Multicomputers

William C. Athas
and
Charles L. Seitz

Computer Science Department
California Institute of Technology

5244:TR:87

Multicomputers*

William C. Athas, Charles L. Seitz
Department of Computer Science
California Institute of Technology
Pasadena CA 91125

5244:TR:87
June 1987

Written for the DARPA SC workshop in Syracuse NY
29 June – 1 July 1987

Abstract

This report outlines the history, current status, current developments, and plans for the message-passing concurrent computers, or *multicomputers*, developed in the Submicron Systems Architecture Project at Caltech. These systems include the Cosmic Cube and its commercial descendants, two second-generation cosmic cubes currently in development, and the Mosaic C, a fine grain multicomputer whose nodes are single VLSI chips.

Section 1 introduces the physical architectures, with particular attention to the characteristics of the message-passing networks.

Section 2 describes the programming environments for the first and second generation medium grain size "cubes."

Section 3 describes a fine grain concurrent object-oriented programming notation called Cantor, which currently runs on cubes and sequential systems, and which will be used for application programming of the Mosaic C.

* The research described in this report was sponsored in part by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597, and in part by grants from Intel Scientific Computers and Ametek Computer Research Division.

1. Multicomputer Architecture

1.1 Idealized Architecture

When abstracted to the level of the application programmer, a *multicomputer,* also referred to as a *message-passing concurrent computer,* can be regarded as N computers called "nodes" connected by a communication network:

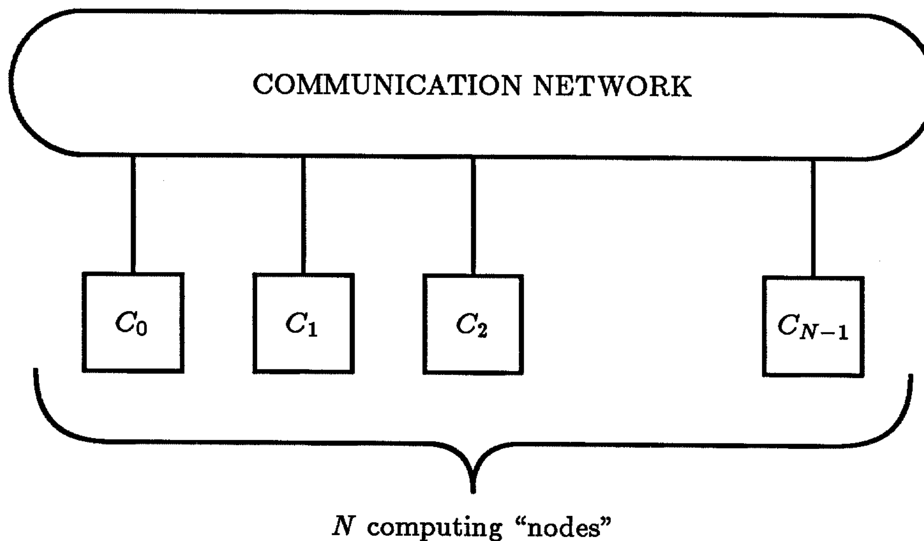

Fig 1: Programmer's model of a message-passing multicomputer

Each of the N node computers executes programs concurrently, and coordinates its activities with the other nodes only by sending and receiving messages. There is no memory shared between the nodes, as in the other principal type of MIMD architecture, the shared-memory multiprocessor, and no global address space. Since each node computer has its own separate memory and address space, multicomputers are sometimes referred to as *distributed memory* systems.

A resident operating system (kernel) in each node supports multiprogramming within a single node, such that each node may contain many processes (or objects). The number of concurrent processes involved in a single computation can accordingly be much larger than N. Since messages are directed to processes (or objects), the results of a computation do not depend on process placement, to within the computation being deterministic, with the result that the programming environments are extremely portable. Message-passing programs are routinely run not only on multicomputers, but also on shared memory multiprocessors, across networks of workstations, and on sequential computers.

We believe that multicomputers are reasonably "general purpose" concurrent computers. Even the first generation of these machines have proved to be applicable to a wide range

of engineering and scientific computations, and there have been hundreds of research papers and reports published on the concurrent formulations and results of these computations.

1.2 History and Plans

The *Cosmic Cube* [Seitz 85], an experimental machine that members of our research group designed (1981), built several copies of (1982-83), and developed the system software for (1981-85), and its commercial "hypercube multicomputer" descendants made by Intel, Ametek, and N-CUBE (all introduced in 1985), are first generation examples of this architecture. Node (grain) sizes of these systems range from a 7-chip node (one custom VLSI chip with processor and communications, plus 6 dRAM chips) in the N-CUBE, packaged with 64 nodes per circuit board, to nodes that require two circuit boards (regular node board plus vector accelerator board) in the Intel iPSC VX. All of these first generation systems use an underlying communication network of a direct binary n-cube (hypercube), and the number of nodes in a single system ranges in the commercial machines from 4 (2-cube) to 1024 (10-cube). Although all of the first-generation systems are ***homogeneous***, that is, the nodes are nominally identical[1], multicomputers can equally well be constructed as heterogeneous systems with a variety of types of nodes specialized for different computations, so long as all the nodes employ the same communication protocols.

Our research group at Caltech is currently in the midst of developing *second generation* multicomputers at what we regard as the two most interesting design points in node size:

- *Second-generation "Cosmic Cubes"* – In joint projects with Ametek and Intel, we are developing two different medium grain size multicomputers based on commercial processors and RAM. The speed of a single node is about *one* order of magnitude faster than the nodes in the first generation systems, or similar to that of a modern workstation computer. The message communication is about *three* orders of magnitude faster than the first generation systems. We expect that the improvement of two orders of magnitude in the *relationship* between communication and computing performance will dramatically widen the application span and will also simplify the programming of these systems.

- *Mosaic C* – The Mosaic C is a fine grain multicomputer system. The node consists of a 14 MIPS processor, 20 MB/s routing communication channels, and 16KB of RAM, integrated onto a *single VLSI chip*. We expect our first prototype of 1024 nodes to be running in June 1988, and a full-size machine of 16,384 nodes in June 1989. Although the performance of such a fine grain system – 200,000 MIPS for the 16K-node machine – is quite phenomenal, its programming requires a similarly fine grained approach, such as the use of the Cantor programming language described in section 3.

While wanting to avoid the "hype" of referring to these as "second generation" systems, the realization of a set of new ideas about the physical design and programming of this class of systems makes the second generation "cubes" and the Mosaic C quite different from the first generation systems. While structurally similar, and programmed in similar

[1] The Intel systems are sometimes configured with subcubes of the machine having different types of accelerators.

ways, the parametric change in the relationship between computation and communication performance makes these second generation machines virtually a different architecture.

The effect of faster communications is to extend the application span of multicomputers from easily partitioned and distributed problems (*eg,* matrix operations, PDE solvers, finite element analysis, finite difference methods, distant or local field many-body problems, FFTs, ray tracing, distributed simulation of systems composed of loosely coupled physical processes) to "high flux" [Ullman 84] or "communication-limited" problems (*eg,* searching, sorting, concurrent data structures [5209:TR:86], graph problems, signal processing, image processing, and distributed simulation of systems composed of many tightly coupled physical processes) that place extravagant demands on the communication network for high bandwidth, low latency, and non-local communication.

In order to place these developments into a long term perspective, one observes that in accordance with Steve Squires's model of technology development in the DARPA Strategic Computing program, the first generation machines use relatively unaggressive technology to establish the computational model, programming methods, and applications. In the case of the cosmic cube type of multicomputers, the first generation even resulted in some cost-effective commercial machines. The second generation systems achieve much higher performance by a combination of organizational improvement and VLSI technology. Since multicomputers are scalable in size and technology, we expect that a third generation will be achievable with little organizational change from the second generation, but only the application of VLSI technology advances.

For example, the message systems for the second generation "cubes" are already close to physical limits imposed by the wiring bisection of the network and the rate at which the custom VLSI routing chips can drive wires. The use of smaller feature size VLSI will scale processor and communication performance together. Thus we do not expect in the evolution of these architectures beyond the second generation that the relationship between communication and computing performance will change significantly. The organizational improvements that will be most important in the third generation will be in the instruction processors, not only their performance, but their ability to switch context more efficiently than processors designed to be used in PCs and workstations. One approach is to design the processors to be "message-driven."

One might summarize this projection of the evolution of the medium grain size cosmic cube type systems as follows:

Generation:	*First*	*Second*	*Third*
Years:	1983-87	1988-92	1993-97
Typical node			
MIPS	0.5	5	50
Mflops scalar	0.1	1	10
Mflops vector	10	20	200
memory (MB)	0.5	4	32
Communication latency (100B msg)			
neighbor (μs)	1000	5	0.5
non-local (μs)	5000	5	0.5

We leave it to the reader to scale the performance and memory capacity of the nodes to that of a system.

The research, design studies, simulations, and software development that have led to the second generation designs have included many inventions that may be individually interesting to the architects, but not necessarily to application programmers and users. What may in some cases be of interest to application programmers are the characteristics of the communication networks, which we now describe in detail.

1.3 Communication Networks

If the message communication network were as ideal as shown in this abstract view of Fig 1 of N completely connected computers (the message-passing counterpart of the PRAM model), one could predict the performance of a particular concurrent computation based entirely on the speed of the individual computers and on the way in which the concurrent formulation of the computation distributes the parts of a computation. Complete connections and contention buses are impractical for large N, so a multicomputer communication network is implemented as a direct network such as the binary n-cube network pictured in Fig 2, in which messages are routed through intermediate nodes.

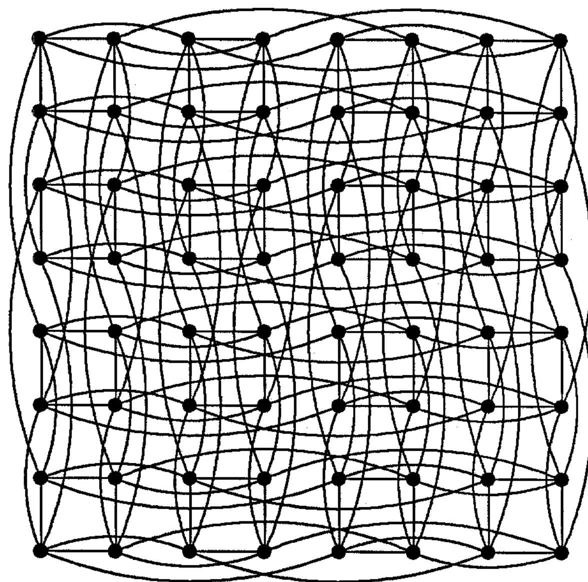

Fig 2: Binary n-cube network ($n = 6$) projected onto two dimensions

Direct networks exhibit paths of different lengths, so that in communication-intensive computations one can take advantage of placing processes not only to balance the computing load, but also to localize message traffic. Relatively localized message traffic helps reduce congestion in the communication network, but is not essential. For example, in simpler application programs the processes may be distributed systematically [5203:TR:85] or an optimal static placement can be computed [5184:TR:85], while in the more general case one may use a variety of algorithmic techniques [5242:TR:87] to determine process placement. Another important case is when the concurrent parts are small and the degree of concurrency is much larger than N, for which near-perfect load balancing is achieved statistically.

All of the first-generation machines use a direct binary n-cube (hypercube) to connect $2^n = N$ node computers. Except for the original cosmic cube, which has since been reprogrammed to use "wormhole" routing, these machines use store-and-forward routing, so that the message latency is linear in message distance for short messages. Longer messages are fragmented into packets, and at the packet level the first generation machines employ cut-through routing [Kermani 79] similar to that used in the Connection Machine [Hillis 85].

The "wormhole" routing used in our second generation machines is much like cut-through routing, in that the head of a packet proceeds if possible directly from incoming to outgoing channel, but if the outgoing channel is already in use, the packet is blocked rather than queued in the node. Wormhole routing is distinctive to multicomputers and multiprocessor switches, in which one can assert flow control at the level of the single parallel data item passed on a channel, and cannot, like cut-through routing, be applied to packet networks. It is the use of wormhole routing that allows us in the second generation machines to make the message latency insensitive to the distance the message must travel in the network. It also does not use storage bandwidth in routing nodes, a crucial advantage, since in the limit the node storage bandwidth is the most precious resource in a multicomputer system.

There are many reasons for the choice of the binary n-cube in the first generation machines, including the existence of a simple deadlock-free routing algorithm, logarithmic diameter, and log-time algorithms that exchange information along successive dimensions (similar to Batcher sort) to compute global quantities. However, our second generation machines do not use binary n-cubes, but meshes of routing chips [Dally 86], *eg:*

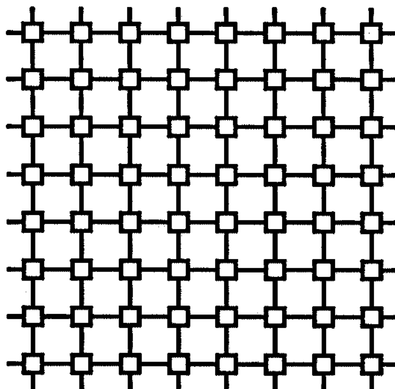

Fig 3: 8×8 *mesh form of 8-ary 2-cube*

Messages are routed on this mesh by "wormhole" routing by an algorithm that assures that deadlock cannot occur.

The mesh is a variant of the bidirectional k-ary n-cube [Seitz 84] connecting $N = k^n$ nodes, but with the "end-around" (torus) connection eliminated. The use of networks of lower dimension (n) and higher radix (k) is justified by analyses [5209:TR:86, Dally 87] that show that for k-ary n-cubes of given wire bisection, the latency is minimized for messages of typical length by networks of relatively small dimension, *eg*, for $n = 2$ when $N \leq 2^{10}$ or $n = 3$ for $2^{11} \leq N \leq 2^{14}$.

The network component of message latency with wormhole routing is given by

$$T_n = T_c(pD + \lceil \frac{L}{W} \rceil)$$

where:

T_c is the "cycle time" of the network – the time to convey W bits. ($0.040\mu s$)

p is the number of cycles necessary to extend the wormhole path through one node. (2)

D is the number of channels the message traverses, or *distance.*

L is the *length* of the message in bits.

W is the *width* of the communication channel – the number of bits conveyed in one cycle. (8)

Typical values of the parameters for the second generation cubes are shown in parentheses. Thus for example, the time required to send a 200 bit message to a node distance 6 is $0.040(12 + 25) = 1.5\mu s$. A corresponding time, including the software overhead of store and forward routing, on the first generation cubes would be about 6 ms. Notice that the latency is not strongly dependent on distance, since the distance and length dependent terms appear as a sum, and the second term is nearly always dominant. Message locality is thus not much of an issue in the second generation machines except as extreme numbers of non-local messages may cause the latency to increase due to congestion (blocking) in the network. It is in this case that it may become desirable to localize message traffic. In order to examine the severity of the congestion, the networks for the second generation cubes have been simulated extensively using the cosmic cubes and a 128-node Intel iPSC.

In addition to their performance for messages produced and consumed within the machine, another feature of the communication network of these machines that is important for I/O intensive applications is the I/O bandwidth of $4\frac{W}{T_c}\sqrt{N}$ in each direction around the edge of the mesh.

1.4 Configurations

Machines such as the second generation cubes can be scaled and configured in several ways:

1. *Scale N* – $N = 256$, a 16×16 mesh, is the "centerline" design point for the second generation cubes. Unlike the binary n-cubes, which are scaled by changing n, these meshes are scaled by changing k. This network scaling has a consequence that smaller machines have a surplus of communication capability, while it is possible in larger machines to drive the message system into a condition of moderate congestion if message traffic is very intense and and non-localized.

2. *Node memory* – We believe that 4MB will be the preferred and typical node memory size, but one should be able to expand the node memory to larger sizes, such as 16MB, if desired.

3. *Accelerators* – One can install accelerators or "object experts" [5209:TR:86] on each node of a multicomputer to speed up operations on particular data types. We hope and expect that a variety of accelerators, such as vector floating point accelerators, will become available for the second generation multicomputers just as they are for workstations and PCs. Accelerator configurations may be heterogeneous.

4. *Peripherals* – I/O devices (disks, displays, communications, etc) can be installed either on individual nodes or around the edges of the mesh. We have studied, for example, the idea of placing small disk servers at one edge of the mesh to run the nodes as virtual memory computers. Even when swapping is disk-limited, we found for $k = 16$ that swapping would consume less than 1% of the available communication capacity of the network. The aggregate I/O bandwidth appears to be more than adequate for most applications.

2. Cosmic Cube Programming Environments

Instead of designing and implementing new programming languages for the cosmic cubes, we preferred to use existing sequential programming languages for process code, with a library of functions or procedures to control message and process spawning operations. In this way we start with highly evolved programming notations for which someone has already written compilers and most of the libraries necessary to develop interesting application programs. Cube process code may be written in any sequential programming language — *eg*, C, Pascal, FORTRAN, LISP, assembly, etc — for which one has a suitable compiler or assembler for the node processor and a library of routines to invoke system functions. One can mix processes written in different source languages in a single computation, with only the usual precaution of understanding the possible differences in data representations. At Caltech we most often use C, so we shall use C syntax for illustrations in this report.

Although this approach of using existing programming notations can be extended down to fine grain concurrency on the medium grain size cubes, and to fine grain multicomputers such as the Mosaic, it becomes increasingly awkward as the size of individual processes or objects gets smaller. Thus for fine grain concurrent programming we find it much more effective to use programming notations specifically designed for fine grain concurrency, such as the Cantor notation described in section 3.

2.1 Message Handling

The concurrent architectures of the Caltech "cubes" and the Mosaic support message-passing directly in hardware. The sending of a message from one node to another requires two steps. The first step is the transmission of the message as a sequence of one or more packets through the routing network. The second step is the reassembly of packets into messages and the processing of the message by the receiving node.

The receiving node must perform a minimal amount of message processing for each packet delivered. This minimal processing usually consists of the received message being queued in memory for the process to which it is addressed, although other actions may also be invoked, including discarding the message, packet reassembly, or using the message to create a code or data segment for a process being spawned. Each message delivered must be sufficiently processed that future messages are not permanently blocked from delivery. The property of freedom from deadlock for the routing network is dependent upon this minimal processing; every message that is delivered to a node must be consumed.

The requirement that each message upon delivery at its destination node result in some action to process the message is called the ***reactive property***. Message-driven programming is a programming style that allows the programmer to exploit the reactive property of the message-passing hardware. Each message is given a ***tag*** as it is sent, and this tag determines the method for handling the message when it is delivered. For example, a tag can specify that the message be queued, discarded, processed immediately, acknowledged, or turned into a code or data segment.

The overhead for this initial processing in accordance with the tag is minimal. The tag is used as an index into a table to select a *handler* for the message. The handler consists of a segment of executable code and a data segment that is used to hold both persistent and temporary variables. After the table lookup is performed, the code for the handler is executed. This code has the property that it will not execute indefinitely. This property is crucial because only one handler can be active per node. The handler must therefore process a message in bounded time so that future messages are not permanently blocked from delivery.

Within this fundamental framework, one may build a variety of handlers to provide the programmer a set of message functions that are convenient for the programming language or specific application. In this section we shall describe two of these environments: first, the process model environment that has been in use for the past three years on the cosmic cubes; second, a reactive model that is a protected version for the application programmer of the system interface between the node kernel and the handlers.

2.2 Process model programming

In this section we shall describe the programming environment that has been used most extensively for writing application programs for the cosmic cube and Intel iPSC. As an historical note, the node operating system kernel that supports this model was first implemented as the "cosmic kernel," the original node operating system for the cosmic cube. Exactly the same functions are now implemented on the cosmic cubes with the "reactive kernel" that was written for the second generation cubes, together with a cosmic handler. The latest version (7) of the Cosmic Environment host runtime system also employs a reactive model at its lowest level with a cosmic handler. In this way the first generation cubes and host systems have been used as the porting base for the system software for the second generation cubes.

The host runtime system, called the *Cosmic Environment*, has been distributed to numerous other universities, and allows the same message-passing programs that run on cubes to be run on single Unix systems or across Unix machines that are connected by conventional networks. This programming environment is described completely in the "C Programmer's Guide to the Cosmic Cube" [5203:TR:85][2].

For the purposes of this exposition, the cosmic cube and the Intel iPSC may be considered to be interchangeable. The hardware details of the particular cube are largely invisible to the C programmer, although programmers are generally conscious in the first generation systems of the relationship between node numbers and message distance on a binary n-cube.

2.2.1 Process Model

The basic unit of computation in this environment 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.*

[2] This technical report can be ordered from the Caltech Computer Science Librarian, Caltech 256-80, Pasadena CA 91125, at a cost of $9.00 postpaid. This technical report also includes instructions on obtaining a copy of the Cosmic Environment software.

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, even if they happen to be in the same node.

Processes located outside the cube nodes, either in the *cube host* (the machine that provides a message path between a cube and local network) 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 many 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*[3] – 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 is useful both as the host system and as a tool for the early development of programs for the cube.

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 binary n-cube numbering of nodes. For processes that must run

[3] Intel refers to this software, licensed for use with the Intel iPSC, as the "seamless" environment.

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*. A process group and cube, or a subcube of a larger cube, is allocated by the user with the host utility getcube *[cube type] [cube dimension]*. Cubes are normally not time-shared, but *space-shared*. For example, a 7-cube might be allocated with one user running on a 6-cube, two others on 4-cubes, and with the remaining 5-cube being free. This mode of sharing has proven to be most appropriate for multicomputers, since it allows a user to select a number of nodes that is appropriate to the number of processes and load balancing characteristics of a particular computation, produces predictable run times, and does not allow one user's processes to upset the load balance of another 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:

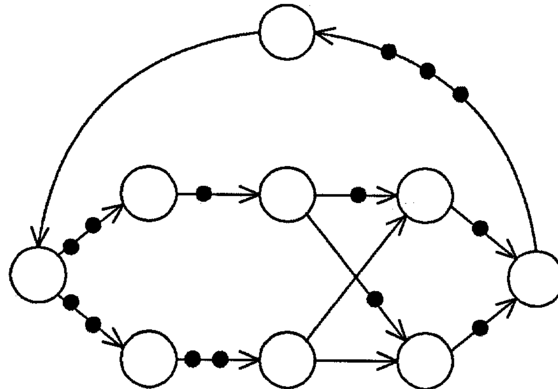

Figure 4: Example of a process structure graph

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 physical communication within a single node, between adjacent nodes, along a path in the n-cube graph, or to a host, but these communications are all logically equivalent. 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. Once a process is loaded or *spawned* it will not migrate to another node. There are a number of functions that are used to spawn new processes, most commonly `spawnf` to spawn a process from a file, and `spawnp` to spawn a process as a duplicate of an existing process. For simple applications one may use utility programs available on the hosts to handle program loading.

2.2.2 The Cosmic Kernel

The original 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. The system calls and corresponding C functions are called *primitive* functions to indicate that they correspond directly to kernel primitives. The primitive `sendq`, `recvq`, `probe`, `flick`, and `block` functions operate in a way that allows a process to manage many concurrent message activities. In addition to these primitive functions, there is an extensive library of C functions to assist the programmer.

The most distinctive characteristic of the cosmic kernel message functions is that a process is able to *exercise discretion* in the messages that are received. Messages are received selectively by *type*, and the process is able to accept messages in an order different from the order in which they are received by exploiting the type mechanism. Messages that have been received by the node but not by the process are queued by the system until called for. This particular feature of the cosmic cube primitives is popular for scientific programming styles in which pointers and dynamic storage allocation is shunned.

The application programmer needs also to understand the concurrency, queueing, and message order properties of the message system itself. 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.

Process creation in the cosmic cube is dynamic and is accomplished by an *outer kernel* process, called the SPAWN process, in each node. This is the one area where the cosmic cube and Intel iPSC are not entirely compatible. Due to its Xenix (rather than Berkeley Unix) cube host, the iPSC has no outer kernel, and has a more restricted set of functions for creating processes. The cosmic cube's 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. Similar functions are provided in the reactive kernel by using handlers rather than processes. The application programmer generally invokes these outer kernel features through functions and host utility programs that interact with outer kernel processes, rather than through message exchanges.

2.3 The Reactive Primitives

The very small network component of the message latency in the second generation cubes makes it essential that we streamline the message handling in the nodes. In addition, the experience with the first generation cubes suggests that we should make lower level message primitives than those that the cosmic kernel supports available to application programmers and particularly to developers of programming language systems – *eg*, Prolog, LISP, Ada, etc – and application systems – *eg,* simulators.

Our choice of the lowest level message primitives for the second generation cubes has also been influenced by object-oriented programming and by the Actor model of computation [Agha 86]. All of these considerations and influences led to the design of a "message-driven" or "reactive" node kernel. The distinction between processes and objects is rather slight in this model. The reactive primitives, although illustrated here as called from C user processes, are similar to those used between the reactive kernel and a handler, and in the message operations in Cantor intermediate code.

Messages are sent and received from a heap maintained by the kernel, and accessed both by processes (objects) and by the message system. Messages are sent by allocating message space in the heap by:

```
p = xmalloc(len);
```

then sending the message built in that space by:

```
xsend(p,node,pid);
```

which also deallocates the space. That is, `xsend(p,...)` is like `free(p)` except that it also sends a message. Thus there is no need for blocking or feedback that the message has been sent, as is done in the cosmic environment via the message descriptor lock variable.

Messages are received by:

```
p = xrecv();
```

which is semantically like `malloc`, except that the length of the block allocated is determined by the length of the message received. Space thus allocated can later be freed after the message is used, either by the `xfree(p)` function, or by building another message in the space and freeing the space by the `xsend(p,...)` function.

Just as the `malloc` function in the Unix environment may return the NULL pointer if there is no space available, `xrecv` function will return the NULL pointer if no message is present. However, `xrecv` returns the NULL pointer only if there are no messages in the node's entire receive queue. Calling `xrecv` when the next queued message is for another object (process) causes the kernel to save the state of this object and start running the

other object. Thus the appearance of xrecv in the code marks "choice points" for switching execution to another object, and it is in this sense that the scheduling is "message-driven." It is still possible for an object to do work while waiting for a message, as in:

```
if (p = xrecv()) digest(p);
else do_other_work();
```

So long as an object (process) is making progress, one does not force a context switch unless a *priority message* is received. The blocking receive can be expressed as a busy-wait loop:

```
while ( !(p = xrecv()) );
```

which has no inefficiency, since if the NULL pointer is returned the node does not have anything else to do anyway.

These purely reactive primitives are not quite adequate for all purposes. For example, remote procedure calls (RPCs) and similar transactions implemented by messages require an acknowledge message that can be distinguished from other messages. RPCs are used extensively by the system for functions such as spawning new objects, even if they tend not to be used in application programs. Accordingly, a RPC is implemented directly in this environment by a system call that changes the way in which the handler treats messages for this object. Only a message that returns to the object tagged as a RPC acknowledge will allow the object to run again, and other messages received in this interval are queued.

Similarly, as another example of an extension of the reactive model, the kernel can send and a node is interrupted by the receipt of priority messages. As one example of their use, when it is necessary to use time- rather than message-driven scheduling, the system timer is encoded as a priority message. Priority messages are also used for system functions such as object spawning, swapping, and error conditions.

Although this set of low-level primitives encourages object-oriented programming styles, it is easy to express any other message primitives in terms of these, and this may be done either at the handler or library level. The results of this formulation of the message primitives are a very fast and simple node kernel that is also compatible with all existing "cube" programs. This point is important: We expect that with a suitable set of handlers for the reactive kernel, the second generation cubes will be able to run programs written for *any* other first or second generation cube.

2.4 Concurrent Formulations

The design objectives of the programming systems described here 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, and only the host runtime system "knows about" sockets and the physical hosts of the cubes on that network. The kernel and host runtime system handle messages of different lengths by different protocols, but these hardware dependencies and protocols are completely invisible to the programmer. Other machines with different hardware communication structures are with their own version of the cosmic or reactive environment 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 message-passing 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 cosmic cube host runtime system allows exactly the same programs that run on the cosmic cubes and iPSC cubes 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, 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, or on the systolic algorithms that have been developed for regular VLSI computational arrays. 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 be deferred until the interdependence and computational demands of different processes is known. The decision can even be deferred to run time choices made at the moment at which a process is to be spawned. One of the major differences between the Cosmic Environment and Cantor is that Cantor manages the distribution of objects without requiring any specification from the programmer.

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 first than to the second generation cubes. Messages should be reasonably infrequent compared with computation, for the reason that there is an appreciable operating system overhead for each message.

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 show that even after the constraints above are met, it is nearly always possible to find a process placement that reasonably minimizes the distance a message must travel [5184:TR:85]. 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 the second generation systems are expected to be essentially indifferent to message distance.

3. Fine Grain Concurrent Programming

Program writing for future scientific programs should depend less upon manual techniques for hosting application programs onto a single exotic architecture and more upon automated techniques that are capable of exploiting concurrency from a variety of concurrent machine architectures. For these future systems, the task of the program writer is to express a computation so that all opportunities for concurrency can be exploited. The task of the compiler, runtime system, and other program writing tools is to manage the concurrent resources of the target architecture for the programmer.

In our work with multicomputers, achieving these objectives would make programming medium grain systems more convenient. These objectives are, however, essential for writing interesting programs for fine grain systems such as the Mosaic. A 16K node machine in which each node contains 16K bytes of RAM will not be programmed in assembly language except by the most intrepid hackers.

3.1 Cantor

Cantor [5232:TR:86] is a programming notation for writing message-driven programs using concurrent objects. Cantor programming systems exist on a variety of sequential and concurrent computers including the Caltech Cosmic Cube, Intel iPSC, and the Sequent. The objects of Cantor are akin to the "rock bottom" actors of actor semantics. Each object is an independent computing agent that interacts with other objects solely by message-passing. An object is made up of three parts: a set of private variables that persist between receiving messages, a list of variables that describe the expected contents of the next message to be received, and a sequence of actions that describe how the object will react to the next message that it receives. For each object generated by a Cantor program, a handler is also generated. A set of handlers is also generated when a Cantor program is compiled.

Cantor objects share the reactive property. Normally a Cantor object is at rest and remains dormant until a message arrives for the object. The response of an object to receiving a message is determined by the contents of the message, the current contents of the private variables, and a sequence of procedural statements that define the fundamental "actions" of an object. These actions include creating new objects and sending more messages. An object in Cantor must process a message in a bounded number of steps. This property is ensured by the Cantor syntax. Because Cantor objects are reactive and because objects process messages in bounded time, they are equivalent to message handlers.

The composition of a node number and message handler tag generates a unique identifier for an object. The object identifier is part of the value domain of Cantor's semantics and is called a *reference*. Message-passing is based upon the sending object possessing a reference to a destination object. After the send action is completed by the sending object, the message exhibits arbitrary delay in travelling from sender to destination. Likewise, the action of creating a new object may also exhibit arbitrary delay. While the send action requires no value to be produced, creating a new object yields a reference value for the new object. Thus the generation of the reference value is instantaneous and may be computed

on before the object is instantiated. For an object to process a message, the object must be instantiated.

3.2 Flow Analysis

The decoupling between the reference to an object and the instance of an object permits references to be speculated upon before an object is required by a program. For the interval during which the reference value to an object exists but the object has not yet been built, the reference value is called a *future*. Generation of futures prior to executing a program is called future flow and can be used to improve load balancing and to preserve locality among communicating objects. The problem of future flow during compilation is equivalent to the problem of constant propagation used in optimizing compilers. Value propagation graphs are constructed to denote the dependencies between the execution points where values become defined and the execution points where values are used. By generating a future for each of the definition vertices and propagating the reference value through the other vertices, a subset of the reference values that will be used by a program are generated as futures at compile time. These futures can be represented as vertices of a graphs that show the connectivity of objects and also the genealogy of objects.

The construction of the future graph is thwarted by data dependencies that cannot be determined without running the program, and by nondeterminancy which is introduced by the message-passing semantics. A second type of flow analysis that is not obstructed by data dependencies and message nondeterminancy is to predict how an object will be used based upon its propensity to create new objects and send more messages. From the semantics of message-driven programs, concurrency is introduced only when an object sends more messages than it receives. The propensity to send more messages is called the Send Factor (SF), and is defined as the maximum number of messages an object can send in response to processing a single message. Likewise, the New Factor (NF) is the maximum number of new objects the object can create in response to processing a message. The Send Factor and New Factor for each type of object can be calculated at compile time.

Probably the most important application of the Send Factor and New Factor is the placement of new objects at runtime. For example, when creating a new object, the new object can be determined to be capable of either:

- Generating concurrency: $SF > 1$.
- Sustaining concurrency: $SF = 1$.
- Sustaining concurrency: $SF < 1$.

If all objects for a program have a Send Factor less than or equal to one, then the program is sequential. The Send Factor and New Factor are useful for making heuristic decisions about the placement of new objects. For example, if the Send Factor and New Factor are greater than one, then the new object should be placed on a distance node to prevent excessive congestion of messages and objects within a small patch of the host multicomputer.

3.3 An Example

To illustrate Cantor as a notation for expressing concurrent programs, let us take the eight queens problem. The task is to place eight queen pieces on an 8 by 8 chessboard so

that no queen is in jeopardy of capture. The queen game piece captures any other piece that lies along the same row, column, or diagonal.

The search for the 92 solutions of the eight queens problem is massively concurrent. From the capture rule, there will be only one queen per row and column. A concurrent search can therefore be organized either by row or by column. Assuming a column by column search, a queen is placed in a row of column 1. The rows of column 2 are then searched to find a safe position for the second queen. After a safe position has been found, the rows of column 3 are searched to find a safe position for the third queen. Each subsequent column search involves checking the previous row and column pairs to be sure that the new position is safe. For the case where no safe row can be found, the partial solution is discarded.

A sequential search would involve backtracking once an impossible configuration is exposed. The backtracking step systematically removes the current emplacements of queens and then continues with a new placement. For the concurrent search, backtracking is not necessary because all solutions are equally pursued. The only action taken in response to a detected impossible configuration is to discard the configuration.

The program fragment of Fig 5 performs the concurrent search. The names `qlist`, `queen`, and `make_q` denote object definitions and not objects. An object definition is a template for building an object, thus all objects result from object definitions. A program consists of a finite number of object definitions that are then used to generate possibly an infinite number of objects. The `make_q` object definition is used to start the concurrent search by creating 8 new objects using the `queen` definition. Each of these objects will search for solutions starting with a different row number for column 1. The `queen` definition advances the search column by column. The `qlist` definition is used to maintain lists of partial solutions. The object definitions for starting the program and queueing the answers for display are not shown.

The eight queens program has been executed on sequential computers and also on concurrent computers such as the Caltech Cosmic Cube and Intel iPSC. The concurrent computers demonstrate that the program is concurrent because the time to find the 92 solutions decreases by applying more computing nodes during program execution. An alternate approach to studying the dynamics of the concurrency is to define an *abstract implementation* for the execution environment and then measure the utilization of different resources.

The abstract implementation is idealized so that every opportunity for concurrency within a Cantor program is exploited. To achieve this ideal environment, the number of computing nodes fluctuates to adapt to the the number of concurrent objects extant in the Cantor program. Message delivery between nodes is made instantaneous and the bandwidth between two nodes is infinite; thus locality is inconsequential. After a message is delivered to a node, the message is processed in a single time quantum. The time quantum is the same for all messages and objects. To simplify the analysis of the measurements, the processing of messages is assumed to be synchronized across all the nodes. Each of these synchronized steps is called a *sweep*. Computations are organized into sweeps. For each sweep, every object that has one or more messages enqueued is allowed to process one message. All of the messages sent in response to processing a message are immediately delivered and all new objects are immediately instantiated.

The execution of a program consists of a number of sweeps. For each sweep the following

data are tabulated:

- total number of messages
- total number of active objects
- total number of objects
- number of messages delivered this sweep

An active object is defined as an object that has one or more messages queued for it. The total number of active objects is called the Concurrency Index (CI). The object count and concurrency index for the eight queens program are shown in Figs 6 and 7. The printing phase of the solutions was removed because it is completely sequential. The program finds all 92 solutions in less than 300 sweeps, or roughly a solution every 3 sweeps. The cost for this performance is an immodest number of objects, peaking out at slightly over 5,000. Most of these objects, however, are purely transitory. The final object count is 830, which includes the 92 solutions requiring 9 objects per solution.

The abstract implementation represents a best case for the execution of the eight queens program. Although no proof exists, the conjecture is that any program which behaves poorly under the abstract implementation will perform poorly upon any physically realizable implementation. A variation upon the abstraction implementation is to fix the number of nodes and evaluate program performance. Figs 8 and 9 show the object count and concurrency index graphs for the eight queens program in which the number of nodes is limited to 64. Because the mapping between objects and nodes is no longer one-to-one, a placement algorithm is necessary to assign objects to nodes. The algorithm used for these graphs attempts to balance the load of objects across the node by determining the node with the smallest object count and then placing the new object on this node.

The concurrency index graph for 64 nodes shows clipping; that is, the concurrency index limits at the number of nodes. This clipping is the desired response, because it indicates that all nodes are in use. The concurrency index also shows that high levels of concurrency occur over a large interval of sweeps. By limiting the number of nodes to 64, the time to completion is increased by a factor of 6, indicating that the average level of concurrency for this program is approximately 384 active objects per sweep.

The placement algorithm used for these two graphs required that a global information be obtained before placing a new object. The graph of Fig 10 compares the performance of three placement algorithms, the previous algorithm (o), the previous algorithm except that the message count per node is used instead of the object count (m), and random placement (r). The number of sweeps for each of the strategies is plotted against the number of nodes that were used to execute the program. The horizontal dashed line denotes the sweep count for the abstract implementation.

Random placement performs remarkably well, requiring no global information and virtually no overhead. The performance of random placement can be understood by considering the two cases of many more active objects than processing nodes and many more processing nodes that active objects. For the case in which the number of active objects is much smaller than the number of nodes, the probability that two or more objects are

assigned to the same node is small. For the case in which the number of active objects is much greater than the number of nodes, then by the weak law of large numbers, the probability that a node will be assigned an active object increases with the number of assignments made.

```
qlist (row,col,next) ::
*[ case (cmd) of
   "check" : (rn,cn,caller)
             if rn = row or (cn - col) = abs (rn -row)
                then send (false) to caller
                else if next = "nil"
                        then send (true) to caller
                        else send ("check",rn,cn,caller) to next
                     fi
             fi
   "copy" : (caller)
            if next = "nil"
               then send (qlist(row,col,"nil")) to caller
               else send ("copy",self) to next
                    [ (1) send (qlist(row,col,1)) to caller ]
            fi
   "get" : (caller) send (row,col,next) to caller
 ]

queen (ql,cn,out) ::
% ql --- list of valid queen positions
% cn --- column number
*[ (rn)
  send ("check",rn,cn,self) to ql
  [ (reply) if reply
               then send ("copy",self) to ql
                    [ (nql)
                      nql := qlist(rn,cn,nql)
                      if cn = 8
                         then send ("insert",nql) to out
                         else send (1) to queen(nql, cn+1, out)
                      fi ]
            fi
   ]
   if rn < 8 then send (rn+1) to self else exit fi
]

make_q(out) ::
*[ (i) send (1) to queen(qlist(i,1,"nil"),2,out)
       if i < 8 then send (i+1) to self else exit fi ]
```

Fig 5: Concurrent Eight Queens Program

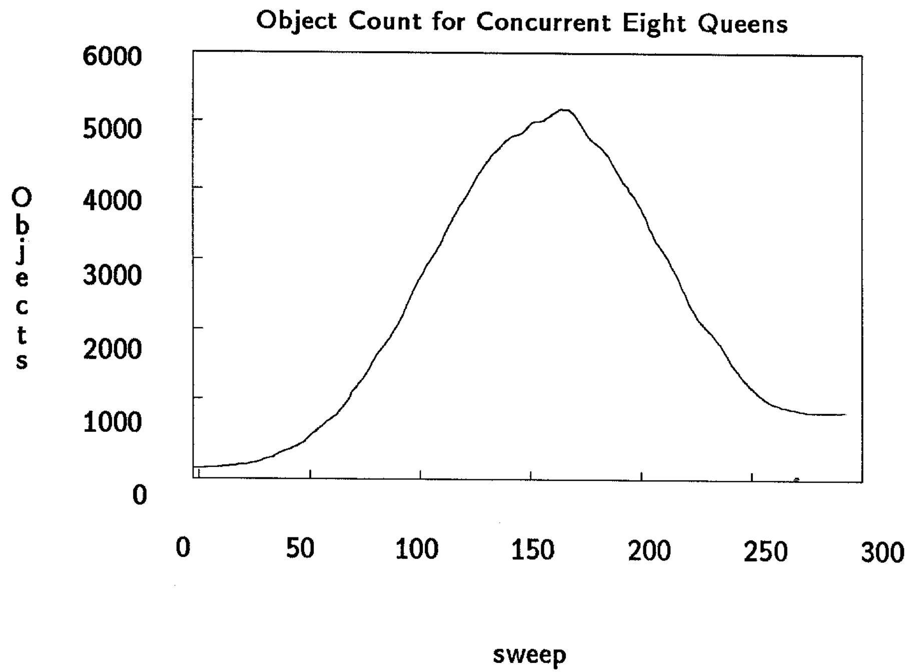

Fig 6: Object Count for Concurrent Eight Queens ($N = \infty$)

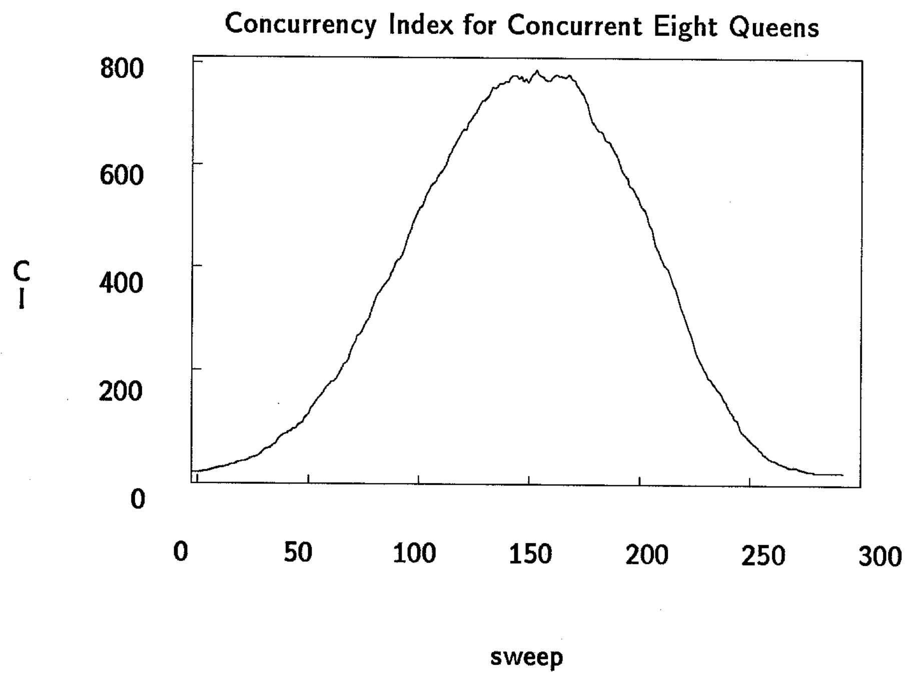

Fig 7: Concurrency Index for Concurrent Eight Queens ($N = \infty$)

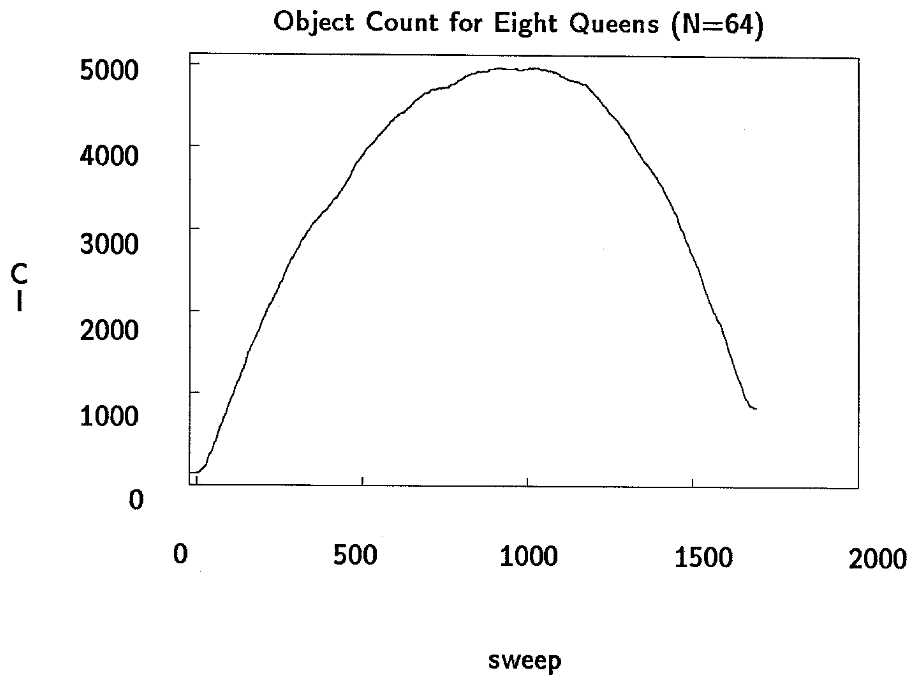

Fig 8: Object Count for Eight Queens ($N = 64$)

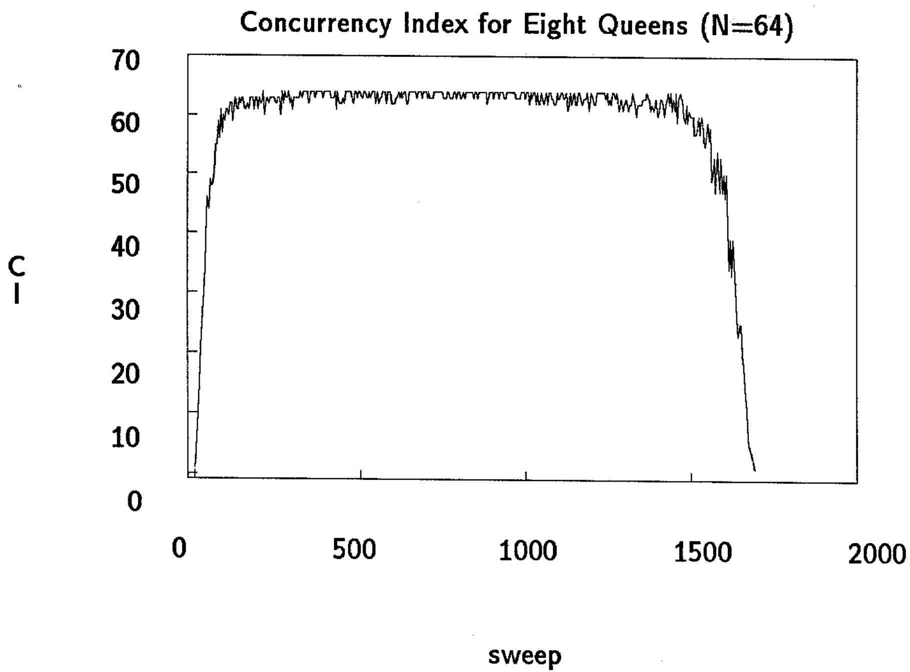

Fig 9: Concurrency Index for Eight Queens ($N = 64$)

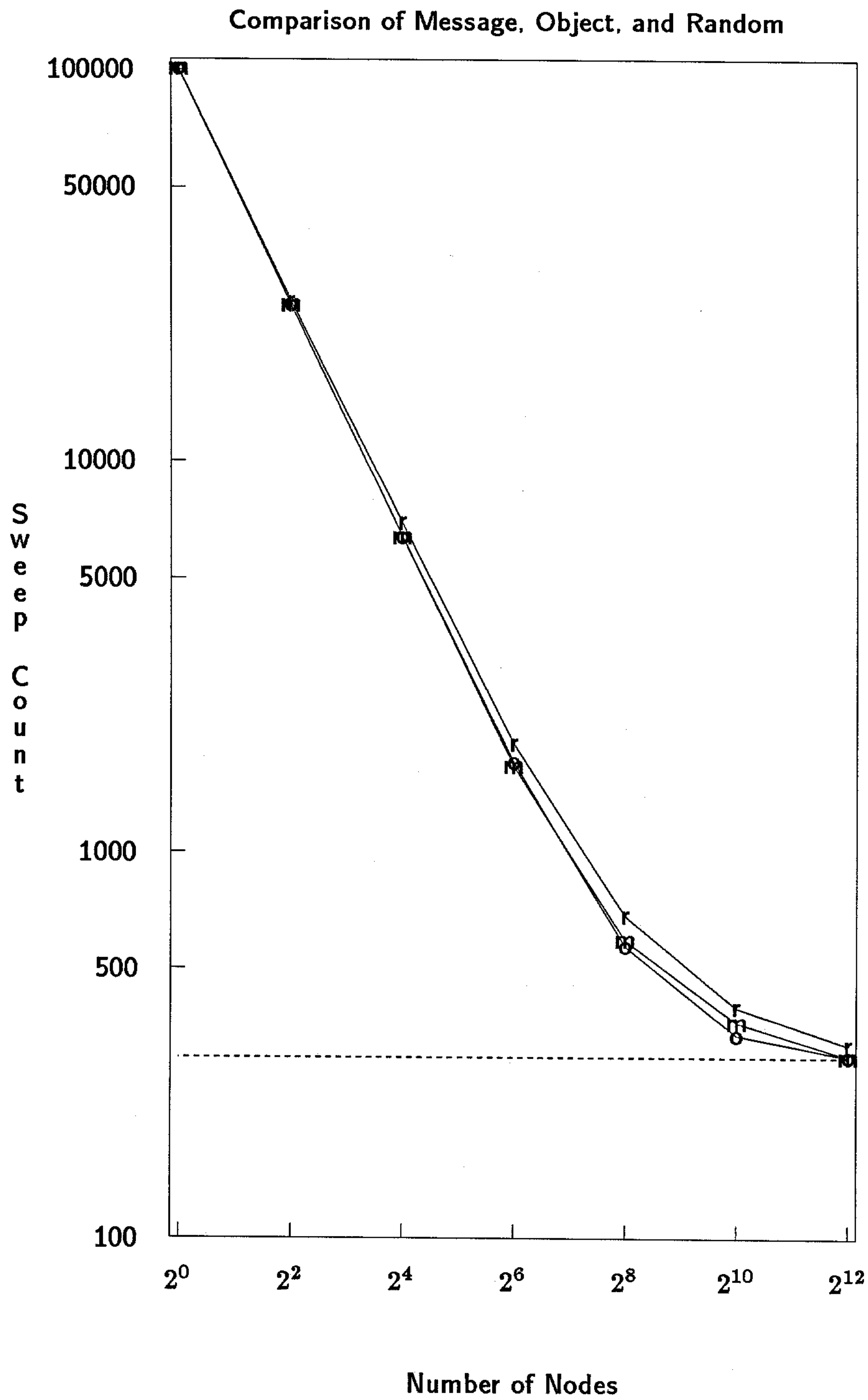

Figure 10: Speedup Comparison for Eight Queens

4. Bibliography

[Agha 86] Gul A Agha, *Actors: A Model of Concurrent Computation in Distributed Systems,* MIT Press, 1986.

[Dally 86] William J Dally, Charles L Seitz, "The Torus Routing Chip", *Distributed Computing,* vol 1, no 4, pp 187-196, Springer International, 1986.

[Dally 87] "Wire-Efficient VLSI Multiprocessor Communication Networks," *Proc of the 1987 Stanford Conference on Advanced Research in VLSI,* MIT Press, 1987.

[Hillis 86] W Daniel Hillis, *The Connection Machine,* MIT Press, 1985.

[Hoare 78] C A R Hoare, "Communicating Sequential Processes," *CACM* vol 21, no 8, pp 666-677, 1978.

[Kermani 79] P Kermani and L Kleinrock, "Virtual Cut-through: A new computer communication switching technique," *Computer Networks* 3:267-286, 1979.

[Seitz 84] Charles L Seitz, "Concurrent VLSI Architectures", *IEEE Transactions on Computers,* vol C-33, no 12, pp 1247-1265, Centennial Issue, December 1984.

[Seitz 85] Charles L Seitz, "The Cosmic Cube", *CACM,* vol 28, no 1, pp 22-33, January 1985.

[Ullman 84] J D Ullman, "Flux, sorting, and supercomputer organization for AI applications," *J of Parallel and Distributed Computing* 1 pp. 133-151, 1984.

Caltech Computer Science Technical Reports:

[5184:TR:85] Craig S Steele, "Placement of Communicating Processes on Multiprocessor Networks."

[5203:TR:85] Wen-King Su, Reese Faucette, and Chuck Seitz, "C Programmer's Guide to the Cosmic Cube."

[5209:TR:86] William J Dally, "A VLSI Architecture for Concurrent Data Structures."

[5232:TR:86] William C Athas, Charles L Seitz, "Cantor User Report."

[5242:TR:87] William C Athas, "Fine Grain Concurrent Computation."