EXPERIMENTS WITH VLSI ENSEMBLE MACHINES

Charles L Seitz

5102:TR:83
Computer Science
California Institute of Technology
Pasadena CA 91125
October 1983

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

This paper to appear in
Journal of VLSI and Computer Systems, Volume 1, Number 3
© Computer Science Press, 1984
Experiments with VLSI Ensemble Machines*

Charles L. Seitz†

Abstract — Our ongoing research at Caltech in VLSI architecture and design is entering a phase in which we are doing experiments with the design, programming, and applications of working concurrent systems of useful size and performance. The three systems to be discussed, cosmic cube, mosaic, and super mesh, are all structured as ensembles of identical elements that operate concurrently and communicate by message passing. These systems differ in many significant respects: element size, communication plan, MIMD vs SIMD, and consequently, in programming style and applications. This paper presents the common principles and issues that have guided these designs, reports preliminary results, and discusses plans for research and system building experiments that we hope and expect will yield significant results over the next several years.

1. INTRODUCTION

Considerations at many levels, from VLSI design to applications, have led the research in VLSI architectures at Caltech to investigations of a family of programmable machines that are ensembles [38] of identical, concurrently operating, small computers. The term "ensemble" is meant to suggest that the computers perform as a group, in close cooperation or harmony, similar to a musical ensemble. The computation to be performed is orchestrated by the programmer. Improvisation is possible, and is interesting to think about, but will not be discussed here. These machines do not run ordinary sequential programs.

In order that these machines be strictly scalable in VLSI implementations, the individual computers are small and communicate only by queued message passing. There is no storage or other system resource that is global or shared except for the communications. The individual computers are referred to as "nodes", as in a computer network. Messages may be sent between the nodes over channels in a regular communication plan such as a tree, mesh, shuffle, or hypercube.

In the MIMD machines, messages are queued and may be routed through intermediate nodes on the way to nodes that are not neighbors in the communication graph. Thus a multiple process message passing computational model, a process or object oriented programming style, and simple extensions of common sequential programming notations, are natural for these machines. This

*The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-72-C-0567.

†Computer Science 256-80, California Institute of Technology, Pasadena CA 91125.
comparatively unrestrictive model has proved to be quite satisfactory for the numerically demanding physical simulation and modeling applications that have been studied and implemented to date.

It is not strictly necessary to the structural or computational model that these MIMD machines be homogeneous, that is, composed of only one type of node, or that the communication plan be regular. Such machines are simply easier to design, build, program, and maintain than heterogeneous and/or irregularly connected ensembles.

The SIMD machine that we have designed and simulated is quite different from the MIMD machines. It is necessarily homogeneous. Communication between nodes is synchronized by the broadcast instruction stream, and the communication graph is restricted for physical reasons to a mesh. Its applications are highly regular computations that can be described by sequential programs that operate on array structures.

Our early investigations of algorithms and programming disciplines for machines based on these principles [7], [20], [8], [24], including extensive simulations, left many questions open that could not be addressed by simulation. The dangerous itch to give our ideas a realistic test, and design studies that indicated that such machines were quite feasible to build in a university and offered extraordinary cost/performance characteristics, led us to initiate three system building experiments in the 1980–1982 period. The following paragraphs briefly summarize the structure and status of these machines.

Cosmic cube, an experimental MIMD homogeneous machine with 64 nodes connected in a Boolean 6-cube, is in operation and regular use. The complexity of the present cosmic node, which includes 128K bytes of read-write storage, 8K bytes of read-only storage, 16-bit instruction processor, floating point coprocessor, and 8 communication channels, is representative of a node that could be integrated onto a single chip with about 1 micron feature size technology. Thus we think of the cosmic cube as a system that could be built with single chip nodes in about 5 years from now. In anticipation of this advanced process technology, we have built a “hardware simulation” of the nodes from commercially available integrated circuit components in order to experiment with the applications, algorithms, programming, and engineering of such systems. This 64 node machine has sufficient performance, about 10 times a VAX11/780, on problems for which it is suited that it has attracted many Caltech scientists to become users, which is indeed part of our experiment. Also, the nodes have sufficient storage for interesting experiments with distributed operating systems. More advanced versions of this style of machine are now being designed.

It was a premise of the mosaic experiment that the node would be a single chip with as much capability and performance as we could achieve with acceptable yield in standard MOSIS [11], [26] fabrication. These nodes are thus much smaller, but are otherwise identical in structure to the cosmic node. Each node includes only 4K bytes of storage with a 16-bit processor and 4 communication channels. Mosaic nodes can be connected in a variety of fixed degree (4) communication plans, including the tree, mesh, chordal ring, shuffle, and cube connected cycle. The single-chip version of this node has not yet been fabricated, pending investigations of its yield from sample yields of its principal parts. However, small mosaic systems using a version of the mosaic node with processor and communication channels on a MOSIS chip together with 8K bytes of fast commercial storage chips have been assembled and run test programs. We expect to use the single chip node to assemble a 1024 node mosaic shuffle or mesh during 1985. This 1024 node system will be capable of executing several billion instructions per second, or about 50 million (32-bit mantissa) floating point operations per second, which is quite remarkable for a system of only 1024 chips packaged into a cubic foot
or so. Even with the meager storage capacity of each node, the application span of mosaic includes many highly concurrent problems in signal and image processing, and finite element and matrix computations.

A fine grain SIMD array machine dubbed super mesh has been designed and simulated, and its very small repeated node is being laid out as a custom chip. Multiple nodes per chip are readily achieved. Although this "array" machine is aimed at very high floating point performance and efficiency, it should not be confused with a "peripheral array processor." Rather, its array structure makes it similar to Illiac IV [5], STARAN [2], the ICL DAP [18], or to a systolic [22] or computational array in which (unlike the CMU programmable systolic chip [14]) the nodes operate from a single, or at most several, instruction streams. This project is aimed at an extreme point of cost/performance versus generality, and is more an experiment in VLSI design, synchronization over large volumes, and optimal arithmetic algorithms, than in architecture and programming. Although we can report only simulation results for this machine, we expect it to exhibit extraordinary cost/performance characteristics, but for a limited class of highly regular computational problems.

One must state immediately the precaution that not even the larger grain versions of these experimental machines are claimed to be efficient for all types of computations performed on conventional mainframes. It is not yet known, and is certainly one of the objectives of our research to discover, how far we can extend these machines toward irregular applications.

2. PRINCIPLES and DESIGN ISSUES

These three system building experiments have a certain consistency in principles of design, operation, and programming, outlined in the subsections below.

2.1 Cost/performance of VLSI Architectures

It is the cube law scaling of the switching energy with feature size in constant electric field scaling of MOS technologies [12], [32] that is the payoff, promise, and driving force for the advances in these technologies. Scaling feature size and voltage in MOS systems reduces the power and area per device quadratically with the scaling factor, and thus reduces the cost by approximately this factor. The power per unit area is held constant in this scaling. The transit time and circuit delays are reduced linearly with the scaling factor. The product of the power per device and device delay, which is equivalent to the energy for each switching event, thus scales by a cube law. The switching energy or power delay product is a fundamental figure of merit of a technology that has a direct relation to the cost/performance of systems implemented in that technology.

We have tried to apply the very stringent requirement on a "VLSI architecture" that the cost/performance of a system scale as closely as possible with the switching energy. This scaling is difficult to achieve in practice in large, tightly-coupled systems due to the dominant contribution of communication in area, power, and delay. At the risk of trying to demonstrate the counterpart of the famous proof that bumblebees cannot fly, analyses of scaling large, tightly-coupled systems, including the cost in area and energy of long distance communication on and between chips, the communication delay in drivers, and the delay in the diffusive propagation of signals in scaled wires [36], [33], [34], [38], [4], indicate that there are diminishing returns in performance for area. In these cases the scaling of transit time in the MOS switches and small MOS circuits is not reflected in system performance because the delay of the interconnect does not scale.
At the same time, we insist that the architectures be open-ended in performance, but see few opportunities for performance improvements in single process systems. The instruction execution rates of computers ranging from single chips to supercomputers are surprising tightly bunched, predominantly between 1 and 100 million instructions per second (Mips), even though the range of costs is spread across more than 5 orders of magnitude. (Granted, a Cray-1 instruction is not only executed 100 times faster than an 8-bit micro's, but also on much longer words, but the difference in word size relates to the parallelism rather than to instruction rate.) Other evidence for this performance limit, or diminishing returns in performance for cost, is that the rate of improvement in execution bandwidth of processors on the high-speed end of the spectrum has been decreasing [9]. In documenting this trend, Butbee has conjectured from a mathematical fit to the history of computer performance that these high-speed single process machine designs are within about an order of magnitude of an asymptote.

For ensemble machines these requirements of (1) cost/performance scaling with technology advances, and (2) open-ended performance, imply that systems be scalable to very large numbers of nodes without sacrificing cost/performance. These requirements are satisfied directly by the limited size and loose coupling of the nodes. However, this family of machines is clearly more specialized than the usual sequential computer, but to a degree that can only be answered by doing the experiment.

### 2.2 Performance Leverage in Concurrency

In order to see what kind of performance can be achieved in cost effective designs with systems based on the high complexity but modest performance MOS technology, one inevitably looks to systems that support multiple concurrent processes. Major advances in performance, say factors of 1000 over today's fastest computers, evidently require that the degree of concurrency be very large indeed. Such goals require more than the evolution of the single process computer to the multiprocessor, but to machine organizations based on very different algorithms, principles of operation, and programming.

Today's fastest computers, and even many microprocessors, exploit concurrentities that can be discovered in the process of interpreting an instruction stream. Techniques such as instruction prefetch, overlap or pipelining of arithmetic operations, and caching to exploit locality in storage references, allow typically and in aggregate a 10-fold concurrency. These machine designs have been tuned to exploit the typical concurrency that can be found in programs [21] by optimization, vectorization, and decomposition of the sequential program to fit concurrent processing elements.

Ensemble machines can achieve strikingly higher degrees of concurrency from concurrent formulations of the whole computation. We do not perform automatic decomposition and vectorization of existing sequential programs, except to the extent that our compilers and processors can exploit the concurrentities in the innermost iterations and expression evaluations in the code that defines individual processes. Rather, we want to turn the size of a large computation against itself. Many massive computations — eg matrix computations, differential equation solution, many-body problems, Fourier transforms, simulation of multiple degree of freedom systems, image and signal processing — have natural concurrent formulations whose degree of concurrency is large and grows with the problem size.

Since we do not know how to automate the discovery of efficient concurrent formulations for such problems, we leave to the programmer the task of inventing and expressing a concurrent and spatially distributed algorithm for a whole computational problem. It will be interesting to learn
how far we can go with this approach, and whether the programming task can be simplified by starting, for example, from a mathematical [19] rather than procedural specification of the problem.

2.3 Message passing

The model of computation, and the target for the problem formulation, for the ensemble machines discussed here is a collection of sequential processes that communicate by message passing. A static process structure, or snapshot of a dynamic process structure, can be represented as a graph of process vertices connected by arcs (directed edges) that represent reference. (When process A possesses reference to process B, a message from A to B is possible.) The hardware communication structure of these machines can be represented similarly, and the mapping of a computation onto an ensemble machine is an embedding of the computation graph onto the machine graph. The embedding reveals both the locality of communication achieved and load balancing properties of the mapping.

In SIMD array machines message passing is a highly synchronized parallel exchange of data. However complex the computation in a node may be, these machines readily accommodate only problems with quite regular formulations that fit their hard-wired communication structure.

Unlike SIMD array machines, which are effectively limited to problems with these regular formulations, the MIMD message passing ensembles can deal also with computations with highly irregular, and even dynamically changing, communication topologies. Here the message passing model seen by the programmer may be substantially abstracted from the hardware structure, in two ways. First, messages are routed in the communication subsystem, so that formulations need not fit any particular hard-wired communication structure. Second, synchronization between message sending and receiving is potentially quite loose, due to queueing of messages in the communication subsystem.

This message passing approach organizes multiple computers into a structure similar to a computer network. The queue is a better fit to VLSI engines than the switch required for shared storage multiprocessors, for many of the same reasons that store and forward packet communication is used in digital communication networks instead of circuit switching [20]. As an engineering technique, queued message passing has the advantages over circuit switching (1) that it scales well with system size, because the cycle time of the node need not increase with the size of the system, and (2) that it allows messages to be queued and routed in transit in a way that increases throughput even though it increases latency.

Some proposed concurrent systems produce the appearance of sharing to the programmer, while the underlying mechanisms work by message passing. In our experiments we have made the choice of treating message passing as an explicit primitive, with the result that the communication is expressed directly in a problem formulation or program.

2.4 Applications and Performance Modelling

Caltech scientists in high energy physics, astrophysics, quantum chemistry, fluid mechanics, structural mechanics, seismology, and computer science, are developing concurrent application programs to run on these ensemble machines. The first research paper on scientific results, a quantum field theory computation on a cosmic cube prototype [5], has already been published, and other applications are developing rapidly. Several of us in computer science are involved in this research both as the system builders and also through interests in concurrent computation and applications to VLSI analysis tools.
The general characteristic of all of these computations is that the sequential part is very small. Following a model by Ware [41], "speedup", \( S \), can be defined as:

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

Ware's formula for speedup:

\[
S = \frac{1}{(1 - \alpha) + \alpha/N},
\]

where "\( \alpha \)" is (loosely defined as) the fraction of the computation that can be done concurrently, follows directly from the definition, e.g., for unit time on one node, with \( N \) nodes the concurrent part of the computation requires time \( \alpha/N \), but the part that resists being done concurrently still requires time \( (1 - \alpha) \). This formula reveals the same limitation represented by "Amdahl's argument" [1], that speeding up a concurrent computation, even in the limit where \( N \) becomes arbitrarily large, is limited to \( 1/(1 - \alpha) \), the reciprocal of the fraction of the computation that must be done sequentially.

We are usually less interested in the asymptotic speedup than in the region in which speedup is almost equal to \( N \), since here an ensemble machine is operating with high efficiency, that is, with very few of the nodes being idle. Such efficiency is achieved with concurrent formulations in which the sequential fraction \( (1 - \alpha) \) is sufficiently small, say .0001, that its reciprocal, 10000, is a number substantially larger than the number of nodes applied to the problem. In fact, the sequential fraction of a computation turns out not to be a serious limitation for a large class of important problems. Often \( \alpha = 1 \), except for the program loading and data I/O phases of a computation.

Once this condition of \( \alpha \approx 1 \) is satisfied, the speedup obtained by using \( N \) nodes concurrently is limited by considerations of (1) idle time due to poor load balancing, (2) waiting time due to the large communication latencies that we sanction in the physical design of these machines, and (3) processor time dedicated to processing and forwarding messages. This third consideration can be effectively eliminated by architectural improvements in the nodes, but the other considerations are fundamental to the formulation of a computation and to the assumptions we make about the physical design of these machines.

Analyses of problem formulations or programs provide simple and exact predictions for the performance of highly regular physics computations [15], in which load balancing is easily assured. Analyses of computations that are less regular and more sensitive to latency necessarily use much more complex models of the computation in order to account for variations in process running times, imperfect embeddings and load balancing, and latency. (However, these analyses still show \( S \approx N \) for small \( N \) and an asymptotic \( S \) for large \( N \).) For example, for MOS-VLSI circuit simulation the model evaluation (linearization) phase of the computation, which is typically over 80% of the running time on a sequential computer, shows \( \alpha = 1 \) and a \( N \)-fold speedup on a concurrent machine. However, solving the sparse matrix equations and communicating the intermediate results between parts of the computation in a circuit of arbitrary topology presents a very complex analysis problem. Amongst the many unknowns in experimenting with circuit simulation, as a paradigm of less regular computations that can be performed on ensembles, are the interaction between communication cost and load balancing in the mapping of processes to nodes. Processes here correspond with rows in a sparse but "clumped" admittance matrix. Although the "clumping" can be exploited in this mapping to localize communication, it may also concentrate many of the longer iterations occurring during a signal transient into a single node, thus creating a "dynamic" load imbalance in the computation.
2.5 Grain Size

The attraction of ensembles with smaller nodes, "fine grain" systems, is their cost/performance characteristics. Since the instruction rate of a smaller node can be expected to be comparable to, or with shorter signal paths, even faster than that of the larger node, the fine grain node provides similar performance at less cost. At the same system cost, the smaller node is the basis for an ensemble with more nodes, more performance, similar total storage, and less storage per node. When this tradeoff is applied to comparisons of "fine grain" machines, such as mosaic or super mesh, and "medium grain" machines such as the cosmic cube, one expects [38] that the fine grain machines will have a smaller application span. for two reasons.

First, if the cost/performance advantage of small nodes is to be reflected in the cost/performance of the computation, one cannot use so many nodes that the speedup is in its asymptotic region. Machines with many nodes are efficient only on computations of commensurate size and concurrency. Properly, this problem-dependent limit on the number of nodes that can usefully be kept busy applies as well to coarse grain machines with many nodes, or to comparisons between single process computers and ensembles, and relates to grain size only for comparisons of systems of constant cost.

Second, the small storage of a fine grain node limits the size of a process, either the program complexity (MIMD) or the data (MIMD and SIMD), or both. Progressively finer grain machines violate more and more seriously the old rule of thumb that instruction processors require about a Mbyte of storage per Mips execution rate. A partial reason that we get away with less data storage than one might expect from the node instruction rate, even on extremely protracted computations, is that the data being processed is not all stored in the node, but is also communicated. The working set can grow with the machine size. However, the size and complexity of the concurrent processes—the grain size of the formulation—is generally bounded by the node size.

Optimal cost/performance for a given computation is achieved with a machine with the exact number of nodes required, each exactly large and fast enough for the node process, a common assumption in research papers but a clearly unrealistic expectation for other than very regular and specialized problems. Thus even with a family of machines of different grain size, which is what we seek as a suitable experimental environment, formulations that are of substantially finer grain than that of the target ensemble are the interesting case for attempting to extend the generality of this family of machines. Techniques for mapping processes onto the ensemble [31], many processes per node, or many nodes per process, either statically or in execution, assume a central role in extending highly concurrent computations beyond regular and specialized problems.

3. COSMIC CUBE EXPERIMENT

3.1 Chronology

The cosmic cube design is based in largest part on extensive program modeling and simulations by Charles R (Dick) Lang [24] carried out between 1980 and 1982. In particular, it was from this work that the communication plan of a Boolean n-cube, the bit rates of the communication channels between nodes, and the organization of the operating system primitives were chosen. In earlier work at Caltech [29] and from interactions with Carl Hewitt [17] at MIT, including an August 1981 meeting organized by Hewitt, we recognized the similarity of our ideas to those presented in a paper by Herbert Sullivan [40], and so refer to cosmic cube by the term coined by Sullivan, a homogeneous
machine, a machine "of uniform structure."

The logical design of the cosmic node was done in the fall of 1981 by Erik DeBenedictis. Most of the choices made in this design are fairly easy to explain. First of all, a Boolean n-cube communication plan was used because this network was shown by simulation to provide very good message flow properties in irregular computations. It also contains all meshes of lower dimension, which is useful for regular mesh-connected problems, and the connections required to map FFT algorithms directly. The Boolean n-cube can be viewed recursively; that is, the n-cube that is used to connect \(2^n = N\) nodes is assembled from two \((n - 1)\)-cubes, with corresponding nodes connected by an additional channel. This property simplifies the packaging of machines of varying size. It also explains some of the excellent message flow performance properties of the Boolean n-cube on irregular problems. The number of channels connecting the pairs of subcubes is proportional to the number of nodes, and hence on average to the amount of message traffic they can generate.

The Boolean n-cube, which is often referred to in the parallel processing literature as a direct binary n-cube, is a logarithmic network, like the shuffle [25], or various indirect n-cubes such as the banyan [16] or Omega [25] network, or the "flip" network used in STARAN [3], in that the longest path in a machine of \(N\) nodes is \(\log_2(N) = n\). The shuffle, however, does not contain submeshes, and must be rewired when expanded. The Boolean n-cube is isomorphic to indirect n-cube networks of \((N/2)^n\) 2x2 switches that form an \(N \times N\) switch in \(n\) layers, except that the \(N\) processing nodes connect to the \(N\) paths and route messages at each of the \(n\) layers of the indirect network. Of course the n-cube graph is not of fixed degree. The present cosmic cube nodes, with 6 communication channels, cannot be assembled into an n-cube larger than 64 nodes.

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

The Intel 8086 was chosen as the instruction processor because, at the time, it was the only single chip instruction processor available with a floating point coprocessor, the Intel 8087. This floating point performance was necessary for applications that our colleagues in high energy physics at Caltech, under the direction of physics professor Geoffrey Fox, wished to attempt. The storage size of 128K bytes was the subject of a great deal of internal discussion of "balance" in the design. It was argued that the cost incurred in doubling the storage size would better be spent on more nodes. In fact, this choice is clearly very dependent on target applications and programming style.

The dynamic RAM includes parity checking but not error correction. Each node also includes 8K bytes of read-only storage for initialization, bootstrapping loader, dynamic RAM refresh, and diagnostic testing programs.

A prototype 4 node (2-cube) system on wirewrap boards was assembled and tested in the spring of 1982, and this system has been running concurrent programs and has been in use for software development since July 1982. The homogeneous structure of these machines is exploited to accelerate the software development by use of a small hardware prototype that is essentially similar to scaled up machines. This same tactic is being used in the mosaic project.

With the design thus proved, we had printed circuit boards designed, and went through the
other packaging logistics of assembling a machine of useful size. The present cosmic cube grew from
an 8 node to a 64 node machine over the summer 1983, and this 6-cube machine has been in regular
operation since October 1983. The task of building hardware to provide more cycles for the user
group has been passed to a group at Caltech JPL, with the intention of building approximately 200
more nodes of a program compatible derivative design over the next year. The “mark II” cosmic
nodes have 256K bytes of storage and 9 communication channels, and will be configured as a 7-cube
and two 5-cubes.

Figure 1 is a photograph of the 6-cube in operation. Larger machines would have nodes arrayed
in 2 or 3 dimensions, but for such a small machine, and large ratio of PCB width to spacing, a one-
dimensional projection of the 6-cube is satisfactory. The separate units on the shelf above the long
6-cube box are (left to right) the power supply and two “intermediate hosts” (IHs). These dedicated
IH machines run network and operating system software, and in some cases are used to support
small auxiliary computations in support of the computation running in the cube. The volume of the
system is 6 cubic feet, and power consumption is 700 watts. We also operate a 3-cube machine in
support of software development, since the 6-cube is not readily shared.

3.2 Early Application Programs and Benchmarks

Programs written by physicists Eugene Brooks and Steve Otto have allowed direct benchmarking
of various size cosmic cubes against a VAX11/780. These programs consistently place a 64
node cosmic cube at about 10 times faster than the VAX11/780 on this restricted class of problems.
These programs execute up to 3 million 32-bit floating point operations per second. It is estimated
that the cosmic cube is somewhat less than 1/10th as fast as a Cray-1 for these same problems,
and would be less than 1/20th a Cray-1 on problems requiring 64-bit floating point computations.
Because of the enormous storage bandwidth across 64 nodes, the instruction rate for computations
that do not use floating point compare more favorably with conventional machines. Thus we believe
that for these specialized problems the cosmic cube, which would have about the same leveled out
manufacturing cost as a VAX11/780 processor with the same 8 Mbytes of primary storage, achieves
a cost/performance advantage over conventional mainframes of about 10.

As an example of its applications at Caltech, a lattice computation programmed by Steve Otto,
about 63K bytes of program and 64K bytes of data per node, has now run for more than 2,500 hours
on the 6-cube. This program is a Monte Carlo simulation on a 12x12x12x16 lattice, an investigation of
the predictions of quantum chromodynamics, a theory that explains the substructure of elementary
particles such as protons in terms of quarks and the glue field that holds them bound. After it had
run for about 400 hours, this program surpassed the results produced by the most extensive prior
computations on this problem, produced in 40 hours on a Cray-1. Otto has shown for the first time
in a single computation both the short range Coulombic force and constant long range force [15].
The communication overhead in this computation, the fraction of the time the nodes spend in the
send and receive routines, is only 2.5%.

Amongst the most interesting programs currently in development is a MOS-VLSI circuit simulator
formulated and written by computer science graduate student Sven Mattisson. This program
promises very good performance, and is a vehicle for developing techniques for less regular
computations on ensemble machines. This program uses a nodal admittance formulation for the electrical
network. The admittance matrix is sparse, and “clumped”, but because electrical networks have
arbitrary topology, does not have the crystalline regularity of the physics computations. The problem
is mapped onto the cube by partitioning the admittance matrix in rows, with numerous row processes per node. The linear equation solution phase of the computation, a Jacobi iteration, involves considerable communication, but the linearization (called model evaluation in circuit simulators) that requires about 80% of the execution time on sequential computers is completely uncoupled. Integration and output in computing transient solutions are very small components of the whole computation.

8.2 Programming Environments

Programs for the cosmic cube are developed on conventional computers, written, compiled, simulated, instrumented, and debugged, and then downloaded into the cube through a connection that is managed by the intermediate host. There are 5 layers of software environments in the system:

1. The lowest level is what we will call the machine intrinsic environment. This environment includes the instruction set of the node processor, its I/O communication with channels, and a small initialization and bootstrap loader program stored together with some diagnostic programs in read-only storage in each processor. This level is important principally for machine initialization, loading, and diagnostics. It is a relatively more interesting system than one might at first imagine, since a part of the initialization involves each of the identical nodes discovering by sending messages its position in whatever size cube was specified in the startup packet sent from the intermediate host. This initialization involves n! messages that also check the function of all of the communication channels to be used.

2. The second level environment, referred to as crystalline, is characterized by programs written in notations such as C, in which there is a single process per node, and in which messages are sent by direct I/O operations to a specified channel. This level has proved to be satisfactory for producing efficient application programs for computations that are so regular that they do not require message routing. The programmer uses a pair of system utilities, WBOX and RBOX, to accomplish message sending and receiving. These routines provide the necessary synchronization, and are also instrumented to collect statistics and to provide error checking on the channels. Although it is perfectly possible for the processes in the nodes to differ, in programs such as Otto's lattice computation they do not. Processes in different nodes take different execution paths depending on the data.

3. The third level system supports a limited multiple process environment with a small operating system called the "cosmic kernel" running in each node. Message routing and queueing is accomplished by the kernel, and messages are sent by reference to a process ID in which is embedded the physical node and process name within that node. This system is limited in the sense that it does not support relocation of processes between nodes during execution. This kernel includes process spawning and destruction, scheduling, storage allocation, (host) I/O communication, and monitoring and debugging facilities. In this environment the definition and execution of the multiple process formulation of the problem is independent of its mapping onto physical nodes: that mapping influences only the efficiency. This environment, when it becomes fully developed, is that intended to be seen by users for both regular and irregular application programs.

8.4 Future Homogeneous Machines

Although the present 64 node machine has adequate performance for programming experiments and for accomplishing many useful computations, we are looking ahead to much more powerful systems. Taking 2 λ as the minimum feature size [32] on the silicon, each node for the present cosmic cube requires about 140 Mλ² in aggregate complexity across 78 chips, most of which are not
high-complexity parts. This node could be expected to scale to a single chip within about 5 years. A technology with 1 micron feature size ($\lambda = 0.5$ micron) on chips 6 mm on a side would provide this 140 M$\lambda^2$ complexity. While such a chip is a fairly small increment in complexity over a million bit storage chip, if storage chips and logic continue to be made with different fabrication processes, a single node might better be packaged as two or more chips in a hybrid package. Based on the benchmarks of the cosmic cube, and the expectation that single chip or hybrid nodes would exhibit higher performance, a system of 1024 such chips or hybrids would provide for many computationally demanding problems performance well beyond today’s fastest computers. But such thinking is for the long term.

In the short term, as long as we are experimenting with this class of machines by using largely off the shelf parts, most obvious opportunities for improvement (deficiencies) in the cosmic cube design relate to (1) the low level of integration in the communication sections, and (2) context switching between communication and computation.

In order to address the first deficiency, a custom communication chip called the “fifo buffered transceiver” was designed and tested by Charles Ng [35] for use in a future homogeneous machine. It provides fifo buffering of 4 256-bit packets each direction in each channel, compared with 2 64-bit packets each direction in cosmic cube, and does full duplex communication at 2mbits/sec using only 2 wires compared with 12 in cosmic cube. The layout is small enough to allow 4 channels per chip.

The second deficiency led to the idea that future descendants of the cosmic cube should use two processors per node, one to manage communication and the other for user processes. This natural partitioning of the tasks of a node not only reduces the problems of context switching in a single processor node, it also allows the operating system kernel to run almost entirely in the communication processor, a considerable advantage since many of the operating system functions can be accomplished concurrently with user processes.

There are three limits to scaling a cosmic cube style of machine, one associated with its connection plan, one associated with reliability, and one associated with the size of the node element.

Boolean n-cubes of progressively higher dimension will be required for larger numbers of nodes. The number of communication channels per node exhibits a relatively benign growth as the $\log_2 N$. If the node must be physically larger to accommodate more pins, the structure is always locally wirable, even in two physical dimensions. However, the length of the channels for a machine wired in $D$ physical dimensions doubles each $D$ abstract dimensions, with the result for $D$ limited to two or three (for physical cubes) that the total volume of wire increases more rapidly than the volume required to accommodate the nodes. With present wiring technology and in 2 physical dimensions (printed circuit boards and “motherboard” backplanes), it is easy to wire a 14-cube (16K nodes), difficult to wire a 16-cube (64K nodes), and impractical to wire an 18-cube (256K nodes). Should machines larger than 64K nodes ever be built, they would require an advanced interconnect technology or (more likely) a different connection plan.

Reliability limitations appear to be equally distant, at least if soft error control is included. The mean time between failure (MTBF) observed for the cosmic cube node, exclusive of soft errors in its dynamic RAM primary storage, is well in excess of 100,000 hours. This node MTBF would result in a system MTBF of about a week for a 1K node machine, or an MTBF of an hour or so for a 64K node machine. Even an hour MTBF might be tolerable on a machine whose performance would be about 100 times a Cray-1. Machines of this size are not likely to be built from today’s technology, so the
reliability situation for future machines is much brighter. The homogeneity of these systems makes faults very easy to diagnose and repair. While it is possible to imagine approaches to fault-tolerant operation, our examinations of this problem suggest that it is not as trivial as it may first appear.

One other scaling consideration is whether the node complexity might reach a point of diminishing returns for performance if the node complexity is scaled up, or equivalently, with a node similar in complexity to the cosmic node in a scaled microcircuit technology. Why shouldn’t a node have a Mbyte or more of primary storage? Nodes of this size might well be appropriate for many problems, but since storage can be expected to dominate the silicon area of a fully integrated node, one would expect the cost to increase almost in proportion to the storage size, but with no large change in the instruction rate. Thus one is led to the uncomfortable conclusion that the evolution of systems in the direction of larger nodes is a losing proposition on the basis of cost/performance. Thus we are led to the mosaic class of machines, an experiment with a much smaller node element.

4. MOSAIC EXPERIMENT

Ensembles of single chip mosaic nodes form what might be regarded as a fine grain homogeneous machine. As mentioned in the introduction, the mosaic node is structurally identical to the cosmic node, with the groundrule that it fit on one MOSIS chip. The full design [30], with processor, storage, and channels, fits on a chip 6 mm on a side in 3 micron nMOS technology, with 140,000 transistors. The complexity of this chip, 4000 λ by 4000 λ, or 16 Mλ², and our use of a 3, rather than 1-transistor dynamic RAM cell, allows us only 4K bytes of storage.

While a sample of two is too small to draw any real conclusions, mosaic is at least an example supporting the supposition that larger nodes can be expected to outperform larger nodes in instruction rate. The mosaic node is so much faster than the cosmic node that its subroutine 33-bit mantissa floating point multiply or add is faster than the 24 bit mantissa floating point in the 8087 floating point coprocessor. Although mosaic includes an integer multiply, it is not otherwise specialized for floating point computations, and its instruction rate for more mundane operations, such as MOVE between a 16-bit register and storage, is about 2.5 Mips. The mosaic processor makes 10 storage references per microsecond, typically 2 for refresh, 5 for instructions and data, and 3 for null references or prefetches that are later discarded.

With only 1/32nd the storage size of the cosmic node, the programming style for mosaic is necessarily “very careful”. In particular, these nodes are too small to accommodate much in the way of an operating system. The node is too small for some of the computations currently performed on cosmic cube, unless one breaks up large processes to run (sequentially, if necessary) on groups of mosaic nodes. However, programs written for mosaic show it to be a nearly optimal grain size for many simpler problems, including the matrix and graph problems investigated by Sally Browning [7], many-body and finite element computations, signal and image processing, graphics, and circuit simulation.

Mosaic nodes can be connected in many regular communication plans while still a fixed degree (≤ 4) graph. Mosaic was originally intended for a tree connection. However, it is also well suited for cube connected cycle, shuffle, 2-dimensional mesh, and chordal ring connections. A new “memory mapped” communication section is being designed for mosaic that will allow us to incorporate more ports, and also makes the ports self-timed rather than synchronous.
Although small mosaic systems have been assembled for testing and for programming experiments, the system building project is still in its early stages. The design of the mosaic node is described fully in [30]; the following descriptions will accordingly be brief.

4.1 Cost/performance of Mosaic Hardware

We are planning to complete a 1024 node mosaic system, configured as a shuffle or mesh, during 1985. This system will likely be packaged as 16 PCBs with 64 nodes per board, an assembly essentially similar in regularity, power distribution requirements, and timing, to the dynamic RAM primary storage used in a large mainframe. Mosaic will also use an intermediate host that is a simple variant on that used for cosmic cube. Add-in storage for mainframes, which mosaic board assemblies resemble, currently sells for less than $20 per 64K RAM chip, packaged and powered. The present mosaic node is of comparable complexity and function to a 64K RAM. Even taking a selling price of $50 per chip, to be conservative, a mosaic system with 1024 nodes, just 1024 chips plus clock drivers and interface, might sell commercially for about $50K.

Analysis of a number of sample regular problems indicate that a mosaic system of this size will be capable of sustained floating point speeds in the same 20-80 million floating point operations per second range that is typical of the Cray-1. The Cray-1 is superior to most mainframes in the measure of floating point operations per second per dollar cost, and the cost is $5-10M.

Of course there are numerous points on which such a comparison between mosaic and existing supercomputers is unfair, if not invalid, including (1) differences in generality and programmability, (2) differences in primary storage size, (3) software support, and (4) existence. However the ratio of cost/performance in the order of 100 in comparing mosaic with a conventional mainframe or supercomputer would certainly make these machines attractive for the regular applications for which they are most efficient. Similar arguments apply to many other fine grain concurrent systems.

4.2 Applications and Programming Environment

Early work in programming systems and applications for a programmable tree machine by Sally Browning [7], [8] used a derivative of Hoare's Communicating Sequential Processes (CSP) programming notation for representing numerical and graph algorithms. Peggy Li later developed fast loading algorithms [28] and a prototype operating system [27]. In addition, there have been many algorithms published for computational or systolic arrays that map very easily onto a suitably connected mosaic. This work well represents one style of programming mosaic systems, somewhat analogous to the crystalline programming style used in the cosmic cube. However, we are also studying ways of programming mosaic systems derived from experience in the cosmic cube project, in particular, systems supporting message routing and queuing with multiple processes per node.

Since we lack storage space for much of a run time system in the nodes, the host tools for producing mosaic programs are expected to be more sophisticated than those for cosmic cube. There are many ways in which advanced compilers can assist the programmer. For example, in static assignment of processes to nodes, one can combine processes in a way that relieves the run time system from scheduling. Also, a computation that is specified in processes too large to fit into a single mosaic node can be subdivided to be run on a group of nodes. Whatever conccurrencies could be extracted automatically from the process code might be exploited in execution, but even if there were none, the communication necessary in the group to pass control for the single process could be derived by such a compiler.
5. SUPER MESH EXPERIMENT

Guided largely by seeing how our physics colleagues have been programming certain regular applications on the cosmic cube, we started in 1982 investigating the design of an SIMD machine of quite small grain size, about one quarter of size of the mosaic node. Since this project is in its earliest stages, we will only outline some of our reasoning about its design.

It seemed very wasteful in cosmic cube that the same program was often loaded into all of the nodes of this MIMD machine, even though the execution temporarily follows different branches in different machines. We rationalized this duplication of code as necessary in face of the communication cost of broadcast control. However, if an efficient broadcast mechanism could be devised, an SIMD machine of quite small node size could achieve extraordinary cost/performance characteristics. One might think of this machine as a VLSI incarnation of the Illiac IV.

The approach taken to broadcast control, as well as to provide optimal performance per area in the arithmetic, was to exploit serialism. We use parallel arithmetic in conventional computers largely to balance arithmetic speed against storage cycles, not because it is efficient. Serial addition in systems with very well balanced combinational delays is nearly as fast as parallel addition, and uses only $1/n$th the logic for $n$-bit words. Carry-save serial multiplication has a similar advantage over a multiplier array or add-and-shift multiplier. There is no correspondingly efficient bit serial algorithm known for division (in a binary number representation), but it can be performed through multiplications by Newton's method. Slower serial nodes also simplify the instruction issuing computer and the broadcast of control, indeed allow and encourage this broadcast to occur serially. In somewhat different terms, a way to get the most performance per chip, in a given or scaling technology, is to minimize the area-time product (cost/performance) in the elementary operations, as is done with serial arithmetic, and then to rely on the concurrency of these operations to scale the performance.

The VLSI engine that evolved from this reasoning, an SIMD array machine dubbed super mesh, resembles a computational or systolic array, somewhat as H T Kung and C L Leiserson described them in [22], except that instead of the algorithm being built into the node, the node operates on the contents of its registers in response to the instruction sequence broadcast through the ensemble. Given the existing body of algorithms developed for computational arrays, and the programmability and efficiency of this implementation, we are confident that super-mesh will be an interesting and useful VLSI system.

A strawman design of the node includes 64 registers for 64-bit floating point numbers, and a carry-save technique for the mantissa multiplication. Four nodes fit on one rather large chip at 3 micron feature size. Because of the uniformly short paths through combinational logic, such a machine is (in circuit simulation of critical paths) able to operate at a bit rate of 20 MHz, which for a 56-bit mantissa translates into about 3.3 microseconds (65 clock cycles) per floating point multiply. Thus we expect this chip to achieve over one million 64-bit floating point operations per second, in addition to its storage and communication functions. The design allows operands to be fetched in the register storage and communication to take place concurrently with floating point operations.

This strawman machine turns out to have too much arithmetic performance and too few registers to be a good fit to real problems. It reveals very well a problem inherent in the computational array model [23] that for 2-dimensional problems the array is so fast, $O(n)$ for many problems that the $O(n^2)$ time complexity of serial loading typically exceeds that of computing. A realistic
design will have to include much more storage relative to the arithmetic capability, in effect to share the arithmetic capability amongst more operands, and to permit a broader class of applications. A preferred way to increase the storage is not to increase the number of internal registers, but to provide another (hierarchic) storage level of dynamic RAM chips. Details of the control of this secondary store are currently being worked out; suffice it to say that it fits the array model very easily.

We have been tediously careful in the physical design of this machine to be sure that it can be scaled to arbitrary size, as long as a pair of neighboring nodes can be included in a single equipotential region [37]. Indeed, this project is in part an experiment in physical synchronization techniques. It is the physical structure of the clock and microinstruction distribution that requires that the node interconnection plan be a 1-, 2-, or 3-dimensional mesh, which can be mapped onto a manifold such as a torus.

The physical design of super mesh starts with clock distribution, which is accomplished by a 1-, 2-, or 3-dimensional tessellation of locally coupled self-timed circuits, each containing a Muller C-element and a reference delay. The clock skew between neighbors in the network is bounded to a small fraction of the clock period and is arbitrarily large across a diameter of an arbitrarily large machine. The skew between neighbors is not unbounded as Fisher and Kung correctly assert would occur [13] if one distributes a central clock, an approach that would in any case violate the principle that one cannot communicate a time interval across a spatially unbounded structure, even in one dimension. Instead, each clock source contains its own reference interval, and the tessellation network behaves as an enormous set of coupled ring oscillators, with a period determined by the maximum reference delay, and with small local differences in phase that are consistently positive with respect to a specific “corner”. The local skew in communication between neighbors is accommodated by the \( t_{21} \) time [37] in two phase clocking being larger than the skew.

Broadcast of the microinstruction is serial, starting from the corner mentioned above, but is resynchronized on each step. Thus the whole array is pipelined, with nodes more distant from the corner being progressively more out of phase. This approach is conceptually identical, but carried out on a bit-by-bit basis, to the transformation of broadcast into pipelining that has been described [10] for other computational array structures. This pipelining is completely invisible to the programmer.

The design and simulation of the super node has been accomplished by Wen-King Su [30], hence the name Super mesh.

The programming system visualized for super mesh is entirely different from those of the MIMD machines. Super mesh operates by microinstruction sequences being transmitted to the array, and can be programmed by specifying the node computation in an ordinary sequential programming notation. Actually, to deal with conditionals and boundary conditions, there are several alternative instruction streams transmitted through the array in parallel. The programming of the control computer, and the ability in serial broadcast to include several instruction streams, leaves open the possibility of compiling the process specification into microprogram sequences for one node or groups of several nodes. The use of physical submeshes in the same resource in super mesh as for mosaics for a problem requiring more storage than is provided in a single physical node, and allows also some concurrency in the computation performed per process, a second level of concurrent decomposition that could be extracted in compilation.
6. SUMMARY

With the usual caveat that performance is achieved at the expense of specialization, we are optimistic about our experiments with the programming and engineering of this homogeneous ensemble style of machines, for two reasons:

First of all, these concurrent machines have not proved to be difficult to program by the methods we have described, in which the programmer formulates a computation in terms of concurrent processes. We share the skepticism of others, that one cannot depend on the programmer to write "ultracurrent" programs with many different processes with complex interactions. The applications that we represent as reasonable are those in which the computation is demanding because of the number of processes, and in which the degree of concurrency grows with the problem size. Many important and currently intractable computational problems fit this description.

Second, experience with the design and engineering of these machines indicates that they provide for these demanding problems excellent cost/performance characteristics. These machines can exploit the scaling of feature size in VLSI technology, and be scaled to very large numbers of nodes, to provide present supercomputer performance in very small packages, and capabilities well beyond present supercomputers in systems the size and cost of present mainframes.

These experiments may appear to be motivated in largest part by the "greed for speed," the simple desire to use VLSI to achieve very high performance computation. Performance has indeed been an objective that has made these system building experiments excellent vehicles for developing engineering and programming techniques for concurrent VLSI systems, and has provided a concrete way in which we can measure our success or failure. However, we do not see the engineering and programming as being either fundamentally difficult or an end in itself. Rather, we see these system building experiments as exposing the opportunities and thus stimulating research on the more fundamental problems in concurrent computation.

7. ACKNOWLEDGEMENTS

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0507.

Projects such as those described have become feasible in a university setting largely because of the MOS implementation system (MOSIS) that is maintained and constantly improved by people at USC/ISI, principally under ARPA support. Special thanks are due the whole MOSIS crew, but particularly to Danny Cohen, George Lewicki, Lee Richardson, Paul Losleben, Kathie Fry, Joel Goldberg, Ron Ayres, and Vance Tyree, for the many "special cases" on which we have received special help.

A contribution by Intel Corporation of chips for the cosmic cube is gratefully acknowledged.

These projects involve extensive collaborations at Caltech. I have tried to indicate in the text the work done by numerous Caltech students over the past several years. The intellectual contributions of several Caltech faculty: Alain Martin, Randy Bryant, Lennart Johnsson (now at Yale), and Martin Rem in computer science, and Geoffrey Fox in high energy physics, should be mentioned as well.

The author very much appreciates H T Kung's suggestion of a "position paper" on these Caltech
experiments, the extensive and constructive review of this paper by editors Danny Cohen and H.T. Kung, and reserves for himself the blame for the large residue of arm-waving statements that even these editors could not entirely expunge.

6. REFERENCES


[7] Sally A. Browning, Hierarchically Organized Machines, Section 8.4 in [32].


[37] Charles L. Seitz, System Timing, Chapter 7 in [32].


Figure 1