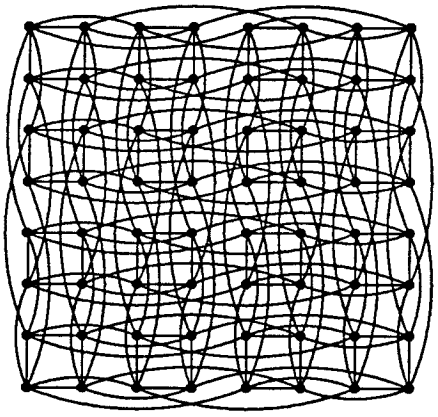
```
(venus:34) cps -o -n33

NODE  PID  STAT  PROCESS
 33     0   S    seed
 33     1   R    seed
 33     2   R    crystal
 33  -128   R    SPAWN
 33  -127   R    SPY
```

The cube load average (cla) utility allows one to monitor the computational load and its balance in all of the cube nodes. It works by entering the process group and sending a message that traverses all of the SPY processes, and then produces a display such as:

```
(venus:35) cla
Loading for a 6-cube:
             AVG      MAX         MIN        SDEV
load avg:   0.92   1.00( 0)   0.04(63)   0.08
free mem:  17664  17664( 0)  17664( 0)      0
```

This display shows that the average time the nodes are spending computing rather than with all processes blocked is .92 – an RMS average across all of the nodes. The largest load

is 1.00, which occurs in node 0 and possibly elsewhere, and the smallest load is .04, which occurs in node 63. The standard deviation of the loads is .08. The display also shows the storage utilization, which in this example is the same for all nodes.



Chapter 7:
Host Functions

7.1 The Cosmic Environment

The *cosmic environment* is a library of functions and a runtime system that supports on Unix host computers the same message-passing functions that exist for cube processes. The library contains functions with the same names and semantics as the primitive and library functions described in chapters 3–5. The `spawnf`, `spawnp`, and `ckill` functions described in chapter 6 are also present in the cosmic environment, but operate only on cube processes, not on host processes. The `cfork` family of functions also exists in the hosts; however, new processes are created only in the machine in which the calling process is running.

This environment also provides an interface between the host machines and the cosmic cubes that enables uniform communication among processes on both sides of the interface. The underlying communication mechanism used in the hosts, Internet sockets, allows processes running on any set of network-connected machines to communicate with processes in the cube and with each other. This environment makes the machine boundaries transparent, except for the byte ordering problem that all networks face.

The cosmic environment can be used with or without a cube. Without a cube it is a system that allows one to run multiple process programs with processes all on one Unix system or distributed across a number of network-connected systems. Messages are passed between processes in the same way as occurs in the cosmic cube. Programs developed in this environment and under certain disciplines can be run on cubes with no more effort than recompiling the process code. When used with a cube, the cosmic environment is the means by which cube processes communicate with host processes. The conventions by which the messages to nonexistent processes are rerouted to the host work together with the cosmic environment to allow cube processes to be relocated into the host. These processes can then be run under a debugger.

This environment also provides an allocation mechanism by which users can share a set of cubes, and a variety of utility programs that simplify user interaction with the system.

Each cosmic environment includes a single dæmon process, the *cube dæmon*, or `cubed`, running on a predetermined host. There is also one cube interface program, `CUBEIFC`, for each cube. Each `CUBEIFC` process must run on its cube's host, and it introduces both a interface into the cosmic environment and a gateway into its cube. The `cubed` process can run on any host. It maintains an account of active host process groups and their affiliation, and an account of available cubes and their allocation. With this information `cubed` oversees the rendezvous of process groups and logical cubes.

7.2 User Interface Routines

Since the physical environment in the cube is different from that in the hosts, there are a few inevitable differences between functions of the same name. Some functions are meaningless; for example, `led` defaults to a no-op function in host processes. The `clock` function is approximated. The `malloc` and `free` functions are the usual Unix functions, and `mallchk` always returns true. All other functions are semantically equivalent to their cube counterparts, but the implementation of some of the functions under Unix has different performance characteristics than their implementation in the cube.

One can best appreciate some of the more subtle differences between the cube and host functions by understanding how the functions incorporated into host processes manage to "simulate" their counterparts within cube processes. Message descriptors in the cube nodes are updated asynchronously by the cosmic kernel. However, since the hosts and their operating systems do not provide for servicing these descriptors in the same way, the descriptors are updated either when message functions are called or when a periodic interrupt signal – the "snooze alarm" – arrives. The periodic interrupt signals are set to occur infrequently in order to avoid tying up host resources. With such infrequent interrupts, the usage:

```
recv(&d);
while (d.lock);  /* busy wait for lock -- BAD construct */
```

is particularly bad in a host process, because it may cause the process to idle loop for several seconds before the interrupt that changes the state of the lock arrives. It is much better to use `flick`, as in:

```
recv(&d);
while (d.lock) flick(); /* lock is changed inside of flick */
```

so that the lock is changed inside the `flick` function.

A version of `flick` exactly as defined for the cube would in any case be inefficient in the hosts. In the cube the construct above constitutes a poll on the receive descriptor lock with the `flick` being used as a null operation. In the cosmic environment `flick` puts the process to sleep until a message arrives or until its "snooze alarm" reawakens it. A `flick` call in the host process with no prospect of messages coming, as might be done to lower the priority of a process, puts the process to sleep for several seconds.

Beyond the set of cube functions described previously, the cosmic environment also includes a set of functions specific to the host processes, and which are defined as null functions for the cube processes. The set of functions for interfacing host processes to the cosmic environment are:

`cosmic_init(node,pid)` sets the `node` and `pid` of this host process and enters the cosmic environment. The `node` defaults (if specified as -1) to `HOST`, and `pid` defaults to 0.

`cosmic_exit()` gracefully exits the cosmic environment.

The following functions may be called *before* `cosmic_init` in order to set the parameters of the environment.

`cosmic_set_alias(alias)` specifies a string `alias` to be used in place of the name of the calling process when the status of process group is inquired. The default `alias` for a process is the name by which the process is invoked.

`cosmic_set_group(group)` specifies in the string `group` the process group in which this process is to participate. It defaults to the string `"group"`.

`cosmic_set_cubedhost(cubedhost)` specifies that this process should participate in the domain of the cosmic environment whose `cubed` process resides in the named `cubedhost`. The host name is a string that may correspond to the Internet address (in `xx.xx.xx.xx` notation) or to any of the hostnames or nicknames of the host to be found in the system host table. At Caltech this parameter defaults to the string `"cit-sol"`. The default `cubedhost` should be changed when the software is run at other sites.

These "`cosmic_set_`" functions may be called multiple times before calling `cosmic_init`. Parameters may be made to revert to "untouched" state by setting numerical parameters to -1 and string parameters to `NULL`. Since the current parameters are sent to `cubed` when `cosmic_init` is called, the `cosmic_set_` functions should not be called after `cosmic_init` is called. If `cosmic_init` has not been called before calling the first message function, it will be called by the message function with the `node` and `pid` set to the defaults.

Programs written for the cosmic environment must be linked with the cube library, *eg*:

```
cc -o prog prog.c -lcube
```

This library is in the file `/usr/lib/libcube.a`.

7.3 Setting process parameters at run time

Parameters such as `(node, pid)` and the `alias` of a host process are usually defined as constants in its program or are computed and set by the process. Some parameters, however, may apply to all processes in the same group, but may change from one job to another. These parameters, such as the group name, are better set at run time.

Those parameters that remain untouched by a process may be defined for that process at run time with the Unix environment string, "`cube`". The string is composed of pairs of words separated by spaces in the form of a parameter name followed by its value. Valid names are:

`node` sets the node of the process.

`pid` sets the pid of the process.

`dim` sets the dimension of the cube to use.

`alias` sets the process name to be printed in status displays.

`group` sets the process group name.

`cubedhost` sets the host on which the dæmon is to be found.

`type` sets the type of cube to be allocated. Valid cube types are `cosmic_cube`, `ipsc_cube`, and `non_cube`. The types may differ from one installation to another.

For example, typing "`setenv cube "dim 6 group YumYum"`" to the C-shell causes the `getcube` utility invoked in that shell to request a cube of dimension 6 if the parameter was not specified, and causes any host process to participate in the process group `"YumYum"` if it did not set its group name. Any parameter names may be replaced with unambiguous prefixes, for example, `setenv cube "d 6 g YumYum"`. The same applies to the cube type.

Parameters untouched in the previous two levels may be set with the format described above in the .cube file in the user's current working directory. Settings specified in this file may apply to many processes, just as those in the previous level, except that the settings specified in a .cube file are typically static and pertain to the physical requirements of the processes. For example, in cases in which there are different types of cubes at one site, the particular type for which the cube programs are compiled may be inscribed in the directory where these programs are kept by specifying the type in a .cube file in that directory.

7.4 The Cube Dæmon

The cosmic environment functions in user programs and in utilities rely on the mechanisms described in this section, but how they work may be of only "cultural" interest to the application programmer.

Like other dæmon processes, cubed starts running as soon as the host is booted, and goes right to sleep waiting for requests from user processes and CUBEIFC processes. The cube dæmon is principally a resource allocator. All processes in the cosmic environment, including the CUBEIFC processes, must first secure a lifeline socket connected to the cubed process. For the CUBEIFC processes and the getcube processes, these lines carry initial parameters exchanged when processes enter the environment. For other host processes, after the initial parameter exchange, these lines are promptly transferred from cubed to their getcube process. If such a line ever breaks, the associated process will die or, in the case of CUBEIFC, will cause the process to restart. The socket connection is the token of right for a process to participate in the cosmic environment.

The CUBEIFC processes are similar to dæmons and are started at boot time. They do not go to sleep, however, until they have successfully rebooted the cube they control and have made connections to the cubed process within a predetermined number of tries. When a CUBEIFC process has successfully secured its lifeline to cubed it identifies itself as the controller of a cube of a certain type and dimension, and the cubed process creates a new entry in its list of active cubes.

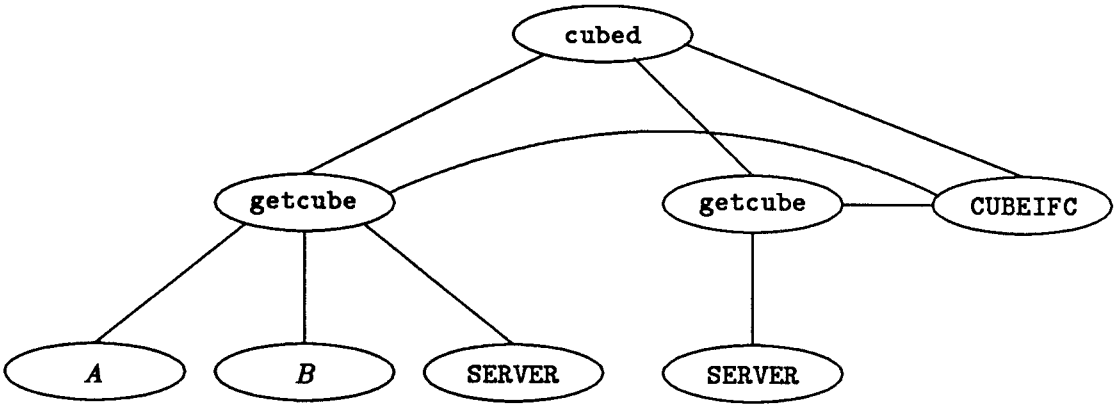
When a CUBEIFC process exits the environment, as when it is killed, cubed will delete its entry from the list of cubes and cut the lifelines of all the groups using that cube. When CUBEIFC dies, it will restart itself automatically, and reboots its cube with a fresh copy of the kernel.

Cubes are allocated with the getcube utility. The type and dimension of the cube to allocate may be specified as arguments. When a getcube process makes a connection to the cubed process, cubed tries to allocate to that process group a cube with dimension no smaller than that requested by that process. If "space-sharing" is enabled, the cube it allocates may be a subcube of a larger cube. If the allocation fails, the process is rejected and its lifeline is cut. If it succeeds, the rest of the processes in the same group are assigned to the same cube as they enter the environment. Each process will receive the size of the cube allocated in reply to their initial connection to cubed. The allocation algorithm awards the smallest cube that fits the requested dimension, with preference given to cubes directly connected to the host of the requesting process.

The configuration of lifeline sockets forms a two level tree with cubed at the root, the CUBEIFC processes as leaf vertices on the first level, the getcube processes as vertices on the first level, and the host processes including (HOST,SERVER) as the leaf vertices on the second level descending from their group's getcube process. Messages from one host process to another go through their socket connections to the getcube process, which both buffers and enforces a flow control discipline on the messages.

For example, the following configuration of host processes and socket connections corre-

sponds to an installation in which there is only one cube, hence only one CUBEIFC process. There are two process groups, each with their own getcube and SERVER processes. One of these process groups includes the user host processes *A* and *B*, while the other process group is getting by with only its SERVER process.



7.5 Unix utility programs for the cosmic environment

A set of five utility programs are currently available for use in the cosmic environment. In the present casual environment any user may generally run any of these utilities, but protection mechanisms are provided for those who want more security.

The `restart` *program*

The `restart` program commands cubed to die and to replace itself with a fresh instance of cubed. This program is usually used when a new version of cubed needs to replace the one currently running. It may also be used to clean up a hopelessly screwed-up situation. If the `-r` flag is set in running cubed, restart across network is disallowed, and only the owner of the cubed process may restart it. If the file `/usr/local/cube/legal_hosts` is present in the cubed host, access to the environment is restricted to those hosts listed in the file.

The `killcube` *program*

The `killcube` utility is used to restart a dead cube when the kernel has crashed or been corrupted by a runaway process. It causes cubed to cut the lifeline of the CUBEIFC process associated with the group from which `killcube` is executed. As a result, the CUBEIFC process will restart itself and will reboot the cube it controls with a fresh copy of the kernel. *Caution: All* host and cube processes in groups that are sharing space within that cube will be purged from the environment. One should use the `freecube` utility if the objective is only to exit the process group and to kill the processes in your part of the cube. If the `-k` flag is set in running cubed, CUBEIFC killing across network is disallowed and only the owner of the CUBEIFC process is allowed to kill it.

The `getcube` *program*

The `getcube` program takes an optional dimension parameter and an optional cube type parameter and sends them to cubed as the specification for the cube to acquire. If a cube of type non-cube is specified, `getcube` serves only to establish a process group and to specify the dimension to be returned by the `cubedim` function, but does not allocate a cube. Otherwise, it also tries to allocate a cube of this dimension, and only if successful will

it enter the environment. This process forks out the (HOST,SERVER) process for the specific cube it owns, and puts itself into the background. The default dimension and cube type may vary at different installations. At Caltech the default is a cosmic cube of dimension 3.

The `freecube` *program*

The `freecube` program commands cubed to cut the lifeline to the processes in this group. This program results in all processes in the process group being killed, so as to free the allocated subcube. If the `-f` flag is set in running cubed, `freecube` across network is disallowed. The `freecube` program accepts a group name as optional parameter.

The `peek` *program*

The `peek` program prints the environment status and group status in the following format. The example below was produced with a peek from inside of the group `wing` by user `chuck`.

```
CUBE DAEMON version 6, up 5 days 14 hours on host sol

   {                } 4d cosmic_cube, b:0000 [    sol 6-CUBE]  3.8d
   {      TV wen-king} 4d cosmic_cube, b:0010 [neptune 6-CUBE] 40.1m
   {    wing chuck   } 5d cosmic_cube, b:0020 [ icarus 6-CUBE]  4.0m
   {wargames reese   } 3d cosmic_cube, b:0000 [neptune 3-CUBE]  4.6d
   {corvette bill    } 2d non-cube,    b:0000 [    sol       ]  1.2h

GROUP {wing chuck}:

   ( -1  20)   red_baron   25s    0r    0q   [ icarus 27292] 10.0s
   ( -1  14)         LAX  115s   25r   20q   [LAX-VAX   788]  4.0m
   ( -1  -1)      SERVER    9s   18r    0q   [ icarus 27285]  4.4m
   (--- ---)     CUBEIFC   22s   87r   21q   [    sol  1173]  4.4m
```

The first part of this status display contains general information about the environment. Each entry with a non-empty group field represents a process group. The format of the entry is:

```
   { wing chuck } :: group name and username of the process group.
 5d cosmic_cube, :: type and dimension of the subcube allocated.
         b:0020 :: base node, in hexadecimal, of the subcube allocated.
       [ icarus :: host on which the group is created.
      6-CUBE ] :: the name given to the cube by the CUBEIFC process.
          4.0m :: the amount of time since the creation of the group.
```

If the group field is empty, the entry represents an available subcube. The format of such an entry is:

```
              { } :: empty, meaning unused.
  4d cosmic_cube, :: type and dimension of the subcube.
         b:0000 :: base node, in hexadecimal, of the subcube.
           [ sol :: host of the CUBEIFC process that owns the cube.
      6-CUBE ] :: the name given to the cube by the CUBEIFC process.
          3.8d :: the time since the last reboot of the cube.
```

The second part of the display contains information pertaining to a process group. This part will be absent if the peek utility is not executed from inside of a group and was not told

to look into any specific group. Each entry represents a process connected to the getcube process of this group. The format of such entries is:

```
( -1  14) :: node and pid of the process in decimal.
      LAX :: name of the process.
     115s :: number of messages sent from the process.
      25r :: number of messages delivered to the process.
      20q :: number of messages waiting to be received by the process.
[ LAX-VAX :: host on which this process is running.
    788 ] :: pid as defined in the host's operating system.
     4.0m :: the amount of time since the process enter the group.
```

The above display shows that cubed has been running for 5 days and 14 hours on host sol. Two cubes are present in the environment. The first, the 6-CUBE, is owned by host sol and is fragmented into three subcubes. A 4-D subcube with base address 10-hex is allocated to user group (TV wen-king). A 5-D subcube with base address 20-hex is allocated to user group (wing chuck). The remaining subcube is free. The second cube, the 3-cube, is wholly allocated to the process group (wargames reese). The host to which that cube is attached cannot be seen from this display because the name neptune reflects the host that started the group. The process group (corvette bill) has no cube allocated, thus all processes in this group are on the hosts.

User bill is running a race track simulator with cars spread out over four VAXes, and is trying to find out why his car skidded while rounding a corner in the cube. User chuck is studying the latest changes in the Federal Aviation Regulations (FARs), and is trying them out in a mock attack on the tower at LAX. User reese is trying to predict the outcome of the Iran-Iraq war. Meanwhile, user wen-king is running signal interceptors and decoders, and is watching something whose description is not fit for print.

7.6 Byte-order and other incompatibilities

The data representations in the VAX hosts, in the SUN hosts, which are based on a Motorola 68010 processor, and in the cube nodes, which are based on the Intel 8086/8087 processors, have several annoying incompatibilities:

- A C int is the same as a short in the 8086, and the same as a long in the VAX or 68010. Thus one should not include int types in messages between the host and cube nodes, but should specify either a short or a long integer.
- The 68010 processors use opposite byte-order conventions from the VAXes and 8086s. The hardware and software interfaces are arranged to guarantee that character strings sent across any interfaces are in proper order. There are conversion routines described below for all other C types.
- The way in which elements in heterogeneous structures get padded out to accommodate the different addressing modes in the types of processors, and by different compilers, may be different. We might not have covered up all the pitfalls here.

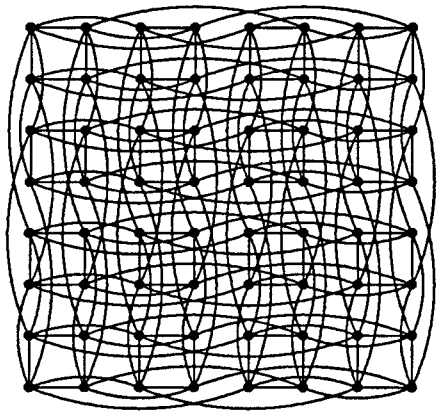
A set of bit-banging routines is included in the library of each host, and is summarized in section 8.6 in the synopsis, to assist in writing host-independent processes. The htoc[s|l|f|d] routines translate from that host's format to the cube's format, while the ctoh[s|l|f|d] routines translate from the cube's format to that host's format. In all cases the routines convert an array of cnt variables of the corresponding type, either short (s), long (l), float (f), or double (d). Since the order of character strings is guaranteed, character conversion routines are unnecessary.

These routines can be used at every opportunity, even in cube processes, where they are defined as null functions, in order to make source process code completely host-independent. Typical usage is to convert a structure immediately before it is sent and immediately after it is received. For example, for a message conveying an array `double vector[XSIZE]`:

```
htocd(vector,XSIZE);     /* convert vector to cube format      */
sendb(&tchan);           /* send it to another process         */
recvb(&tchan);           /* get the transformed vector back    */
ctohd(vector,XSIZE);     /* convert back to this host's format */
```

However, a process that would only be used in a cube or between hosts of the same type might as well omit these conversions routines.

The floating point conversions are not quite perfect. The cube nodes (8087s) and SUNs generally conform to the IEEE standard, and are compatible but for byte-order. For 32-bit floating point, the IEEE standard and the VAX have same precision, but a slightly different exponent range (differs by a factor of 2). For 64-bit floating point, the IEEE standard has 11-bit exponent but VAX has 8. The VAX floating point is more precise but has a smaller range. Therefore, when floating-point numbers are converted to and from VAX format, some numbers will get "crunched".



Chapter 8: Synopsis

8.1 Inner Kernel Functions

Message descriptor:

```
typedef struct{
                short           node;
                short           pid;
                short           type;
                short           seg;
                char            *buf;
                unsigned short  msglen;
                unsigned short  buflen;
                short           lock;
              } MSGDESC;
```

Macro for declaring and initializing external message descriptor:

```
idesc(desc, node, pid, type, buf, len);

    MSGDESC desc;
    short node, pid, type;
    unsigned short len;
    char *buf;
```

node *and* pid *constants:*

```
HOST      /* (  -1) node parameter for host processes */
MAXUPID   /* ( 127) maximum user pid (currently 127)  */
SERVER    /* (  -1) defined in cube/cosmic/cubedef.h  */
SPY       /* (-127) defined in cube/cosmic/cubedef.h  */
SPAWN     /* (-128) defined in cube/cosmic/cubedef.h  */
```

node, pid, *and* type *allowed ranges*

$0 \leq$ node ≤ 63 for cosmic cube processes

$0 \leq$ pid $\leq$ MAXUPID for user processes

$0 \leq$ type ≤ 32767 for user messages

Inner Kernel message functions:

```
send(d);
recv(d);
probe(d);
flick();
block(d);

    MSGDESC *d;
```

Other primitive functions:

```
mynode();
mypid();
cubedim();
clock();
led(on);
stop();
exit();

    int on;
```

8.2 Library functions:

Conveniences:

```
sdesc(d, node, pid, type, buf, length);
ssend(d, node, pid, type, buf, msglen);
srecv(d, type, buf, buflen);
sendb(d);
recvb(d);
ssendb(d, node, pid, type, buf, msglen);
srecvb(d, type, buf, buflen);
cspsend(d);
csprecv(d);
print(format [,arg] );
execute(format [,arg] );

    MSGDESC *d;
    short node, pid, type;
    unsigned short msglen, buflen, length;
    char *buf;
    char *format;
```

The print, execute, and cspsend functions use message type -1 for a reply message.

Storage allocation

```
char *malloc(len);
free(p);
mallchk();

    char *p;
    int len;
```

8.3 SPAWN process messages:

The definitions in this section can be included from `cube/cosmic/cubedef.h`.

SPAWN process pid*:*

```
SPAWN
```

Run state constants:

```
LOADING, READY, SUSPEND, DEAD
```

File spawn message:

```
typedef struct{
                short pid;          /* requested new pid, -1 for any */
                short state;        /* run state of new process      */
                short rtype;        /* type for the reply message    */
                char name[NAMLEN];  /* file name                     */
                } FSPMSG;
```

which is sent with type FSPTYPE. (The constant NAMLEN = 96).

Process spawn message:

```
typedef struct{
                short pid;      /* requested new pid, -1 for any */
                short state;    /* run state of new process      */
                short rtype;    /* type for reply message        */
                short snode;    /* node of proc to be duplicated */
                short spid;     /* pid of proc to be duplicated  */
                } PSPMSG;
```

which is sent with type PSPTYPE.

Spawn reply message:

```
typedef struct{
                short code;     /* success code is 0, error negative */
                short pid;      /* pid of process created */
               } SREPLY;
```

which is received with the rtype specified in the file or process spawn message. (This message will be changed in the near future to a single short, positive for a pid, negative for an error code.

Run state changing message:

```
typedef struct{
                short pid;    /* pid of process effected */
                short state;  /* new run state           */
               } RUNMSG;
```

which is sent with type RUNTYPE (and there is no reply).

8.4 SPAWN convenience functions

Spawn functions

```
spawnf(name,         node, pid, state);
spawnp(snode, spid, node, pid, state);
ckill(               node, pid, state);

    short snode, spid, node, pid;
    char  *state;
    char  *name;
```

The **spawnf** and **spawnp** functions return the **pid** assigned, always positive, or a negative error code. If the **node** parameter of **spawnf** or **spawnp** is **-1**, the process is spawned in all nodes. If the **pid** parameter of **spawnf** or **spawnp** is **-1**, the SPAWN process selects a **pid**. The **node** and **pid** parameters may not both be **-1**.

Fork functions

```
cfork(node, pid);
gfork();
ccore();

   short node, pid;
```

These functions return 1 to the parent, 0 to the child, or a negative error code.

8.5 Compiling

Set **/usr/cube** to be in your search path.

Include file (all cube or host processes):

```
#include <cube/cubedef.h>
```

Extended include file (cosmic cube programs):

```
#include <cube/cosmic/cubedef.h>
```

cc86 *compiler:*

```
cc86 -o prog prog.c -lcube
```

is typical usage, and most of the standard **cc** options also exist.

Object code conversion:

fix86 *[-s stacksize] file*

Host process compiling:

```
cc -o prog prog.c -lcube
```

8.6 Cosmic host environment

Functions:

All functions listed in sections 8.1, 8.2, and 8.4 are available for host processes, and also those listed here.

```
cosmic_init(node, pid);
cosmic_exit();

cosmic_set_alias(alias);
cosmic_set_group(group);
cosmic_set_cubedhost(cubedhost);

    short  node, pid;
    char   *alias, *group, *cubedhost;
```

The following are functions to convert in the host between that particular host's data representations and the cube node data representations. They are defined as null functions in the cube nodes.

```
/*   type          host-to-cube      cube-to-host    */

    short *sp;    htocs(sp,cnt);    ctohs(sp,cnt);

    long *lp;     htocl(lp,cnt);    ctohl(lp,cnt);

    float *fp;    htocf(fp,cnt);    ctohf(fp,cnt);

    double *dp;   htocd(dp,cnt);    ctohd(dp,cnt);
```

8.7 Running programs

Host process parameters:

`node, pid, dim, alias, group, cubedhost, type`

The `node` and `pid` parameters are set in the `cosmic_init` function, or default to the `cube` environment variable, `.cube` file specification, or a default value, respectively.

The `dim` and `type` parameters are set as arguments of the `getcube` utility, or default as above.

The remaining parameters, `alias`, `group`, and `cubedhost`, may be set by the `cosmic_set_` functions before `cosmic_init` is called, or default as above.

Either the `cube` environment variable or `.cube` file is in the form of parameter-value pairs, with unambiguous prefixes permitted.

Legal `type` strings are `"cosmic_cube"`, `"ipsc_cube"`, or `"non-cube"`, and unambiguous prefixes are permitted.

Cosmic environment utilities:

`restart`
kills everything and reloads a fresh copy of `cubed`.

`killcube` *[group name]*
reboots the cube acquired or specified by the *group name*.

`getcube` *[dimension] [cube type]*
acquires a cube and creates the process group.

`freecube` *[group name]*
frees the cube acquired or specified by the *group name*.

`peek` *[group name [user name]]*
generates a status display.

Cosmic cube utilities

`spawnf` *file [node [pid [runstate]]]*
spawns a process from *file*`.cos` with the specified ID and *runstate*. Specifying the *node* parameter as `-1` causes the process to be spawned in all nodes. Legal *runstates* are `r` or `s` (or `n`).

`spawnp` *snode spid [node [pid [runstate]]]*
spawns a copy of a process *(snode,spid)* with the specified ID and *runstate*. Specifying the *node* parameter as `-1` causes the process to be spawned in all nodes.

`ckill` *node pid [runstate]*
changes the run state of the process (*node*, *pid*) to *runstate*. If *runstate* is not specified, it defaults to `d`, which stands for `DEAD`.

`cps` *[options]*
displays the cube process status. Run `cps -u` for a list of options.

`cla`
displays the cube load average and standard deviation.

9. References

[Athas 85] William C Athas, "An Experimental Concurrent Language", Caltech Computer Science Technical Report 5196:TR:85, September 1985.

[Hoare 78] C A R Hoare, "Communicating Sequential Processes", *Communications of the ACM*, 21(8), August 1978.

[InMOS 84] "Occam and the Transputer", InMOS sales document, 1984.

[Kern 78] Brian W Kernighan and Dennis M Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[Kung 80] H T Kung, "The Structure of Parallel Algorithms", in *Advances in Computers*, vol 19, Academic Press, NY, 1980.

[Lang 82] Charles R Lang, "The Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture", Caltech Computer Science Technical Report 5014:TR:82, May 1982.

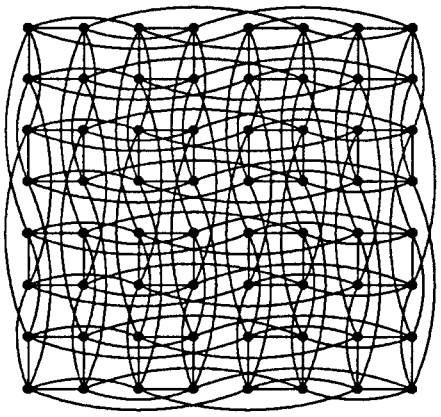
[Mart 84] Alain J Martin, "The Probe: an Addition to Communication Primitives", *Information Processing Letters* **20**, no 1, January 1985. Also as Caltech Computer Science Technical Report 5124:TR:84.

[Seitz 84] Charles L Seitz, "Concurrent VLSI Architectures", *IEEE Transactions on Computers*, vol C-33, no 12, December 1984.

[Seitz 85] Charles L Seitz, "The Cosmic Cube", *Communications of the ACM*, 28(1), January 1985.

[Steele 85] Craig S Steele, "Placement of Communicating Processes on Multiprocessor Networks", Caltech Computer Science Technical Report 5184:TR:85, April 1985.

[5150:DF:84] Wen-King Su, Reese Faucette, Chuck Seitz, "C Programmer's Guide to the Cosmic Cube", Caltech Computer Science internal report (predecessor to this technical report), several versions starting in 1984.



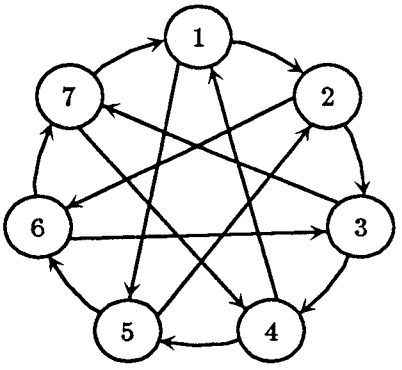
Appendix 1:
N-body Program

A1.1 Concurrent Formulation

This example of computing the time evolution of a system of N bodies that interact by gravitational attraction, or some other symmetrical force, is an paradigm of the concurrent formulation of a "universal interaction" computation. One might expect a distributed computer to do best at "local interaction" problems, but as this program shows, there are also very nice formulations for universal interaction. The program is an elaboration on an example presented in a *CACM* article on the cosmic cube [Seitz 85].

Each of the N bodies interacts with all of the $N-1$ other bodies, but instead of calculating the forces due to all $N(N-1)$ interactions, for the symmetrical force one needs only to calculate half of them. One time step of the computation then requires $N(N-1)/2$ force calculations, and N position and velocity updates (integrations), one for each of the N bodies.

The concurrent formulation is based on N processes. Each of these N identical processes is "host" to one body. The process is responsible for computing the forces due to the interactions with $(N-1)/2$ "guest" bodies, and for updating the position of its own body. The formulation is simplified by assuming N is odd, so that $(N-1)/2$ is an integer. The N-fold concurrency in the formulation reduces the time complexity of the $N(N-1)/2$ force calculations to $O(N)$, and of the N position and velocity updates to $O(1)$.



The *ring process structure* shown above for $N = 7$ is perfect for organizing universal interaction problems. 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 home process ID of body 1 is conveyed through the processes that are host to bodies 2, 3, and 4, with the forces due to these interactions accumulated. The pattern is the same for all the other bodies. After $(N-1)/2$ visits the representations of the bodies are returned over the chordal paths to the process that is host to the body, the forces are combined, and the positions are updated.

Both the cube and host process code for this problem provide opportunities to illustrate many of the more important cosmic cube programming techniques. While in later examples we will spare the reader some of the details, here we will show all of the cube process code, and all of the host process code except for some display driver routines.

A1.2 Cube Process code

In the following process code, each of N identical processes has two instances of a structure body, one for the body for which that process is the "home", `host`, the other for the body that "visits" via messages, `guest`. The structure body contains the state information for a body: its position, velocity, accumulated force, mass, and the ID of its home process.

The entire process structure is spawned and controlled by a process that nominally resides in the hosts. However, the code for the N identical processes that run in the cube will be constructed to be controlled from an arbitrary set of processes.

The message interface to the processes consists of 3 messages:

- A `state` message sent to the process, and which conveys a body structure containing the initial state. This message is distinguished as being type 0, and the process code uses the `state` descriptor to receive this message.
- A `start` message sent to the process, and which conveys a `startup` structure containing the number of bodies, number of time steps, and the ID of the next process in the ring process structure. This message is distinguished as being type 1, and the process code uses the `start` descriptor to receive this message.
- A `result` message sent by the process back to whatever process sent the `start` message. This message conveys a body structure containing the state of the body after the specified number of time steps. This message is of type 0, and the process code uses the `result` descriptor to send this message.

Thus, after all of the N processes have received `state` and `start` messages, the computation can run autonomously for the specified number of time steps before all the processes send their `result` messages back to whatever process started the computation.

The process is then ready to compute again. However, as an additional complication, we allow the control process to send a `start` message that is not preceeded by a `state` message in order to indicate that the final state reported in the `result` message is also to be the initial state for the next set of time steps. This is a good example of a process depending on message order between pairs of processes being maintained, and where a process must behave differently based on the presence or absence of a particular message, in this case the `state` message. The situation is nicely accomodated with a `probe`. By this protocol, the control process might even queue up many `start` messages, but in order to provide a new state in unambiguous sequence with `start` messages, it must preceed a `start` message with a `state` message only after `result` messages have been received in reply to each `start` message.

The inner loops in the computation involve the following communications:

- A sequence of $(N-1)/2$ `body_out` messages to the next process on the ring. The first of these messages convey the `host` body, and the remaining such messages convey the sequence of `guest` body representations that are processed, except for the last.

This message is of type 2, and uses the body_out descriptor.

- A sequence of $(N-1)/2$ body_in messages from the preceeding process on the ring. Each of these messages is used to calculate a force which is accumulated both in the host and guest bodies. These messages are paired with body_out messages, and so are also of type 2, but they use the body_in descriptor.
- The body_bak message corresponds to the last body_in message, after the computation of its interaction. The ID of the process that is home to this body is determined from the guest message. The body_back message is distinguished by type 3, and the same descriptor is used sequentially to send the last guest home and then to receive the corresponding message.

Here, then, is the process code described above. In order not to obscure the communication structure of the process, we present first the core from which we have abstracted the computation of the forces and the updating of the positions. These computations are performed by functions linked from `force.c`, which will be presented later.

```
/* process for an n-body computation, n odd, with symmetrical forces */
#include <cube/cubedef.h> /* cube definitions */

struct body   {
                double pos[3];          /* body position x,y,z          */
                double vel[3];          /* velocity vector x,y,z        */
                double mass;            /* body mass                    */
                double force[3];        /* to accumulate forces         */
                short home_node;        /* ID of body's home process    */
                short home_pid;
                double midpos[3];        /* extrapolated mid time step  */
              } host,guest;

struct startup{
                double tstep;           /* size of integration step     */
                short steps;            /* number of integration steps */
                short n;                /* number of bodies             */
                short next_node;        /* ID of next process on ring   */
                short next_pid;
              } task;

idesc(state,  0, 0, 0, &host, sizeof(host));   /* to receive state */
idesc(start,  0, 0, 1, &task, sizeof(task));   /* to receive task  */
idesc(result, 0, 0, 0, &host, sizeof(host));   /* to send results  */

idesc(body_in , 0, 0, 2, &guest, sizeof(guest)); /* ring send      */
idesc(body_out, 0, 0, 2, &host, sizeof(host));   /* ring recv      */
idesc(body_bak, 0, 0, 3, &guest, sizeof(guest)); /* chordal send & recv */

main()
{
    int i;
    for(;;)                                 /* forever                  */
    {
        led(1);                             /* turn on LED while wating */
```

```
        recvb(&start);                          /* get task through start   */
        if (probe(&state))                      /* check if new init state  */
        {
            recv(&state);                       /* get state, no wait req   */
            host.home_node = mynode();          /* label host with this     */
            host.home_pid = mypid();            /* process being home       */
        }

        RESET(&host,&task);                     /* reset integrator         */
        body_out.node = task.next_node;         /* set body_out channel to  */
        body_out.pid = task.next_pid;           /* next proc in ring        */

        while(task.steps--)                     /* integrate task.steps times    */
        {
            led(task.steps & 0x4);              /* flash while you work          */
            body_out.buf = (char *) &host;      /* first time send out host body */

            for(i = (task.n-1)/2; i--;)         /* (s.n-1)/2 times along the ring*/
            {
                sendb(&body_out);               /* send out the host|guest   */
                recvb(&body_in);                /* receive the next guest    */
                COMPUTE_FORCE(&host,&guest);    /* calculate force           */
                body_out.buf = (char *) &guest; /* prepare to pass the guest */
            }

            body_bak.node = guest.home_node;    /* setup destination ID to   */
            body_bak.pid  = guest.home_pid;     /* guest's home, and         */
            sendb(&body_bak);                   /* send guest back           */
            recvb(&body_bak);                   /* envoy returns (in guest)  */
            ADD_FORCE_TO_HOST(&host,&guest);    /* combine the forces        */
            UPDATE(&host,&task);                /* time step, zero force     */
        }

        result.node = start.node;  /* send new host state back to whatever */
        result.pid = start.pid;    /* process sent the startup message     */
        sendb(&result);
    }
}
```

One might note that if some exceptional condition were to occur in the time evolution of the system, such as a collision or close encounter, the function that computes the forces, COMPUTE_FORCE, could report this event back to the control process by a message of some distinguished type.

Now, with this N-body process core, the following `force.c` routines complete the cube process code.

The computation of the force is for a gravitational attraction with G equal to unity. Mass, distance, or time may be scaled to normalize the computation for a particular value of G.

The UPDATE routine performs a simple Euler integration using the slopes found at an extrapolated half-time-step point, instead of the current point, to project the velocity and location for the next point.

The RESET routine computes the projected half-time-step point for each body before entering the integration loop. The point is re-computed by the UPDATE routine after each integration.

```
/* integrations routines, force calculations */
#include <math.h>
#include <cube/cubedef.h>

struct body   {
                double pos[3];          /* body position x,y,z          */
                double vel[3];          /* velocity vector x,y,z        */
                double mass;            /* body mass                    */
                double force[3];        /* to accumulate forces         */
                short home_node;        /* ID of body's home process    */
                short home_pid;
                double midpos[3];       /* extrapolated mid time step   */
              };

struct startup{
                double tstep;           /* size of integration step     */
                short steps;            /* number of integration steps  */
                short n;                /* number of bodies             */
                short next_node;        /* ID of next process on ring   */
                short next_pid;
              };

COMPUTE_FORCE(host,guest)
    struct body *host,*guest;
{
    double FORCE[3];
    double dx,dy,dz,dist2,inetforce;

    dx = guest->midpos[0] - host->midpos[0];    /* compute offset vector */
    dy = guest->midpos[1] - host->midpos[1];
    dz = guest->midpos[2] - host->midpos[2];

    dist2 = dx*dx + dy*dy + dz*dz;              /* square of distance */
    inetforce = (host->mass * guest->mass) / (dist2 * sqrt(dist2));
    FORCE[0] = inetforce * dx;
    FORCE[1] = inetforce * dy;
    FORCE[2] = inetforce * dz;                  /* force vector */

    host->force[0] += FORCE[0];
    host->force[1] += FORCE[1];
    host->force[2] += FORCE[2];                 /* force to host */
    guest->force[0] -= FORCE[0];
    guest->force[1] -= FORCE[1];
    guest->force[2] -= FORCE[2];                /* force to guest */
}
```

```
ADD_FORCE_TO_HOST(host,guest)
    struct body *host,*guest;
{
    host->force[0] += guest->force[0];
    host->force[1] += guest->force[1];
    host->force[2] += guest->force[2];
}

UPDATE(host,task)
    struct body *host;
    struct startup *task;
{
    double factor;

    factor = task->tstep;
    host->pos[0] += host->vel[0]*factor;        /* integrate velocity */
    host->pos[1] += host->vel[1]*factor;
    host->pos[2] += host->vel[2]*factor;

    factor /= host->mass;
    host->vel[0] += host->force[0]*factor;      /* integrate position */
    host->vel[1] += host->force[1]*factor;
    host->vel[2] += host->force[2]*factor;

    factor = task->tstep/2;                     /* predict next mid-point */
    host->midpos[0] = host->pos[0] + host->vel[0]*factor;
    host->midpos[1] = host->pos[1] + host->vel[1]*factor;
    host->midpos[2] = host->pos[2] + host->vel[2]*factor;

    host->force[0] = host->force[1] = host->force[2] = 0; /* clear force */
}

RESET(host,task)
    struct body *host;
    struct startup *task;
{
    double factor;

    factor = task->tstep/2;                     /* predict mid-point */
    host->midpos[0] = host->pos[0] + host->vel[0]*factor;
    host->midpos[1] = host->pos[1] + host->vel[1]*factor;
    host->midpos[2] = host->pos[2] + host->vel[2]*factor;
}
```

A1.3 Host process code

The sunbody program is run on the host after acquiring a cube of appropriate dimension with the getcube utility. It is run with:

sunbody *body-file steps timestep*

The *steps* argument is an integer specifying the number of integration steps to perform between reporting the new states. The *timestep* argument is a real number which taken as the size of the time step used in the integrator. The *body-file* contains a description of the initial states of the bodies in the following format:

```
1  0    700   -200 0 0    0   4 0
3  0    850   -120 0 0    0   6 0
2  0   1000    -50 0 0    0  14 0
6  0   8000      0 0 0    0   0 0
7  0   1000     50 0 0    0 -14 0
5  0    850    120 0 0    0  -6 0
4  0    700    200 0 0    0  -4 0
```

The first two numbers in each line are the node and pid of the cube process, followed by the mass, position (x y z), and the velocity (vx vy vz) of the body.

For each body entry, a process is spawned by sunbody. These processes are chained into a ring in the same order as their bodies appear in the list. Although processes can be placed anywhere, it is best to minimize the distance between adjacent processes, as was done in the above example.

The details of the process spawning, done directly by the process by messages to the SPAWN processes, are shown in the `install(node,pid)` function.

The time evolution of the system then proceeds indefinitely, with the sunbody process maintaining a continuous display of the positions of the bodies, and a summary of the center of mass and total energy of the system. Because of an extra `start` message, produced by the `send_task` function, the host computation is overlapped with the cube computation.

All the sending and receiving of messages is consolidated into the `send_task`, `send_value`, and `recv_value` functions, which use the "block" variants of the basic `send` and `recv` functions. The changes in number representations, using the `ctoh*` and `htoc*` functions, are also performed here.

Here is the complete code of the sunbody program. The display driver routines are contained in another file not shown here.

```
/* Host process code */
#include <stdio.h>
#include <cube/cosmic/cubedef.h>
#include <math.h>

struct body   {
                double pos[3];          /* body position x,y,z          */
                double vel[3];          /* velocity vector x,y,z        */
                double mass;            /* body mass                    */
              };

struct obj    {
                struct body body;       /* describes a body             */
                struct obj *next;       /* used for making linked list  */
                short node;             /* ID of body's home process    */
                short pid;
              } *objhead = NULL, *newhead = NULL;

struct startup{
                double tstep;           /* size of integration step     */
                short steps;            /* number of integration steps  */
                short n;                /* number of bodies             */
                short next_node;        /* ID of next process on ring   */
                short next_pid;
              };

int    bodycnt;         /* number of bodies there are.                          */
int    steps;           /* number of integrations steps per position report     */
double timestep;        /* time interval used for integration.                  */
double cm_vel[3];       /* stores the velocity of the center of the mass.       */
double energy;          /* stores the energy of the system.                     */

install(node,pid)       /* spawns the process "nbody" in the cube               */
    short node,pid;
{
    FSPMSG sp;
    MSGDESC sd;

    sp.pid = pid;
    sp.state = READY;
    sp.rtype = FSPTYPE;
    bcopy("nbody",sp.name,7);           /* set up message to spawner.       */
    htocs(&sp,3);                       /* change to cube byte order.       */

    sdesc(&sd,node,SPAWN,FSPTYPE,&sp,sizeof(sp));
    sendb(&sd);                         /* send it to the spawner, and      */
    recvb(&sd);                         /* ignore spawner's reply.          */
}
```

```
summarize(obj,cm_vel,energy)    /* energy and net velocity of the system     */
    double *cm_vel;             /* center of mass                            */
    double *energy;
    struct obj *obj;
{
    double mass,dist,dx,dy,dz;
    struct obj *tmp;

    cm_vel[0] = cm_vel[1] = cm_vel[2] = *energy = mass = 0;
    for(; obj; obj = obj->next)
    {

        cm_vel[0] += obj->body.vel[0]*obj->body.mass;   /* cm's momentum.    */
        cm_vel[1] += obj->body.vel[1]*obj->body.mass;
        cm_vel[2] += obj->body.vel[2]*obj->body.mass;
        mass += obj->body.mass;                         /* cm's mass.        */
        for(tmp = obj; tmp->next; tmp = tmp->next)      /* potential energy  */
        {

            dx = tmp->body.pos[0] - tmp->next->body.pos[0];
            dy = tmp->body.pos[1] - tmp->next->body.pos[1];
            dz = tmp->body.pos[2] - tmp->next->body.pos[2];
            dist = sqrt(dx*dx + dy*dy + dz*dz);
            *energy -= tmp->body.mass*tmp->next->body.mass/dist;

        }
        *energy += 0.5 * obj->body.mass *               /* kinetic energy    */
                   ( obj->body.vel[0]*obj->body.vel[0] +
                     obj->body.vel[1]*obj->body.vel[1] +
                     obj->body.vel[2]*obj->body.vel[2]);

    }
    cm_vel[0] /= mass;
    cm_vel[1] /= mass;
    cm_vel[2] /= mass;                  /* convert momentum to velocity      */

                                        /* print it out on the screen        */
    fprintf(stderr,"cm moving at (%lg %lg %lg)", cm_vel[0],cm_vel[1],cm_vel[2]);
    fprintf(stderr," energy %lg\n",*energy);
    fflush(stderr);
}

zero_cm_vel(obj,cm_vel)                 /* makes initial velocity of cm zero */
    struct obj *obj;
    double *cm_vel;
{
    int i;

    if(cm_vel[0] || cm_vel[1] || cm_vel[2])
```

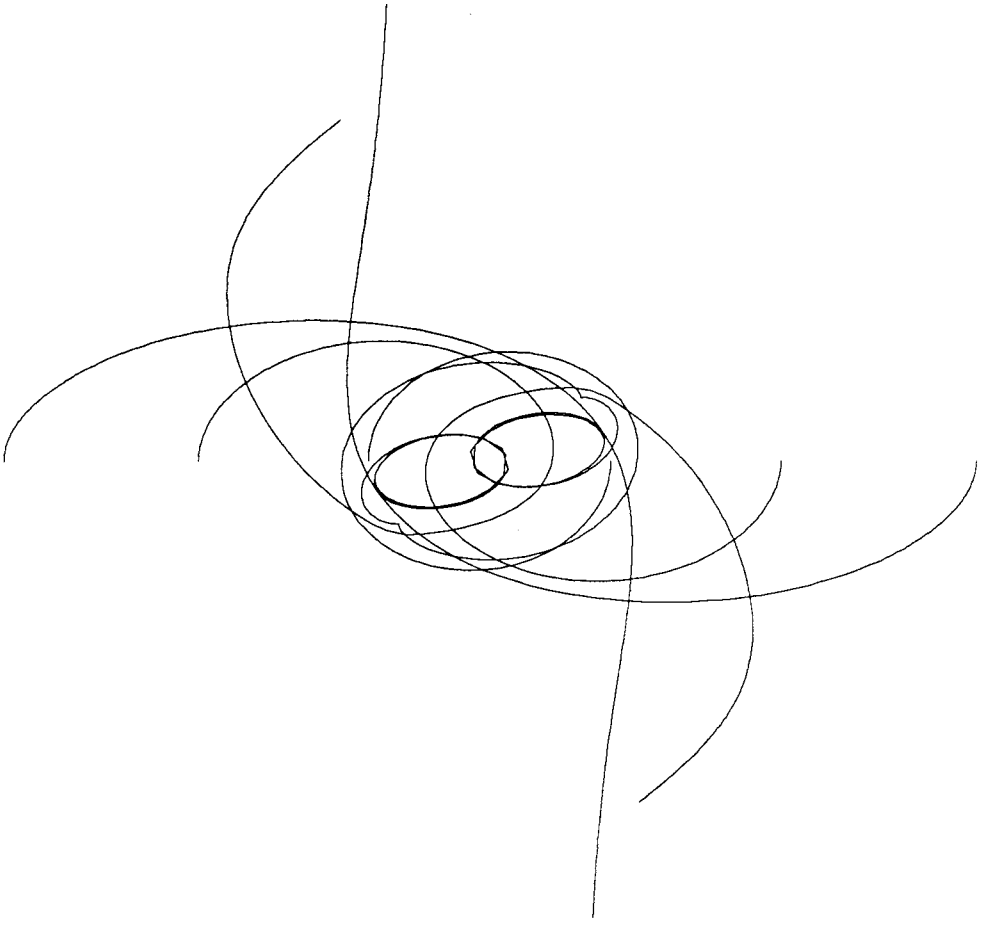
```
    {                                       /* if cm's velocity is not zero. */
        for(; obj; obj = obj->next)         /* subtract it from all bodies.  */
            for(i = 3; i--;) obj->body.vel[i] -= cm_vel[i];
        fprintf(stderr,"center of mass's movement zeroed\n");
        summarize(objhead,cm_vel,&energy);  /* re-calculate vel and energy.  */
    }
}

read_file(name)         /* reads description of bodies and where to put them */
    char *name;
{
    FILE *fp;
    char line[100];
    int node,pid;
    struct obj *obj;

    if(!(fp = fopen(name,"r")))                 /* open the file             */
    {
        fprintf(stderr,"Bad file %s\n",name);
        cosmic_exit();
    }

    for(bodycnt = 0; fgets(line,100,fp); bodycnt++) /* one line per body     */
    {
        obj = (struct obj *) malloc(sizeof(struct obj));
        /* node pid mass x y z vx vy vz */
        sscanf(line,"%d %d %lf %lf %lf %lf %lf %lf %lf",
                &node,&pid,&obj->body.mass,
                &obj->body.pos[0], &obj->body.pos[1], &obj->body.pos[2],
                &obj->body.vel[0], &obj->body.vel[1], &obj->body.vel[2]);

        obj->node = node;
        obj->pid = pid;
        obj->next = objhead; objhead = obj;     /* first list for old data   */

        obj = (struct obj *) malloc(sizeof(struct obj));
        *obj = *objhead;
        obj->next = newhead; newhead = obj;     /* second list for new data  */
        install(node,pid);                      /* spawn process             */
    }

    if(!(bodycnt & 1))
    {
        fprintf(stderr,"I need an odd number of bodies\n");
        cosmic_exit();
    }
}

send_value(obj)                 /* sends initial values to cube processes    */
    struct obj *obj;
```

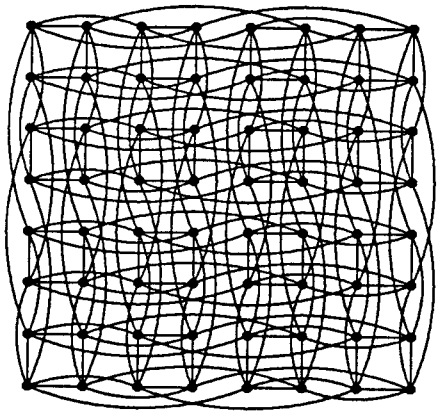
```
{
    MSGDESC sd;
    struct body bd;

    for(; obj; obj = obj->next)
    {
        bd = obj->body;
        htocd(&bd,7);           /* change to cube number representation      */
        sdesc(&sd,obj->node,obj->pid,0,&bd,sizeof(bd));
        sendb(&sd);
    }
}

send_task(obj)                  /* sends task messages to the cube processes */
    struct obj *obj;
{
    struct obj *first;
    MSGDESC sd;
    struct startup task;

    task.tstep = timestep;      /* set parameters common to all processes    */
    task.steps = steps;
    task.n = bodycnt;
    htocd(&task,1);             /* change to cube number representation      */
    htocs(&task.steps,2);
    first = obj;

    for(; obj; obj = obj->next)
    {
        if(obj->next)           /* set ID of next process in the ring        */
        {                       /* to be that of the next on the linked list.*/
            task.next_node = obj->next->node;
            task.next_pid  = obj->next->pid;
        } else

        {
            task.next_node = first->node;  /* if this is the last body       */
            task.next_pid  = first->pid;   /* wrap around the the first.     */
        }
        htocs(&task.next_node,2);  /* change to cube number representation   */
        sdesc(&sd,obj->node,obj->pid,1,&task,sizeof(task));
        sendb(&sd);
    }
}

recv_value(obj)                 /* receives new values from cube processes   */
    struct obj *obj;
{
    int i;
    struct obj *o;
    struct body bd;
    MSGDESC rd;
```

```
    for(i = bodycnt; i--; )     /* wait for bodycnt messages to arrive      */
    {
        sdesc(&rd,0,0,0,&bd,sizeof(bd));
        recvb(&rd);             /* receive the message                      */
        ctohd(&bd,7);           /* change to host number representation     */

        for(o = obj; o->node != rd.node || o->pid != rd.pid; o = o->next);
        if(o) o->body = bd;                 /* store it in the proper place */
    }
    summarize(obj,cm_vel,&energy);
}

main(argc,argv)
    int argc;
    char *argv[];
{
    struct obj *tmp;

    if(argc < 4){puts("<data file> <steps/display> <time step>"); exit(1);}
    steps = atoi(argv[2]);
    timestep = atof(argv[3]);           /* process command line parameters  */

    cosmic_init(HOST,0);                /* enters cosmic environment        */
    read_file(argv[1]);                 /* read bodies from file            */
    summarize(objhead,cm_vel,&energy);  /* initial energy and cm's velocity */
    zero_cm_vel(objhead,cm_vel);        /* adjust cm's velocity to zero.    */
    send_value(objhead);                /* send out initial values.         */

    open_graphic();                     /* initialize display device        */
    for(send_task(objhead);;)           /* first task msg, then forever     */
    {
        send_task(objhead);             /* queue another task msg           */
        recv_value(newhead);            /* recv new values into the new list */

        display(objhead,newhead);       /* display the results.             */
        tmp = objhead;
        objhead = newhead;              /* switch the two lists.            */
        newhead = tmp;
    }

    cosmic_exit();  /* leave cosmic environment.  It never gets executed.   */
}
```

A1.3 Sample Output

A sample output for the initial conditions illustrated above is:





Appendix 2: $3x + 1$ Sieve

In contrast to the universal interaction of n-body program presented in Appendix 1, this second example is a problem formulated with very sparse interaction. The initial phases of the computation are used to compute a reasonably even division of effort, and to distribute the resulting tasks to processes. Then during the largest part of the computation, there is no communication at all between the processes as each works on its own independent part of the problem. Finally, the communication capabilities of the cosmic cube are used again to collect the results. The pattern of distributing the tasks, a process structure that we call the "time-on-target" tree, is useful in many such situations. For example, this same algorithm is used by the broadcast spawn process.

A2.1 The $3x + 1$ Problem

The "$3x + 1$ problem" is concerned with the iterates of the function that takes even integers x to $x/2$ and odd integers to $3x + 1$. For example, if one starts with the number 7 a sequence:

```
7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1 [4 2 1]*
```

is generated. It is *conjectured* that for any initial integer the sequence will always reach 1. The search for a counterexample to this conjecture has been done by an exhaustive test of the integers up to 2^{40} by Nabuo Yoneda at the University of Tokyo, but without success. No one has been able to prove the conjecture either. The problem *appears* to be intractably hard; thus its fascination. Paul Erdős commented that "Mathematics is not yet ready for such problems." It is a popular joke that the $3x + 1$ problem is a conspiracy to slow down mathematical research. The author of the program presented here (Seitz) first encountered this problem in 1967, while he was a graduate student at MIT, and he has burned countless computer cycles through the years in various computer's operating system idle loops while exploring the problem.

An excellent survey of "The $3x + 1$ problem and its Generalizations"* traces the problem to work of Lothar Collatz starting in 1932, and describes what is known about it. The mathematics of the problem is very interesting, but is beyond the scope of what is required to explain this simple programming example.

The cosmic cube program presented here uses an approach of trying to prove the conjecture by sieving (eliminating) expressions rather than searching the integers. The expressions considered fall into classes in the form $2^k n + T_k$, where T_k is a set of terms, each

* Jeffrey C Lagarias, *American Mathematical Monthly*, January 1985.

element of which is less than 2^k. Starting with $k = 0$, one has only the primitive expression $2^0n + \{0\} = n$. Since n might be either even or odd, one cannot iterate this expression. So, consider next the even and odd integers separately. No even integer – no integer represented by the expression $2n$ – could be the least counterexample since $2n \Rightarrow n$, which is equal to or less than than $2n$ for all $n \geq 0$, and any successor in the sequence produced by a counterexample is also a counterexample. Thus one can eliminate from consideration all expressions in the form $2n$. However, the odd integers, represented by the expression $2n + 1$, follow the iteration: $2n + 1 \Rightarrow 6n + 4 \Rightarrow 3n + 2$. One cannot discriminate whether $3n + 2$ is even or odd, and no element of the sequence is smaller than the starting element. So, the set of expressions that cannot be sieved out at this level are $2^1n + \{1\}$.

The next set of expressions to consider is in general $\{2^k(2n) + T_k\} \cup \{2^k(2n + 1) + T_k\}$, which are then sieved, for example:

$2^1(2n) + 1 = 4n + 1 \Rightarrow 12n + 4 \Rightarrow 6n + 2 \Rightarrow 3n + 1$ (eliminated).

$2^1(2n + 1) + 1 = 4n + 3 \Rightarrow 12n + 10 \Rightarrow 6n + 5 \Rightarrow 18n + 16 \Rightarrow 9n + 8$ (retained).

to form the set $\{2^{k+1}n + T_{k+1}\}$, which in this case is $2^2n + \{3\}$.

If in proceeding in this way one could eventually eliminate all expressions – that is, if one could construct the entire binary tree down to expressions that are eliminated –, the conjecture would be proved by demonstrating the non-existence of a least counterexample. There is no serious expectation of this outcome; the "proof" strategy of the program is intended only to be an efficient way to search for counterexamples. If the problem is in fact intractable, the experimenter is guaranteed run out of patience or the machine will run out of bits before the conjecture is either proved or a counterexample is found.

The following program output shows the breadth of the tree of expressions constructed in this way. Shown in {} for the first 8 levels are the sets of terms (edited from the output of another program) that cannot be eliminated.

```
Cube version of Collatz sieve, running on 6-cube

2^0*n + set of 1 terms    { 0 }
2^1*n + set of 1 terms    { 1 }
2^2*n + set of 1 terms    { 3 }
2^3*n + set of 2 terms    { 3 7 }
2^4*n + set of 3 terms    { 11 7 15 }
2^5*n + set of 4 terms    { 27 7 15 31 }
2^6*n + set of 8 terms    { 27 59 7 39 15 47 31 63 }
2^7*n + set of 13 terms   { 27 91 123 71 39 103 79 47 111 31 95 63 127 }
2^8*n + set of 19 terms
2^9*n + set of 38 terms
2^10*n + set of 64 terms
2^11*n + set of 128 terms
2^12*n + set of 226 terms
2^13*n + set of 367 terms
2^14*n + set of 734 terms
2^15*n + set of 1295 terms
2^16*n + set of 2114 terms
2^17*n + set of 4228 terms
2^18*n + set of 7495 terms
2^19*n + set of 14990 terms
2^20*n + set of 27328 terms
```

```
2^21*n + set of 46611 terms
2^22*n + set of 93222 terms
2^23*n + set of 168807 terms
2^24*n + set of 286581 terms
2^25*n + set of 573162 terms
2^26*n + set of 1037374 terms
2^27*n + set of 1762293 terms
2^28*n + set of 3524586 terms
2^29*n + set of 6385637 terms
2^30*n + set of 12771274 terms
2^31*n + set of 23642078 terms
```

Total running time to produce the tabulation above was about 25 hours on the 6-cube, with arithmetic done in arbitrary precision. Because this sieve on expressions eliminates at the last levels all but about 1% of the integers from consideration as possible counterexamples, it greatly reduces the space in which to search. However, even by this technique, a duplication of the test for counterexamples up to 2^{40} would require, based on some partial tests, about 500 hours on the 6-cube. This experiment was not tried in full, since the sieve program revealed some additional properties of this curious problem that suggest a number of other shortcuts.

A2.2 Concurrent Formulation

The approach used to distributing the work of this $3x + 1$ sieve can be applied to many problems involving only sparse interaction. Let us describe this technique first in terms of the $3x + 1$ sieve.

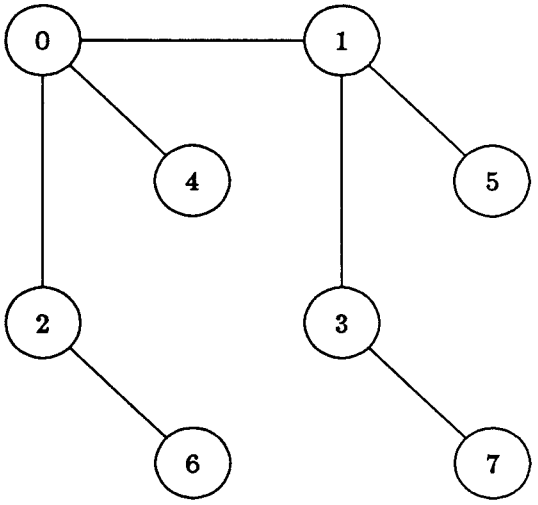
The process structure for the $3x + 1$ sieve program is built with an identical process in every node, each of which starts by waiting for a message. A message sent to one of these processes conveys a `MESSAGE` structure:

```
typedef struct {
    unsigned short size;       /* size of set of terms */
    short          rp_node;    /* return path node     */
    short          rp_pid;     /* return path pid      */
    unsigned short maxsplit;   /* 2^(number of splits allowed (dim of cube)) */
    unsigned short split;      /* 2^(dimension on which to split next) */
    unsigned short logmaxcoef; /* log base 2 of max coef */
    unsigned short logcoef;    /* log base 2 of current coef */
    unsigned short term[ARRAYSIZE][DIGITS+1];  /* set of terms */
} MESSAGE;
```

The `MESSAGE` structure conveys the `logcoef` ($= k$), the `term` array, and its `size`, which together represent a set of expressions in the form $2^k n + T'_k$. We use the notation T'_k to indicate that this set is not yet sieved. The task is potentially splittable as long as the whole cube is not already used and k has not reached its specified limit value, `logmaxcoef`. As long as the task is splittable, the process performs a "breadth-first" search in which sieves the input expressions and then forms the set of expressions $2^{k+1} n + T'_{k+1}$. This set T'_{k+1} may be as much as twice as large as T'_k.

If the resulting list is larger than a `SPLITSIZE` parameter, the task is split. The process keeps half of the task and "gives away" the other half. The "time-on-target" tree is an orderly way to build this tree on a binary n-cube. Although it could be any process, let us assume that the process in node 0 receives the first message. It may then split first

by sending a task message to its neighbor along dimension 0 (the process in node 1). It includes in this message and in its own state, using the variable `split`, that the dimension on which to split next is 1. Then the processes in nodes 0 and 1 continue until they decide, independently, to split along dimension 1, and so on until the splitting is inhibited by having reached the dimension of the cube or the specified depth in the search. The following graph shows the process structure that would be built in a 3-cube:



Although this structure is embedded into a binary n-cube while using only neighbor connections, that is not its most important property. We call this process structure the *time-on-target* tree to indicate how the "keep half, give away half" strategy results at each generation, including the last, in all the processes reaching the "target" of starting out at more or less the same time. Someone suggested calling this structure "the family tree of the immortal amoeba", but the authors were concerned about keeping this guide clean.

This same stategy also works well for the broadcast spawn process, as discussed in section 6.4, and is the mechanism used by the `gfork` function. These programs are a somewhat simpler application of this technique in that they do not do any intermediate computation, such as the $3x+1$ sieve process performs in order to increase the size of the set being sieved.

Implicit in the formulation of the $3x + 1$ program is the assumption that by making the set of terms large enough, one is dividing the work approximately equally. This assumption is experimentally justified, even using the split by which the $2^k(2n) + T_k$ terms are kept and the $2^k(2n + 1) + T_k$ terms are given away. The choice of the SPLITSIZE parameter allows one to make a tradeoff between how quickly the problem gets dispersed and how well balanced the parts are.

A2.3 Cube process code

The first challenge of writing programs for exploring the Collatz conjecture is that the problem starts getting interesting and outside the range of conventional machines at about the same time that one runs out of numerical precision in C `long` integers. The Collatz iterations starting with integers in the range of 2^{16} can reach values in the range of 2^{32}. In order that one can have absolute confidence in the results of these very long-running computations, a package of arbitrary precision integer arithmetic routines is used, and included from `bignum.c`. Also, the file `coldef.c` contains definitions common to the several parts of the program, in particular, it includes the definition of the MESSAGE type, the

parameter that controls when the task is to be split, SPLITSIZE, and the parameters of the arbitrary precision arithmetic routines. Any change to this file requires remaking both host and cube process code.

The bignum functions manipulate numbers represented as arrays of DIGITS+1 shorts, where x[0] is the low-order digit and x[DIGITS-1] is the high-order digit, in radix RADIX. RADIX must be a power of 2, and is equal to 2^{14} except when performing consistency tests on the code. x[DIGITS] indicates the array index in which the first non-zero entry is found, or is 0 for the number 0. Some arithmetic operations are thus done from high-order to low-order digit. For example, to get something of the flavor of these routines, here is the "greater than" function:

```
gtr(x,y)  /* returns true if x is strictly greater than y */
unsigned short x[DIGITS+1], y[DIGITS+1];
{
    int i;
    if (x[DIGITS] > y[DIGITS]) return(1);
    for (i = y[DIGITS]; i >= 0; i--) {
        if (x[i] > y[i]) return(1);
        if (x[i] < y[i]) break;
    }
    return(0);
}
```

the other functions used in the Collatz program can be summarized as follows:

halve(x)	divides x by 2.
threehalves(x)	performs $x \leftarrow x + \frac{x}{2}$ for even x and $x \leftarrow x + \frac{x+1}{2}$ for odd x.
exp(x, y)	where x is a short performs $y \leftarrow 2^x$
add(x,y)	performs $y \leftarrow x + y$.

The function bcopy, the same function as that in the default C library in Unix systems, is also used.

The cube process code is then otherwise explained by the discussion of the formulation and by the comments:

```
#include <cube/cubedef.h>
#include "coldef.c"
#include "bignum.c"

/* version 6 June 1985; first version 20 December 1984 */

/* The wander function returns false if an expression in the form
((2^logcoef)*n + term) progresses in the Collatz sequence to an
expression that is smaller for any n > 0, and returns true if the
expression progresses to an expression with an odd coef, so that
even/odd cannot be distinguished. */

wander(logcoef, term)
unsigned short logcoef, term[DIGITS+1];
{
    unsigned short coef[DIGITS+1], tcoef[DIGITS+1], tterm[DIGITS+1];
    short i;
```

```
    exp(logcoef,coef);
    bcopy(coef,tcoef,sizeof(tcoef));
    bcopy(term,tterm,sizeof(tterm));

    for (;;) {
        if ( (gtr(coef,tcoef)) && (!gtr(tterm,term)) )  /* reduced to      */
            return(0);                                  /* smaller expr    */

        if (odd(tcoef[0]))              /* can't distinguish even and odd */
            return(1);                  /* return true to include in next */

        if (odd(tterm[0])) {            /* coef even, term odd            */
            threehalves(tcoef);         /* tcoef += tcoef/2      (even)   */
            threehalves(tterm);         /* tterm += (tterm+1)/2  (odd)    */
        }

        else {                          /* tcoef and tterm both even      */
            halve(tcoef);               /* tcoef = tcoef/2                */
            halve(tterm);               /* tterm = tterm/2                */
        }
    }
}

/* countleaf is a depth-first (recursive) leaf counting function.  When
called with logcoef and term representing ((2^logcoef)*n+term), it counts
into unsigned long count the number of derivative expressions in a tree
with leaves in the form (2^logmaxcoef*n+term). */

int toggle = 0;
unsigned long count;

countleaf(logcoef, logmaxcoef, term)
unsigned short logcoef, logmaxcoef, term[DIGITS+1];
{
    if (wander(logcoef, term)) {
        if (logcoef == logmaxcoef) {
            count++;
            led(--toggle&1);  /* blink light to show I'm working */
        }
        else {
            unsigned short tterm[DIGITS+1];
            exp(logcoef, tterm);
            add(term, tterm);
            countleaf(logcoef+1, logmaxcoef, term);
            countleaf(logcoef+1, logmaxcoef, tterm);
        }
    }
}
```

```
/* make 2 instances of structure MESSAGE for receiving and then
 * sending the sieve task */
MESSAGE t1, t2;

/* for returning the count of the number of leaves below this vertex */
unsigned long replyword;

/* declare and intialize message descriptors */
idesc(taskin, 0, 0, 0, &t1, sizeof(t1));
idesc(taskout, 0, 0, 0, &t2, sizeof(t2));
idesc(reply, 0, 0, 1, &replyword, sizeof(replyword));

main()
{
unsigned int i1, i2, descendants;
unsigned short coef[DIGITS+1];

while(1) {

/* first receive the initial conditions into t1 */

    led(0);           /* led off while this process waits */
    recvb(&taskin);   /* receive task to be performed     */
    led(1);           /* led on to show started           */
    descendants = 0;  /* number of descendants of this process */

/* while splitable, proceed breadth-first */

    while ((t1.split < t1.maxsplit) && (t1.logcoef < t1.logmaxcoef)) {
        for (i1 = i2 = 0; i1 < t1.size; i1++)  {      /* sieve */
            if (wander(t1.logcoef, t1.term[i1]))
                bcopy(t1.term[i1], t2.term[i2++], sizeof(t2.term[0]));
        }
        exp(t1.logcoef, coef);   /* make 2^logcoef -> coef */
        t2.size = i2;

/* if the sieved list of terms in t2 is smaller than SPLITSIZE,
generate the next list of trial terms into t1 */

        if (t2.size < SPLITSIZE) {
            for (i1 = i2 = 0; i2 < t2.size; i2++) {
                bcopy(t2.term[i2], t1.term[i1++], sizeof(t1.term[0]));
                add(coef, t2.term[i2]);
                bcopy(t2.term[i2], t1.term[i1++], sizeof(t1.term[0]));
            }
            t1.size = i1;
            t1.logcoef = t1.logcoef +1;
        }
```

```
/* if the sieved list of terms exceeds the SPLITSIZE, leave the list
in place in t2, and produce the 2nd set of terms into t1 to continue
*/

        else {
            for (i2 = 0; i2 < t2.size; i2++)  {
                bcopy(t2.term[i2], t1.term[i2], sizeof(t1.term[0]));
                add(coef, t1.term[i2]);
            }
            t1.size = t2.size;
            t1.logcoef = t2.logcoef = t1.logcoef + 1;

/* now set up rest of t2 and MSGDESC task to send this part of the job
off to another process on the time-on-target tree */

            t2.rp_node = mynode();
            t2.rp_pid = mypid();
            t2.logmaxcoef = t1.logmaxcoef;
            t2.maxsplit = t1.maxsplit;
            taskout.node = t2.rp_node ^ t1.split;
            taskout.pid = t2.rp_pid;
            t1.split = t2.split = t1.split << 1;
            sendb(&taskout);
            descendants++;  /* increment count of descendants */
        }
    }

/* having exited the while clause, the sieve is either at a state
where coef = maxcoef, or is no longer splittable (is a leaf process),
or both.  In either case we can proceed depth-first starting with
t1.coef and each member of the t1.term list, and count the number of
cases down to maxcoef. */

    count = 0;
    for (i1 = 0; i1 < t1.size; i1++)
        countleaf(t1.logcoef, t1.logmaxcoef, t1.term[i1]);

/* now for getting the answer back */

    led(1);
    while (descendants--) {
        recvb(&reply);
        count += replyword;
    }
    replyword = count;
    reply.node = t1.rp_node;
    reply.pid = t1.rp_pid;
    sendb(&reply);
```

```
} /* end while(1) */
} /* end main()   */
```

A2.4 Host process code

This version of the host process code for the $3x + 1$ sieve, `hcol` is very simple. The program starts with some preliminaries to format the argument variables, the `logmaxcoef`, and `logcoef` and `term` as optional parameters that default to 0. Thus one can ask the sieve to search for descendents of the tree vertex corresponding to $2^7 n + 27$ to level 25 by `hcol 25 7 27`. The process then sets its `alias` to `Collatz`, enters the environment with ID `(HOST,0)`, prints the specification of the problem, sends a message to the process in node 0, and finally prints the resulting count of the number of leaves of this tree.

```
#include <stdio.h>
#include <cube/cosmic/cubedef.h>
#include "coldef.c"

MESSAGE t;
MSGDESC sd,rd;
unsigned long reply;

main(argc,argv,envp)
    int argc;
    char **argv,**envp;
{
    unsigned short maxsplit, logmaxcoef, logcoef, term;
    int i, j;

    if ( argc < 2 || argc > 4) {
        fprintf(stderr,"usage: logmaxcoef [logcoef] [term]\n");
        exit();
    }

    argc--;
    argv++;

    if(argc) { logmaxcoef = atoi(*argv++); argc--; } else logmaxcoef = 0;
    if(argc) { logcoef    = atoi(*argv++); argc--; } else logcoef = 0;
    if(argc) { term       = atoi(*argv++); argc--; } else term = 0;

    cosmic_set_alias("Collatz");
    cosmic_init(HOST,0);

    printf("Cube version of Collatz sieve, running on %d-cube\n",cubedim());
    printf("logmaxcoef= %hd, logcoef= %hd, term= %hd\n",
            logmaxcoef,      logcoef,      term);

    t.size = 1;
    t.rp_node = mynode();
    t.rp_pid = mypid();
```

```
    t.maxsplit = (1 << cubedim());
    t.split = 1;
    t.logmaxcoef = logmaxcoef;
    t.logcoef    = logcoef;
    t.term[0][0] = term;
    for (i=1; i<=DIGITS; i++) t.term[0][i] = 0;

    sd.node = 0;
    sd.pid = sd.type = 0;
    sd.buf = (char *) &t;
    sd.buflen = sd.msglen = sizeof(t);
    htocs(&t, sizeof(t)/2);
    send(&sd);

    rd.type = 1;
    rd.buflen = sizeof(long);
    rd.buf = (char *) &reply;
    recvb(&rd);
    ctohl(&reply,1);

    printf("2^%hd*n + set of %d terms\n", logmaxcoef, reply);

    exit();
}
```

A2.5 Checkpointing

The $3x+1$ sieve can run for such long periods that it is valuable to be able to save intermediate results in case the cube or host systems might crash. Thus we use this opportunity to illustrate a simple example of breaking up a computation so that results are saved at points where the amount of information is reasonably small. For this sieve a search to a large depth, such as 32, can be broken up by searching independently from each of many vertices below the root. For example, the tree at level 7 has 13 terms, and one can use these as independent starting points. One then need only write a script, such as:

```
date
hcol 32 7 31
date
hcol 32 7 95
date
hcol 32 7 63
date
freecube
```

that will run the host program a number of times. Here is an illustration of starting and running the program with the script, which is called run:

```
(icarus:27) getcube
6D sub-cube allocated
(icarus:28) spawnf col -1 0
col spawned in all nodes, pid 0
(icarus:29) run > col32.output &
```

```
[1] 943
(icarus:30) peek

 CUBE DAEMON version 6, up 22 hours 53 minutes on host sol

    {Collatz chuck} 6d cosmic cube, b:0000 [ icarus 6-cube]  1.0m
    {             } 2d cosmic cube, b:0000 [  ceres 3-cube]  4.6h
    {  group kiat } 2d cosmic cube, b:0004 [cit-vax 3-cube]  4.5h

 GROUP {Collatz chuck}:

    ( -1   0)   Collatz   1s   0r   0q   [icarus 7945]  6.0s
    ( -1  -1)    SERVER   6s   4r   0q   [icarus 7940] 59.0s
    (--- ---)   CUBEIFC   6s   9r   0q   [   sol 2291]  1.0m

(icarus:31) cla
Loading for a 6-cube:
            AVG      MAX         MIN        SDEV
load avg:   0.12   1.24( 0)   0.04( 8)   0.23
free mem:  94528  94528( 0)  94528( 0)    NaN
(icarus:32) cla
Loading for a 6-cube:
            AVG      MAX         MIN        SDEV
load avg:   1.00   1.00( 0)   1.00( 0)   0.00
free mem:  94528  94528( 0)  94528( 0)    NaN
```

The session include checking the cube load average right after the distribution phase of the computation, and about a minute later after the work has spread to all of the nodes.

Finally, about 10 hours later, the results show up in the file `col32.output`:

```
Tue Jun 4 20:49:41 PDT 1985
Cube version of Collatz sieve, running on 6-cube
logmaxcoef= 32, logcoef= 7, term= 31
2^32*n + set of 4584700 terms
Wed Jun 5 01:20:29 PDT 1985
Cube version of Collatz sieve, running on 6-cube
logmaxcoef= 32, logcoef= 7, term= 95
2^32*n + set of 1316172 terms
Wed Jun 5 02:35:21 PDT 1985
Cube version of Collatz sieve, running on 6-cube
logmaxcoef= 32, logcoef= 7, term= 63
2^32*n + set of 4584700 terms
Wed Jun 5 07:03:08 PDT 1985
```

Some of the previously unsuspected structure of this problem is revealed in that the number of cases that pass through the sieve at the 32nd level are the same starting with $2^7n + 31$ and $2^7n + 63$.

CALIFORNIA INSTITUTE OF TECHNOLOGY

PASADENA, CALIFORNIA 91125

COMPUTER SCIENCE 256-80

Cosmic Environment Software Distribution

The *cosmic environment* software described in the "C Programmer's Guide to the Cosmic Cube" is available for internal use to most university, government, and non-profit research organizations under the arrangements outlined below. Software distributions are available to other organizations only through a license with Caltech. It is expected that another version of this software will be available through Intel Scientific Computers.

This software also comes with a network-based update mechanism that can be used to distribute it conveniently between hosts on TCP/IP networks. This mechanism is the recommended way of obtaining a distribution for ARPAnet hosts.

In order to get a copy, send a letter to the undersigned that includes statements:

1. of your intended use of this software (brief).
2. that you agree not to redistribute this software outside of your organization, and in no case outside of the United States, and will keep the files protected on your systems.
3. that you agree that this software is for your internal use, and that you will not sell access to or otherwise permit use of these programs to people outside your own organization.
4. that you understand that this is experimental, unsupported software that may or may not work as described, and that you hold Caltech harmless from any loss or damage that may result from its installation or use.

We are able answer questions about this software as our time allows, and are happy to have bug reports, but only by electronic mail to "`cube@cit-sol.arpa`". It is most helpful if a bug report includes the smallest example of code that you can devise that demonstrates the bug.

If you are installing this software on a machine that is not an ARPAnet host, please include with your letter a blank 600' 1/2 inch tape labeled with your postage address. We will mail the C sources to you in Unix `tar` format. The files can be placed in any source directory. The installation instructions are included in the `make` file.

If you are installing this software on an ARPAnet host, you do not need to send a tape. We hope you don't. Just send us your ARPAnet mail address and the name of the host on which you intend to maintain your source directory. We will then enter the name of the host on which you maintain the sources into a permission table,

and will send you a message that includes a C program that you can compile and run to bring the sources over the ARPAnet to your machine. This program makes a socket connection to the cube dæmon that runs on `cit-sol.arpa`, and brings the files across the network. This same program can be used later for automatic updates, which are selective by timestamp in order to minimize the volume of messages sent through the network.

Requests to use the cosmic cubes or Intel iPSC d7 operated by the Caltech Computer Science project group should be addressed to "`chuck@cit-vlsi.arpa`".

Charles L Seitz
Computer Science 256-80
California Institute of Technology
Pasadena, CA 91125

Deadlock Free Message Routing in Multiprocessor Interconnection Networks

William J. Dally
Charles L. Seitz

5183:DF:85

draft of paper for IEEE Transactions on Computers

May 10, 1985

Abstract

A deadlock-free routing algorithm can be generated for arbitrary interconnection networks using the concept of virtual channels. A necessary and sufficient condition for deadlock-free routing is the absence of cycles in the channel dependency graph. Given an arbitrary network and a routing function, the cycles of the channel dependency graph can be removed by splitting physical channels into groups of virtual channels. This method is used to develop deadlock-free routing algorithms for k-ary n-cubes, for cube connected cycles, and for shuffle-exchange networks.

Index Terms - Interconnection networks, communication networks, concurrent computation, parallel processing, message passing multiprocessors, graph model.

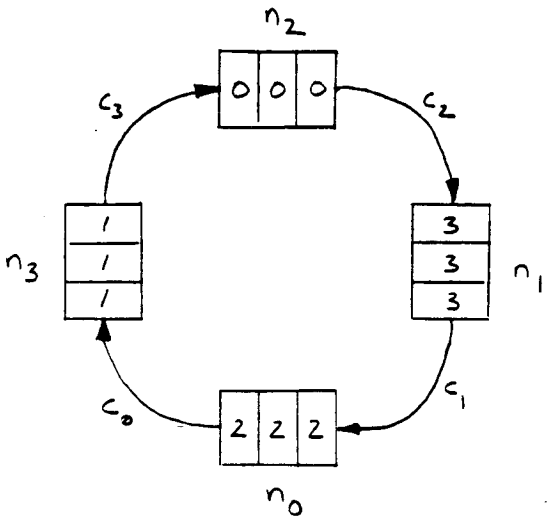


Figure 1: Deadlock in a 4-Cycle

1 Introduction

Message passing concurrent computers such as the Caltech Cosmic Cube [1] consist of many processing nodes that interact by sending messages over communication channels between the nodes. Deadlock in the interconnection network of a concurrent computer occurs when no message can advance toward its destination because the queues of the message system are full [2]. Consider the example shown in Figure 1. The queues of each node in the 4-cycle are filled with messages destined for the opposite node. No message can advance toward its destination; thus the cycle is deadlocked. In this locked state, no communication can occur over the deadlocked channels until exceptional action is taken to break the deadlock.

Reliable concurrent computation requires a routing algorithm that is provably free of deadlock. In this paper we present a general method for constructing deadlock-free routing algorithms for arbitrary networks. In Section 2 we state our assumptions and develop the necessary and sufficient conditions on a *channel dependency graph* for a routing to be deadlock-free. These conditions are used in Section 3 to develop a method of constructing deadlock-free algorithms for arbitrary networks. This method is based on the concept of virtual channels: groups of channels that share a physical channel but each have their own queue. Using virtual channels, we develop deadlock-free routing algorithms for arbitrary k-ary n-cubes in Section 4, for cube-connected cycles in Section 5, and for shuffle-exchange networks in Section 6. In each of these examples, physical channels belonging to cycles are split into a group of virtual channels. The virtual channels are ordered; routing is restricted to visit channels in decreasing order to eliminate cycles in the channel dependency graph.

2 Deadlock Free Routing

We assume the following:

- A message arriving at its destination node is eventually consumed.
- A node can generate messages destined for any other node.
- Between any two processing nodes i and j the order of messages transmitted directly from i to j must be preserved.
- There is room to queue at least two message fragments (possibly a single bit each) for each channel.
- Nodes can produce messages at any rate subject to the constraint of available queue space (source queued).

The following definitions develop a notation for describing networks, routing functions and configurations. A summary of notation is given below.

Definition 1 An *interconnection network*, I is a strongly connected *directed* graph, $I = G(N,C)$. The vertices of the graph, N, represent the set of processing nodes. The edges of the graph, C, represent the set of communication channels. The source node of channel c_i is denoted s_i and the destination node d_i. Associated with each channel, c_i, is a queue with capacity $\text{cap}(c_i)$ containing $\text{size}(c_i)$ messages. If the queue for channel c_i contains a message destined for node n_d then $\text{member}(n_d, c_i)$ is true. There may be redundant channels between two nodes. For purposes of analysis, we add an internal channel c_{n_i} to C for each node n_i. Messages originating at node n_i are considered to have arrived on c_{n_i}.

Definition 2 A *routing function* $\mathbf{R} : C \times N \mapsto C$ maps the current channel, c_c, and destination node, n_d, to the next channel c_n on the route from c_c to n_d, $\mathbf{R}(c_c, n_d) = c_n$. A channel is not allowed to route to itself, $c_c \neq c_n$. Note that this definition restricts the routing to be memoryless in the sense that a message arriving in channel c_c, destined for n_d has no memory of the route that brought it to c_c. However, this formulation of routing as a function from $C \times N$ to C has more memory than the conventional definition of routing as a function from $N \times N$ to C. Making routing dependent on the current channel rather than the current node allows us to develop the idea of channel dependence.

Definition 3 A *channel dependency graph*, D, for a given interconnection network, I, and routing function, $\mathbf{R}$, is a directed graph, $D = G(C, E)$. The vertices of D are the channels of I. The edges of D, are the pairs of channels connected by $\mathbf{R}$:

$$E = \{(c_i, c_j) | \mathbf{R}(c_i, n) = c_j \text{ for some } n \in N\}. \tag{1}$$

Note that, since channels are not allowed to route to themselves, there are no 1-cycles in D.

Definition 4 A *configuration* is an assignment of a subset of N to each queue. A configuration is legal if

$$\forall c_i \in C,\ \text{size}(c_i) < \text{cap}(c_i). \tag{2}$$

Definition 5 A *deadlocked configuration* for a routing function, **R**, is a non-empty legal configuration of channel queues $\ni$

$$\forall c_i \in C,\ (\forall n \ni \text{member}(n, c_i),\ n \neq d_i \text{ and } c_j = \mathbf{R}(c_i, n) \Rightarrow \text{size}(c_j) = \text{cap}(c_j)) \qquad (3)$$

In this configuration no message is one step from its destination and no message can advance because the queue for the next channel is full. A routing function, **R**, is *deadlock-free* on an interconnection network, I, if no deadlock configuration exists for that function on that network.

Summary of Notation

I	interconnection network, a directed graph $I = G(N, C)$,
N	the set of nodes,
n_i	a node,
C	the set of channels,
c_i	a channel,
c_{n_i}	the internal channel of node n_i,
s_i	the source node of channel c_i,
d_i	the destination node of channel c_i,
$\text{cap}(c_i)$	the capacity of the queue of channel c_i,
$\text{size}(c_i)$	the number of messages enqueued for channel c_i,
$\text{member}(n, c_i)$	true if a message destined for node n is enqueued for channel c_i,
R	a routing function $\mathbf{R} : C \times N \mapsto C$,
D	the channel dependency graph.

Theorem 1 A routing function, **R**, for an interconnection network, I, is deadlock-free iff there are no cycles in the channel dependency graph, D.

Proof:

$\Rightarrow$ Suppose a deadlock-free network has a cycle in D. Since there are no 1-cycles in D, this cycle must be of length two or more. Thus one can construct a deadlocked configuration by filling the queues of each node in the cycle with messages destined for a node two channels away where the first channel of the route is along the cycle. Contradiction.

$\Leftarrow$ Suppose a network with no cycles in D is deadlocked. Since D is acyclic one can assign a total order to the channels of C so that if $(c_i, c_j) \in E$ then $c_i > c_j$. Consider the least channel in this order with a full queue, c_l. Every channel, c_n, that c_l feeds is less than c_l, and thus does not have a full queue. Thus, no message in the queue for c_l is blocked and one do not have deadlock. ■

3 Constructing Deadlock-Free Routing Algorithms

Given a routing function, **R**, and a network, I, that is not deadlock-free, we can construct a deadlock-free routing ($\mathbf{R}'$,I) by removing the cycles from (**R**,I). In some cases, such as a cycle with channels in only one direction, it is not possible to remove edges without reducing the set of nodes that can be reached from a given node. In these cases we can break cycles by splitting each physical channel along the cycle into a group of *virtual channels*. Each group of virtual channels shares a physical communication channel; however, each virtual channel requires its own queue. The purpose of virtual channels is to associate different priorities to messages traversing the same physical channel depending on the channel from which a message arrives and the node to which the message is destined. By assigning priorities so that a message's priority always increases as it moves closer to its destination, we can construct deadlock-free routing algorithms.

Consider for example the case of a unidirectional four-cycle as shown in Figure 2A, $N = \{n_0, \ldots, n_3\}$, $C = \{c_0, \ldots, c_3\}$. The interconnection graph, I is shown on the left and the dependency graph, D is shown on the right. We pick channel c_0 to be the dividing channel of the cycle and split each channel into high virtual channels, $c_{10}, \ldots, c_{13}$, and low virtual channels, $c_{00}, \ldots, c_{03}$, as shown in Figure 2B. We can consider the internal channel of node n_i to be numbered c_{2i}. Priority in use of the physical channel is given to the low virtual channel.

Messages at a node numbered less than their destination node are routed on the high channels and messages at a node numbered greater than their destination node are routed on the low channels. Channel c_{00} is not used. We now have a total ordering of the virtual channels according to their subscripts: $c_{13} > c_{12} > c_{11} > c_{10} > c_{03} > c_{02} > c_{01}$. Thus, there is no cycle in D and the routing function is deadlock-free. In the next three sections we apply this technique to three practical communications networks. In each case we add virtual channels and restrict the routing to route messages in order of decreasing channel subscripts.

Some possible implementations of virtual channels are shown in Figure 3. A parallel implementation of virtual channels is shown in Figure 3A. Virtual channels between nodes n_i and n_j are multiplexed over a physical channel c_p. Queues $Q_{0p}, \ldots, Q_{kp}$ in n_i contain messages enqueued for transmission over virtual channels $c_{0p}, \ldots, c_{kp}$ all of which are multiplexed onto physical channel c_p. A priority encoder (PE), selects the highest priority non-empty queue, Q_{lp}, and enables the next packet of its data onto the physical channel. The data includes routing information which selects the next queue along the route for the message in n_j. When there is room in this queue, the packet is acknowledged and removed from Q_{lp}. For multi-packet messages, the routing information can be transmitted just once and stored in node n_j.

As long as a high-priority queue, Q_{hp}, is non-empty it blocks any lower priority queues, $Q_{0p}, \ldots, Q_{(h-1)p}$ from transmitting their contents on the physical channel. This blocking of lower priority channels may be undesirable for message-flow performance, but does not introduce a deadlock. In a practical system a more complex protocol could be used to grant lower priority virtual channels access to the physical channel while a high priority channel is blocked.

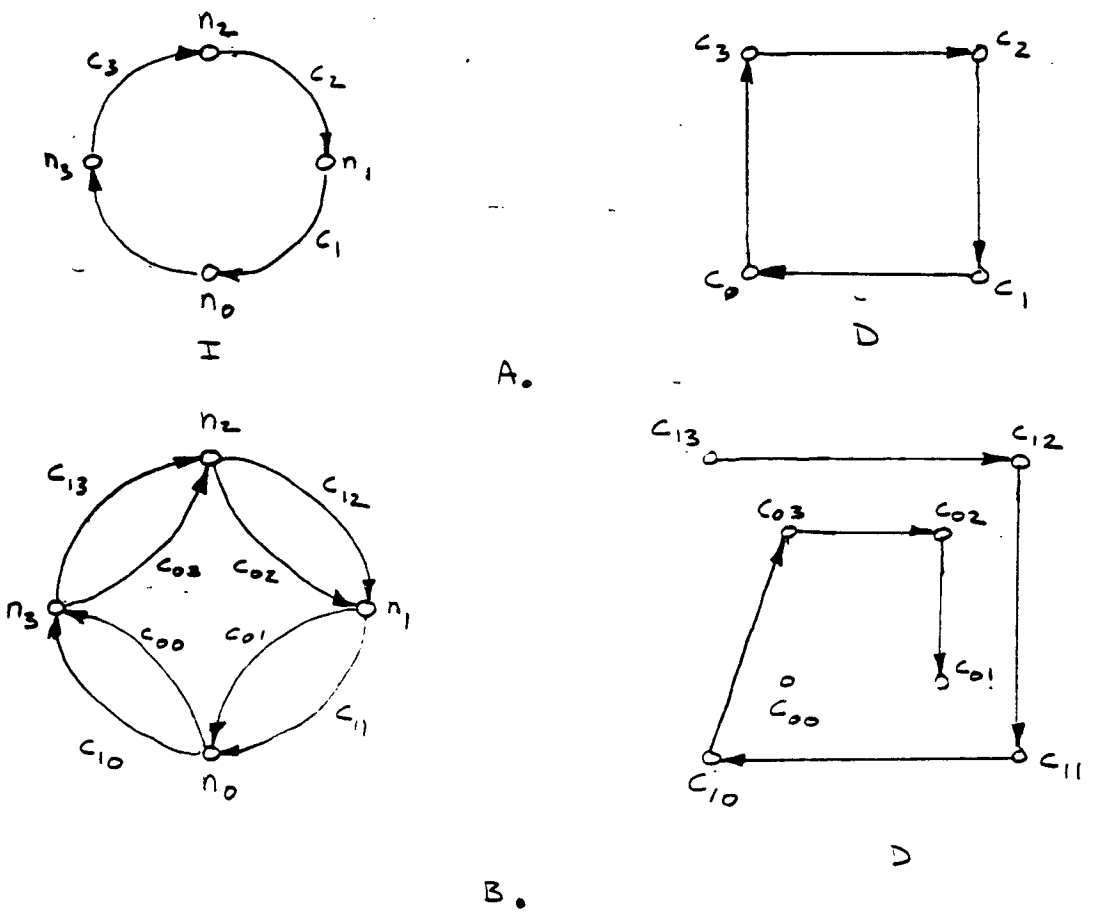
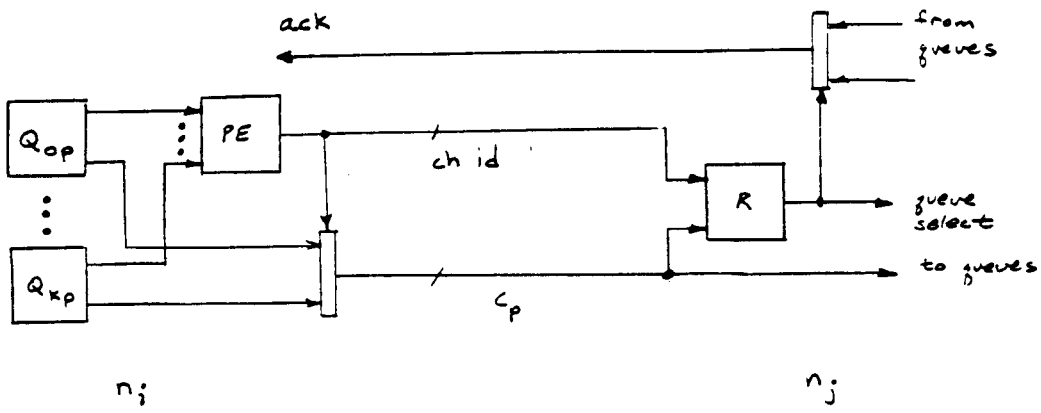
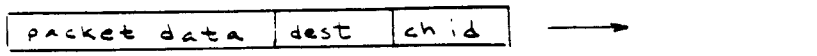


Figure 2: Breaking Deadlock by Adding Virtual Edges



A - Parallel Implementation



B - Format for Serial Implementation

Figure 3: Implementation of Virtual Channels

The same protocol can be implemented serially using the format shown in Figure 3B. The virtual channel id, routing information and data are transmitted serially from n_i to n_j. When this packet is accepted by n_j, an acknowledge signal is transmitted back to n_i to remove the packet from its queue and enable transmission of the next packet.

4 K-ary n-cubes

The E-cube routing algorithm [3,4] guarantees deadlock free routing in binary n-cubes. In a cube of dimension d, we denote a node as n_k where k is an d-digit binary number. Node n_k has d output channels, one for each dimension, labeled $c_{0k}, \ldots, c_{(d-1)k}$. The E-cube algorithm routes in decreasing order of dimension. A message arriving at node n_k

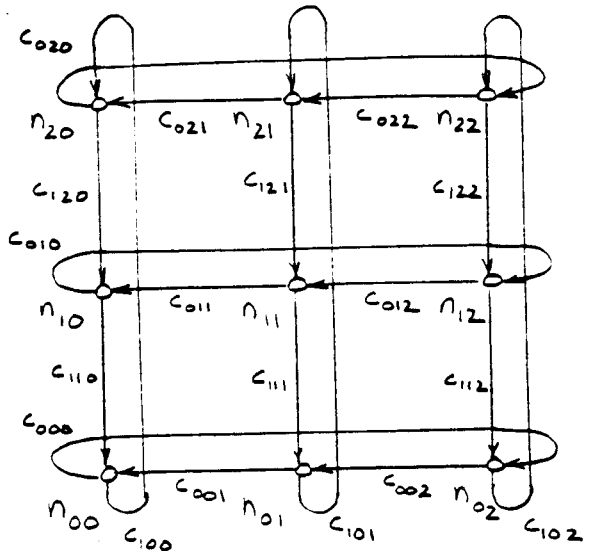


Figure 4: 3-ary 2-cube

destined for node n_l is routed on channel c_{ik} where i is the position of the most significant bit in which k and l differ. Since messages are routed in order of decreasing dimension and hence deceasing channel subscript, there are no cycles in the channel dependency graph and E-cube routing is deadlock-free.

Using the technique of virtual channels, this routing algorithm can be extended to handle all k-ary n-cubes: cubes with dimension n and k nodes in each dimension. Rings and toroidal meshes are included in this class of networks. This algorithm can also handle mixed radix cubes. Each node of a k-ary n-cube is identified by an n-digit radix k number. The i^{th} digit of the number represents the node's position in the i^{th} dimension. For example, the center processor in the 3-ary 2-cube of Figure 4 is n_{11}. The channels are identified by the number of their source node and their dimension. For example, the dimension 0 (horizontal) channel from n_{11} to n_{10} is c_{011}. To break cycles we divide each channel into an upper and lower virtual channel. The upper virtual channel of c_{011} will be labeled c_{0111}, and the lower virtual channel will be labeled c_{0011}. To give internal channels the lowest priority, they are labeled with a dimension higher than the dimension of the cube. To assure that the routing is deadlock-free, we restrict it to route through channels in order of descending subscripts. Priority is always given to the message from the channel with a lower subscript.

As in the E-cube algorithm we route in order of dimension, most significant dimension first. In each dimension, i, a message is routed in that dimension until it reaches a node whose subscript matches the destination address in the i^{th} position. The message is routed on the high channel if the i^{th} digit of the destination address is greater than the i^{th} digit of the present node's address. Otherwise the message is routed on the low channel. It is

easy to see that this routing algorithm routes in order of descending subscripts, and is thus deadlock-free.

Formally, we define the routing function:

$$\mathbf{R_{KNC}}(c_{dvn}, n_j) = \begin{cases} c_{d1(n-r^d)} \text{ if } (\mathrm{dig}(n,d) < \mathrm{dig}(j,d)) \wedge (\mathrm{dig}(n,d) \neq 0), \\ c_{d0(n-r^d)} \text{ if } (\mathrm{dig}(n,d) > \mathrm{dig}(j,d)) \vee (\mathrm{dig}(n,d) = 0), \\ c_{i1(n-r^d)} \text{ if } (\forall k > i, \mathrm{dig}(n,k) = \mathrm{dig}(j,k)) \wedge \\ \qquad (\mathrm{dig}(n,i) \neq \mathrm{dig}(j,i)). \end{cases} \tag{4}$$

Where $\mathrm{dig}(n,d)$ extracts the d^{th} digit of n, and r is the radix of the cube. The subtraction, $n - r^d$, is assumed to be performed modulo r.

Assertion **1** The routing function, $\mathbf{R_{KNC}}$, correctly routes messages from any node to any other node in a k-ary n-cube.

Proof: By induction on dimension, d.

For $d = 1$, a message, destined for n_j, enters the system at n_i on the internal channel, c_{d0i}. If $i < j$, the message is forwarded on channels, $c_{01i}, \ldots, c_{010}, c_{00r}, \ldots, c_{00(j+1)}$ to node n_j. If $i > j$, the path taken is, $c_{00i}, \ldots, c_{00(j+1)}$. In both cases the route reaches node n_j.

Assume that the routing works for dimensions $\leq d$ Then for dimension $d+1$ there are two cases. If $\mathrm{dig}(i,d) \neq \mathrm{dig}(j,d)$, then the message is routed around the most significant cycle to a node $n_k \ni \mathrm{dig}(k,d) = \mathrm{dig}(j,d)$, as in the $d = 1$ case above. If $\mathrm{dig}(i,d) = \mathrm{dig}(j,d)$, then the routing need only be performed in dimensions d and lower. In each of these cases, once the message reaches a node, n_k, $\ni \mathrm{dig}(k,d) = \mathrm{dig}(j,d)$, the third routing rule is used to route the message to a lower dimensional channel. The problem has then been reduced to one of dimension $\leq n$ and the routing reaches the correct node by induction. ■

Assertion **2** The routing function $\mathbf{R_{KNC}}$ on a k-ary n-cube interconnection network, I, is deadlock-free.

Proof: Since routing is performed in decreasing order of channel subscripts, $\forall c_i, c_j, n_c \ni \mathbf{R}(c_i, n_c) = c_j$, $i > j$, the channel dependency graph, D is acyclic. Thus by Theorem 1 the route is deadlock-free. ■

5 Cube Connected Cycles

The cube-connected cycle (CCC) [5] is an interconnection network based on the binary n-cube. In the CCC, each node of a a binary n-cube is replaced with an n-cycle and the cube connection in the n^{th} dimension is attached to the n^{th} node in the cycle. A CCC of dimension 3 is shown in Figure 5.

Each node in the CCC is labeled with the position of its cycle, (an n-bit binary number), and its position within the cycle. For example, in Figure 5, processor 2 in cycle 111 is labeled n_{2111}. There are two channels out of each node: an in-cycle channel and an out-of-cycle channel. The in-cycle channel is split into three virtual channels. One set of virtual channels is used to rotate a message around the cycle to get to the most significant node in the cycle. These channels are labeled c_{2d0ccc} where d is the dimension of the node, (its position in the cycle), and ccc is the cycle address. The next set of virtual channels is used to decrement the dimension during the E-cube routing of the message between cycles. These channels are labeled c_{1d1ccc}. The out-of-cycle channels, labeled c_{1d0ccc}, are used to toggle the bit of the current cycle address corresponding to dimension d. Note that the channels c_{1d0ccc} are actually physical channels. These connections are not shared with any other channels. The third set of virtual channels is used to rotate the message around the cycle to its destination once it is in the proper cycle. These channels are labeled c_{0d0ccc}. As above, we will restrict our routing to route through channels in order of descending subscripts. Priority is given to messages from channels with lower subscripts.

The routing algorithm proceeds in three phases. During phase 1, messages are routed around the cycle using the first set of virtual channels until they reach a node with dimension greater than or equal to the position of the most significant bit in which the destination cycle address differs from the current cycle address. During phase 2 we route the message to the proper cycle using a variant of the E-cube algorithm. At each step of phase 2, we find the most significant dimension, i, in which the current cycle and destination cycle addresses differ. The message is routed around the current cycle until it reaches the node with dimension i and is then routed out of the cycle. When the message arrives in the destination cycle it is routed around the cycle using the third set of virtual channels to reach its destination node. It is easy to see that routing is always performed in order of decreasing channel numbers, and thus the routing is guaranteed to be deadlock-free.

While most cube-connected cycles are binary, this routing algorithm can be extended for k-ary cube-connected cycles, that is, cycles with k cycles in each dimension. The only modification required is to split each out-of-cycle channel into two virtual channels. For simplicity, however we will analyze the routing for the binary case only. Formally, we define the routing function:

$$\mathbf{R_{CCC}}(c_{vdxn}, n_{d'j}) = \begin{cases} c_{2(d-1)0n} & \text{if } (v \geq 2) \wedge (d > 0) \wedge \\ & \quad (\exists i > d \ni \operatorname{dig}(i,n) \neq \operatorname{dig}(i,j)), \\ c_{1(d-1)1n} & \text{if } (v \geq 1) \wedge (x = 0) \wedge \\ & \quad (\forall i > d \operatorname{dig}(i,n) = \operatorname{dig}(i,j)) \wedge \\ & \quad (\operatorname{dig}(d,n) \neq \operatorname{dig}(i,j)), \\ c_{1(d-1)0n} & \text{if } (v \geq 1) \wedge (x = 0) \wedge \\ & \quad (\forall i \geq d \operatorname{dig}(i,n) = \operatorname{dig}(i,j)), \\ c_{1d0(n-r^d)} & \text{if } (v = 1) \wedge (x = 1) \wedge \\ & \quad (\forall i \geq d \operatorname{dig}(i,n) = \operatorname{dig}(i,j)), \\ c_{1d1(n-r^d)} & \text{if } (v = 1) \wedge (x = 1) \wedge \\ & \quad (\forall i > d \operatorname{dig}(i,n) = \operatorname{dig}(i,j)) \wedge \\ & \quad (\operatorname{dig}(d,n) \neq \operatorname{dig}(i,j)), \\ c_{0(d-1)0n} & \text{if } (x = 0) \wedge (n = j) \wedge (d' < d), \\ c_{0(d)0n} & \text{if } (x = 1) \wedge (n = j) \wedge (d' < d). \end{cases} \tag{5}$$

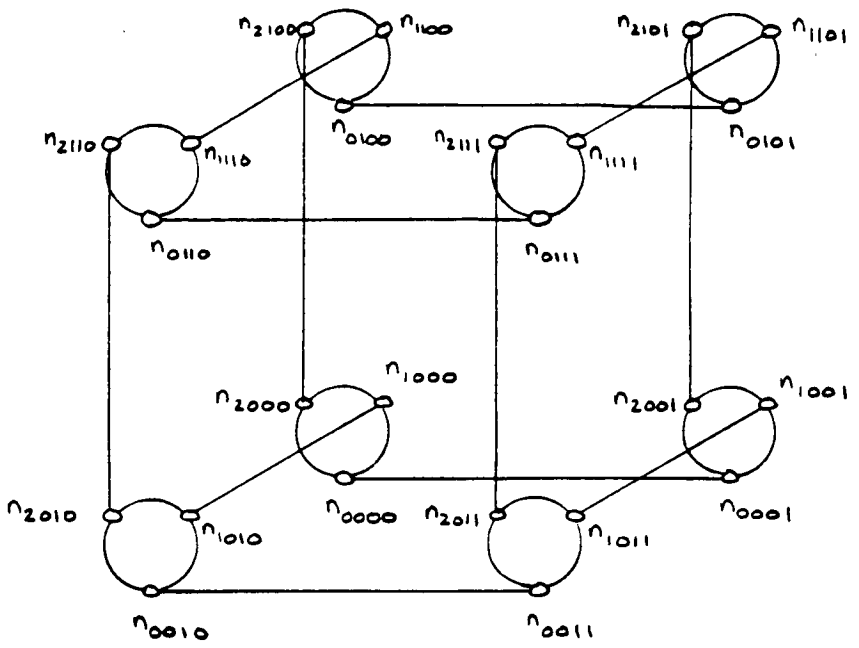


Figure 5: Cube Connected Cycle of Dimension 3

The following two assertions apply to the routing function, $\mathbf{R_{CCC}}$.

Assertion 3 $(v < 2) \Rightarrow \mathrm{dig}(i, n) = \mathrm{dig}(i, n') \; \forall i > d$.

Proof: The only way v can be decreased to less than 2 is for the right side of the assertion to hold. Initially $v = 3$, the message arrives on the internal channel. Then as long as $\exists i > d \ni \mathrm{dig}(i, n) \neq \mathrm{dig}(i, n')$, the first routing rule forwards the message along channels for which $v = 2$. When the arriving channel has $d = 0$, the current node has $d = m$ (where m is the dimension of the CCC). In this case the first routing rule forwards the message along a channel for which $v = 1$. The assertion holds since there are only m digits in n and n', so there is no $i > m$ for which the i^{th} digit of these two cycle addresses differ.

Once $v < 2$, the right side of the assertion continues to hold because of routing rules 2,3,4 and 5. We prove this by induction on d. For $d = m$, the assertion holds as stated above. If we assume the assertion holds for $d = i$, then for $d = i - 1$ it also holds since routing rules 2 and 5 route the messages out-of-cycle to toggle the d^{th} digit of n if the addresses disagree in that digit. ■

Assertion 4 $(v = 0) \Rightarrow (n' = n) \wedge (d' < d)$.

Proof: The only way to decrease v to zero is by routing rules 6 and 7 which require the right side of the assertion. By Assertion 3, when $v < 2$ and $d = 0$, after one or two more

traversals, the right side of the assertion will be met. Once $v = 0$, no out-of-cycle channels will be used so n does not change. ■

Assertion **5** The routing function, $\mathbf{R}_{CCC}$, correctly routes messages from any node to any other node in a k-ary cube-connected cycle.

Proof: Since there are only a finite number of channels, and $\mathbf{R}_{CCC}$ routes in order of decreasing channel subscripts, v will eventually be decreased to 0. Then by Assertion 4, $n' = n$ and $d' < d$, so the message will be rotated about the cycle until it reaches its destination. ■

Assertion **6** The routing function $\mathbf{R}_{CCC}$ on a k-ary cube-connected cycle interconnection network, I, is deadlock-free.

Proof: As in the case of k-ary n-cubes, since routing is performed in decreasing order of channel subscripts, $\forall c_i, c_j, n_c \ni \mathbf{R}(c_i, n_c) = c_j$, $i > j$, the channel dependency graph, D is acyclic. Thus by Theorem 1 the route is deadlock-free. ■

6 Shuffle-Exchange Networks

The shuffle-exchange network [6], shown in Figure 6, provides two channels out of each node: a shuffle channel and an exchange channel. The shuffle channel from node n_i has as its destination the node n_j where the binary representation of j is the left rotation of the binary representation of i, denoted here $j = \mathrm{rol}(i)$. The exchange channel from n_i routes messages to n_k where the binary representations of k and i differ in the least significant bit.

The exchange channel out of n_i is labeled c_{1i}. The shuffle channel is labeled c_{0i}. For the shuffle-exchange network we split each channel into n virtual channels where $N = 2^n$. That is, we have one virtual channel for each bit of node address. Readers understanding the relationship between the binary n-cube and the shuffle will find this assignment of virtual channels unsuprising. The virtual channels are labeled c_{dxi}. Where $0 \leq d \leq n-1$, $x \in \{0, 1\}$, and $0 \leq i \leq N$. The internal channel at each node is labeled with $d = n$ to give it lower priority than any other channel.

The routing algorithm, like the E-cube algorithm, routes a message toward its destination one bit at a time beginning with the most significant bit. At the i^{th} step of the route, the $n - i^{\text{th}}$ bit of the destination address is compared to the least significant bit of the current node address. If the two bits agree, the message is forwarded over the shuffle channel to rotate the node address around to the next bit. Otherwise, the message is forwarded over the exchange channel to bring the two bits into agreement and then over the shuffle channel to rotate the address. At the i^{th} step messages are forwarded over channels with $d = n - i$. Since d is always decreasing and, during a single step, the exchange channel is used before the shuffle channel, messages are routed in order of decreasing virtual channel subscripts.

Formally, we define the routing function:

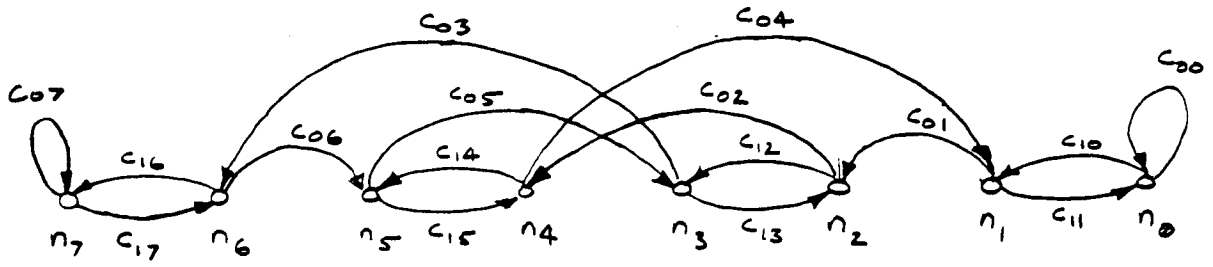


Figure 6: Shuffle-Exchange Network, $N = 8$

$$\mathbf{R_{SEN}}(c_{dxi}, n_j) = \begin{cases} c_{(d-1)0\mathrm{rol}(i)} & \text{if } (x = 0) \wedge (\mathrm{dig}(d-1, i) = \mathrm{dig}(0, j)), \\ c_{(d-1)1\mathrm{rol}(i)} & \text{if } (x = 0) \wedge (\mathrm{dig}(d-1, i) \neq \mathrm{dig}(0, j)), \\ c_{d0(i \oplus 1)} & \text{if } (x = 1). \end{cases} \tag{6}$$

Assertion 7 If a message is routed on channel c_{d0i} destined for node n_j, then $\forall m \geq d$, $\mathrm{dig}(m, j) = \mathrm{dig}(m, k)$, where k is i rotated left d bits.

Proof: By induction on dimension, d. For $d = n-1$, a message is only routed on the shuffle connection, $x = 0$, of dimension $n-1$ if $\mathrm{dig}(n-1, i) = \mathrm{dig}(0, j) \Rightarrow \mathrm{dig}(n-1, i) = \mathrm{dig}(n-1, k)$ by the definition of k. Since there is only one possible value for m, the assertion is satisfied.

If the assertion is true for dimension d, then after routing on connection $c_{(d-1)0i}$, the assertion also holds for $d-1$ by the same argument: a message is only routed on the shuffle connection, $x = 0$, of dimension $d-1$ if $\mathrm{dig}(d-1, i) = \mathrm{dig}(0, j) \Rightarrow \mathrm{dig}(d-1, i) = \mathrm{dig}(d-1, k)$. ■

Assertion 8 The routing function, $\mathbf{R_{SEN}}$, correctly routes messages from any node to any other node in a shuffle-exchange network.

Proof: From Assertion 7, after routing on channel c_{00i}, the message will be at its destination. It may reach its destination before this. Since the function routes in order of decreasing channel subscripts and there are a finite number of channels messages will reach their destinations. ■

Assertion **9** The routing function, $\mathbf{R_{SEN}}$, on a shuffle-exchange network, I, is deadlock-free.

Proof: Since routing is performed in order of decreasing channel numbers, D is acyclic and the routing is deadlock-free. ■

7 Conclusion

We have shown how a deadlock-free routing algorithm can be constructed for an arbitrary communications network by introducing virtual channels. This technique has been applied to construct deadlock-free routing algorithms for k-ary n-cubes, for cube-connected cycles, and for shuffle exchange networks.

The use of virtual channels to construct deadlock-free routing functions is motivated by the definition of a routing function that maps $C \times N$ to C, rather than the conventional definition of a routing function that maps $N \times N$ to C. By including C in the domain of the routing function, we explicitly define the dependencies between channels. These dependencies are represented by a channel dependency graph D. A necessary and sufficient condition for deadlock-free routing is that D be acyclic.

To develop deadlock-free routing algorithms for specific networks we assign a subscript to each virtual channel using a mixed radix notation. Routing is performed in order of decreasing subscripts. Since the subscripts define a total order on the channels, there are no cyclic dependencies and the routing is deadlock-free.

The cost of implementing virtual channels need not be high. Each virtual channel requires its own queue, but the queue size can be as small as the unit of data that is transmitted on each handshake, possibly a single bit. With single bit queueing, virtual channels can be used to implement the low latency wormhole routing technique [7].

The availability of deadlock-free routing algorithms encourages the investigation of different interconnection topologies. While O(logN) diameter networks such as the binary n-cube and the shuffle are attractive because of their richness of interconnection, these networks are almost always embedded in a grid for physical implementation. In keeping with the VLSI imperative of making form fit function, high bandwidth grid interconnections may turn out to be more attractive.

References

[1] Seitz, Charles L., "The Cosmic Cube," *CACM*, 28(1), January 1985, pp. 22-33.

[2] Kleinrock, Leonard, *Queueing Systems,* Wiley, 1976, Vol. 2, pp. 438-440.

[3] Sullivan, H. and Brashkow, T.R., "A Large Scale Homogeneous Machine," *Proc. 4th Annual Symposium on Computer Architecture,* pp 105-124, 1977.

[4] Lang, Charles R., *The Extension of Object-Oriented Languages to a Homogeneous Concurrent Architecture,* Caltech Ph.D. Thesis, 5014:TR:82, 1982, pp. 118-124.

[5] Preparata, F.P. and Vuillemin, J.E., "The Cube-Connected Cycles: A Versatile Network for Parallel Computation," *Proc. 20th IEEE Symposium on the Foundations of Computer Science,* pp. 140-147.

[6] Stone, H.S., "Parallel Processing with the Perfect Shuffle," *IEEE Transactions on Computers,* vol. C-20, February 1971, pp. 153-161.

[7] Seitz, C. et. al, *Wormhole Chip Project Report,* Winter 1985.