Performance Characteristics of Scientific Computation on the Connection Machine

Roldan Pozo
&
A. E. MacDonald

CU-CS-440-89 June 1989

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Performance Characteristics of Scientific Computation on the Connection Machine†

Roldan Pozo and A.E. MacDonald

CU-CS-440-89 June 1989

Roldan Pozo
Center for Applied Parallel Processing
and Department of Computer Science
Campus Box 430
University of Colorado @ Boulder
Boulder, Colorado 80309-430
(303) 492-6361 e-mail roldan@tigger.colorado.edu

A.E. MacDonald
Forecast Systems Laboratory
National Oceanic and Atmospheric Administration
325 Broadway
Boulder, Colorado 80303
(303) 497-6378

† The computations described in this paper were performed on the CM-2 at the Center for Applied Parallel Processing provided to the University of Colorado by the Defense Advance Projects Research Agency.

Abstract

The Connection Machine 2 (CM) is a massively parallel SIMD supercomputer: a fully configured machine consists of 64K (65,536) processors running in parallel. Our benchmark results indicate that such a machine can sustain speeds of over 5 GFLOPs for certain parallel triad operations. This paper describes some performance characteristics of scientific computations on the CM-2. To assess the value of this machine in the scientific computing field, we present benchmark results for floating point operations and regular-patterned interprocessor communication. Furthermore, as an indication of this machine's performance on a real application, we chose to implement a 3-dimensional atmospheric (primitive equation) model, a standard in numerical weather forecasting. As a set of three dimensional coupled partial differential equations, it is an example of a very important class of scientific problems. Although this model problem does not parallelize perfectly, it is shown that a performance of 1 GFLOP is sustained. We conclude by presenting characteristics of parallel algorithms that are well suited to this machine with suggestions on how to best utilize the CM hardware for maximum GFLOP performance.

1 Introduction

Massively parallel computers have begun to make a presence in the supercomputing field. Many researchers agree that the numerical solution of many challenging problems in aerodynamics, climate modeling, structural analysis, medicine, biology, physics, vision, and other important fields will require supercomputers with thousands, even millions, of processors.

The Connection Machine 2, developed at Thinking Machines Corporation, Inc. (TMC) in Cambridge, Massachusetts, is a single instruction multiple data stream (SIMD) distributed memory computer consisting of 65,536 processors and delivering supercomputer performance. Future plans to design a similar 1 million processor machine are being considered.

What performance can we expect out of such computers? Simply quoting theoretical peak performance rates is insufficient, since these figures are rarely sustained when running everyday code. Fortunately, a good portion of the work spent by large application codes can usually be traced to several computation-intensive subroutines. Thus, having reliable performance figures for numerical operations allows application programmers to formulate execution-time estimates for their applications, before running a single line of code.

This paper presents *expected* performance characteristics of the Connection Machine on numerical computations. We benchmark floating point operations: primitive operations (+,*,-,/), math library functions (`sin, exp, sqrt`), triad operations (`ax + y`), and communication costs for regularly-patterned two-dimensional topologies.

Furthermore, we discuss some suggestions for best utilization of the CM hardware: maximizing performance through the use of virtual processors, pipelining the Weitek 3132 FPU effectively, using triad operations to minimize use of the Sprint Chip, and so on.

Finally, as a typical example of running a real application on the Connection Machine, we have implemented a finite difference version of an atmospheric primitive equation model. The primitive equations are currently predominant in operational weather forecasting world wide, and thus allow the test of a massively parallel architecture on a very important, complex and practical problem.

2 The Connection Machine

This section presents a brief description of the Connection Machine (CM) architecture, hardware, and function. It is not intended as a comprehensive discussion – we have omitted several layers of detail and instead presented just enough of the basic concepts so that the reader can realize some of the issues involved in making the best utilization of its hardware. A more thorough treatment of the design and implementation of the Connection Machine can be found in Hillis [3] and TMC[4].

2.1 Overview

The Connection Machine is a massively parallel single instruction multiple data stream (SIMD) air-cooled supercomputer. A fully configured machine consists of 64K (65,536) bit-serial processors utilizing 2,048 Weitek 3132 floating point units (FPU), rated at 16 MFLOPs each. Thus, a Connection Machine that keep each of its Weiteks optimally busy can reach speeds in the 32 GFLOP neighborhood, although that theoretical peak is not usually obtained. This paper describes the performance that *can* be obtained in practice.

Although the CM proper consists of 64K processors, smaller configurations, *subcubes*, of sizes 8K, 16K, and 32K, are also available. The machine accessible to us for benchmarking was an 8K processor, 6.7 MHz CM at the Center for Applied Parallel Processing (CAPP) at the University of Colorado, Boulder.

The benchmarks conducted in this paper were measured on a CM with a 6.7 MHz clock, and thus performance figures should be scaled accordingly to other CM machine running at different clock rates. That is, if your site is equipped with an 8 MHz CM, then your performance figures should be higher by a factor of 1.19.

2.2 Processors

The basic architectural unit of the CM are the bit-serial processors. Sixteen bit-serial processors are fabricated on a single physical chip; thus the full CM contains 4096 such chips. Every two chips (32 bit processors) share a Weitek 3132 floating point unit (FPU).

It is these Weitek chips that provides the CM with its tremendous number-crunching strength. When properly pipelined, each Weitek 3132 can produce one (single-precision) floating point multiply and one addition result per clock cycle. For example, a CM with a 8 MHz clock can run each Weitek at 16 MFLOPs, thus accounting for the peak 32 GFLOP figure quoted above.

2.3 The Sprint chip

How do the bit-serial processors interface with the Weiteks? This is the key to understanding CM floating point performance. There is a "glue" chip, the Sprint chip, that performs this service. Each group of 32 bit-serial processors shares a Sprint chip and a Weitek. At each clock step, each of the 32 processors deposits a *single bit* of its single-precision argument into the Sprint chip. After 32 clock steps, the Sprint chip contains 32 single-precision (32 bit) floating point values —one from each processor. The Sprint chip then feeds these 32 values into the Weitek for processing and then performs the reverse operation when retreiving results from the Weitek back to the processors.

Notice that a single bit-processor cannot pipeline a Weitek bit by bit, because the Weitek works only with 32-bit numbers. That is, at each clock step, each bit-processor can only produce one bit, while the Weitek can digest 32 bits; one way to make up for this mismatch of speed is to draw a bit from *each of the 32 bit-serial processors in parallel* then feed these values *sideways* to the Weitek. This is currently what the Sprint chip does.

This has two important ramifications:

1. To make best use of the floating point hardware, one should always **utilize *all* 32 processors sharing the Weitek** — it takes the same amount of time (32 clock steps) to load a single value as to load all 32 values into the Sprint chip [1] Thus, if one is not loading data between the Sprint chip and *all* 32 processors, efficiency is being lost.

2. **One should minimize traffic through the Sprint chip by attempting to keep intermediate results in the Weitek registers.** Storing intermediate results back in the processor memory requires a transposition through the Sprint chip, only to be loaded back again for the next operation. Such needless traffic through the Sprint chip can become a serious bottleneck for floating point operations. Triad operations (discussed in a later section) achieve their high GFLOP rates by keeping the intermediate results in the Weitek.

2.4 Memory

Each bit processor has 8196 (8K) bytes associated with its local memory space; thus the CM contains 512 MBytes of core memory. Future plans of TMC include a memory upgrade to 32K bytes per processors, resulting in a machine with 2 Gigabytes of core memory.

Each processor memory is divided into stack and heap regions. No code is stored in processor memory —only data. Each bit processor can access its local memory at 5 megabits per second, providing a memory bandwidth of 300 gigabits per second [4], [6].

2.5 Processor Communication

Since the CM is a distributed memory machine, processors need a way to communicate with each other. Communication between bit-processors on the same physical chip occur

[1]Originally, some programmers were masking some of the physical processors, thinking that this would squeeze out higher performance numbers —instead, it brought them down.

through a Benes network. The 4096 physical *chips* in CM are connected in a boolean cube topology: each chip forms the node of a 12 dimensional hypercube. Hypercube topologies have the attractive feature that *ring*, *tree* and *N-dimensional grid* topologies can be efficiently imbedded in them [1].

In a machine with P processors, each processors has its own unique address, numbered from 0 to $P-1$. General communication between CM chips occurs through "routers"; every group of 16 bit-serial processors share a router which is responsible for forwarding outgoing requests from the chip onto the network, and servicing incoming message requests from other chips residing in the Connection Machine. A processor can request a value from any processor by specifying that processor's cube address, the beginning memory location of the data object, and its bit length.

For applications that utilize regular communication patterns along N-dimensional grids, faster communication primitives exist that do not entail the overhead of using the chip router. For example, in a 2-dimensional grid topology, each processor can simultaneously retrieve data from its immediate south neighbor. This type of regular communication is termed NEWS (each letter representing the four respective directions: north, east, west, south) to distinguish it from the more general (router) communication.

2.6 Front End

The Connection Machine is not a stand-alone computer: it stores data but no code. Program development is carried out on the front-end or host computer: currently either a Sun Microsystems workstation, Symbolic 3600 system, or DEC VAX 8000 mainframe. That is, program development (compiling, editing) and its *execution* occur on the front-end. At run-time, the front-end attaches to the CM and takes control of it, issuing a command stream in a synchronized, lockstep fashion.

An important concept to realize about the Connection Machine is that its bit-serial chips are inherently slow. Unlike other supercomputers that push chips to near-ultimate speeds with liquid-cooled circuits and the like, the basic architecture of the CM's bit-serial processors is by comparison rather conservative. It is only the *aggregate* collection of the bit-serial processors (all 64K of them) that form a significant source of processing power.

This has one important ramification: any sequential section of your code should be run on the front-end. That is, all scalar (non-parallel) data should reside on the front-end, and all parallel data should reside on the CM. Never attempt to use the CM as scalar processor —a few bit-serial processors are no match for a VAX 8550. Furthermore, the cost of shipping the data to and from the CM would dominate the computation.

2.7 Paris

Although the CM is inherently a bit-serial computer, the application programmer normally does not program at this level. TMC provides a low-level virtual machine interface —a set of atomic commands collectively named Paris (for PARallel Instruction Set.)

Paris is a library of atomic instructions which the front end host executes to direct the actions of the CM. All the current high level CM languages —C*, *LISP, and CM FORTRAN– eventually become compiled into Paris instructions. Some Paris commands instruct each processor to perform a floating point multiplication, or retrieve data from a neighboring processor; for example, the C/Paris instruction

```
CM_f_cos_2_1L(y, x, 23, 8);
```

instructs the active processors in the CM to calculate the cosine of all floating point **x** values (in parallel) and store the result in the parallel variable **y**, (i.e., $y = \cos(x)$.) The last two arguments indicate that the floating point variables are in IEEE single-precision (23 bits for the mantissa, 8 bits for the exponent.)

The programmer can call Paris instructions directly as functions using the standard C, or FORTRAN compilers normally available on the front end, provided they link with the proper library. We will not provide a full discussion of the Paris command set —there are over 800 functions and the complete Paris 5.0 manual is quite hefty. Suffice it to say that programming directly in Paris provides the most direct control of the CM's hardware.

The Paris programming model presents a large number of homogeneous processors, (similar in power to a conventional microprocessor) each with some local memory and able to address any other processor. Each processor carries out orders from the same instruction stream; optionally, some processors may choose to turn themselves "inactive" and remain idle during the duration of certain instructions.

2.8 Virtual Processors

An elegant feature of the Connection Machine is its availability to utilize virtual processors. The CM can be programmed as if it had many more processors than it physically has. Normally, one associates a single data point (e.g. a grid point) with each processor; although 64K processors may at first appear overwhelming, many scientific applications (specifically 3D models) require literally millions of data points.

The ratio of physical processors present on a specific CM to the number of virtual processors requested by the application is termed the **vp ratio.** Each physical processor can *simulate* more than one *virtual* processors by dividing its memory space into equal parts and sequentially serving the virtual processors.

We have found the support of virtual processors to be an extremely useful feature of the CM. It facilitates programming by eliminating the intermediate-level decomposition (mapping of data points to processors) for regular structures altogether. Furthermore, the use of high vp ratios not only allows one to run extremely large distributed data objects, but actually *enhances* the performance of the CM. This phenomenon is demonstrated in the next section.

3 Floating Point Performance

Fundamental floating point operations were measured on an 8K CM-2 available to us at the University of Colorado in Boulder. Because of the varying configurations possible for CM-2 (8K, 16K, 32K, and 64K processors) it has become a standard practice to scale benchmark results to a full 64K machine when reporting performance figures; this performance scaling is reasonable only for operations involving local (NEWS) communication, or operations involving no communication at all (see McBryan [2], Farhat et. al. [8], and Sato & Swarztrauber [6].)

To capture the performance of the CM-2 architecture we benchmarked the Paris primitives directly. These primitives are the "common denumerators" of all the current CM languages. By using the Paris primitives directly we can report times that reflect the characteristics of the underlying hardware and are independent of the specific higher-level language being used.

The recorded benchmarks for basic floating point operations (*, +, -, /) are shown in Fig. 1, illustrating that most operations perform in the 1.5 - 2.5 GFLOP range. The operation $\vec{x} \Leftarrow \vec{ab}$ denotes a pair-wise parallel multiply, i.e. each processor P_i simply multiplies a_i by b_i. The symbol λ denotes a scalar variable whose value is broadcast throughout the CM during the execution of that instruction.

We note that **higher vp ratios result in higher machine performance.** Here we have an interesting situation where a physical processor *simulating* many virtual processors actually runs more efficiently. Two fundamental reasons are that (1) the cost of interpreting a single Paris instruction is amortized over a larger set of processors, and (2) pipelining of Weitek FPU and Sprint chip can be better utilized.

For most of these functions, their GFLOP performance levels off for vp ratios greater than 8, so it is not completely necessary that we always utilize the machine with the highest vp ratio possible to obtain good floating point performance. This is best illustrated in Fig. 2, where we have plotted the performance curve for several floating point operations over different vp ratios. However, for any code that also uses NEWS communications, we will see that higher vp ratios *do* make an important difference.)

The results also suggest that one should **avoid divisions in their code.** Floating point divisions are expensive —roughly four times slower than a multiplication. If an application requires the repetitive division of a particular parallel variable, it may better to compute with its reciprocal –changing division to multiplications.

3.1 Math Library Functions

Many benchmarks of supercomputers do not contain performance figures for transcendental and irrational functions (`exp`, `sin`, `log`, `sqrt`, etc.) This is unfortunate, since many scientific applications require the extensive use of these primitives. Although these functions

CM BASIC FLOATING POINT OPERATIONS
GFLOP Ratings using various VP configurations

Operation	vp ratio							
	1	2	4	8	16	32	64	128
$\vec{x} \Leftarrow \lambda\vec{x}$	1.75	2.09	2.31	2.43	2.50	2.53	2.55	2.56
$\vec{x} \Leftarrow \lambda\vec{a}$	1.70	2.09	2.30	2.43	2.50	2.53	2.55	2.56
$\vec{x} \Leftarrow \vec{x}/\lambda$	1.50	2.10	2.30	2.43	2.50	2.53	2.55	2.56
$\vec{x} \Leftarrow \vec{x}+\lambda$	1.57	1.82	2.00	2.08	2.13	2.16	2.17	2.17
$\vec{x} \Leftarrow \vec{a}-\lambda$	1.57	1.83	1.98	2.08	2.13	2.16	2.17	2.17
$\vec{x} \Leftarrow \vec{a}+\lambda$	1.57	1.82	1.98	2.08	2.13	2.15	2.17	2.17
$\vec{x} \Leftarrow \vec{x}-\lambda$	1.57	1.82	1.98	2.08	2.12	2.16	2.17	2.17
$\vec{x} \Leftarrow \vec{a}\vec{b}$	1.53	1.79	1.96	2.06	2.10	2.13	2.15	2.15
$\vec{x} \Leftarrow \vec{x}+\vec{a}$	1.40	1.61	1.74	1.81	1.85	1.87	1.88	1.88
$\vec{x} \Leftarrow \vec{a}-\vec{b}$	1.40	1.60	1.72	1.80	1.84	1.86	1.86	1.87
$\vec{x} \Leftarrow \vec{a}+\vec{b}$	1.38	1.60	1.73	1.80	1.84	1.86	1.86	1.87
$\vec{x} \Leftarrow \vec{x}/\vec{a}$	0.66	0.68	0.69	0.70	0.70	0.70	0.70	0.70
$\vec{x} \Leftarrow \vec{a}/\vec{b}$	0.53	0.55	0.56	0.56	0.56	0.56	0.56	0.56

Figure 1: Gigaflop ratings for fundamental floating point operations.

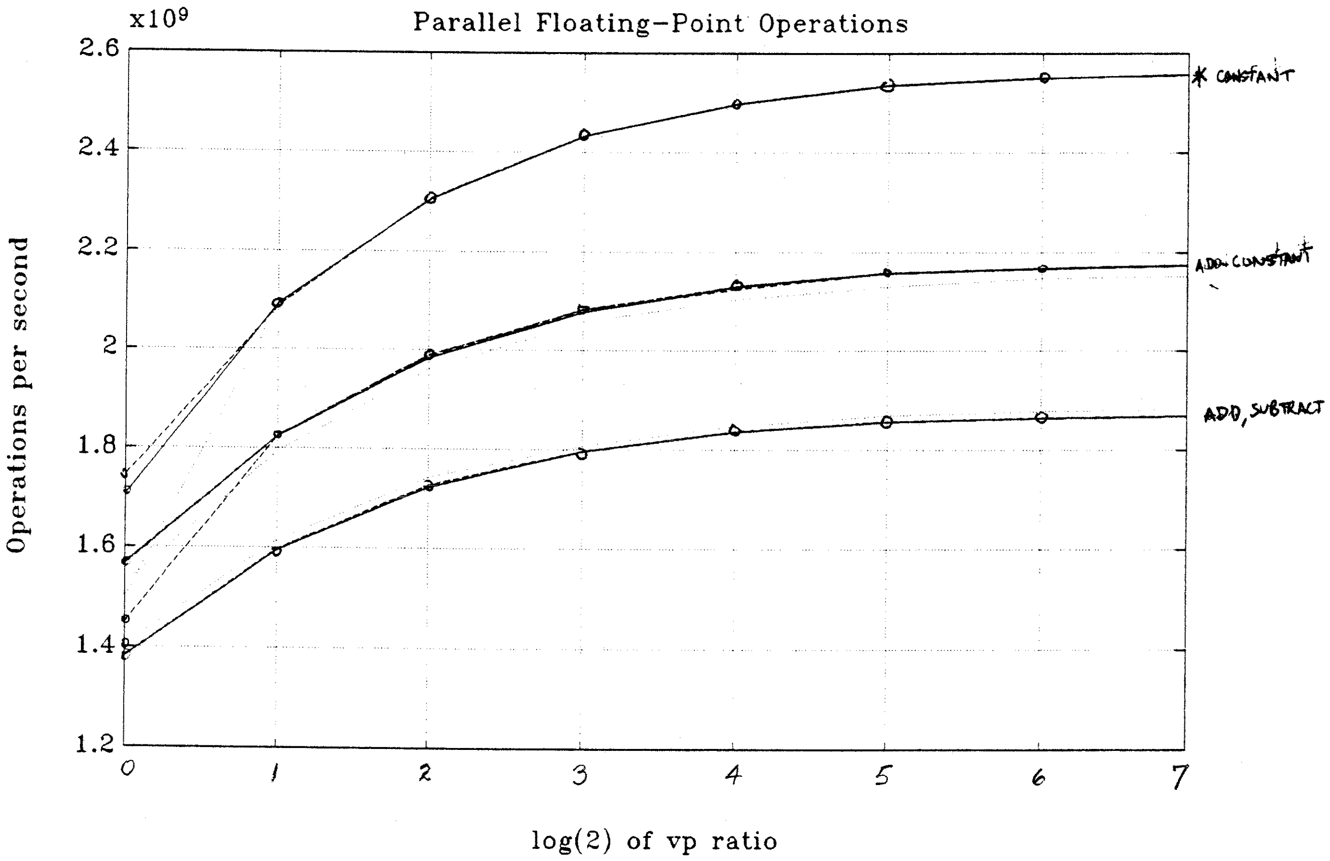

Figure 2: Efficiency improvement of operations with higher virtual processor (vp) ratios.

CM TRANSCENDENTAL PARIS FUNCTIONS
Millions of Operations per Second

Paris transcendental functions	vp ratio							
	1	2	4	8	16	32	64	128
$\vec{y} \Leftarrow \sqrt{(\vec{x})}$	331.8	334.9	336.9	337.6	338.9	338.9	338.9	338.9
$\vec{y} \Leftarrow \sin(\vec{x})$	203.6	205.1	206.1	206.4	206.6	206.9	206.9	206.9
$\vec{y} \Leftarrow \cos(\vec{x})$	240.0	241.8	242.8	242.8	243.1	243.1	243.5	243.5
$\vec{y} \Leftarrow \tan(\vec{x})$	180.4	181.0	181.6	182.0	182.0	182.0	182.0	182.0
$\vec{y} \Leftarrow \exp(\vec{x})$	223.1	230.8	237.0	239.7	241.2	242.4	242.5	242.8
$\vec{y} \Leftarrow \ln(\vec{x})$	277.3	284.4	288.8	290.8	292.3	292.8	293.3	293.3

Figure 3: Paris Math Library Functions

can be built up by the programmer from the basic +, -, * operations, they are not as efficient as the microcoded library counterparts.

The specific values are presented in Fig. 3. Notice that their performance does not improve significantly for increasing vp ratios, unlike the previous set of floating point operations. This gives an indication that the Weitek 3132 FPU is being kept quite busy, even at low vp ratios.

3.2 Triad Operations

A key to obtaining better utilization of the floating point hardware is through the triad operations, introduced in the Paris 5.0 release [2]. The triad operations take the place of two primitive floating point operations and allow the CM to reach speeds of over 5 GFLOPS. Of course, one should attempt to **use triad operations whenever possible.** Unfortunately, at the time of this paper, neither the *LISP or C* compiler parse numerical expressions to generate these triad operations.

We note that most of the triad operations are roughly twice as fast as single operations. In other words, the second operation performed in the triads is almost free. This reinforces the concept that a significant portion of the time is spent moving data back and forth between the bit-serial processors and the Weitek FPU. Keeping intermediate results in the Weitek FPU is the key to good performance.

Triad operations always take four arguments, their format is

```
CM_f_op1_op2_1L(arg1, arg2, arg3, arg4, ...)
```

and perform

```
arg1 = (arg2 op1 arg3) op2 arg4
```

For example,

```
CM_f_mult_const_add(a, b, c, d, 23, 8)
```

performs

```
a = (b * c) + d
```

where `a, b, d` are parallel variables and `c` is a scalar whose value is broadcast during the function execution. The last two arguments, `(23,8)`, specify IEEE single precision (23 bit mantissa, 8 bit exponent, 1 sign bit). This particular Paris function performs near 5.3 GFLOPS. (Of course, we are counting each triad function as two floating point operations.) Fig. 4 summarizes their performance.

We must also comment on the performance of the last two triad operations in Fig. 4. They are *not* much faster than their single-operation counterparts. In fact, `CM_f_mult_add_const_1L` is only 1% faster than performing a `CM_f_add` and a `CM_f_mult_const` separately. This leads us to believe that these two triad functions have not been optimized at all.

[2]With the latest 5.1 release, new optimized Paris instructions will be available that overlap communication and computation.

CM TRIAD PARIS FUNCTIONS
GFLOP Ratings

Paris triad functions	vp ratio							
	1	2	4	8	16	32	64	128
$\vec{x}a + b$	2.42	3.79	4.42	4.81	5.04	5.16	5.27	5.27
$(\vec{x} + a)b$	2.41	3.80	4.42	4.82	5.04	5.16	5.23	5.26
$(\vec{x} + \vec{a})b$	3.13	3.65	4.00	4.17	4.28	4.34	4.36	4.37
$\vec{x}a + \vec{b}$	3.12	3.64	3.98	4.17	4.28	4.43	4.36	4.37
$\vec{x}\vec{a} + \vec{b}$	2.88	3.27	3.50	3.64	3.70	3.74	3.76	3.77
$\vec{x}\vec{a} - \vec{b}$	2.88	3.27	3.50	3.64	3.70	3.74	3.76	3.77
$(\vec{x} + \vec{a})\vec{b}$	2.88	3.27	3.50	3.64	3.70	3.74	3.76	3.77
$(\vec{x} - \vec{a})\vec{b}$	2.88	3.27	3.50	3.64	3.70	3.74	3.76	3.77
$(\vec{x} + \vec{a})b$	1.56	1.90	2.13	2.26	2.33	2.37	2.39	2.40
$\vec{x}\vec{a} + \vec{b}$	1.57	1.91	2.13	2.26	2.43	2.38	2.40	2.40

Figure 4: Performance figures for traid operations.

4 NEWS Communication

Many scientific applications take place on two-dimensional and three-dimensional grids. Finite difference solvers for partial differential equations, for instance, need processors to communicate with their local neighbors on the grid —solving Poisson's equation on a rectangular grid with a Jacobi relaxation scheme is one common example. In such instances, one can configure the Connection Machine as a two-dimensional grid so that processors can communicate efficiently with neighbors directly north, east, west, or south. This type of communication occurs in a significant body of scientific code. In contrast, general (router) communications are intrinsically more difficult to benchmark, since their performance is communication-pattern specific [4]. Some performance studies of general communication in finite element applications have been discussed in [8].

Since the CM is a bit-serial machine, the transfer length of data items are expressed in bits. Lengths of data items to be transferred are limited to 128 bits (for an 8K machine), so we see that the CM has been designed for very fine-grained communication . Furthermore, we see that the start-up costs for this type of communication is minimal —generally, the time required to transfer a floating-point number is equivalent to only 2 to 4 floating point operations. Figure 5 shows the latency encountered for transmitting messages of varying length to a nearest neighbor in the NEWS topology. As one would expect, the relationship between message-length and latency is rather linear, with an almost negligible startup cost. One (single-precision) floating point value can be transferred through a NEWS coordinate by *every* virtual processor in a 1024x1024 grid (on 8K machine) in about 6.9 mS. — this translates to over 1 Gbyte/sec transfer rate for the full 64K machine utlizing NEWS communication.

Figure 6 summarizes the ratio of communication to computation varying NEWS grid sizes. We compare the time required to send a single floating point number to the time required to compute a floating-point multiply.

We note that when implementing NEWS communication, it is **always beneficial to use as high a vp ratio as possible**. This way, many of the "communications" between virtual processors occur actually on the same physical chip as inexpensive memory transfers. For this reason, NEWS communication is always more efficient at high vp ratios, and unlike computational operations, does not taper off. Figure 6 reveals that NEWS communications become efficient at a faster rate than floating point operations. Thus, for example, using 64x16K grid (vp ratio of 128), the cost of a NEWS communication in the north/south direction is only 1.5 times more expensive than an floating-point multiply (which is already efficient at that vp ratio.)

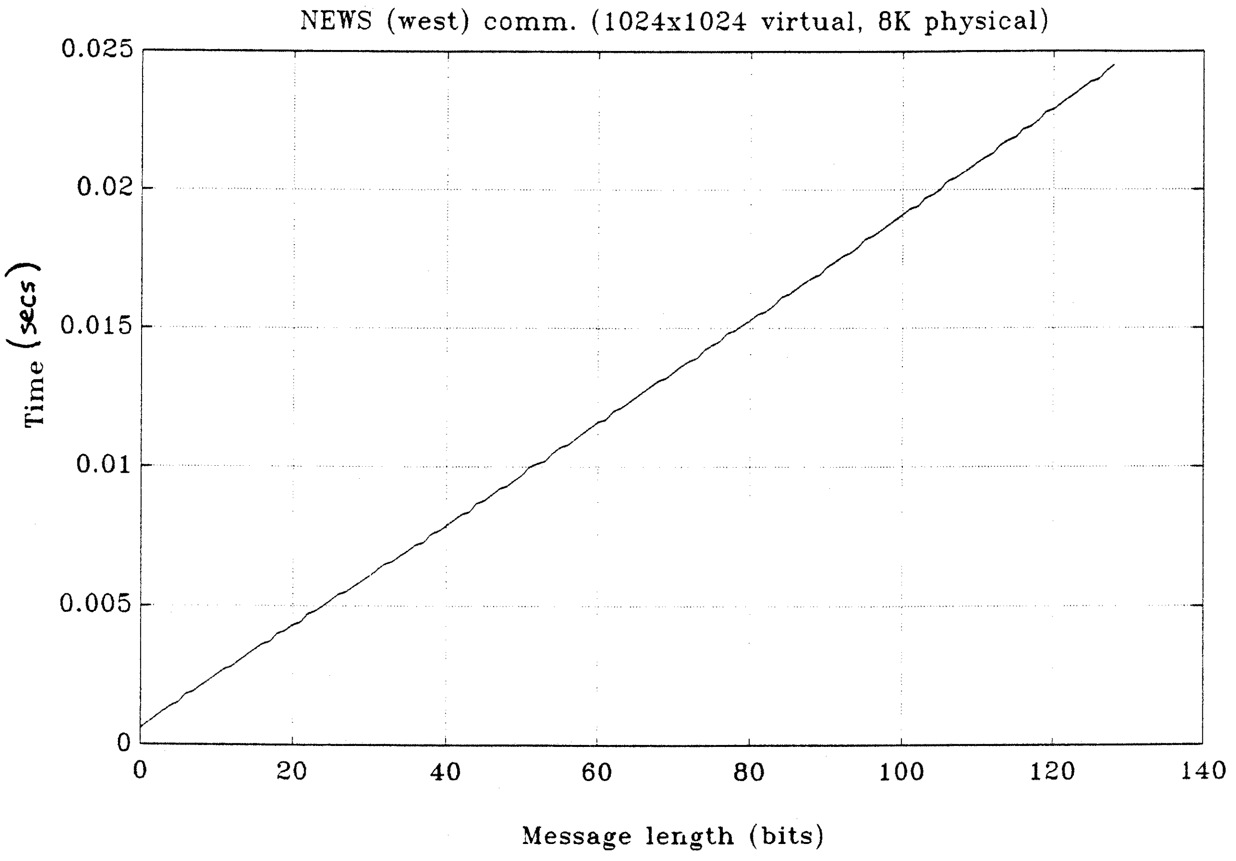

Figure 5: The elapsed time (secs) for *every* virtual processor to complete a West NEWS communication is nearly linear in the message length.

Comm/computation Ratio for various NEWS grids

(NX)

(NY)	64		128		256		512		1024	
128	3.48	3.48	3.04	4.67	2.34	5.05	1.99	5.24	1.75	5.34
256	4.66	3.06	3.25	3.27	2.43	3.36	1.98	3.41	1.78	3.45
512	5.02	2.36	3.36	2.45	2.48	2.46	2.01	2.49	1.79	2.51
1K	5.20	1.99	3.41	2.01	2,49	2.03	2.02	2.02	1.77	2.03
2K	5.31	1.78	3.45	1.78	2.51	1.77	2.03	1.77		
4K	5.34	1.65	3.48	1.66	2.51	1.67				
8K	5.40	1.59	3.46	1.62						
16K	5.44	1.56								

Figure 6: Ratio of communication (time required to transfer a single floating-point value) to computation (time required for a floating point multiply) for various NEWS grid sizes. The left figure in each column represents the east/west axis, the right figure represents north/south axis.

5 Application Benchmark: Primitive Equation Model Test

As an indication of the Connection Machine's performance on a real scientific application, we have studied a standard model routinely used in numerical weather prediction — a finite difference version of an atmospheric primitive equation model. As a set of three dimensional coupled partial differential equations, it is an example of a very important class of problems for which massively parallel computers are thought to be well suited.

Primitive equation models are currently the basis for almost all operational weather prediction. They consist of the Eulerian equations of gas dynamics, modified primarily by replacing of the vertical equation of momentum with the hydrostatic approximation [9]. They have proved remarkably successful in global weather prediction, with significant skill in advanced models now extending out near 10 days.

It was felt that the practical importance and widespread use of the primitive equations would make study of their use and performance on the Connection Machine valuable. Currently, such models are generally formulated spectrally for global and hemispheric domains, and in finite difference form for limited or regional domains. We have chosen to look at the finite difference formulation because of its obvious fined grained and local character. A similar study has been conducted by Tuccillo [7].

Typically, such models consist of a dynamic model (the primary equations) and other routines which approximate physical processes such as radiation, friction and convection. This study only addresses the dynamic model, with plans to study some of the more detailed physics at a later date.

5.1 Dynamic Model

The primitive equation is formulated in a Cartesian grid, with the x direction oriented east, the y direction north and the z direction corresponding to the vertical. It models a large volume of air (the domain consisting of approximately a 5000 by 5000 km square) as a three dimensional lattice. Each lattice point holds six (floating point) variables:

- wind vector: U (West wind), V (North wind), and W (vertical wind)
- pressure: P
- temperature: T
- air density: D

The equations relate these six variables (c_v, f and r are physical constants) by the following system of partial differential equations.

$$\begin{aligned}
\frac{\partial U}{\partial t} &= -U\frac{\partial U}{\partial x} - V\frac{\partial U}{\partial y} - W\frac{\partial U}{\partial z} - \frac{1}{D}\frac{\partial P}{\partial x} + fV && (1) \\
\frac{\partial V}{\partial t} &= -U\frac{\partial V}{\partial x} - V\frac{\partial V}{\partial y} - W\frac{\partial V}{\partial z} - \frac{1}{D}\frac{\partial P}{\partial y} - fU && (2) \\
\frac{\partial T}{\partial t} &= -U\frac{\partial T}{\partial x} - V\frac{\partial T}{\partial y} - W\left(\frac{\partial T}{\partial z} - \frac{Tr}{c_v}\left(\frac{\partial \ln D}{\partial z}\right)\right) && (3) \\
P &= D \cdot r \cdot T && (4) \\
0 &= \frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} + \frac{\partial W}{\partial z} && (5) \\
\frac{\partial P}{\partial z} &= -g \cdot D && (6)
\end{aligned}$$

5.2 Prognostic and Diagnostic Fields

The U, V, and T fields are the "primary" (prognostic) variables; the other "secondary" (diagnostic) variables can be computed directly from the primary fields by the last three equations: (eq. 4) is simply the ideal gas equation (r is the gas constant), (eq. 5) is the incompressibility equation, and (eq. 6) is the hydrostatic equation stating that the pressure at any point is proportional to the weight of the air residing above it.

5.3 Boundary Conditions

The model employs constant boundary conditions for the lateral (y-z and x-z planes) boundaries, and the top horizontal boundary. Since the z=0 plane corresponds to ground level, the vertical wind velocity, W, is zero everywhere on that lower horizontal boundary.

The model is explicit: knowing the values of these six variables at time t we calculate their values at time $t + \Delta t$ by the equations above. Thus, given the initial values of the six variables at each lattice point, the model simply evolves through time in steps of size Δt.

5.4 Numerical Solution

A numerical solution to the Primitive Equations consists of several parts, roughly one for each equation. $\overline{A}_x$ denotes the central difference approximation to $\partial A/\partial x$. The algorithm is rather straightforward: at each time step,

1. `calc_dUdt()`
$$\frac{\partial U}{\partial t} \approx -U\,\overline{U}_x - V\,\overline{U}_y - W\,\overline{U}_z - \frac{1}{D}\,\overline{P}_x + fV$$
(eq. 1)

2. `calc_dVdt()`
$$\frac{\partial V}{\partial t} \approx -U\,\overline{V}_x - V\,\overline{V}_y - W\,\overline{V}_z - \frac{1}{D}\,\overline{P}_y - fU$$
(eq. 2)

3. `calc_dTdt()`
$$\frac{\partial T}{\partial t} \approx -U\,\overline{T}_x - V\,\overline{T}_y - W\left(\overline{T}_z - \frac{Tr}{c_v}\left(\overline{\ln D_z}\right)\right)$$
(eq. 3)

4. `Adams()` :Calculate new values of U, V, and T using Adams-Bashforth time differencing. For example,
$$U^{(i+1)} = U^{(i)} + \left[\frac{3}{2}\left(\frac{\partial U}{\partial t}\right)^{(i)} - \frac{1}{2}\left(\frac{\partial U}{\partial t}\right)^{(i-1)}\right]\Delta t$$

5. `smooth()` : diffuse U, V, T fields by 5-point weighted ($\omega_{i,j}$) grid averaging, e.g.
$$U_{i,j,k} = \omega_{i,j}U_{i,j,k} + (1-\omega_{i,j})\frac{U_{i-1,j,k} + U_{i+1,j,k} + U_{i,j+1,k} + U_{i,j-1,k}}{4}$$

6. `calc_pressure()` : new P value using (eqs. 6 and 4)
$$P_k = P_{top}e^{(g/r\int_{top}^{k} 1/T_z dz)}$$

7. `calc_vertical_wind()`: update W using ($\frac{\partial W}{\partial z}$) (eq. 5)
$$W_k = W_{bottom} + \int_{bottom}^{k} -\left(\frac{\partial U}{\partial x} + \frac{\partial V}{\partial y}\right)_z dz$$

8. `calc_density()`: update D using (eq. 4)
$$D = \frac{P}{rT}$$

5.5 Parallelism

Let N_x, N_y, N_z be the respective axes lengths of the x, y, and z dimensions of the 3D lattice. We make this distinction (as opposed to assuming a cubic N^3 lattice) because the number of vertical points in weather models, Nz, is usually much smaller than the number of data points in the x, y directions.

The Primitive Equation model exhibits reasonable parallelism. Each of the first five functions (`calc_dUdt()`, `calc_dVdt()`, `calc_dTdt()`, `Adams-Bashforth()`, and `smooth()`) can be computed simultaneously at each of the $N_x \times N_y \times N_z$ data points in the 3D lattice.

Calculating the atmospheric pressure and the vertical wind requires, however, computing a vertical integration at *each* data point (see eq. 6, 7). This presents us with two choices

1. Perform calculations on vertical levels sequentially. That is, we only exploit 2D parallelism for each x, y plane. This gives us a running time of $O(N_z)$.
2. calculate these vertical integrals using parallel prefix operators. This would give us an optimal running time of $O(\log N_z)$.

The first choice is easy to program: we only have a one-dimensional loop running the vertical levels. The second choice is a little more involved but gives a better (at least theoretically better) running time.

At the time that we first implemented the Primitive Equation model on the Connection Machine, both C* (version 5.0) and Paris (version 4.3) allowed the use of only two-dimensional grids, thus we had little choice about which method we would code. *LISP was the only language on the Connection Machine that allowed us to use a three-dimensional grid topology. The latest version of Paris (5.0) incorporates the use of N-dimensional grids and we currently porting our code to this latest version. This report discusses the only the 2D C* and C/Paris version together with the 3D *LISP version.

5.6 Timings

The following table presents the time required for an 8K Connection Machine to compute 100 time steps of the Primitive Equation model on a 128x128x32 grid. Although we present the times for the *LISP and C/Paris code side by side, it may be misleading to compare them directly. The *LISP code exhibits 3D parallelism, while the C/Paris exhibits only 2D parallelism by running over the z index sequentially. This 2D restriction for C/Paris has been removed in the latest release, and we will provide new timing results as they become available.

In either case, we can perform 100 time steps on this grid in less than a minute. Assuming that a full 64K Connection Machine could execute a 256x256x64 grid in roughly the same time, these measurements suggest that the Primitive Equation model (including interprocessor communication, but not initial pre and post-processing) would sustain about 1.1 GFLOPs in practice.

Primitive Equation Benchmark
Results for 100 time steps on a 256x256x32 grid

Function	Approx MFLOPS (64K) (Mflops)	*LISP (3D) (secs)	C/Paris 4.3 (2D) (secs)
calc-dUdt	914.4	7.80	7.64
calc-dVdt	1007.7	7.64	8.39
calc-dTdt	1053.0	5.61	5.89
3 * Adams	2077.7	3.03	2.63
3 * smooth	799.0	9.45	14.47
calc-pressure	1798.2	9.31	3.73
calc-density	894.4	0.94	0.83
vertical-wind	487.4	8.57	9.81
Total		52.35	52.64

6 Future CM Enchancements

At the time of composing this paper, the following is a list of proposed enhancements to the Connection Machines hardware and programming environment:

- **New optimized instructions:** with the relase of Paris 5.1, new instructions will be introduced that further optimize floating point computations by essentially **overlapping nearest-neighbor communication with computation.** For example, an instruction such as `CM_f_news_add_1L` will retrieve a data value from the processor's east neighbor and add it to an intermediate result in *one* atomic operation. Such computations are typical of calculating a central difference, a common operation in finite difference codes for solving partial differential equations, for example.

- **New routines in microcode:** Furthermore, Paris 5.1 will include some optimized numerical routines such as the Fast Fourier Transform (**FFT**), and **matrix multiplication** supplied in microcode.

- **Upgrade to double-precision hardware:** Currently, the CM hardware (Weitek 3132) supports only single-precision floating point representation. Double-precision (in fact *arbitrary* precision) is available by performing it on the bit-serial processors (without the FPUs), but it is about two orders of magnitude slower. Future plans of TMC include the upgrade to double-precision floating point units for the CM.

- **More memory:** The memory capacity of the CM will from 256K RAM chips to 1 MG RAM chips resulting in the storage of 2 Gigabytes for a 64K processor machine (each physical processor's memory space will increase from 8K bytes to 32K bytes.) This will allow the use of even higher vp ratios, and the storage of even larger distributed data structures.

- **New languages:** Thinking Machines Inc. will soon release their new CM FORTRAN, a new high-level language following many of the proposed standards of FORTRAN 8x. The introduction of CM FORTRAN will certainly increase interest from the supercomputing application programmers. We will supply benchmarks for the performance of the CM FORTRAN compiler as it becomes available.

7 Conclusions

Can a massively parallel architecture compete with traditional vector supercomputers? These preliminary benchmarks suggest that computers such as the Connection Machine can be serious contenders.

Our benchmark results show that the Connection Machine is capable of sustaining speeds of several GFLOPs in practice: 1.5-2.5 GFLOPS for single-precision floating point operations, and 2.5-5.3 GFLOPS for triad operations. For a real scientific application -even one that does not parallelize ideally, such as the atmospheric primitive equation forecast- sustained speeds of over 1 GFLOP are achievable. Similar studies have been carried out by Sato & Swarztrauber [6], running a shallow-water equation model at 1.7 GFLOPs on the CM, and by McBryan [2], demonstrating a Conjugate Gradient algorithm running at 3.8 GFLOPs.

Throughout this paper we have provided suggestions as to how the application programmer can best utilize the CM's hardware for floating point performance. We briefly summarize the major points :

- Use a **high vp ratio** as possible, especially for near-neighbor (NEWS) interprocessor communication. Problems must be *large* — containing millions of data points — to utilize the CM hardware efficiently. The Connection Machine gets its supercomputer power not from a few fast processors, but from many, relatively slow, chips working in parallel. Therefore, there must be enough work to keep all processors busy. Our performance characteristics show that the CM performs even better at high *virtual processor* (vp) ratios.

- Utiliize **triad operations** (Paris 5.0 and later) whenever possible. Many of the triad operations effectively double the CM's floating point performance by optimizing use of the Sprint chip and Weitek FPU. A key issue concerning the success of CM compilers will be their ability to parse numerical expressions into triad operations. The *LISP compiler employs such optimization; however, the current C* compiler requires this parsing to be done by hand.

- **Optimize use of the CM stack.** This also should be an issue for the compiler, but nevertheless is a good source for speed improvement. Consider the following psuedocode (similar to C*) for retrieving a value from the North neighbor,

  ```
  y = NORTH(x);
  ```

  The following is a typical breakdown of the compiler-generated code, with each step corresponding to a Paris instruction:

```
(1) allocate temp variable, T, on CM stack.
(2) CM_get_from_north(T, x)
(3) CM_move(y,T)
(4) pop variable T off the CM stack.
```

whereas the single Paris function

```
        CM_get_from_north(y,x)
```

accomplishes the same thing. It is clear that successful compilers for the CM will employ considerable optimization.

These preliminary benchmarks are certainly not comprehensive of the complete CM's capabilities —with over 800 Paris functions, a thorough examination of each function would result in a rather lengthy paper. Furthermore, we did not discuss other features of the CM hardware, such as the Datavault disk I/O, or its graphics capabilities. These topics will be dealt with in later papers.

Current work is underway to continue benchmarking many of the parallel prefix operators present in Paris, together with studying performance attributes of global communication. Further work will examine the performance of the upcoming Paris 5.1 instructions for optimized communication/computation, and the new microcode numerical routines. Of particular interest will be the performance of the new CM FORTRAN compiler. These new benchmark figures will be presented as results become available.

References

[1] G. S. Almasi, A. Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings, Reading, Mass., 1989.

[2] O. McBryan, “New Architectures: Performance Highlights and New Algorithms”, *Parallel Computing*, Vol. 7, No. 3, (1988) pp. 447-499.

[3] W. D. Hillis, *The Connection Machine*, MIT Press, Cambridge, Mass., 1985.

[4] Thinking Machines Corporation, “Connection Machine Model CM-2 Technical Summary”, *Technical Report Series*, HA87-4.

[5] Thinking Machines Corporation, “Communications Architecture in the Connection Machine System”, *Technical Report Series* , HA87-3.

[6] R. K. Sato, P. N. Swarztrauber, “Benchmarking the Connection Machine 2”, *Proceedings of the Supercomputing Conference*, Orlando, FL, November 14-18, 1988.

[7] J. J. Tuccillo, “Numerical Solution of the Primitive Equations on the Connection Machine”, Proceedings of the Supercomputing Conference, Santa Clara, CA, May 1989.

[8] C. Farhat, N. Sobh, and K. C. Park, “Dynamic Finite Element Simulations on the Connection Machine,” *Internation Journal of High Speed Computing*, Vol. 2., 1989.

[9] G. J. Haltiner, R. T. Williams, *Numerical Prediction and Dynamic Meteorology*, John Wiley & Sons, New York, 1980.