# Architecture and Applications of the Connection Machine

**Lewis W. Tucker and George G. Robertson**
**Thinking Machines Corp.**

**Massively parallel computer architectures have come of age. We describe here the architecture, evolution, and applications of the Connection Machine system.**

As the use of computers affects increasingly broader segments of the world economy, many of the problems to which people apply computers grow continually larger and more complex. Demands for faster and larger computer systems increase steadily. Fortunately, the technology base for the last twenty years has continued to improve at a steady rate—increasing in capacity and speed while decreasing in cost for performance. However, the demands outpace the technology. This raises the question, can we make a quantum leap in performance while the rate of technology improvement remains relatively constant?

Computer architects have followed two general approaches in response to this question. The first uses exotic technology in a fairly conventional serial computer architecture. This approach suffers from manufacturing and maintenance problems and high costs. The second approach exploits the parallelism inherent in many problems. The parallel approach seems to offer the best long-term strategy because, as the problems grow, more and more opportunities arise to exploit the parallelism inherent in the data itself.

Where do we find the inherent parallelism and how do we exploit it? Most computer programs consist of a control sequence (the instructions) and a collection of data elements. Large programs have tens of thousands of instructions operating on tens of thousands or even millions of data elements. We can find opportunities for parallelism in both the control sequence and in the collection of data elements.

In the control sequence, we can identify threads of control that could operate independently, thus on different processors. This approach, known as *control parallelism*, is used for programming most multiprocessor computers. The primary problems with this approach are the difficulty of identifying and synchronizing these independent threads of control.

Alternatively, we can take advantage of the large number of independent data elements by assigning one processor to each data element and performing all operations on the data in parallel. This approach, known as *data parallelism*,[1] works best for large amounts of data. For many applications, it proves the most natural programming approach, leading to significant decreases in execution time as well as simplified programming.

Massively parallel architectures containing tens of thousands or even millions of processing elements support this "data-parallel" programming model. Early examples of this kind of architecture are ICL's Distributed Array Processor (DAP),[2] Goodyear's Massively Parallel Processor (MPP),[3] Columbia University's Non-Von,[4] and others.[5] Each of these machines has some elements of the desired architecture, but lacks others. For example, the MPP has 16K (K = 1,024) processing elements arranged in a two-dimensional grid, but interprocessor communication is supported only between neighboring processors.

The Connection Machine provides 64K physical processing elements, millions of virtual processing elements with its *virtual processor* mechanism, and general-purpose, reconfigurable communications networks. The Connection Machine

0018-9162/88/0800-0026$01.00 ©1988 IEEE

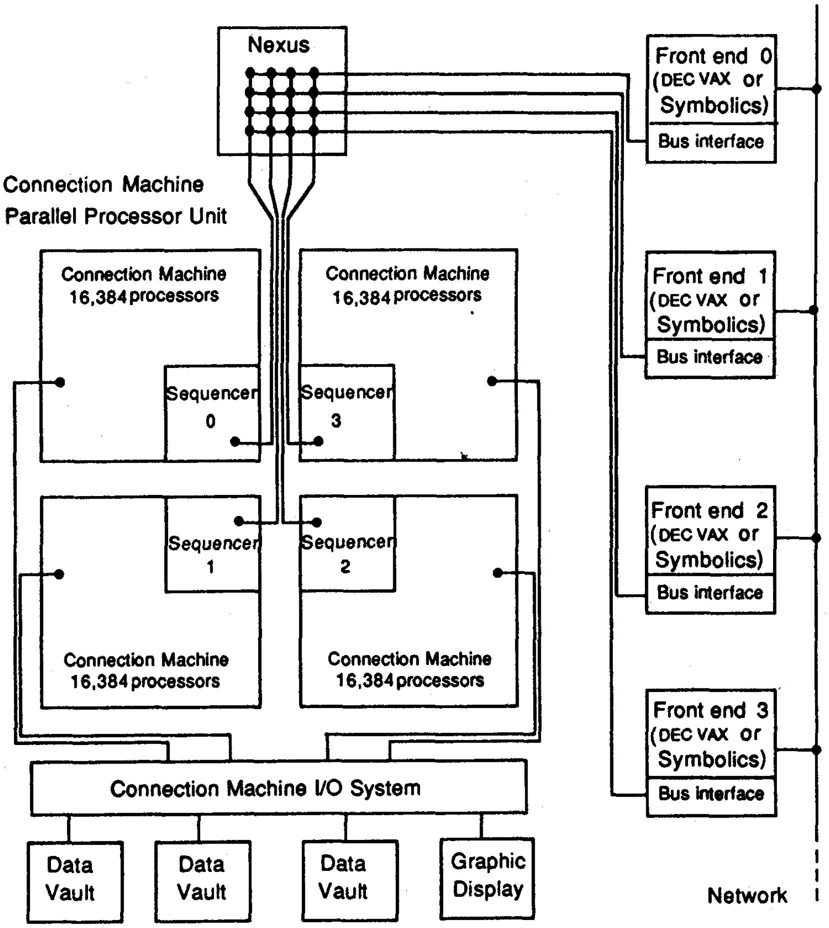

Nexus
Connection Machine Parallel Processor Unit
Connection Machine 16,384 processors
Sequencer 0
Connection Machine 16,384 processors
Sequencer 3
Sequencer 1
Connection Machine 16,384 processors
Sequencer 2
Connection Machine 16,384 processors
Connection Machine I/O System
Data Vault
Data Vault
Data Vault
Graphic Display
Front end 0 (DEC VAX or Symbolics)
Bus interface
Front end 1 (DEC VAX or Symbolics)
Bus interface
Front end 2 (DEC VAX or Symbolics)
Bus interface
Front end 3 (DEC VAX or Symbolics)
Bus interface
Network

**Figure 1. Connection Machine system organization.**

encompasses a fully integrated architecture designed for data-parallel computing.

## Architecture of the Connection Machine

The Connection Machine is a data-parallel computing system with integrated hardware and software. Figure 1 shows the hardware elements of the system. One to four front-end computer systems (right side of Figure 1) provide the development and execution environments for system software. They connect through the nexus (a 4 × 4 cross-point switch) to from one to four sequencers. Each sequencer controls up to 16,384 individual processors executing parallel operations. A high-performance, data-parallel I/O system (bottom of Figure 1) connects processors to peripheral mass storage (the DataVault) and graphic display devices.

System software is based upon the operating system or environment of the front-end computer, with minimal visible software extensions. Users can program using familiar languages and programming constructs, with all development tools provided by the front end. Programs have normal sequential control flow and do not need new synchronization structures. Thus, users can easily develop programs that exploit the power of the Connection Machine hardware.

At the heart of the Connection Machine system lies the parallel-processing unit consisting of thousands of processors (up to 64K), each with thousands of bits of memory (four kilobits on the CM-1 and 64 kilobits on the CM-2). As well as processing the data stored in memory, these processors when logically interconnected can exchange information. All operations happen in parallel on all processors. Thus, the Connection Machine hardware directly supports the data-parallel programming model.

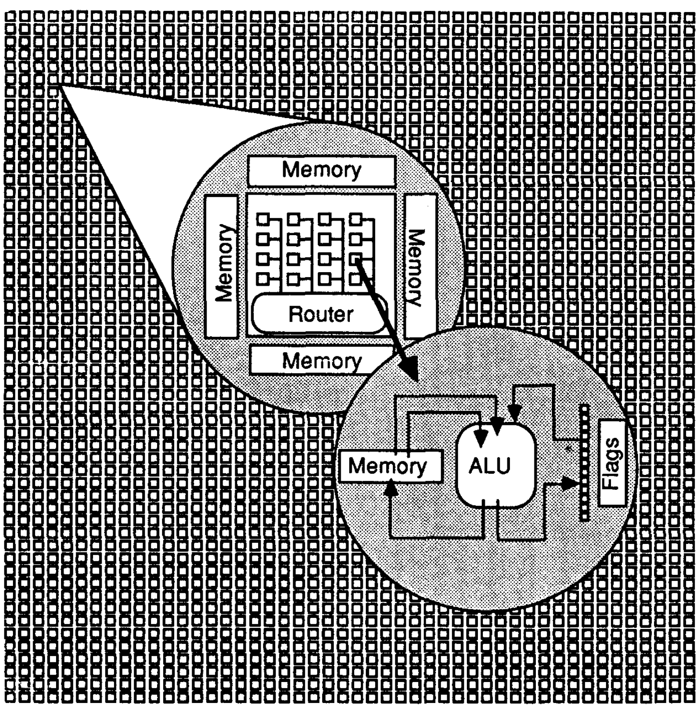


**Figure 2. CM-1 data processors.**

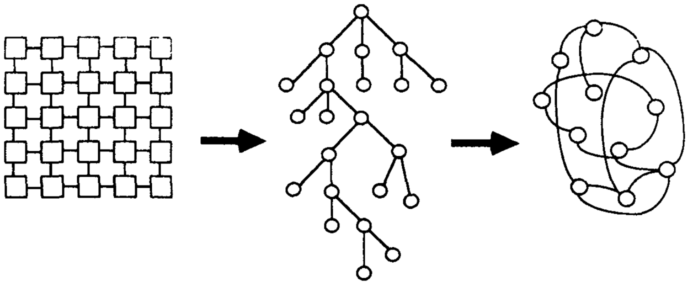


**Figure 3. Complex problems change topology.**

**CM-1: First implementation of CM concept.** Hillis originally conceived the Connection Machine architecture while at MIT and described it in his thesis.[6] The design of the Connection Machine began in 1980 at the MIT AI Laboratory, where the basic architectural design and prototype custom integrated circuits were developed. It became clear that private enterprise would have to get involved to actually build the machine, so Thinking Machines was founded in 1983.

The Connection Machine Model CM-1 was designed at Thinking Machines during 1983 and the first half of 1984. By the end of 1984, with funding from the Defense Advanced Research Projects Agency, Thinking Machines had built the first 16K-processor CM-1 prototype. A demonstration of its capabilities took place in May 1985, and by November the company had constructed and successfully demonstrated a full 64K-processor machine. Thinking Machines commercially introduced the machine in April 1986. The first machines went to MIT and Perkin-Elmer in the summer of 1986.

As illustrated in Figure 1, the CM-1 contains the following system components:

- up to 64K data processors,
- an interprocessor communications network,
- one to four sequencers, and
- one to four front-end computer interfaces.

Although part of the original Connection Machine design, the I/O system was not implemented until the introduction of the CM-2.

*CM-1 data processors and memory.* The CM-1 parallel-processing unit contains from 16K to 64K data processors. As shown in Figure 2, each data processor contains

- an arithmetic-logic unit and associated latches,
- four kilobits of bit-addressable memory,
- eight one-bit flag registers,
- a router interface, and
- a two-dimensional-grid interface.

The data processors are implemented using two chip types. A proprietary custom chip contains the ALU, flag bits, and communications interface for 16 data processors. Memory consists of commercial static RAM chips, with parity protection. A fully configured parallel-processing unit contains 64K data processors, consisting of 4,096 processor chips and 32 megabytes of RAM.

A CM-1 ALU consists of a three-input, two-output logic element and associated latches and memory interface (see Figure 2). The basic conceptual ALU cycle first reads two data bits from memory and one data bit from a flag. The logic element then computes two result bits from the three input bits. Finally, one of the two results is stored in memory and the other result, in a flag. The entire operation is conditional on the value of a context flag; if the flag is zero, then the results for that data processor are not stored.

The logic element can compute any two Boolean functions on three inputs. This simple ALU suffices to carry out all the operations of a virtual-machine instruction set. Arithmetic is carried out in a bit-serial fashion, requiring 0.75 microsecond per bit plus instruction decoding and overhead. Hence, a 32-bit Add takes about 24 microseconds. With 64K processors computing in parallel, this yields an aggregate rate of 2,000 million instructions per second (that is, two billion 32-bit Adds per second).

The CM processing element is a reduced-instruction-set-computer processor. Each ALU cycle breaks down into subcycles. On each cycle, data processors execute one low-level instruction (called a *nanoinstruction*) issued by the sequencer, while the memories can perform one read or write operation. The basic ALU cycle for a two-operand integer Add consists of three nanoinstructions: LoadA to read memory operand A, LoadB to read memory operand B, and Store to store the result of the ALU operation. Other nanoinstructions direct the router and NEWS (north-east-west-south) grid, and perform diagnostic functions.

*CM-1 communications.* Algorithm designers typically use data structuring techniques to express important relationships between data elements. For example, an image-understanding system usually employs a two-dimensional grid to represent the individual pixels of the image. At a later stage in the processing, however, a tree data structure or relational graph might represent more abstract relationships such as those between objects and their parts (see Figure 3).

On a serial machine with sufficient random access memory, pointers to memory elements are used to implement complex data structures. In a data-parallel architecture, however, individual data elements are assigned to individual processors and interprocessor communication expresses the relationships between the elements of very large data structures.

The CM-1 was designed with flexible interprocessor communication in mind and supports several distinct communication mechanisms:

- *Broadcast communications* allow immediate data to be broadcast from the front-end computer or the sequencer to all data processors at once.

**The CM-1 was designed with flexible interprocessor communication in mind.**

- *Global OR* is a logical OR of the ALU carry output from all data processors, which makes it possible to quickly discover unusual or termination conditions.
- *Hypercube communication* forms the basis for the router and numerous parallel primitives supported by the virtual-machine model. The topology of the network consists of a Boolean *n*-cube. For a fully configured CM-1, the network is a 12-cube connecting 4,096 processor chips (that is, each 16-processor chip lies at the vertex of a 12-cube). An example of a parallel primitive implemented with the Hypercube is Sort, which runs in logarithmic time; sorting 64K 32-bit keys takes about 30 milliseconds.
- The *router* directly implements general pointer following with switched message packets containing processor addresses (the pointers) and data. The router controller, implemented in the CM processor chips, uses the Hypercube for data transmission. It provides heavily overlapped, pipelined message switching with routing decisions, buffering, and combining of messages directed to the same address, all implemented in hardware.
- The *NEWS grid* is a two-dimensional Cartesian grid that provides a direct way to perform nearest-neighbor communication. Since all processors communicate in the same direction (north, east, west, or south), addresses are implicit and no collisions occur, making NEWS communication much faster (by about a factor of six) than router communication for simple regular message patterns.

*CM-1 sequencer, nexus, and front-end interface.* The CM-1 sequencer—a specially designed microcomputer used to implement the CM virtual machine—is implemented as an Advanced Micro Devices 2901/2910 bit-sliced machine with 16K 96-bit words of microcode storage. A Connection Machine contains from one to four sequencers. The sequencer's input is a stream of high-level, virtual-machine instructions and arguments, transmitted on a synchronous 32-bit parallel data path from the nexus. The sequencer outputs a stream of nanoinstructions that controls the timing and operation of the CM data processors and memory.

The CM-1 nexus—a 4 × 4 cross-point switch—connects from one to four front-end computers to from one to four sequencers. The connections to front-end computers are via high-speed, 32-bit, parallel, asynchronous data paths, while the connections to sequencers are synchronous. The nexus provides a partitioning mechanism so that the CM can be configured as up to four partitions under front-end control. This allows isolation of parts of the machine for different users or purposes (such as diagnosis and repair of a failure in one partition while other partitions continue to run). When more than one sequencer is connected to the same front-end through the nexus, they are synchronized by a common clock generated by the nexus.

The front-end bus interface supports a 32-bit, parallel, asynchronous data path between the front-end computer and the nexus. The FEBI is the only part of the CM-1 that lies outside of the main cabinet. It resides as a board in the system bus of the front end (any DEC VAX containing a VAXBI I/O bus and running Ultrix, or any Symbolics 3600 series Lisp machine).

**CM virtual-machine model.** The CM virtual-machine parallel instruction set, called Paris, presents the user with an abstract machine architecture very much like the physical Connection Machine hardware architecture, but with two important extensions: a much richer instruction set and a virtual-processor abstraction.

*Paris.* Paris provides a rich set of parallel primitives ranging from simple arithmetic and logical operations to high-level APL-like reduction (parallel prefix) operations,[7] sorting, and communications operations. The interface to Paris between the front end and the rest of the Connection Machine reduces to a simple stream of operation codes and arguments. The argu-

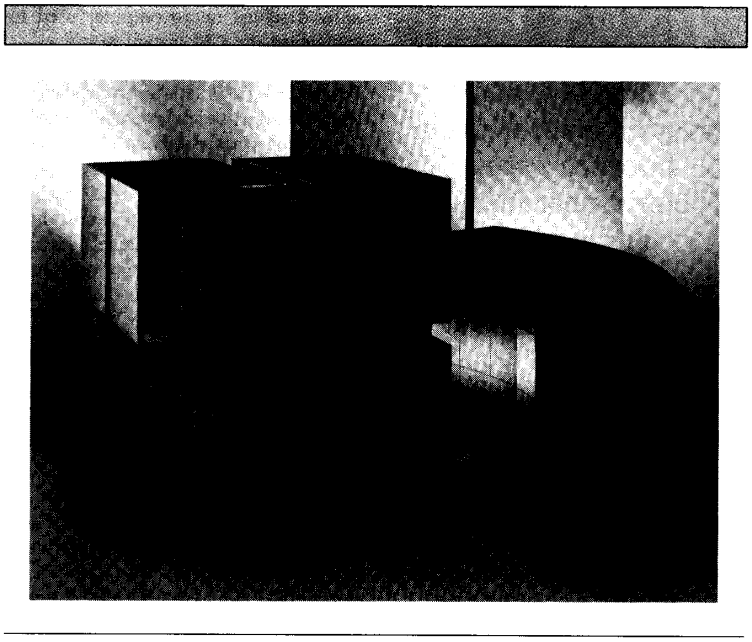


**Figure 4. Connection Machine Model CM-2 and DataVault.**

ments usually describe fields to operate on, in the form of a start address and bit length. Arguments can also be immediate data, broadcast to all data processors.

Most of Paris is implemented in firmware and runs on the sequencers, where the opcode/argument stream is parsed and expanded to the appropriate sequence of nanoinstructions for the data processors. Since Paris defines the virtual-machine instruction set, we use the same name for the assembly language of the Connection Machine.

*Virtual processors.* Data-parallel applications often call for many more individual processors than are physically available on a given machine. Connection Machine software provides for this through its virtual-processor mechanism, supported at the Paris level and transparent to the user. When we initialize the Connection Machine system, the number of virtual processors required by the application is specified. If this number exceeds the number of available physical processors, the local memory of each processor splits into as many regions as necessary, with the processors automatically time-sliced among the regions.

For example, if an application needed to process a million pieces of data, it would request $V = 2^{20}$ virtual processors. Assume the available hardware to have $P = 2^{16}$ physical processors, each with $M = 2^{16}$ bits of memory (the size for CM-2 memory; $M = 2^{12}$ bits of memory for the CM-1). Then each physical processor would support $V/P = 16$ virtual processors.

This ratio $V/P$, usually denoted $N$, is called the *virtual-processor ratio,* or *VP-ratio.* In this example, each virtual processor would have $M/N = 2^{12}$ bits of memory and would appear to execute code at about $1/N = 1/16$ the speed of a physical processor. In fact, virtual processors often exceed this execution rate, since instruction decoding by the sequencer can be amortized over the number of virtual processors.

**CM software environment.** The Connection Machine system software uses existing programming languages and environments as much as possible. Languages are based on well-known standards. Minimal extensions support data-parallel constructs so that users need not learn a new programming style. The Connection Machine front-end operating system (either Unix or Lisp) remains largely unchanged.

Fortran on the Connection Machine system uses the array extensions in the draft Fortran 8x standard (proposed by American National Standards Institute Technical Committee X3J3) to express data-parallel operations. The remainder of the language is the standard Fortran 77. No extension is specific to the Connection Machine; the Fortran 8x array extensions map naturally onto the underlying data-parallel hardware.

The *Lisp and CM-Lisp languages are data-parallel dialects of Common Lisp (a version of Lisp currently being standardized by ANSI Technical Committee X3J13). *Lisp gives programmers fine control over the CM hardware while maintaining the flexibility of Lisp. CM-Lisp is a higher-level language that adds small syntactic changes to the language interface and creates a powerful data-parallel programming language.

The C* language is a data-parallel extension of the C programming language (as described in the draft C standard proposed by ANSI Technical Committee X3J11). C* programs can be read and written like serial C programs. The extensions are unobtrusive and easy to learn.

The assembly language of the Connection Machine, Paris, is the target language of the high-level-language compilers. Paris logically extends the instruction set of the front end and masks the physical implementation of the CM processing unit.

**Evolution of the CM-2.** Experience gained during the first year of in-house use of the CM-1 led to the initiation of a project to build an improved version of the machine: the Connection Machine Model CM-2.

The design team established several goals for the CM-2: increasing memory capacity, performance, and overall reliability while maintaining or improving ease of manufacturing. To continue to support the CM-1 customer base, the designers wanted the CM-2 to be program compatible with the previous machine. Finally, the designers wanted the CM-2 to incorporate a high-speed I/O system for peripheral data storage and display devices.

To satisfy these goals, CM-2 kept the basic architecture of the CM-1 (see Figure 1). To increase performance, the CM-2 incorporates a redesigned sequencer and CM processor chip and an optional floating-point accelerator. A four-fold increase in microcode storage in the sequencer allowed for improvements in performance and functionality in the virtual-machine implementation. A sixteen-fold increase in memory capacity brought total memory capacity up to 512 megabytes. Enhanced reliability resulted from adding error correction to the memory system, and diagnostic capability was improved by increasing the number of data paths with error detection (parity). A redesigned NEWS grid increased functionality and ease of manufacture. CM-2 implemented an I/O system to support a massively parallel disk system (called the DataVault) and a high-speed color graphics system.

The CM processor chip underwent redesign in late 1985 and early 1986. The first prototype of the CM-2 was working by the end of 1986. The company commercially introduced the CM-2 (see Figure 4) in April 1987, and delivered about a dozen machines to customers in the fall of 1987. The first DataVault was delivered at the end of 1987.

*CM-2 data processors and memory.* The CM-2 data processor strongly resembles the CM-1 data processor. The major differences are

- 64 kilobits of bit-addressable memory instead of four kilobits,
- four one-bit flag registers instead of eight,
- an optional floating-point accelerator,
- a generalized NEWS-grid interface to support $n$-dimensional grids,
- an I/O interface, and
- increased error-detection circuitry.

The CM-2 data processors are implemented using four chip types. A proprietary custom chip contains the ALU, flag bits, router interface, NEWS-grid interface, and I/O interface for 16 data processors, and part of the Hypercube network controller. The memory consists of commercial dynamic RAM chips, with single-bit error correction and double-bit error detection. The floating-point accelerator consists of a custom floating-point interface chip and a floating-point execution chip; one of each is required for every 32 data processors. A fully configured 64K-processor system contains 4,096 processor chips, 2,048 floating-point interface chips, 2,048 floating-point execution chips, and half a gigabyte of RAM.

*CM-2 floating-point accelerator.* In addition to the bit-serial data processors described above, the CM-2 parallel-processing unit has an optional floating-point accelerator closely integrated with the processing unit. This accelerator has two options: single precision or double precision. Both options support IEEE standard floating-point formats and operations, and increase the rate of floating-point calculations by a factor of more than 20. Taking advantage of this speed increase requires no change in user software.

**The DataVault combines high reliability with fast transfer rates for large blocks of data.**

The hardware associated with each of these options consists of two special-purpose very-large-scale-integration chips: a memory-interface unit and a floating-point execution unit for each pair of CM-2 processor chips. Because the floating-point units access memory in an orthogonal manner to the CM-2 processors, the memory-interface unit transposes 32-bit words before passing them to the floating-point unit.

Firmware that drives the floating-point accelerator stages the data through the memory-interface unit to the floating-point execution unit. In general, it takes $5N$ stages to implement a floating-point operation, for a virtual-processor ratio of $N$. However, the firmware is pipelined so as to require only $3N+2$ stages instead of $5N$ stages.

*CM-2 communications.* The CM-2 communications are basically the same as those of the CM-1 with two exceptions. First, we redesigned the router to improve reliability, diagnostic capability, and performance. For example, we enhanced its performance by providing hardware for en route combining of messages directed to the same destination. The combining operations supported include Sum, Logical OR, Overwrite, Max, and Min.

The second major difference lies in the nature of grid communications. We completely redesigned grid communications for the CM-2. We wanted to increase flexibility and functionality while simplifying the overall system architecture and increasing manufacturing ease and reliability. We accomplished this by replacing the two-dimensional NEWS grid with a more general $n$-dimensional grid implemented on top of the Hypercube (by grey-encoding addresses). We enhanced flexibility and functionality by supporting high-level-language concepts with $n$-dimensional-grid nearest-neighbor communication. Thus, programmers can employ one-dimensional to sixteen-dimensional nearest-neighbor grid communication according to the requirements of the task. Manufacturing ease and reliability are enhanced because we removed the separate set of cables for the NEWS grid.

*CM-2 I/O structure.* The Connection Machine I/O structure moves data into or out of the parallel-processing unit at aggregate peak rates as high as 320 megabytes per second using eight I/O controllers. All transfers are parity-checked on a byte-by-byte basis.

A Connection Machine I/O bus runs from each I/O controller to the devices it controls. This bus is 80 bits wide (64 data bits, eight parity bits, and eight control bits). The I/O controller multiplexes and demultiplexes between 256-bit processor chunks and 64-bit I/O-bus chunks. The controller also acts as arbitrator, allocating bus access to the various devices on the bus.

*DataVault.* Since standard peripheral devices do not operate at the speeds that the CM system itself can sustain, we had to design a mass-storage system capable of operating at very high speed. The DataVault combines high reliability with fast transfer rates for large blocks of data. It holds five gigabytes of data, expandable to ten gigabytes, and transfers data at a rate of 40 megabytes per second. Eight DataVaults, operating in parallel, offer a combined data transfer rate of 320 megabytes per second and hold up to 80 gigabytes of data.

The design philosophy followed for the Connection Machine architecture served for the DataVault as well: we used standard technology in a parallel configuration. Each DataVault unit stores data spread across an array of 39 individual disk drives. Each 64-bit data chunk received from the Connection Machine I/O bus is split into two 32-bit words. After verifying parity, the DataVault controller adds seven bits of error-correcting code and stores the resulting 39 bits on 39 individual drives. Subsequent failure of

**Table 1. Example applications.**

| Field | Application |
|---|---|
| Geophysics | Modeling geological strata using reverse time migration techniques |
| VLSI Design | Circuit simulation and optimization of cell placement in standard cell or gate array circuits |
| Particle Simulation | *N*-body interactions, such as modeling defect movement in crystals under stress and modeling galaxy collisions |
| Fluid-Flow Modeling | Cellular autonoma and Navier-Stokes-based simulation, such as turbulance simulation in helicopter rotor-wake analysis and fluid flow through pipes |
| Computer Vision | Stereo matching, object recognition, and image processing |
| Protein-Sequence Matching | Large database searching for matching protein sequences |
| Information Retrieval | Document retrieval from large databases, analysis of English text, and memory-based reasoning |
| Machine Learning | Neural-net simulation, conceptual clustering, classifier systems, and genetic algorithms |
| Computer Graphics | Computer-generated graphics for animation and scientific visualization |

any one of the 39 drives does not impair reading of the data, since the ECC allows detection and correction of any single-bit error.

Although operation is possible with a single failed drive, three spare drives can replace failed units until repaired. The ECC provides 100-percent recovery of the data on the failed disk, allowing a new copy of this data to be reconstructed and written onto the replacement disk. Once this recovery is complete, the database is considered healed. This mass-storage system architecture leads to high transfer rates, reliability, and availability.

*Graphics display.* Visualization of scientific information is becoming increasingly important in areas such as modeling fluid flows or materials under stress. The enormous amount of information resulting from such simulations is often best communicated to the user through high-speed graphic displays. We therefore designed a real-time, tightly coupled graphic display for the Connection Machine. This system consists of a 1,280 × 1,024-pixel frame-buffer module with 24-bit color and four overlays (with hardware pan and zoom) and a high-resolution, 19-inch color monitor. The frame buffer is a single module that resides in the Connection Machine backplane in place of an I/O controller. This direct backplane connection allows the frame buffer to receive data from the Connection Machine processors at rates up to one gigabit per second.

**CM-2 engineering and physical characteristics.** The cube-shaped Connection Machine measures 1.5 meters a side and is made up of eight subcubes. Each subcube contains 16 matrix boards, a sequencer board, and two I/O boards, arranged vertically. This vertical arrangement allows air cooling of the machine. Power dissipation is 28 kilowatts. Each matrix board has 512 processors and four megabytes of memory. The matrix board has 32 custom chips implementing the processors and router, 16 floating-point chips, 16 custom floating-point memory-interface chips, and 176 RAM chips. The nexus board occupies the space between the subcubes. Each front end has one front-end interface board. Red lights on the front and back, with one light for each CM chip (4,096 altogether), assist troubleshooting.

Conservative engineering throughout ensures that the machine is manufacturable, maintainable, and reliable. The power of the machine arises from its novel architecture rather than from exotic engineering. It incorporates few types of boards; one board in particular—the matrix board—is replicated 128 times in a 64K machine. The matrix board is a 10-layer board with 9-mil trace widths.

The chip technology is also current state of the art and conservative. The CM-1 chip was implemented on a 10,000-gate, two-micron, complementary-metal-oxide-semiconductor gate array. The CM-2 chip is implemented on two-micron CMOS standard cells and has about 14,000 gate equivalents.

The massive parallelism of the machine makes it possible to provide a particularly powerful and fast set of hardware diagnostics. For example, the entire memory (which accounts for a large percentage of the silicon area of the machine) can be tested in parallel. Diagnostics can isolate a failure to a particular chip or pair of chips and one wire connecting them more than 98 percent of the time. Diagnostics, in combination with the error-detection hardware on all data paths, leads to a reliable and maintainable system, with mean time to repair well under one hour.

**CM-2 performance.** We can measure the Connection Machine's performance in a number of ways. Since the machine uses bit-serial arithmetic, the speed of integer arithmetic and logical operations will vary with word length; the languages implemented on the machine take advantage of this and use small fields whenever possible. For example, 32-bit integer arithmetic and logical operations run at 2,500 million instructions per second, while eight-bit arithmetic runs at 4,000 MIPS.

The speed of the machine also depends on how many processors take part in a particular calculation, and on the virtual-processor ratio. In some cases, higher virtual-processor ratios lead to higher instruction rates because the physical processors are better utilized.

Sustained floating-point performance has been shown to exceed 20 gigaflops (billions of floating-point operations per second) for polynomial evaluation using 32-bit floating-point-precision operands. When the function to be computed involves interprocessor communication such as a 4K × 4K-element matrix multiply, sustained performance typically exceeds five gigaflops.

We can express the performance of the Connection Machine communications systems in either of two ways: bandwidth or time per operation. Grid communications performance varies with the choice of grid dimensionality, grid shape, and virtual-processor ratio. A two-dimensional-grid Send operation takes about three microseconds per bit. For 32-bit, two-dimensional grid operations, that translates into 96 microseconds or 20 billion bits per second of communications bandwidth.

Router communications performance is somewhat harder to measure because it depends on the complexity of the addressing pattern in the message mix, and thus the number of message collisions. For a typical message mix, 32-bit general Send operations take about 600 microseconds. This translates into about three billion bits per second of communications bandwidth for typical message mixes (peak bandwidth exceeds 50 billion bits per second).

The performance of the Connection Machine I/O system depends on the number of channels in use. Each of up to eight I/O channels has a bandwidth of 40 million bytes per second, for a total bandwidth of 320 million bytes per second. This peak bandwidth has been observed on an installed DataVault disk system. The typical sustained bandwidth on the DataVault is 210 million bytes per second, which makes it possible to copy the entire contents of the Connection Machine memory (512 megabytes) to disk in about 2.4 seconds.

The preceding discussions show that the Connection Machine achieves significant gains in performance over conventional computers through the use of a data-parallel model in a fine-grained, massively parallel architecture without using any exotic technology. But, how broadly applicable is this data-parallel model? In the remainder of this article, we will illustrate the breadth of applications by describing a range of applications already developed on the Connection Machine.

## Applications of the Connection Machine

Performance measures of massively parallel architectures tell only part of the story. One of the significant revelations that occurred with the introduction of the Connection Machine concerned the surprising number of different application areas suitable for this technology.

**The general-purpose nature of the Connection Machine permits it to be applied equally well to numeric and symbolic processing.**

Although data-parallel programming requires a different approach towards computation, programmers quickly adapted. In fact, they often found that many systems are naturally expressed in a data-parallel programming model. Fears that parallel programming would require a massive reeducation effort proved unfounded.

The partial list of applications given in Table 1 illustrates the range of applications developed for the Connection Machine. In general, most applications required less than a person-year of effort and were prototyped in a matter of months. The list includes examples from engineering, materials science, geophysics, artificial intelligence, document retrieval, and computer graphics. Far from being an architecture designed for special domains, the general-purpose nature of the Connection Machine permits it to be applied equally well to numeric and symbolic processing.

We include here discussions of three applications from the fields of VLSI design, materials science, and computer vision. These examples illustrate the use of parallelism in a range of areas and the role interprocessor communication plays in supporting the data-structure requirements of each application. Grid-based communication finds primary application in regularly structured problems such as particle simulations, while general routing supports the differing topologies of circuit simulation and computer vision.

Consult Waltz[8] for detailed descriptions of several other applications.

**Molecular dynamics.** Since the 1960s, materials science has been a key technology in designing jet-engine turbine blades and other high-technology products. To understand materials more fully, designers often perform simulation studies. Unfortunately, macro-level experiments, whether direct or computer-simulated, have not successfully explained important behaviors such as metal fatigue. That requires simulation at the molecular level. Here a major problem arises. Studies of perfect crystals generally offer little help in understanding real-world materials, as evidenced by the fact that the strengths predicted by such studies often exceed those measured in actual metals by twenty to fifty times. Defects in the crystalline structure alter its properties and dramatically increase the computational complexity of simulation studies. Often the interactions of ten thousand to one million atoms need to be simulated to accurately represent the real-world behavior of materials.

Molecular-dynamics simulation is extremely computation-intensive, and the necessity of computing high-order interactions on systems of millions of particles poses major problems for conventional machines. Data-parallel architectures permit the investigator to see in minutes what would take hours on traditional hardware. The Connection Machine virtual-processor mechanism allows the investigator to simulate systems many times larger than would a physical-processor system alone.

A commonly used model for molecular dynamics uses the Lennard-Jones 6/12 potential to describe the interactions between uncharged, nonbonded atoms. Two approaches permit the study of this behavior on the Connection Machine. In both cases, processors are assigned to individual particles. Each processor contains the $(x,y,z)$ position of the particle, the velocity, and the force on the particle from a previous time step of the simulation. All calculations are performed using 32-bit floating-point precision.

In the first approach, we consider interactions between all particles. Particles are arbitrarily assigned to processors. The NEWS grid circulates the information about each particle throughout the CM such that each processor can compute the force its particle receives from every other particle in the system. All processors use the result of these forces to update their own $(x,y,z)$ position in parallel. Depending upon the purpose of the simulation, we might observe several hundred time steps. For large systems, each processor computes tens of thousands of force calculations during a single time step.

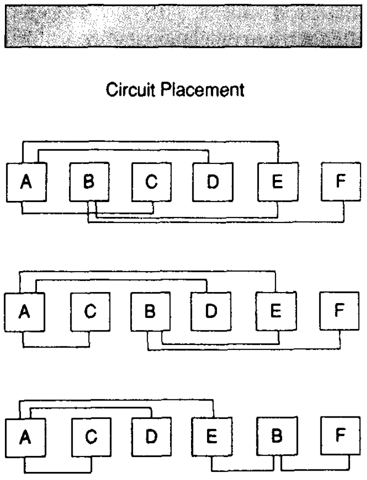


**Figure 5. VLSI cell placement. Given an initial placement of cells, individual cells swap positions to reduce the total wire length.**

This approach makes no assumption about which particles lie near others and hence dominate the overall force calculation. Dramatic improvements in execution time are possible if we know the location of nearby particles in advance. If we assume that the particles of the simulation have a regular structure and do not drift far from their initial relative positions (as in modeling solid-phase materials at low temperatures), we can take an alternative approach.

In the second approach, particles are assigned to processors according to their spatial configuration. In a manner akin to convolution, nearest-neighbor communication is used to calculate the forces from a surrounding $K \times K$ neighborhood of particles. Particles are free to drift, but we assume they stay roughly ordered relative to each other. Since the force calculation is dominated by these near neighbors, we can impose a cutoff in the force calculation. We simply ignore interactions of distant particles.

Even in large-scale simulations, the number of near neighbors remains relatively small. Thus, we need to compute only a fraction of the total number of force calculations. As expected, this results in a significant reduction in the time required for simulation. Systems as large as a million molecules when tested showed sustained execution rates in excess of 2.6 gigaflops.

Researchers have applied this data-parallel approach to studying defect motion in two-dimensional crystals of 16,384 atoms. They introduce stress by gradually shifting the top and bottom rows of atoms in opposite directions. They then observe the resulting movement, or percolation, of the defects. This work has contributed greatly to the study of stress-failure modes in various materials.

**VLSI design and circuit simulation.** Computer-aided design tools are routinely used in VLSI design, but as the size of circuits increases, two aspects of the design process become increasingly important and computationally expensive: cell placement and circuit simulation.

*Cell placement.* Semicustom VLSI circuits with more than 10,000 cells or parts have become quite common. Placing this number of parts on a chip while simultaneously minimizing the area taken up by interconnecting wire has proved a difficult and time-consuming problem. Minimization of area occupied is important because in general the smaller the area, the higher the expected yield. Designers have applied various automated methods to this problem. One method, employing *simulated annealing,*[9] produces good optimization results on a broad spectrum of placement problems. Unfortunately, for circuits having 15,000 parts, simulated annealing may require 100 to 360 hours on a conventional computer. On a 64K-processor Connection Machine, the time required reduces to less than two hours.

Simulated annealing is an optimization procedure related to an analogous process in materials science wherein the strength of a metal is improved by first raising it to a high temperature and then allowing it to cool slowly. When the metal is hot, the energy associated with its molecules causes them to roam widely. As the metal cools, molecules lose their energy and become more restricted in their movements, settling into a compact, stable configuration. This general optimization procedure, already applied to a variety of problems, supports parallel implementation.

Designers apply simulated annealing to VLSI placement in the following way. Given an initial arbitrary placement of cells for a VLSI circuit, the designer first computes the size of the silicon chip required. Improvement in the layout results from swapping cells to reduce the total wire length (see Figure 5). If the only exchanges permitted are those that reduce total wire length, the placement pattern would seek a local minimum—not necessarily optimal.

Simulated annealing, however, accepts a certain percentage of "bad" moves in order to expose potentially better solutions, avoiding the classic trap of local minimums. The percentage of such "bad" moves the system will accept is governed by a parameter usually expressed as a temperature. Starting with a high temperature, a large number of cells are permitted to change position over great distances. Gradually, the temperature parameter is lowered, restricting the movement of cells and the number of exchanges. Ultimately, the system converges to a near-optimal solution as only nearby "good" exchanges are permitted.

Implementation of simulated annealing in parallel on the Connection Machine is straightforward. Individual cells, potential locations for cells, and nodes where wires between cells connect are represented in individual processors. The need to exchange information between cells at arbitrary locations illustrates the utility of the general router-based communications mechanism of the Connection Machine.

Simulated annealing takes place as follows. According to the given temperature parameter, a certain percentage of cells initiate a swap, calculate the expected change in wire length, and accept or reject the move. Potential exchanges are chosen by randomly generating the addresses of other cells. (Exchanges in general are not independent when performed in parallel, so restrictions on cell movements are enforced to ensure the consistency of the cell rearrangement.) This process repeats as the temperature parameter slowly lowers, restricting cell movement until the solution is acceptable to the designer.

*Circuit simulation.* A second area of importance to the electronics industry is VLSI circuit simulation. Unfortunately, our ability to simulate large circuits has not kept pace with the increasing number of components. Again, this results from the computation-intensive nature of simulation. Consequently, often only circuits with several hundred transistors are simulated at a time. Just as computer-aided design tools have become essential for successful implementation of VLSI, designers need simulation at the level of analog waveforms to verify circuit design.

Real circuits operate according to the characteristics of each component and

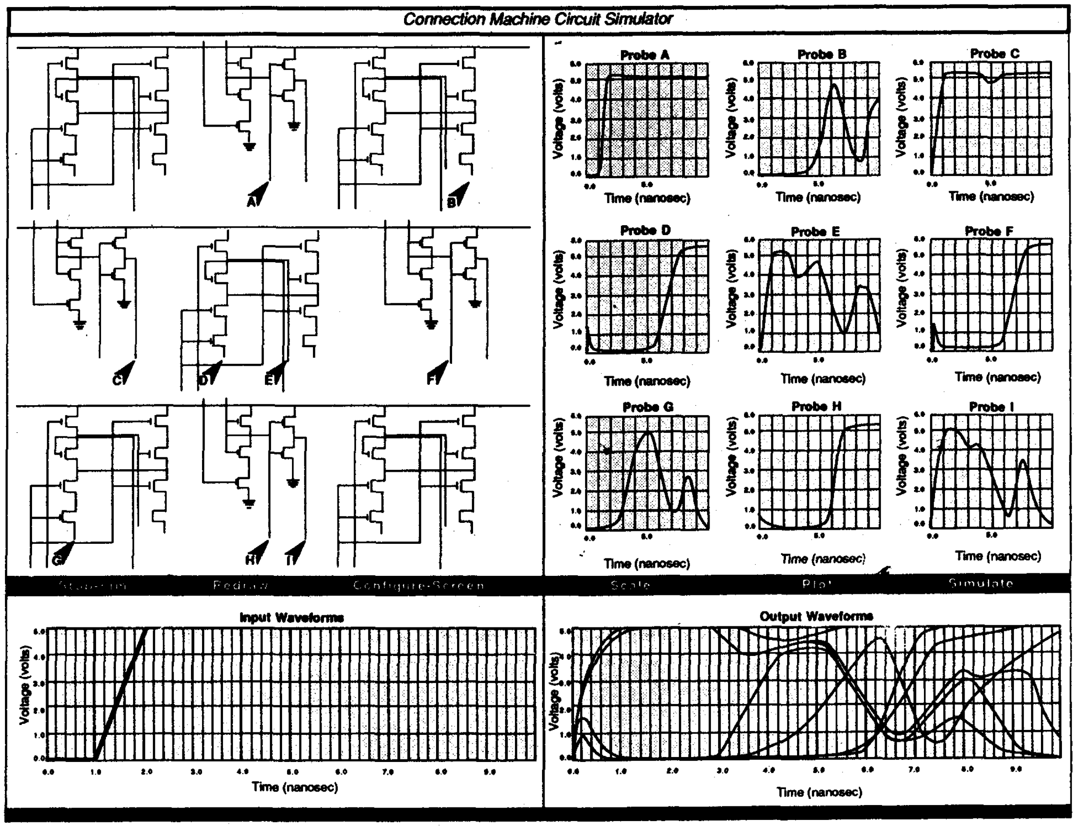


**Figure 6. Simulation output of VLSI circuit. Waveforms from multiple probe points are displayed for a given input.**

through enforcement of Kirchoff's law, which states that at each node the current entering the node is exactly balanced by the current leaving the node. In computer simulation, this leads to the application of an iterative relaxation method at each time step, which brings the circuit into equilibrium.

A Gauss-Jacobi relaxation algorithm for solving a system of nonlinear differential equations has been employed to simulate VLSI circuits.[10] As implemented on a Connection Machine, individual components (such as transistors, resistors, or capacitors) and nodes are each assigned to individual processors. Pointers express the connectivity of the circuit. During simulation, the pattern of communication directly reflects the topology of the circuit.

A simulation time step starts with all nodes sending their voltages to the terminals of the components to which they connect. In the initial time step, we know only the potentials at the power supply and ground nodes (others are assumed zero). Each element computes its response and sends this information back to connecting nodes. Each node processor checks to see if all currents balance. If so, the time step is complete; otherwise, each node processor updates its expected potential according to the relaxation rule, and the cycle repeats. In practice, each time step typically requires three to four iterations and accounts for one-tenth of a nanosecond simulated time. Figure 6 shows a typical simulation.

Because all operations take place in parallel, simulations of circuits with more than 10,000 devices can run in a few minutes on a 64K-processor Connection Machine. Simulations of circuits having more than 100,000 transistors have run in less than three hours.

**Computer vision.** Problems in computer vision pose a major challenge for today's systems not only because the problem is large (256K pixels in a typical video image), but also because we need a variety of data structures to meet the differing representational requirements of vision. We can easily see how to apply massively-parallel systems to pixel-level image processing, but it is often less clear how parallelism applies to higher-level concerns, as in the recognition of objects in a scene.

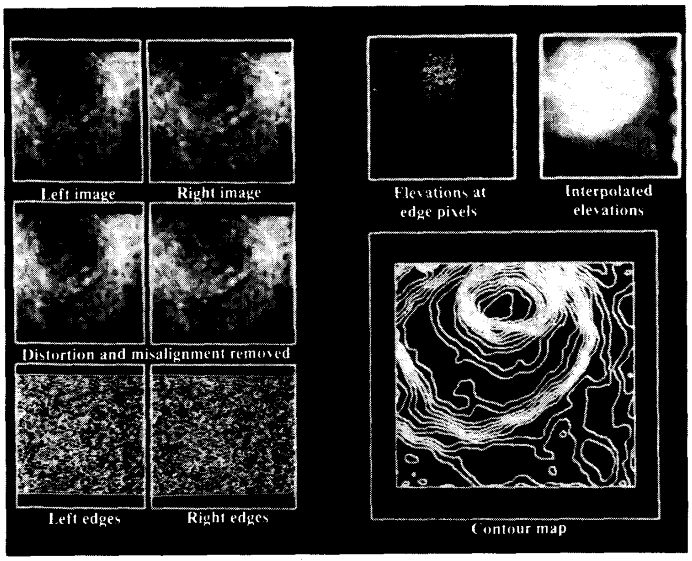


**Figure 7. Using a stereo pair of left and right terrain images, the system corrects for any geometric distortions, detects edge points, computes elevations, and generates a contour map for the given terrain.**

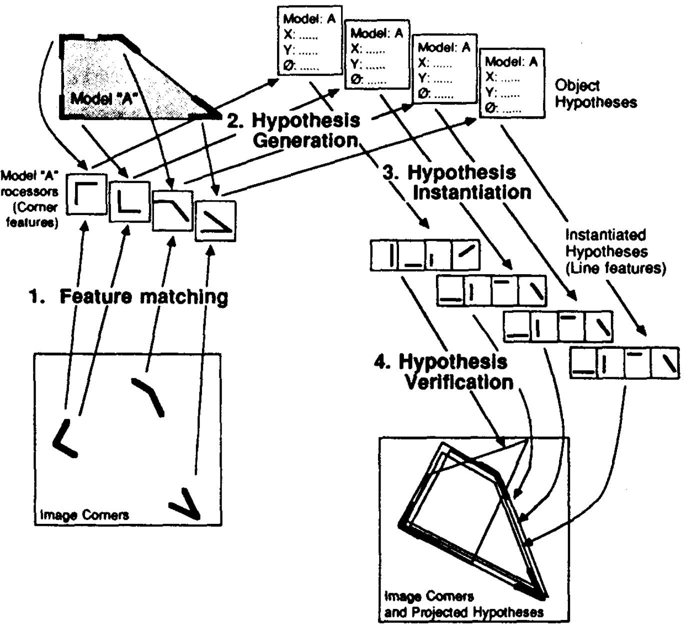

Model: A
X: ......
Y: ......
Ø: ......
Object Hypotheses
Model "A"
2. Hypothesis Generation
3. Hypothesis Instantiation
Model "A" Processors (Corner features)
Instantiated Hypotheses (Line features)
1. Feature matching
4. Hypothesis Verification
Image Corners
Image Corners and Projected Hypotheses

**Figure 8. Parallel object-recognition steps.**

Therefore, we show here two examples in computer vision: production of a depth map from stereo images, and two-dimensional-object recognition.

*Depth map from binocular stereo.* Binocular stereo is a method for determining three-dimensional distances based on comparisons between two different views of the same scene. It is used by biological and machine vision systems alike. Drumheller and Poggio[11] have implemented a program that solves this problem in near-real time and illustrates aspects of performing image analysis on the Connection Machine.

Given a stereo image pair, the program produces a depth map expressed in the form of a topographic display. First, two images (left and right) are digitized and loaded into the Connection Machine, with pixel pairs from right and left images assigned to individual processors. Next, a difference-of-Gaussians filter is applied to each image to detect edge points, which form the primitive tokens to be matched in the two images. All pixels perform this difference-of-Gaussians convolution in parallel.

To determine the disparity of features between left and right images, the Connection Machine system performs a local cross-correlation between the two images. It slides the right image over the left using NEWS communications. At a given relative shift, the two edge-point patterns are ANDed in parallel, producing a new image that has the Boolean value *t* only at points where the left and right images both contain an edge.

Then each processor counts the number of *t*'s in a small neighborhood, effectively computing the degree of local correlation at each point. This occurs at several relative shifts, with each processor keeping a record of the correlation score for each shift.

Finally, each processor selects the shift (disparity) at which the maximum correlation took place.

Since the above process determines disparity (and distance) only at edge pixels, we must "fill in" the non-edge pixels by interpolation using a two-dimensional heat-diffusion model. The resulting depth map is displayed either as a gray-scale image with density as a function of height from the surface, or in standard topology map format. The entire process, from loading in a pair of 256 × 256-pixel stereo images to production of a contour map, completes in less than two seconds on a

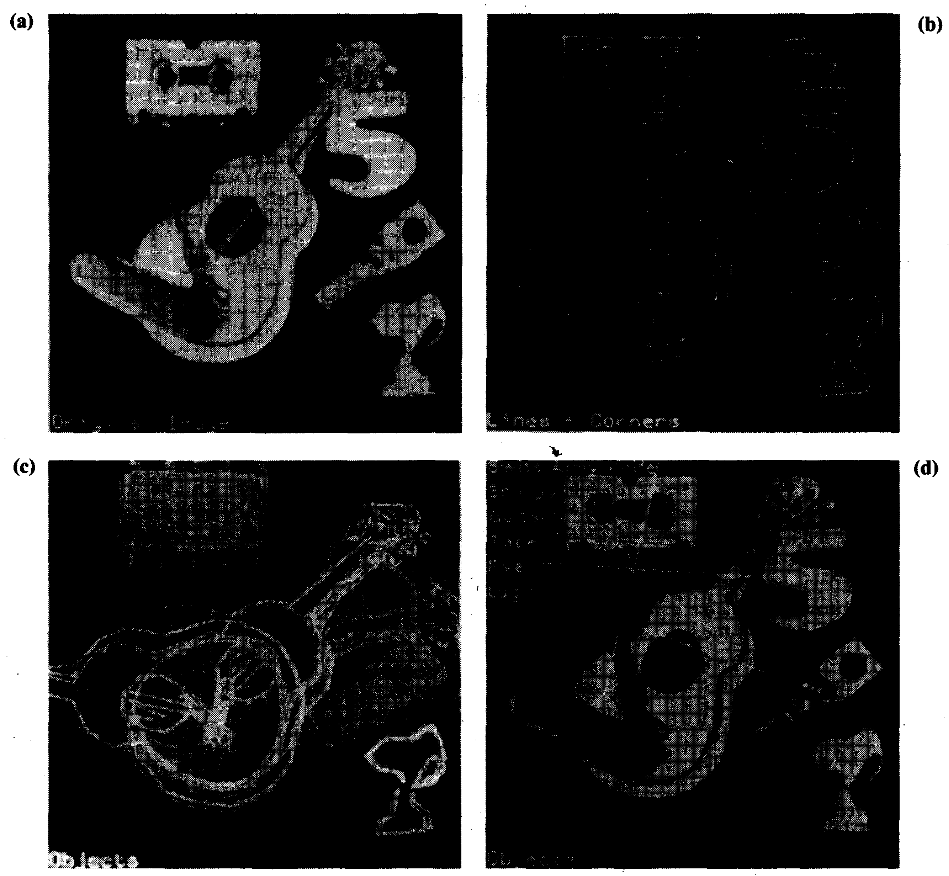


**Figure 9. Object recognition example showing original image (a), feature extraction (b), hypothesis generation (c), and recognition and labeling of objects (d).**

64K-processor Connection Machine (see Figure 7).

*Object recognition.* Recognition of objects in natural scenes remains an unsolved problem in computer science despite attracting research interest for many years. It is important in many applications, including robotics, automated inspection, and aerial photo-interpretation.

In contrast to many of the systems designed to date, the object-recognition system described in this section was designed to work with object databases containing hundreds of models and to search each scene in parallel for instances of each model object.[12] While this system is limited to recognition of two-dimensional objects having rigid shapes, it is being extended to the three-dimensional object domain.

The general framework for the approach taken here is massive parallel hypothesis generation and test. Whereas object-recognition algorithms traditionally use some form of constraint-based tree search, here searching is effectively replaced by hypothesis generation and parameter space clustering. In this scheme, image features in the scene serve as events, while features of each model serve as expectations waiting to be satisfied. Hypotheses arise whenever an event satisfies an expectation.

On the Connection Machine, objects known to the system are represented simply as a collection of features—generally straight line-segments and their intersections at corners. Each feature is assigned to its own processor. A single object is therefore distributed over a number of processors, enabling each feature to actively participate in the solution.

Features useful for hypothesis generation are those that constrain the position and orientation of a matching model object. A single point, line, or patch of color is by itself not sufficient. In a two-dimensional world, however, the intersection of two lines that matches an expected model corner can be used to generate a hypothesis that an instance of the model object exists—albeit translated and rotated such that the corresponding features come into alignment.

Figure 8 illustrates this parallel hypothesis generation and verification method. An unknown scene is first digitized and loaded into the Connection Machine memory as an array of pixels, one per processor. Edge points are marked and straight line segments are fitted to edge points using a least-squares estimate. Intersecting lines are grouped into corner features and matched in parallel with corresponding corner features in the model database.

Whenever a match between an image and model feature occurs, a hypothetical instance of the corresponding model object is created and projected into the image plane. A hypothesis-clustering scheme, related to the Hough transform, is next applied to order hypotheses according to the support offered by mutually supporting hypotheses. Although this still leaves many possible interpretations for each image feature, the Connection Machine can quickly accept or reject thousands of hypotheses in parallel using a template-like verification step.

Verification results from having each instance check the image for evidence supporting each of its features in parallel. Hypotheses having strong support for their expected features are accepted over those with little support. Expected features of an object that are occluded or obscured in the scene do not rule out the hypothesis, they only weaken the confidence the system has in the object's existence. Competitive matching between instances for each image feature finally resolves any conflicts that arise in the overall scene interpretation. Results appear in Figure 9.

The time required from initial image acquisition to display of recognized objects is approximately six seconds on a 64K-processor Connection Machine. Experiments have shown this time to be constant over a range of object database sizes from 10 to 100. We expect optimization of this task to reduce the time to less than one second.

The Connection Machine represents an exciting approach to dealing with the large and complex problems that a growing number of people wish to solve with computer systems. The size of problems has grown at a rate faster than improvements in technology, forcing us to find innovative ways of using current technology to achieve quantum leaps in performance.

We have described the architecture and evolution of the Connection Machine, which directly implements a data-parallel model in hardware and software. From the applications described here and elsewhere, it seems clear to us that data-level parallelism has broad applicability. We expect that its applicability will continue to grow as people continue to require solutions to larger and more complex problems. □

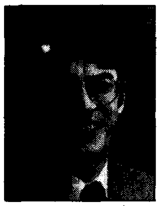
## Acknowledgments

Numerous people have contributed to the development of the architecture and applications of the Connection Machine. We wish to offer special thanks to Danny Hillis, Brewster Kahle, Guy Steele, Dick Clayton, Dave Waltz, Rolf-Dieter Fiebrich, Bernie Murray, and Mike Drumheller for their help in preparing this article.

This work was partially sponsored by the Defense Advanced Research Projects Agency under contract no. N00039-84-C-0638.

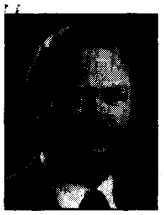
## References

1. W.D. Hillis and G.L. Steele, "Data-Parallel Algorithms," *Comm. ACM,* Vol. 29, No. 12, 1986, pp. 1,170-1,183.
2. P.M. Flanders et al., "Efficient High Speed Computing with the Distributed Array Processor," *High-Speed Computer and Algorithm Organization,* Kuch, Lawrie, and Sameh, eds., Academic Press, New York, 1977.
3. K.E. Batcher, "Design of a Massively Parallel Processor," *IEEE Trans. Computers,* Vol. C-29, No. 9, 1980.
4. D.E. Shaw, *The Non-Von Supercomputer,* tech. report, Dept. of Computer Science, Columbia Univ., New York, August 1982.
5. L.S. Haynes et al., "A Survey of Highly Parallel Computing," *Computer,* Vol. 15, No. 1, 1982, pp. 9-24.
6. W.D. Hillis, *The Connection Machine,* MIT Press, Cambridge, Mass., 1985.
7. G.E. Blelloch, "Scans as Primitive Parallel Operations," *Proc. Int'l Conf. Parallel Processing,* Aug. 1986, pp. 355-362.
8. D.L. Waltz, "Applications of the Connection Machine," *Computer,* Vol. 20, No. 1, 1987, pp. 85-97.
9. C.D. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi, "Optimization by Simulated Annealing," *Science,* Vol. 220, May 1983, pp. 671-680.
10. R.A. Newton, and A.L. Sangiovanni-Vincentelli, "Relaxation-Based Electrical Simulation," *IEEE Trans. Computer-Aided Design,* Vol. CAD-3, No. 4, 1984.
11. M. Drumheller and T. Poggio, "On Parallel Stereo," *Proc. 1986 IEEE Int'l Conf. on Robotics and Automation,* April 1986, pp. 1,439-1,488.
12. L.W. Tucker, C.R. Feynman, and D.M. Fritzsche, "Object Recognition Using the Connection Machine," *Proc. IEEE Conf. Computer Vision and Pattern Recognition,* June 1988, pp. 871-877.



**Lewis W. Tucker** is a senior scientist at Thinking Machines Corp., where he is engaged in the application of parallel architectures to machine vision. His interests include image understanding, parallel algorithms, computer architecture, and machine learning.

Tucker earned his BA from Cornell University in 1972 and a PhD in computer science from the Polytechnic Institute of New York in 1984.



**George G. Robertson** is a principal engineer at Xerox PARC, working on user interface research and artificial intelligence. Previously, at Thinking Machines, he worked on massively parallel systems and artificial intelligence. His publications are primarily in the areas of user interfaces, programming language design, operating systems design, distributed systems, and machine learning.

Robertson received an MS in computer science from Carnegie Mellon University and a BA in mathematics from Vanderbilt University.

Readers may write to Tucker at Thinking Machines Corp., 245 First St., Cambridge, MA 02142.