RIACS

Research Institute for Advanced Computer Science
NASA Ames Research Center

An Assessment of the Connection Machine

Robert Schreiber

RIACS Technical Report 90.40 June, 1990

An Assessment of the Connection Machine

Robert Schreiber

The Research Institute of Advanced Computer Science is operated by Universities Space Research Association, The American City Building, Suite 311, Columbia, MD 244, (301)730-2656

Work reported herein was supported by the NAS Systems Division of NASA and DARPA via Cooperative Agreement NCC 2-387 between NASA and the University Space Research Association (USRA). Work was performed at the Research Institute for Advanced Computer Science (RIACS), NASA Ames Research Center, Moffett Field, CA 94035.

An Assessment of the Connection Machine

Robert Schreiber*
Research Institute for Advanced Computer Science
Mail Stop 230-5, NASA Ames Research Center
Mountain View, CA 94035
e-mail: schreiber@riacs.edu

April 13, 1990

Abstract

The CM-2 is an example of a connection machine. The strengths and problems of this implementation are considered. Then important issues in the architecture and programming environment of connection machines in general are considered. These are contrasted with the same issues in MIMD multiprocessors and multicomputers.

1 Introduction

VLSI technology continues to evolve. In today's submicron CMOS technologies, million transistor chips with 50 Mhz clocks capable of two 64-bit floating point operations per clock have been built. In the coming decade, a further four-fold reduction in feature size and 16 fold increase in density is likely: thus, the 1600 Megaflop processor chip is coming. These developments will lead to parallel teraflop systems in the late 90s. There are going to be strong MIMD shared memory contenders as well as multicomputers; I also expect that the DARPA sponsored development at Thinking Machines will lead to SIMD massively parallel systems that get there as well.

This paper is meant to be a critical assessment of the connection machine project. The conclusions expressed here are entirely mine. They have been

*This work was supported by the NAS Systems Division and/or DARPA via Cooperative Agreement NCC 2-387 between NASA and the Universities Space Research Association (USRA).

developed with the attention, criticism, and help of my colleagues, especially Creon Levit and John Gilbert, but I am solely responsible for these views.

My conclusion is that the connection machine is a breakthrough in machine architecture. It clearly shows the possibility of achieving a major gain in sustained performance through the use of simple but highly replicated hardware. It is also a breakthrough in that the fundamental programming model, data parallelism, is much easier to use and think about than MIMD programming systems. TMC has made several nice extensions to the simplest SIMD programming model: parallel remote reference (called **pref**), parallel remote store (called **pset**) with combining operators, segmented parallel prefix operations (called **scan**), and nearest-neighbor grid communication (called **news**). These add significant power to the languages.

Nevertheless, there are a number of serious weaknesses in the current TMC implementation (the CM-2) of the ideal connection machine. After discussing these, I will give my view of the unsolved and difficult problems of software and hardware that will need to be addressed over the next decade in order for the supercomputer user community to "cash in" on the connection machine breakthrough.

1.1 Outline

The contents of the report are these.

1. Introduction.
2. Connection machines: what are they?
3. Uses of the CM-2 at RIACS and NASA Ames.
4. Abstract CM architecture.
5. The hardware implementation.
6. The programming model.
7. The programming tools.
8. In contrast: The MIMD parallel computers.
9. The future of TMC.

2 Connection machines: what are they?

The connection machines are implementations of the abstract CM model proposed by Hillis in his 1985 PhD thesis [6]. The idea is massively parallel computing. There are so many processors that one thinks in terms of an unlimited number. The memory is purely distributed and local so that by the replication of local access the overall memory bandwidth is huge. (Hillis rightly starts from the thesis that limited bandwidth to memory is the crux of the problem of speed in highly parallel computing. His solution is memory parallelism to go along with the processor parallelism.) These ideas were not new to Hillis. The MPP and the DAP had these features and came earlier. But in Hillis' connection machine, there is a communications network so that the programmer may access remotely stored data through the network just as if it were locally stored (the connections). The programming style is based on the idea of one virtual processor per element of the chosen data structure. Programs freely use the connection system through the primitive mechanisms of parallel remote references and parallel remote store with combining operators to resolve multiple stores to the same memory cell. They also use it through the parallel prefix, or **scan** pseudo-operator.

Hillis minimizes the importance of the SIMD — MIMD dichotomy in his thesis. The initial CM implementation was SIMD as a matter of convenience, not principle. But the users of the CMs have become accustomed to the SIMD style. And they have found it far easier to program a single threaded SIMD machine than have users of the MIMD alternatives. It seems very probable that connection machines will continue to be SIMD, at some level. (The chief problem with the SIMD implementation of the CM-2 is that at most four simultaneous users can be on the machine at one time.)

3 Uses of the CM-2 at RIACS and NASA Ames

At RIACS and NASA Ames we have had a lot of experience in implementation of complex numerical methods on the CM. The NAS Systems Division has ported the flow code ARC3D [7]. RIACS personnel have implemented a particle simulation of hypersonic flow [1], [2]; efficient matrix multiplication codes [8]; some multilevel iterative methods for elliptic differential equations [3], [4]; and a sparse Cholesky factorization [5]. RIACS and NASA scientists are developing codes for geophysical fluid dynamics (Lewis and Frederickson) and for preconditioned conjugate gradient iterations and ex-

plicit finite volume flow codes using unstructured grids (Hammond, Barthes, and Schreiber). This work has been done in *lisp and c* with some code in Paris; we have now begun to use CM Fortran as well.

4 Abstract CM architecture

The CM and other highly parallel machines derive much of their power form the simplicity and scalability of their hardware. The hardware consists of ICs of a very small number of types, all VLSI, that are laid out in replicated patches over a number of PC boards. The ideal system uses only one or a few VLSI parts per processor with a few DRAM memory chips along side. Machines with any other hardware strategy are invariable too expensive, too power hungry, and too unreliable. Architectures that discourage this approach are likely to fail in the long run.

The router is the most important, innovative, and difficult to build feature of CM architectures. It is vital, however, in that it provides the hardware support needed to implement the very general communication mechanism that programs for complex problems require. (The corresponding feature in vector machines, hardware scatter/gather, is equally important in that domain.) Because arbitrarily complex communication is implemented entirely by router hardware, no layer of software intervenes between application code and message passing. This reduces the latency for communication to levels where it is possible to use it every few operations, not every few thousand as on current MIMD multicomputers.

Concerning the processor architecture, the issue of whether 1-bit, 8-bit, or 64-bit processors should be used is important for these reasons:

i) 1-bit architectures are most efficient in that all processor hardware is employed all of the time;

ii) A corollary of point (i) is that narrow processors are extremely efficient at logical operations (1 bit data), pixel operations (8–16 bit data), and integer operations (16 or 32 bit data);

iii) For a given total gate count, one has more 1-bit processors than, say, 8-bit processors. Thus, more algorithm parallelism is needed to use a 1-bit machine than an 8-bit machine of the same cost;

iv) Local indirect addressing is very difficult in 1-bit architectures, since a multiple bit address (16 – 24 bits, typically) must be read from

memory to support a 1-bit read; this may be a fatal problem;

v) In 1-bit architectures it is not useful to integrate a fast scalar unit tightly with the parallel array, since the time to extract data from memory over a 1-bit data path is prohibitive. Given wider paths to memory, there can be considerable value in this kind of hybrid system.

The tension between points (i) and (ii), which favor 1-bit, and (iii) – (v), which favor wider processors, makes the 4-16 bit range a very attractive design point in today's technologies: this allows 10^4 – 10^5 processors in machines with prices in the \$1 – \$10 million range. In future designs, the capability for local addressing of local memory should be provided, perhaps at some reduced throughput. To do this on the current CM requires storage of data in a "slicewise" manner, in which a 32-bit word is spread over the 32 processors on 2 chips that share one Sprint–Weitek combination. With better hardware implementation of floating point and indirect addressing, this distinction should disappear in the future.

4.1 The instruction set

Recent research into instruction set architectures has shown that simple instruction sets that provide direct control of the hardware without an intervening layer of microcode are the best approach. These architectures are called RISCs (Reduced Instruction Set Computers).

The instruction set of the CM-2, Paris, implements a very high level machine model. It is virtualized. It is a memory–memory architecture. It has an enormous number of instructions because it is aimed at too high a level of abstraction. It is implemented by microcode, which makes it slower. It hides important machines features, most notably the Weitek chip registers, the latency of the paths to memory and the details of the router. It seems to me that Paris was designed before the virtues of the RISC approach to instruction sets was well known and understood. At this point, Paris is a significant liability for TMC. Future compilers should bypass Paris. It should be supported as a high-level language only for compatibility purposes.

The fact that Paris is a memory – memory instruction set is particularly unfortunate, since this wastes the most valuable machine resource, memory bandwidth. In RISCs, memory is referenced only by load and store instructions; this allows memory traffic to be scheduled to hide the latency of memory and to avoid unnecessary loads and stores. But in memory – memory architectures, in which the processor registers (and the CM-2 has

many of these) aren't visible to the programmer or compiler, every temporary value is stored to and loaded from memory, and can't be loaded until the processor already is waiting for it.

Because of its complexity, Fortran probably won't use more than a fraction of Paris.

Because of the deficiencies of Paris, it is common for programmers to resort to even lower level coding, microcode in some form, to get the best possible performance out of the CM hardware for their applications. There are several well known examples: the 5Gflop matrix multiplier, the TMC implementation of FFT, Vavasis' fast spreads for the QR factorization, the LU decomposition codes in the new linear algebra library, for instance. This is symptomatic of Paris failure as an instruction set for the CM-2 hardware.

I don't want to sound completely negative about Paris. It does have terrific restaurants.

Virtualization in the instruction set, implemented by microcode, is a bad idea. The alternative is for software to create "strip-mining" loops around non-virtual instructions to implement virtualized operations. The advantage of this approach is that these loops are exposed to compiler optimization. For example, the compiler can detect that in the Fortran statement

```
forall (i = 1:n), a(i,i) = 0.
```

that the number of virtual processors for which any activity is required is in general less than the number of data elements per physical processor, and can act accordingly. Not so the CM microcode.

The implementation of `news` communication in Paris with virtualization is illustrative of the way memory bandwidth is wasted. Paris virtualizes by placing a group of contiguous virtual processors in a single physical processor. Consider the *lisp code

```
(*set a!!  (+!!  (news!!  b!!  -1) (news!!  b!!  1) ) )
```

which replaces $a(i)$ by $b(i-1)+b(i+1)$. Let there be V virtual processors for every physical processor. Processor P_1, for example, stores $a(V),\ldots,a(2V-1)$ and $b(V),\ldots,b(2V-1)$. A good implementation would move only two values into each processor, regardless of the VP ratio. In the case of P_1, only $b(2V)$ from processor P_2 and $b(V-1)$ from processor P_0. But Paris has to move $2V$ values! Only two move into a processor from its neighbors. The other $2(V-1)$ are moved from one memory cell to another within the processor. Why? Because the microcode that implements Paris only

sees two `news!!` instructions, which move every element of b to the two neighboring virtual processors.

In summary, a lower-level instruction set that is not virtual and in which machine registers are visible would allow the user or the compiler to make better use of the hardware's resources. It would make the CM-2 far more effective. Paris also uses the router wires inefficiently. Better communications software that implements important communication patterns, such as `news`, `power-of-two-news`, and spreads would be another significant improvement to the CM-2.

5 The hardware implementation

The CM-2 is implemented with gate arrays. Without a full custom implementation, future CMs will not keep pace with the microprocessor based machines. Today, chips with 2 – 3 times the clock rate and 2 times the data path to memory are easy, low-risk designs. No architecture is so good that it can be implemented in routine technology and still survive in this very competitive market.

How should floating point be handled? The CM-2 does it by strapping a Weitek chip onto the board as a coprocessor for each pair of processor chips. A glue chip (the "Sprint" chip of the CM-2) is needed for serial – parallel conversion. The latency of floating point operations is quite high due to this long pipeline, so only highly virtualized operations are efficient. Finally, the memory of 32 processors is not fast enough to keep the Weitek fully supplied with data; but Paris does not provide visibility of the Weitek registers, so there is little software can do to alleviate the memory bottleneck. The moral of the story: design the floating-point in, don't strap it on. The floating point should run at memory speed (one operation to one memory reference). And the floating point pipe stages and registers need to be visible in the instruction set, RISC style.

The CM uses a front-end for two distinct jobs: running the operating system and controlling the parallel array. This makes the front-end a bottleneck. In the future these roles should be separated, and a fast control processor that is tightly integrated with the parallel array should be developed. It should be capable of fast scalar operations.

The current CM avoids the complication of a memory hierarchy entirely. This likely will continue to be viable and is a major advantage. On the other hand, with the increase in CMOS VLSI density and the resulting speed of

the processor chip that will come during the 90's, a system in which all memory references are off-chip may become untenable: some cache on the processor chip may be required in a future generation of the machine. (New packaging and mounting techniques may make this unnecessary by allowing for much wider data paths between processor chips and memory chips.)

The CM uses no virtual addressing. For computations on large data arrays, which is what the CM is supposed to do, virtual addressing can be quite problematic due to highly nonlocal addressing of data. Moreover, local indirect addressing would be impossible if it required that every such address undergo a virtual to physical translation in the processor: there is no room for a TLB and other such mechanisms at each CM processor.

The router is the key to the CM. The current machine has a router that runs at roughly 1% of floating point speed. This make router-intensive algorithms unattractive in terms of their performance. But many such algorithms are important in many areas, for everything from graph theory to geometry to AI. So a true CM needs a faster router. This is *the* difficult hardware problem for this type of machine and the one into which research money should flow fastest.

The CM does not do particularly well on problems for which the underlying communication scheme is grid oriented. A better hardware grid may be the answer for some applications, but a faster router should have the top priority. Part of the problem may well be fixed by simplifying and devirtualizing the instruction set, as was indicated earlier.

The I/O structures of the CM-2, the data vault and frame buffer, are excellent. They illustrate well that with respect to I/O, data parallel machines have an advantage: parallelism in the processors is matched to parallelism in the memory is matched to parallel I/O in a balanced, simple, high bandwidth system.

6 The programming model

6.1 The connection machine programming model

The programming style for the connection machine is called "data parallelism". In essence, whole arrays are acted upon, elementwise, in parallel. Three levels of abstraction are possible:

i) All operations are done on arrays of one element per physical processor. This is the programmer's view at the assembly language (microcode

on the CM) level.

ii) Operations are done on arrays of one element per virtual processor. Each virtual processor simulates V virtual processors. Thus, array sizes are a multiple of the machine size. The CM 'assembly" language Paris works at this level. (With the restriction that V is a power of two.)

iii) Arbitrary arrays and subarrays may be used as operands. Fortran 90 works at this level.

The third of these levels is the most appropriate for serious scientific computing and is, fortunately, soon to be available.

The introduction of the virtual processor model by TMC was a very important step in the direction of useful programmability in SIMD parallel machines; even today, it is not available on the MIMD multicomputers.

A very important advantage of the SIMD programming models is reproducibility of results. Because there is a single thread of control, a program produces the same results every time it is run (assuming the same input data). This greatly eases the problem of debugging. MIMD models on the other hand are nondeterministic. Thus, known bugs are hard to track down since they may be ephemeral.

The problem with writing applications programs on the CM are threefold. First, Amdahl's law plays a role: you need to keep most of the hundreds of thousands of virtual processors busy almost all the time. Only extremely parallel algorithms work well. Second, irregular communication through the router is very expensive and should be sparingly used. On later instances of the CM this should be made less so. This will allow the applications programmers greater freedom in their choice of algorithms and data structures and will make it possible to solve problems with irregular topological structures more easily. Finally, the peculiar characteristics of the CM-2 instruction set often require that the programmer get involved with coding at an unnecessarily low level. This ought not be true in the future.

Let me summarize my thoughts on the programming environment.

- Connection machine programming is essentially no more difficult than sequential machine programming.

- Very innovative algorithms are needed on connection machines because of parallelism (Amdahl) and because the data are distributed so no processor sees more than a small part of the problem.

- Fortran 90 is a very promising approach to the programming of many but not all parallel supercomputing situations.

7 The programming tools

For numerical computation, *lisp has little to offer. The Fortran now under development is better in these ways:

- It is quite close to the standard Fortran 90; it is divorced from Paris entirely;
- It allows multiple VP sets and arrays of any size without any fuss;
- It provides the most natural syntax for arrays and iteration, Fortran's traditional strengths;
- It provides dynamic storage allocation, correcting one of Fortran's traditional weaknesses.
- Its intrinsics are useful.

The current CM Fortran needs some additional extensions.

- There is no provision for `pset` with combining operators;
- There are no scans, segmented or otherwise;
- There is limited control over the layout of arrays in the CM;
- Nested `where` constructs should be allowed.

Unfortunately, the birth of CM Fortran has been slow and very painful. As of today, the implementation fails to support the full language. Array valued functions, a key feature, are not implemented. There is no interactive debugger, a feature that I don't find really important, but others would like. More important, very little is known about what optimizations the compiler will do and how it will do them. A very interesting question is the implementation of `forall`. This construct gives the language a lot of additional expressive power. Whether it can be compiled into good code, and how, remains to be seen. How the compiler will manage memory and how it will map data to the machine are also open issues.

It is quite inconvenient to have to rely on access to the machine in order to debug. Offline development of CM Fortran programs, or time sharing of the CM should be a high priority.

Further development of the parallel variants of C is of little value.

8 In Contrast: The MIMD Parallel Computers

8.1 Shared memory and distributed memory MIMD

The first question to be answered in determining the direction for supercomputing is one of architecture. The Von Neumann line of machines, the climax of which is the current vector multiheaded supercomputers (two evolutionary steps away from Von Neumann already) cannot continue to evolve to meet NASA's needs. For the future, there are several alternative branches.

Today's (Cray, NEC, Fujitsu, Hitachi) supercomputers are MIMD multiprocessors that share a unified memory space. A number of highly parallel derivatives of these machines are now under development. To build such scaled up versions, a new memory architecture that employs many memory modules and a processor – memory switching network is necessary. This makes coherent caching at the processor difficult, so some software controlled caching is becoming popular. Latency for access to nonlocal memory is high on these machines: 5μsecs is typical (whereas floating point arithmetic takes a few tens of nanoseconds, at most).

The one advantage of these systems is that they can attempt to support the current programming model: simultaneous multiple users, many Unix processes, the illusion of a flat memory with equal access by all processes, and automatic compiler extraction of parallelism from sequential code.

The multicomputers are an alternative. (They have also been called "message passing" machines, but I prefer the name multicomputer.) In these, each processor has its own memory and may not address the memory of another processor directly. Synchronization and communication are accomplished by messages, sent by one and received by one or several processors. Peak performance compares favorably with the shared memory alternatives, but not by much: the difference, I feel, is due more to the use of very high performance stock microprocessors. In both classes of machine, hardware costs are roughly the same, with slightly less hardware devoted to interconnect in the multicomputers.

In early multicomputers, memory per node was inadequate. Large programs or large shared data structures that had to be copied in every node

were therefore ruled out. The economics of hardware technology now make it cheap to have several tens of megabytes per node (at the node per board level) so that this is not a problem any longer.

Latency for communication and synchronization is due essentially to the cost of the operating system call needed to send or receive a message. In the current i386 based Intel machine, that latency is roughly 300 μsecs. The vendors hope that new faster microprocessors and reimplementation of the code will reduce this to as little as 10 μsecs, but there is no certainty that they will be able to do so. Unlike the original, and indeed the current hypercubes, these new systems will use a message routing subsystem connected as a grid in two dimensions and implemented by full custom VLSI devices. This has essentially eliminated hardware as a source of significant message passing latency. Bandwidth, however, is still hardware limited by the channel width (which is now 8 bits).

The fundamental difference between these two architectural species is that the shared memory machines use hardware to generate messages on program demand, and the messages (words or cache lines) are a few tens of bytes long. The avoidance of a software layer to traffic with remote memory greatly reduces the latency that can be achieved. On the multicomputers, the programmer has the burden of explicitly decomposing the data into its separate local data structures; this can enhance performance given the current state of compiler technology. It results in fewer messages with more information in each, thereby allowing for increased utilization of the network. It also makes programming these machines hard, especially when a computation is irregular or dynamic: as in a moving local mesh refinement solver for unsteady transonic flow, for example.

8.2 MIMD programming models

For the MIMD multicomputers, at the assembly language level, there is one program per node with explicit use of messages to handle data sharing and synchronization. This model is currently the only one supported by the manufacturers. (This pill usually comes with a C or Fortran flavored sugar coating). While an optimizing compiler shields the programmer from the peculiarities of the node architecture (and allows for portability between machines with different nodes) the programmer sees the fact that the machine has no unified memory space. Several alternatives currently under study by university and commercial researchers include Linda, a programming system that simulates in software an associative shared data space; virtual shared

memory simulated using software; compiler optimization to partition the data and the work of an unpartitioned program, inserting message passing calls as needed.

In shared memory machines, access to shared variables is tricky. Semaphores are needed to insure proper synchronization of writes and reads. A number of other synchronization mechanism, such as the barrier, are available to the programmer. Access to these synchronization tools can be expensive. So is access to nonlocal memory.

Some current research directions in simplifying the programming of these machines are:

- The development by the Parallel Computing Forum of a standard set of extensions to Fortran 77 to allow the programmer various ways of explicitly expressing parallelism in a program.
- The development by machine and compiler vendors of Fortran 77 and Fortran 90 compilers that automatically find and exploit parallelism at an outer loop level.
- Operating systems that allow the amount of parallelism used in a job to vary as the characteristics of the computation vary.
- Dynamic scheduling mechanisms that balance the load between processors at run-time.
- Compiler analysis of entire programs (interprocedural analysis) to allow for better optimization.
- Automatic decomposition of programs into tasks that require relatively little communication (automatic blocking of algorithms).

9 The future of TMC

Here are my chief recommendations:

1. Redesign the processor chip in a full custom, high density CMOS technology. Increase the clock rate.
2. Design a new, simplified instruction set in which each processor is a load/store architecture. This could be done for the CM-2 now.

3. Implement virtualization through appropriate code generation by the compilers. The instruction set should not be virtual.

4. Build support for floating point arithmetic into the processor chip.

5. Make the router faster in relation to the processors.

6. Provide local indirect addressing.

7. Develop a robust optimizing compiler for a full Fortran 90, with appropriate TMC extensions to support segmented `scan`, `pset` with combining, and nested `where`.

Based on discussions between NASA and RIACS and the TMC staff, I believe that TMC is aware of all of these issues and is working to correct the problems and accentuate the strengths. I hope that this will result in a machine that realizes to an even greater extent the promise that data parallel architectures have for very large scale scientific computing. Thus I expect to see a healthy TMC delivering production supercomputer solutions that, for the right applications, far surpass what is offered by the traditional supercomputer houses.

References

[1] Leonardo Dagum. Implementation of a hypersonic rarefied flow particle simulation on the Connection Machine. Technical Report 88.46, Research Institute for Advanced Computer Science, 1988.

[2] Leonardo Dagum. A fast sorting algorithm for a hypersonic rarefied flow particle simulation on the Connection Machine. Technical Report 89.44, Research Institute for Advanced Computer Science, 1989.

[3] Paul O. Frederickson. Totally parallel multilevel algorithms. Technical Report 88.34, Research Institute for Advanced Computer Science, 1988.

[4] Paul O. Frederickson. Totally parallel multilevel algorithms for sparse elliptic systems. Technical Report 89.10, Research Institute for Advanced Computer Science, 1989.

[5] John R. Gilbert and Robert Schreiber. Massively parallel sparse Cholesky factorization. Technical Report, Research Institute for Advanced Computer Science, 1990.

[6] W. Daniel Hillis. *The Connection Machine.* Cambridge, MA: MIT Press, 1985.

[7] Dennis C. Jespersen and Creon Levit A computational fluid dynamics algorithm on a massively parallel computer. *International Journal of Supercomputer Applications* 3:4 (1989) pp. 9-27.

[8] Walter F. Tichy. Parallel matrix multiplication on the Connection Machine. Technical Report 88.41, Research Institute for Advanced Computer Science, 1988.