Iaval Research Laboratory
Iashington, DC 20375-5000



NRL Report 9289



# Benchmarking the Connection Machine

M. A. YOUNG

*Signal Processing Branch*
*Acoustics Division*

November 21, 1990

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188



**1. AGENCY USE ONLY** *(Leave blank)*

**2. REPORT DATE**
November 21, 1990

**3. REPORT TYPE AND DATES COVERED**
Final

**4. TITLE AND SUBTITLE**
Benchmarking the Connection Machine

**5. FUNDING NUMBERS**

**6. AUTHOR(S)**
Michael A. Young

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Naval Research Laboratory
Washington, DC 20375-5000

**8. PERFORMING ORGANIZATION REPORT NUMBER**
NRL Report 9289

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

**11. SUPPLEMENTARY NOTES**

**12a. DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for public release; distribution unlimited.

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*

This paper presents the results of benchmarking the Connection Machine CM-2 with an efficient, highly parallel implementation of the Livermore Loops. Re-coded in CM Fortran, only a few of the kernels required significant modification. The loops have been one of the most often run benchmark suites and form a good testbed for parallel machines due to the many varied computational structures. Analysis and discussion of the single and double precision Mflop rates are presented for large vector lengths on a 32K CM-2. For applications involving large vector lengths, a large amount of computation, and minimal general communication the CM-2 performs extremely well. Gigaflop performance was attained on the computationally intensive kernels.

procedures (KR)

**14. SUBJECT TERMS**
Livermore loops
Benchmarking
Parallel processing
Performance
Connection Machine

**15. NUMBER OF PAGES**
25

**16. PRICE CODE**

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | UL |

NSN 7540-01-280-5500

Standard Form 298 (Rev 2-89)
Prescribed by ANSI Std Z39-18
298-102

## CONTENTS

# BENCHMARKING THE CONNECTION MACHINE

## 1. INTRODUCTION

Performance of various computers is compared by running programs across different machines and comparing execution times (*benchmarking* the computers). Scientific or engineering benchmarks are usually measured in Mflops (millions of floating point operations per second). The current state of benchmarking supercomputer architectures is not very clear. Performances of a specific supercomputer on various benchmarks may vary greatly, making the judgment extremely difficult. Naturally, certain benchmarks may be more suited to a particular machine's architecture. Running standard benchmarks, without modification, across various supercomputers can show the effectiveness of the compilers in using the available resources. This allows comparison with an optimized code implementation.

To measure the true capability of an architecture may require some restructuring of the code. This customization for a given machine can provide dramatic increases in performance. Automatic vectorizing compilers help to alleviate this task of customization but presently cannot look at whole routines. The performance of highly parallel machines is greatly dependent on communication and the overall communication network of a particular code. It is important to look closely at the overall problem/algorithm rather than to make a line-by-line conversion [1].

Many installations develop their own set of benchmarks, specific to the particular institution specialization, and send these to prospective vendors to compare various machines. Kernels are excerpts extracted to be representative of the programs run at a given installation. This report measures the performance of the Connection Machine model CM-2, manufactured by Thinking Machines Corporation, relative to other supercomputers and provides some insight into its strengths and weaknesses. The Livermore Loops were selected as the representative kernels to benchmark the CM-2.

Although there is not a universally accepted set of benchmark programs, the Livermore Loops are widely used [2]. The Livermore Loops consist of Fortran kernels that Lawrence Livermore National Laboratory (LLNL) extracted from actual production codes of a number of representative application areas [3,4]. Figure 1 lists the kernels.

Machine dependencies, such as input/output (I/O) and memory management, are not present in the Livermore Loops. Originally developed to benchmark serial machines, the kernels also form a good test set for parallel machines. As reported by Frank McMahon of LLNL, the kernels are good predictors of the actual production performance [4].

Manuscript approved June 22 1990.

| Kernel | Title |
|---|---|
| 1 | Hydro fragment |
| 2 | ICCG excerpt (Incomplete Cholesky - Conjugate Gradient) |
| 3 | Inner Product |
| 4 | Banded Linear Equations |
| 5 | Tridiagonal Elimination, below diagonal |
| 6 | General Linear Recurrences |
| 7 | Equation of State fragment |
| 8 | A.D.I. (Alternating Direction Implicit) Integration |
| 9 | Integrate Predictors |
| 10 | Difference Predictors |
| 11 | First Sum |
| 12 | First Difference |
| 13 | 2-D Particle in Cell |
| 14 | 1-D Particle in Cell |
| 15 | Casual Fortran |
| 16 | Monte Carlo Search Loop |
| 17 | Implicit Conditional Computation |
| 18 | 2-D Explicit Hydrodynamics fragment |
| 19 | General Linear Recurrence Equations |
| 20 | Discrete Ordinates Transport |
| 21 | Matrix Product |
| 22 | Planckian Distribution |
| 23 | 2-D Implicit Hydrodynamics fragment |
| 24 | Find location of first minimum in array |

**Fig. 1 — Kernel List**

## 2. THE CONNECTION MACHINE

The Connection Machine model CM-2 is a data parallel machine made up of 64K (K = 1024) processors. The CM-2 works best on large amounts of data because it is a Single Instruction Multiple Data (SIMD) computer, which means an instruction may operate in parallel on many data elements. Another approach to parallel processing is Multiple Instruction Multiple Data (MIMD) architectures, which can have multiple independent instructions operating on different data elements. A SIMD architecture like the CM-2 is much easier to program than a MIMD architecture because SIMD does not require the control synchronization needed by MIMD architectures.

Each CM processor is a 1-bit-wide custom processor with 64K, 256K, or 1024K bits of memory. It has an arithmetic logic unit (ALU) and a router interface to perform communication among the processors. Communication among processors is done by a high-speed routing network, and a much faster grid communication device is used for nearest-neighbor communication. The router allows any processor to perform data transfer between itself and any other processor. Collisions occur when several processors send messages to the same processor. In this case there are message-combining operations (bitwise logical, numerically largest, or integer sum of all messages). Each CM-2 processor chip contains one router node serving the 16 data processors on the chip. For a fully configured CM-2, each router node is connected to 12 other router nodes forming a 12-dimensional hypercube connecting the 4K processor chips. Within a CM processor chip, full crossbar interconnections are provided.

All program development and execution takes place on the front end (Symbolics, DEC VAX, or Sun 4). Multiple front-end bus interfaces (FEBIs) from the front end allow, through the Nexus (a bidirectional switch), multiple users to access separate sections of the CM-2 (one per section of 8K or 16K processors). The number of simultaneous users depends on the number of FEBIs (maximum of 4). Symbolics is a single-user machine.

The commands that direct the CM-2 are issued from the front end. These commands make up the Parallel Instruction Set (Paris), which is similiar to the assembly language instruction set of a standard computer. The Paris instructions from the front end are broken down by the CM microsequencer into low-level data processor operations. Each parallel processing unit or section, either 8K or 16K processors, has its own sequencer. Depending on the overall machine size, a section has either 8K or 16K processors. A 64K machine would have four sections of 16K processors, and a 32K machine would have four sections of 8K processors. On the 32K machine a user could have 1, 2, or 4 sequencers corresponding to 8K, 16K, or 32K processors. The configuration of the sequencers is dependent on the Nexus, which can be quickly reconfigured.

The CM-2 may also have a floating point accelerator (FPA) option (single or double precision) that increases the rate of floating point calculations by more than a factor of 20. The coprocessors, manufactured by Weitek, consist of a memory interface unit and a floating point execution unit. Each coprocessor is assigned to two CM-2 processor chips (32 physical processors). The floating point execution chip can store 32 values of a given precision. The chip is used for operations such as integer multiply, floating point multiply, and addition. Two memory references are required for each 64-bit floating point processor. The extra memory reference is required since the floating point processor's data path is only 32 bits wide. A large degradation in performance results if the data type does not match the associated floating point processor data type. A 32-bit (64-bit) floating point data type uses 23 (52) bits for the significand, 8 (11) bits for the exponent, and 1 (1) bit for the sign. Douglas et al. [5] thoroughly discuss the CM-2 data processor architecture.

A physical processor's memory may be partitioned and serially executed to simulate a machine with more processing nodes than the actual number of physical processors. This virtual processor (VP) mechanism is transparent to the user [5,6]. For example, on a machine with 64K physical processors, a VP set of size (1024,1024) would require that each physical processor simulate 16 virtual processors. This VP set is said to have a VP ratio of 16 and have 16*64K = 1024K virtual processors. The maximum VP ratio is dependent on the physical processor memory for a particular machine. The use of virtual processors can dramatically increase the performance of floating point operations by allowing the floating point chips to pipeline. Douglas et al. [5] show that a rate of 2600 Mflops would be expected for a 32-bit floating point multiply if the physical processors would cycle through their virtual processors one at a time. Since the memory and float bus are idle at different stages of the multiply, they can be used to start the next virtual processor, causing pipelining and increasing the Mflop rate to 4300. Reference 6 gives more details of the CM-2 hardware.

The programming environment consists of three high-level languages *LISP, C*, and CM Fortran. *LISP and C* are parallel extensions of Common LISP and the C programming language, respectively. CM Fortran [7] consists of the majority of Fortran 77 with some of the array extensions and removed extensions outlined in the draft S8 of the ANSI Fortran 8x standard (x3.9–198x) [8,9]. All three languages compile into Paris. The programming environment also includes three interfaces for calling Paris (LISP/Paris, C/Paris, and Fortran/Paris) along with library packages such as *Render (a graphics processing package) and CMSSL (a scientific subroutine library). For a program written in C/Paris, standard C code directs the front-end (serial) operations whereas the Paris calls direct the handling of data residing on the CM-2 and any transfers of data between the CM-2 and the front end. LISP/Paris and Fortran/Paris are similar interfaces except the serial operations are programmed in Common LISP and Fortran 77, respectively. The Livermore Loops are coded in release 0.7 of CM Fortran.

## 3. CM FORTRAN

CM Fortran [7] consists of a mixture of serial and parallel array operations. Serial operations are executed by the front-end computer by using its own memory and CPU. The parallel operations are executed on the CM-2 where each processor concurrently executes its own data point. Multidimensional arrays are allocated on the CM-2, one element per processor.

Major array features that have been adapted from the proposed 8x standard [8] include array assignment, constructors, and sections (Fig. 2). The *where* statement and *block where* construct, Fig. 3, are also featured. These allow you to operate conditionally on array elements depending on their values. Especially useful in CM Fortran are the scan operations, or parallel prefix operations, *sum* and *spread* (Figs. 4 and 5), where the dimension the scanning is done across is specified. The advantage of these scanning operations is that while communicating, the processors perform a combining operation (add, min, max, ...). *Sum* is a scan-with-addition combining operation, and *spread* is a special scan that adds a dimension by copying data. Other useful functions are *eoshift* (end off *shift*) and *cshift* (circular *shift*), which shift elements of an array along a specified dimension (Fig. 6). The following declarations are assumed in Figs. 2–8 below which compare code written in both Fortran 77 and CM Fortran.

```
real a(n),b(n),c(m,n),d(m,n)
integer i1(n),i2(n)
```

**Fortran 77**

```
do 10 i=1,m
do 10 j=1,n
d(i,j) = c(i,j) * 30.0
10 continue
```

**CM Fortran**

```
d = c * 30.0
```

Fig. 2 — *Do Loops*

**Fortran 77**

```
do 10 i=1,n
if (a(i) .ne. 0.0) then
b(i) = 3.0/a(i)
else
b(i) = 0.0
10 endif
```

**CM Fortran**

```
where (a .ne. 0.0)
b = 3.0/a
elsewhere
b = 0.0
endwhere
```

Fig. 3 — *Where*

**Fortran 77**

```
q = 0.0
do 10 i=1,n
10 q = q + a(i) * b(i)
```

**CM Fortran**

```
q = sum(a*b)
```

Fig. 4 — *Sum*

The removed extensions that have been implemented include vector-valued subscripts and the *forall* statement. The *forall* statement can do indirect addressing and scattering of data along with segmented scans, in which partial results are computed. Compilation of the *forall* statement generates a get (send) router communication if the addressing is done on the right-hand (left-hand) side of the assignment statement. Figs. 7 and 8 show this get and send communication, respectively. The send router communication is approximately twice as fast as the get router communication.

**Fortran 77**

```
do 10 i=1,n
do 10 j=1,m
10 c(j,i) = a(i)
```

**CM Fortran**

```
c = spread(a,1,m)
```

Fig. 5 — *Spread*

**Fortran 77**

```
do 10 i=1,m
do 20 j=1,n-2
d(i,j) = c(i,j+2)
20 continue
d(i,n-1) = c(i,1)
d(i,n) = c(i,2)
10 continue
```

**CM Fortran**

```
d = cshift(c,dim=2,shift=2)
```

Fig. 6 — *Cshift*

**Fortran 77**

```
do 10 i=1,n
a(i) = d(i1(i),i2(i))
10 continue
```

**CM Fortran**

```
forall (i=1:n) a(i) = d(i1(i),i2(i))
```

Fig. 7 — *Forall (get)*

**Fortran 77**

```
do 10 i=1,n
a(i1(i),i2(i)) = d(i)
10 continue
```

**CM Fortran**

```
forall (i=1:n) a(i1(i),i2(i)) = d(i)
```

Fig. 8 — *Forall (send)*

## 4. CODING PROCEDURES

On serial computers, the Livermore Loops are executed without modification. The massively parallel architecture of the CM-2 requires that the loops be explicitly changed to use the array features of CM Fortran. The original Fortran kernels were converted to CM Fortran (see the Appendix) and in most cases the same algorithm was used. Most of the code conversion involved a simple mapping of each element of a vector or matrix to a virtual processor and then performing simultaneous operations on these elements as in Figs. 2–6. A few of the kernels (5, 11, 19, 23) involved recurrence and were coded with a cyclic reduction algorithm [10]. With recurrences it becomes more difficult to generate an O(1) (i.e., a single array statement) solution, so a cyclic reduction method of O(logn) was used to increase performance for the sequential O(n) problem. This involved the only major change to the algorithmic structure of the kernels (5, 11, 19, 23). Fig. 9 shows this cyclic reduction technique.

**Fortran 77**

```
x(1) = a(1) * x(0) + d(1)
do 1 j=2,n
x(j) = a(j) * x(j-1) + d(j)
1 continue
```

**CM Fortran**

```
x = d
x0 = x(0)
do i=1,log2n
j = -(2**(i-1))
x = x + a * eoshift(x,1,j,x0)
a = a * eoshift(a,1,j,0.0)
enddo
x(n) = x(n) + x0 * a(n)
```

Fig. 9 — *Cyclic Reduction*

General communication, handled by the router, can be a bottleneck when implementing code on the CM-2. Programs that transfer or access data randomly would use general communication, whereas programs with a more structured communication involving neighboring processors would use the much quicker grid communication. The best performance will usually be obtained by minimizing router communication and using grid communication when needed. Grid communication is approximately 16 times more efficient than general communication [7]. The particular communication that will be used for a CM Fortran statement can be found by inspecting the Paris commands in the assembler output generated by the compiler.

The *cshift* and *eoshift* commands under compilation generate either a general communication or a series of grid communications, depending on whether the distance of the shift is less than 17. The communication costs involved in the assignment of array sections are similiarly dependent on the offset involved. The following declarations are assumed for Fig. 10 which shows when interprocessor communication (either general or grid) is required.

```
real a(16384),b(16384),c(16000)
```

| Statement | Communication |
|---|---|
| a = b | cost = 0 no communication |
| a(1:16000) = c | cost = 0 no communication |
| a(17:16016) = c | cost = 16 grid communication |
| a(1:16000) = b(2:16001) | cost = 1 grid communication |
| a(1:16000) = b(17:16016) | cost = 16 grid communication |
| a(1:16000) = b(18:16017) | cost = 17 general communication |

Fig. 10 — *Communication*

A general data exchange routing routine was required to perform the communication required in kernels 13 and 14. In kernel 21 (matrix multiply), the dimensions of the vy matrix were increased to put an element in each virtual processor, thus providing a better evaluation of the CM-2 on large matrix multiplication.

## 5. RESULTS

Tables 1, 2, 3, and 4 list the single (32-bit) and double (64-bit) precision Mflop rates for a 16K and 32K CM-2 with 64-bit FPA and 64K bits of memory per processor. Results are presented for different VP ratios. Assignment of weights to floating point operations was made according to McMahon [4], '+, −, * = 1; /, sqrt = 4; exp, sin, etc. = 8; if(x.rel.y) = 1.' The extra computation required for the cyclic reduction algorithm used in kernels 5, 19, and 23 was not counted in computing a Mflop rating. A table entry denoted by a '*' indicates that the VP ratio could not be raised to this level because of insufficient memory.

The highest Mflop performance occurred for kernels 1, 3, 7, 8, 9, 10, 12, 15, 18, 21, and 22, which include the most computationally intensive kernels. Although the computational resources remain fixed, efficiency increases for larger problems, as reflected by the higher Mflop rate vs VP ratio. This results from filling up the pipeline of the FPAs. However, efficiency of kernels involving recurrence does not improve across VP ratios because of a communication bottleneck. Communication-bound problems fare poorly on the CM-2, and ways to minimize router communication must be explored.

Presently, the performance of the recurrences showed little if any improvement as the VP ratio increased because of a communication bottleneck. It is possible to code Kernel 11 with a single Paris scan instruction. We tried to improve the other somewhat more involved recurrences (5, 19, 23) by using Paris scans. Although being very efficient for small vector lengths (<1000), this effort became impractical for larger vector lengths. A multiply scan is needed in which the number of consecutive multiplies grows linearly with the vector length. To perform these multiplies would require more bits in the exponent field thus creating a nonuniform data type that would run dramatically slower (as discussed in Section 2.).

Table 1 —

16K CM-2 **(64-bit hardware)** Single Precision Performance in Mflops

| Kernel | VP ratio | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| 1 | 102.33 | 179.58 | 288.33 | 369.09 | 473.73 | 544.11 | 586.88 |
| 3 | 92.05 | 171.89 | 299.30 | 468.06 | 655.39 | 813.60 | 915.39 |
| 5 | 1.44 | 1.50 | 1.47 | 1.48 | 1.23 | 1.21 | 0.74 |
| 7 | 135.53 | 250.97 | 340.94 | 471.29 | 534 28 | 584.43 | 613.80 |
| 8 | 260.18 | 283.12 | 321.48 | 348.08 | 356.12 | * | * |
| 9 | 558.21 | 777.22 | 863.48 | 906.63 | 938.14 | 941.45 | 946.45 |
| 10 | 212.53 | 231.29 | 239.78 | 247.82 | 247.82 | 250.98 | 252.78 |
| 11 | 1.35 | 1.43 | 1.43 | 1.41 | 1.35 | 1.19 | 0.70 |
| 12 | 100.81 | 170.21 | 220.54 | 242.73 | 259.26 | 273.08 | 278.88 |
| 13 | 0.36 | 0.36 | 0.32 | 0.31 | 0.27 | 0.24 | * |
| 14 | 2.51 | 2.68 | 2.23 | 1.15 | 1.05 | 1.01 | * |
| 15 | 100.11 | 155.30 | 162.62 | 165.88 | 163.66 | 170.14 | 170.54 |
| 18 | 274.46 | 366.77 | 431.21 | 487.43 | 508.03 | * | * |
| 19 | 1.93 | 2.16 | 2.08 | 1.85 | 1.72 | 1.13 | 1.11 |
| 21 | 106.86 | 164.21 | 213.22 | 266.92 | 312.13 | 339.02 | 347.49 |
| 22 | 426.88 | 467.36 | 484.02 | 491.89 | 498.05 | 501.43 | 501.70 |
| 23 | 5.15 | 4.76 | 4.29 | 2.64 | 2.59 | * | * |
| 24 | 37.93 | 60.24 | 70.47 | 84.01 | 101.60 | 99.29 | 117.82 |

* Memory exceeded (64K bits per processor)

Table 2 —

16K CM-2 **(64-bit hardware)** Double Precision Performance in Mflops

| Kernel | VP ratio | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| 1 | 60.77 | 105.32 | 170.18 | 214.40 | 271.24 | 316.39 | 351.33 |
| 3 | 46.71 | 84.38 | 143.63 | 234.76 | 325.61 | 406.10 | 456.11 |
| 5 | .72 | .80 | .85 | .89 | .80 | .74 | .49 |
| 7 | 68.44 | 123.41 | 170.52 | 234.18 | 267.52 | 290.39 | 307.17 |
| 8 | 130.43 | 141.29 | 163.56 | 173.23 | * | * | * |
| 9 | 281.12 | 388.44 | 431.23 | 451.10 | 466.23 | 470.17 | 472.53 |
| 10 | 105.56 | 119.40 | 119.64 | 122.30 | 124.87 | 126.45 | * |
| 11 | .72 | .76 | .81 | .83 | .87 | .74 | .45 |
| 12 | 59.25 | 101.22 | 129.93 | 146.12 | 155.02 | 112.89 | 164.98 |
| 13 | 0.22 | 0.23 | 0.22 | 0.22 | 0.21 | * | * |
| 14 | 1.51 | 1.73 | 1.46 | .82 | .72 | * | * |
| 15 | 49.01 | 77.24 | 78.33 | 82.48 | 83.29 | 83.92 | * |
| 18 | 137.41 | 183.39 | 216.19 | 243.91 | * | * | * |
| 19 | .98 | 1.18 | 1.19 | 1.07 | 1.08 | .71 | .71 |
| 21 | 52.12 | 81.82 | 106.62 | 132.91 | 156.81 | 167.72 | 178.29 |
| 22 | 245.39 | 270.25 | 274.13 | 272.64 | 274.03 | 275.31 | 275.92 |
| 23 | 2.66 | 2.67 | 2.43 | 1.58 | * | * | * |
| 24 | 19.73 | 30.34 | 36.13 | 44.02 | 52.87 | 55.71 | 62.31 |

* Memory exceeded (64K bits per processor)

Table 3 —

32K CM-2 (**64-bit hardware**) Single Precision Performance in Mflops

| Kernel | VP ratio | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| 1 | 204.38 | 358.26 | 569.25 | 734.75 | 939.04 | 1066.65 | 1183.90 |
| 3 | 184.86 | 338.68 | 598.95 | 938.73 | 1297.52 | 1624.29 | 1822.44 |
| 5 | 2.84 | 2.98 | 2.96 | 2.93 | 2.51 | 2.46 | 1.49 |
| 7 | 267.56 | 498.39 | 675.32 | 939.02 | 1068.68 | 1157.24 | 1224.87 |
| 8 | 519.41 | 564.85 | 641.62 | 692.26 | 710.66 | * | * |
| 9 | 1117.82 | 1[illegible]50.83 | 1725.25 | 1810.51 | 1873.20 | 1880.30 | 1889.40 |
| 10 | 421.17 | 465.71 | 475.55 | 492.63 | 498.73 | 499.83 | 501.19 |
| 11 | 2.70 | 2.90 | 2.86 | 2.80 | 2.76 | 2.33 | 1.38 |
| 12 | 203.80 | 342.11 | 439.40 | 484.84 | 519.71 | 546.87 | 553.04 |
| 13 | 0.71 | 0.72 | 0.62 | 0.60 | 0.53 | 0.46 | * |
| 14 | 5.01 | 5.27 | 4.38 | 2.30 | 2.08 | 2.02 | * |
| 15 | 198.88 | 310.29 | 318.86 | 330.38 | 337.95 | 339.71 | 341.93 |
| 18 | 546.89 | 731.00 | 860.75 | 978.16 | 1020.33 | * | * |
| 19 | 3.82 | 4.30 | 4.15 | 3.61 | 3.42 | 2.24 | 2.21 |
| 21 | 210.93 | 326.01 | 424.66 | 531.22 | 622.76 | 676.76 | 702.77 |
| 22 | 848.65 | 932.25 | 967.94 | 989.63 | 993.32 | 1002.16 | 1003.26 |
| 23 | 10.14 | 9.68 | 8.41 | 5.21 | 5.11 | * | * |
| 24 | 75.59 | 118.91 | 138.26 | 167.69 | 201.90 | 211.36 | 233.17 |

* Memory exceeded (64K bits per processor)

Table 4 —

32K CM-2 (**64-bit hardware**) Double Precision Performance in Mflops

| Kernel | VP ratio | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| 1 | 121.27 | 211.10 | 341.46 | 430.21 | 543.34 | 630.71 | 701.67 |
| 3 | 92.70 | 168.55 | 298.91 | 469.99 | 650.88 | 811.45 | 912.12 |
| 5 | 1.45 | 1.61 | 1.72 | 1.77 | 1.58 | 1.48 | .95 |
| 7 | 135.69 | 249.59 | 339.84 | 468.25 | 534.91 | 579.69 | 612.21 |
| 8 | 259.99 | 282.40 | 324.42 | 344.86 | * | * | * |
| 9 | 561.05 | 771.64 | 862.82 | 904.15 | 934.64 | 940.47 | 943.49 |
| 10 | 210.97 | 234.70 | 238.77 | 245.10 | 249.06 | 252.03 | * |
| 11 | 1.42 | 1.54 | 1.64 | 1.67 | 1.72 | 1.47 | .88 |
| 12 | 118.39 | 201.02 | 260.17 | 289.46 | 310.12 | 324.30 | 330.92 |
| 13 | 0.42 | 0.45 | 0.43 | 0.42 | 0.40 | * | * |
| 14 | 3.02 | 3.38 | 2.89 | 1.61 | 1.43 | * | * |
| 15 | 96.23 | 154.83 | 158.77 | 162.24 | 165.11 | 167.38 | * |
| 18 | 273.24 | 368.64 | 433.22 | 487.65 | * | * | * |
| 19 | 1.98 | 2.33 | 2.36 | 2.16 | 2.15 | 1.41 | 1.40 |
| 21 | 104.88 | 161.04 | 212.74 | 266.50 | 310.78 | 334.32 | 355.65 |
| 22 | 490.23 | 540.67 | 547.43 | 543.18 | 546.44 | 551.22 | 551.88 |
| 23 | 5.27 | 5.32 | 4.84 | 3.13 | * | * | * |
| 24 | 38.43 | 60.77 | 71.82 | 88.23 | 106.64 | 111.33 | 123.66 |

* Memory exceeded (64K bits per processor)

Before a critical point, the efficiency for many of the kernels is much lower at smaller VP ratios (shorter vector lengths). This is due to underutilization of the memory bandwidth, and to the start-up and shutdown costs of the FPA pipeline (64 cycles), which constitute a much higher percentage of the overall time required to do a floating point operation at the smaller VP ratios. Data must be processed through a "transposer" chip upon entry and exit from the FPA. A future release of CM Fortran is expected to alleviate this problem and to reduce the start-up and shutdown costs of the FPA pipeline to 2 cycles, greatly increasing the efficiency of code running on small VP ratios.

Table 5 shows the double-precision Mflop results for the Cray X-MP/1 for large vector lengths [4]. Since the Cray is a vector machine, increasing the vector length would result in no measurable performance increase.

Table 5 —

Cray X-MP/1 Double Precision Performance in Mflops [10]

| Kernel | Vector Length 1000 |
|---|---|
| 1 | 164.58 |
| 3 | 151.70 |
| 5 | 14.36 |
| 7 | 187.75 |
| 8 | 145.79 |
| 9 | 157.52 |
| 10 | 61.21 |
| 11 | 12.68 |
| 12 | 74.34 |
| 13 | 5.83 |
| 14 | 22.22 |
| 15 | 5.18 |
| 18 | 110.57 |
| 19 | 13.36 |
| 21 | 108.94 |
| 22 | 65.78 |
| 23 | 13.88 |
| 24 | 3.56 |

## 6. CONCLUSIONS

For applications involving large vector lengths, a large amount of computation, and minimal general communication the CM-2 performs extremely well. For half of the kernels (1, 3, 7, 8, 9, 10, 12, 15, 18, 21, 22, and 24), the CM-2 outperformed by a wide margin the Cray X-MP/1. Kernels 2, 4, 6, 16, 17, and 20 were not implemented because they were either strictly sequential or involved a very small number of floating point operations. References 10, 11, and 12 further discuss vector and parallel architectural results. The results presented in this report are scalable when run on a 64K CM-2 and would allow the Mflop rates to increase by a factor of two. References 13, 14, and 15 compare performances involving actual applications on the CM-2 and other supercomputers.

## 7. ACKNOWLEDGMENTS

I thank Paul Anderson, Robert Whaley, Alex Vasilevsky, Henry Dardy, and Elizabeth Wald for many helpful comments and discussions.

## 8. REFERENCES

1. T. Perry and G. Zorpette, "Supercomputer Experts Predict Expansive Growth," *IEEE Spectrum*, February 1989, pp. 26-33.

2. J. Dongarra, J. Martin, and J. Worlton. "Computer Benchmarking: Paths and Pitfalls," *IEEE Spectrum*, July 1987, pp. 38-43.

3. J. T. Feo, "An Analysis of the Computational and Parallel Complexity of the Livermore Loops," *Parallel Computing* **7**, 163-185, 1988.

4. F. H. McMahon, "The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range," Lawrence Livermore National Laboratory, Livermore, Calif., UCRL-53745(1986).

5. D. C. Douglas, B. A. Kahle, and A. Vasilevsky, "The Architecture of the CM-2 Data Processor," Thinking Machines Corporation Technical Report HA88-1 (1988).

6. "Connection Machine Model CM-2 Technical Summary," Thinking Machines Technical Report HA87-4, Apr. 1987.

7. "CM Fortran Reference Manual," Thinking Machines Corporation, Mar. 1990.

8. "American National Standards for Information Systems Programming Language Fortran S8 (X3.9-198x) version 104," American National Standards Institute, June 1987.

9. M. Metcalf and J. Reid, *Fortran 8x Explained* (Oxford University Press, London, 1987).

10. R. W. Hockney and C.R. Jesshope, *Parallel Computing* (Adam-Hilger, Bristol, 1981).

11. R. W. Hockney, "Characterization of Parallel Computers," in *Parallel and Large-Scale Computers: Performance, Architecture, Applications*, M. Ruschitzka et al., eds. (North-Holland, 1983).

12. J. Levesque and J. Williamson, *A Guidebook to Fortran on Supercomputers* (Academic Press, 1989).

13. R. K. Agarwal and J. L. Richardson, "Development of an Euler Code on a Connection Machine," in *Proceedings of the Conference on Scientific Applications of the Connection Machine*, H. D. Simon, ed. (World Scientific, 1989), pp. 27-37.

14. O. A. McBryan, "Connection Machine Application Performance," in *Proceedings of the Conference on Scientific Applications of the Connection Machine*, H. D. Simon, ed. (World Scientific, 1989), pp. 94-115.

15. R.G. Brickner and C. F. Baillie, "Pure Gauge QCD on the Connection Machine," in *Proceedings of the Conference on Scientific Applications of the Connection Machine*, H. D. Simon, ed. (World Scientific, 1989), pp. 234-260.

## Appendix
## CONVERTED CODE

*Kernel 1 (Hydrodynamics fragment)*

**Fortran 77**

```
do k=1,n
x(k) = q + y(k) * (r * z(k+10) + t * z(k+11))
```

**CM Fortran**

```
n = nvec - 11
x = q + y * (r * z(11:n+10) + t * z(12:n+11))
```

*Kernel 3 (Inner Product)*

**Fortran 77**

```
do k=1,n
q = q + z(k) * x(k)
```

**CM Fortran**

```
q = dotproduct(x,z)
```

*Kernel 5 (Tridiagonal Elimination)*

**Fortran 77**

```
do i=2,n
x(i) = z(i) * (y(i) - x(i-1))
```

**CM Fortran**

```
k2 = log2(nvec)
a = -z
do k = 1,k2 - 1
x = x+a*eoshift(x,1,-(2**(k-1)))
a = a*eoshift(a,1,-(2**(k-1)))
enddo
x = x+a*eoshift(x,1,-(2**(k2-1)))
```

*Kernel 7 (Equation of State fragment)*

**Fortran 77**

```
do k=1,n
x(k) = u(k) + r * (z(k) + r * y(k)) + t * (u(k+3)
+ r * (u(k+2) + r * u(k+1)) + t * (u(k+6) +
r * (u(k+5) + r * u(k+4))))
```

**CM Fortran**

```
n = nvec -6
x = u(1:n) + r * (z + r * y) + t * u(4:n+3) +
r * ( u(3:n+2) + r * u(2:n+1) + t * u(7:n+6)
+ r * ( u(6:n+5) + r * u(5:n+4)))
```

*Kernel 8 (A.D.I. Integration)*

**Fortran 77**

```
n11 = 1
n12 = 2
do kx=2,3
do ky=2,n
du1(ky) = u1(kx,ky+1,n11) -
u1(kx,ky-1,n11)
du2(ky) = u2(kx,ky+1,n11) -
u2(kx,ky-1,n11)
du3(ky) = u3(kx,ky+1,n11) -
u3(kx,ky-1,n11)

u1(kx,ky,n12) = u1(kx,ky,n11) + a11 * du1(ky)
+ a12 * du2(ky) + a13 * du3(ky) + sig
* (u1(kx+1,ky,n11) - 2.0 * u1(kx,ky,n11) +
u1(kx-1,ky,n11))

u2(kx,ky,n12) = u2(kx,ky,n11) + a21 * du1(ky)
+ a22 * du2(ky) + a23 * du3(ky) + sig
* (u2(kx+1,ky,n11) - 2.0 * u2(kx,ky,n11) +
u2(kx-1,ky,n11))

u3(kx,ky,n12) = u3(kx,ky,n11) + a31 * du1(ky)
+ a32 * du2(ky) + a33 * du3(ky) + sig
* (u3(kx+1,ky,n11) - 2.0 * u3(kx,ky,n11) +
u3(kx-1,ky,n11))
```

**CM Fortran**

```
do kx=2,3
du1(2:n) = u1(kx,1,3:n+1) - u1(kx,1,1:n-1)
du2(2:n) = u2(kx,1,3:n+1) - u2(kx,1,1:n-1)
du3(2:n) = u3(kx,1,3:n+1) - u3(kx,1,1:n-1)

u1(kx,2,2:n) = u1(kx,1,2:n) + a11 * du1(2:n)
+ a12 * du2(2:n) + a13 * du3(2:n) + sig
* ( u1(kx+1,1,2:n) -2.0 * u1(kx,1,2:n) +
u1(kx-1,1,2:n) )

u2(kx,2,2:n) = u2(kx,1,2:n) + a21 * du1(2:n)
+ a22 * du2(2:n) + a23 * du3(2:n) + sig
* ( u2(kx+1,1,2:n) -2.0 * u2(kx,1,2:n) +
u2(kx-1,1,2:n) )

u3(kx,2,2:n) = u3(kx,1,2:n) + a31 * du1(2:n)
+ a32 * du2(2:n) + a33 * du3(2:n) + sig
* ( u3(kx+1,1,2:n) -2.0 * u3(kx,1,2:n) +
u3(kx-1,1,2:n) )
enddo
```

*Kernel 9 (Integrate Predictors)*

**Fortran 77**

```
do i=1,n
px(1,i) = px(3,i) + c0 * (px(5,i) + px(6,i)) +
dm28 * px(13,i) + dm27 * px(12,i) + dm26 *
px(11,i) + dm25 * px(10,i) + dm24 * px(9,i) +
dm23 * px(8,i) + dm22 * px(7,i)
```

**CM Fortran**

```
px1 = dm28 * px13 + dm27 * px12 + dm26 *
px11 + dm25 * px10 + dm24 * px9 + dm23 *
px8 + dm22 * px7 + c0 * (px5 + px6) + px3
```

*Kernel 10 (Difference Predictors)*

**Fortran 77**

```
do i=1,n
ar = cx(5,i)
br = ar - px(5,i)
px(5,i) = ar
cr = br - px(6,i)
px(6,i) = br
ar = cr - px(7,i)
px(7,i) = cr
br = ar - px(8,i)
px(8,i) = ar
cr = br - px(9,i)
px(9,i) = br
ar = cr - px(10,i)
px(10,i) = cr
br = ar - px(11,i)
px(11,i) = ar
cr = br - px(12,i)
px(12,i) = br
px(14,i) = cr - px(13,i)
px(13,i) = cr
```

**CM Fortran**

```
ar = cx5
br = ar - px5
px5 = ar
cr = br - px6
px6 = br
ar = cr - px7
px7 = cr
br = ar - px8
px8 = ar
cr = br - px9
px9 = br
ar = cr - px10
px10 = cr
br = ar - px11
px11 = ar
cr = br - px12
px12 = br
px14 = cr - px13
px13 = cr
```

*Kernel 11 (First Sum)*

**Fortran 77**

```
do k=2,n
x(k) = x(k-1) + y(k)
```

**CM Fortran**

```
k2 = log2(nvec)
x = y
do k = 1,k2
x = x +eoshift(x,1,-(2**(k-1)))
enddo
```

*Kernel 12 (First Difference)*

**Fortran 77**

```
do k=1,n
x(k) = y(k+1) - y(k)
```

**CM Fortran**

```
n = nvec - 1
x(1:n) = y(2:n+1) - y(1:n)
```

*Kernel 13 (2-D Particle in Cell)*

**Fortran 77**

```
do ip=1,n
i1 = p(1,ip)
j1 = p(2,ip)
i1 = 1 + mod2n(i1,64)
j1 = 1 + mod2n(j1,64)
p(3,ip) = p(3,ip) + b(i1,j1)
p(4,ip) = p(4,ip) + c(i1,j1)
p(1,ip) = p(1,ip) + p(3,ip)
p(2,ip) = p(2,ip) + p(4,ip)
i2 = p(1,ip)
j2 = p(2,ip)
i2 = mod2n(i2,64)
j2 = mod2n(j2,64)
p(1,ip) = p(1,ip) + y(i2+32)
p(2,ip) = p(2,ip) + z(j2+32)
i2 = i2 + e(i2+32)
j2 = j2 + f(j2+32)
h(i2,j2) = h(i2,j2) + 1.0
```

**CM Fortran**

```
h = 0
i1 = 1 + mod2n(int(p(1,:)),64)
j1 = 1 + mod2n(int(p(2,:)),64)
forall (i=1:n) temp1(i)=b(i1(i),j1(i))
forall (i=1:n) temp2(i)=c(i1(i),j1(i))
p(3,:) = p(3,:) + temp1
p(4,:) = p(4,:) + temp2
p(1,:) = p(1,:) + p(3,:)
p(2,:) = p(2,:) + p(4,:)
i2 = mod2n(int(p(1,:)),64)
j2 = mod2n(int(p(2,:)),64)
p(1,:) = p(1,:) + y(i2+32)
p(2,:) = p(2,:) + z(j2+32)
i2 = i2 + e(i2 + 32)
j2 = j2 + f(j2 + 32)

call library routine to perform scatter operation source array to scatter_add_2 is an array of 1's

temp = 1.0
call scatter_add_2(h,i2,j2,temp)
```

*Kernel 14 (1-D Particle in Cell)*

**Fortran 77**

```
do k=1,n
vx(k) = 0.0
xx(k) = 0.0
ix(k) = int(grd(k))
xi(k) = float(ix(k))
ex1(k) = ex(ix(k))
dex1(k) = dex(ix(k))
enddo
do k=1,n
vx(k) = vx(k) + ex1(k) + (dex1(k) * (xx(k) -
xi(k)))
xx(k) = xx(k) + vx(k) + flx
ir(k) = xx(k)
rx(k) = xx(k) - ir(k)
ir(k) = mod2n(ir(k),512) + 1
xx(k) = rx(k) + ir(k)
enddo
do k=1,n
rh(ir(k)) = rh(ir(k)) - rx(k) + 1.0
rh(ir(k) + 1) = rh(ir(k) + 1) + rx(k)
enddo
```

**CM Fortran**

```
vx = 0.0
xx = 0.0
ix = int(grd)
xi = float(ix)
ex1 = ex(ix)
dex1 = dex(ix)
vx = vx + ex1 + (dex1 * (xx -xi))
xx = xx + vx + flx
ir = xx
rx = xx - ir
ir = mod2n(ir,512) + 1
xx = rx + ir

call library routine to perform scatter operation

call scatter_add_1(rh,ir,1.0-rx)
call scatter_add_1(rh,ir+1,rx)
```

*Kernel 15 (Casual Fortran)*

**Fortran 77**

```
ng = 7
nz = n
ar = .053
br = .073
15 do 45 j = 2,ng
do 45 k = 2,nz
if (j-ng) 31,30,30
30 vy(k,j) = 0.0
goto 45
31 if (vh(k,j+1) - vh(k,j)) 33,33,32
32 t = ar
goto 34
33 t = br
34 if (vf(k,j) - vf(k-1,j)) 35,36,36
35 r = max(vh(k-1,j),vh(k-1,j+1))
s = vf(k-1,j)
goto 37
36 r = max(vh(k,j),vh(k,j+1))
s = vf(k,j)
37 vy(k,j) = sqrt(vg(k,j)**2 + r*r) * t/s
38 if (k-nz) 40,39,39
39 vs(k,j) = 0.0
goto 45
40 if (vf(k,j) - vf(k,j-1)) 41,42,42
41 r = max(vg(k,j-1),vg(k+1,j-1))
s = vf(k,j-1)
t = br
goto 43
42 r = max(vg(k,j),vg(k+1,j))
s = vf(k,j)
t = ar
43 vs(k,j) = sqrt(vh(k,j)**2 + r * r) * t/s
45 continue
```

**CM Fortran**

```
n1 = nvec/8
n2 = 8
m = .false.
m(2:n1,2:n2-1) = .true.
vy(2:n1,n2) = 0.0
vs(n1,2:n2-1) = 0.0
where(m.and.(eoshift(vh,2,1).gt.vh))
t = .053
elsewhere
t = .073
endwhere
where(m.and.(vf.ge.eoshift(vf,1,-1)))
r = max(vh,eoshift(vh,2,1))
s = vf
elsewhere
r=
max(eoshift(vh,1,-1),
eoshift(eoshift(vh,1,-1),2,1))
s = eoshift(vf,1,-1)
endwhere
where (m)
vy = sqrt(vg * vg + r * r) * t / s
endwhere
m(n1,:) = .false.
where(m.and.(vf.ge.eoshift(vf,2,-1)))
r = max(vg,eoshift(vg,1,1))
s = vf
t = .053
elsewhere
r=
max(eoshift(vg,2,-1),
eoshift(eoshift(vg,1,1),2,-1))
s = eoshift(vf,2,-1)
t = .073
endwhere
where (m)
vs = sqrt(vh * vh + r * r) * t / s
endwhere
```

*Kernel 18 (2-D Explicit Hydro fragment)*

**Fortran 77**

```
kn = 6
jn = n
do 70 k=2,kn
do 70 j=2,jn
za(j,k) = (zp(j-1,k+1) + zq(j-1,k+1) -
zp(j-1,k) - zq(j-1,k)) * (zr(j,k) + zr(j-1,k))
/ (zm(j-1,k) + zm(j-1,k+1))

zb(j,k) = (zp(j-1,k) + zq(j-1,k) - zp(j,k) -
zq(j,k)) * (zr(j,k) + zr(j,k-1)) / (zm(j,k) +
zm(j-1,k))
70 continue

do 72 k=2,kn
do 72 j=2,jn
zu(j,k) = zu(j,k) + s * (za(j,k) * (zz(j,k) -
zz(j+1,k)) - za(j-1,k) * (zz(j,k) - zz(j-1,k))
-zb(j,k) * (zz(j,k) - zz(j,k-1)) + zb(j,k+1) *
(zz(j,k) - zz(j,k+1)))

zv(j,k) = zv(j,k) + s * (za(j,k) * (zr(j,k) -
zr(j+1,k)) - za(j-1,k) * (zr(j,k) - zr(j-1,k))
-zb(j,k) * (zr(j,k) - zr(j,k-1)) + zb(j,k+1) *
(zr(j,k) - zr(j,k+1)))
72 continue

do 75 k = 2,kn
do 75 j = 2,jn
zr(j,k) = zr(j,k) + t * zu(j,k)
zz(j,k) = zz(j,k) + t * zv(j,k)
75 continue
```

**CM Fortran**

```
n1 = 8
n2 = nvec
do k=2,6
za(k,2:n2-1) = (zp(k+1,1:n2-2) +
zq(k+1,1:n2-2) - zp(k,1:n2-2) -
zq(k,1:n2-2)) * (zr(k,2:n2-1) + zr(k,1:n2-2))
/ (zm(k,1:n2-2) + zm(k+1,1:n2-2))

zb(k,2:n2-1) = (zp(k,1:n2-2) + zq(k,1:n2-2)
- zp(k,2:n2-1) - zq(k,2:n2-1)) *
(zr(k,2:n2-1) + zr(k-1,2:n2-1)) /
(zm(k,2:n2-1) + zm(k,1:n2-2))

zu(k,2:n2-1) = zu(k,2:n2-1)
+ s * (za(k,2:n2-1) * (zz(k,2:n2-1) -
zz(k,3:n2)) - za(k,1:n2-2) * (zz(k,2:n2-1) -
zz(k,1:n2-2)) - zb(k,2:n2-1) * (zz(k,2:n2-1)
- zz(k-1,2:n2-1)) + zb(k+1,2:n2-1) *
(zz(k,2:n2-1) - zz(k-1,2:n2-1)))

zv(k,2:n2-1) = zv(k,2:n2-1)
+ s * (za(k,2:n2-1) * (zr(k,2:n2-1) -
zr(k,3:n2)) - za(k,1:n2-2) * (zr(k,2:n2-1) -
zr(k,1:n2-2)) - zb(k,2:n2-1) * (zr(k,2:n2-1)
- zr(k-1,2:n2-1)) + zb(k+1,2:n2-1) *
(zr(k,2:n2-1) - zr(k+1,2:n2-1)))

zr(k,2:n2-1) = zr(k,2:n2-1) + t *
zu(k,2:n2-1)

zz(k,2:n2-1) = zz(k,2:n2-1) + t *
zv(k,2:n2-1)
enddo
```

*Kernel 19 (General Linear Recurrence)*

**Fortran 77**

```
do 191 k=1,n
b5(k) = sa(k) + stb5 * sb(k)
191 stb5 = b5(k) - stb5
do 193 i=1,n
k = n - i + 1
b5(k) = sa(k) + stb5 * sb(k)
193 stb5 = b5(k) - stb5
```

**CM Fortran**

```
x0 = 0.0
k2 = log2(nvec)
a = sb - 1.0
stb5 = sa
do k=1,k2 - 1
i2 = -(2**(k-1))
stb5=stb5+a*eoshift(stb5,1,i2,x0)
a=a*eoshift(a,1,i2)
enddo
i2 = -(2**(k2-1))
stb5 = stb5 + a * eoshift(stb5,1,i2,x0)

clean up last one

stb5(nvec)=stb5(nvec)+x0*a(nvec/2)
xend = stb5(nvec)
a = sb - 1.0
stb5 = sa
do k=1,k2-1
i2 = (2**(k-1))
stb5=stb5+a*eoshift(stb5,1,i2,xend)
a=a*eoshift(a,1,i2)
enddo
i2 = (2**(k2-1))
stb5=stb5+a*eoshift(stb5,1,i2,xend)

clean up last one

stb5(1) = stb5(1) + xend * a(1)
```

*Kernel 21 (Matrix Product)*

**Fortran 77**

```
do 21 k=1,25
do 21 i=1,25
do 21 j=1,n
21 px(i,j) = px(i,j) + vy(i,k) * cx(k,j)
```

**CM Fortran**

```
px = matmul(vy,cx)
```

*Kernel 22 (Planckian Distribution)*

**Fortran 77**

```
do k=1,n
y(k) = 20.0
if (u(k) .lt. 20.0 * v(k)) y(k) = u(k) / v(k)
w(k) = x(k) / (exp(y(k)) - 1.0)
```

**CM Fortran**

```
y = 20.0
where (u .lt. 20.0 * v) y = u/v
w = x/(exp(y) - 1.0)
```

*Kernel 23 (2-D Implicit Hydro fragment)*

**Fortran 77**

```
do 23 j=2,6
do 23 k=2,n
qa = za(k,j+1) * zr(k,j) + za(k,j-1) * zb(k,j)
+ za(k+1,j) * zu(k,j) + za(k-1,j) * zv(k,j) +
zz(k,j)
23 za(k,j) = za(k,j) + .175 * (qa - za(k,j))
```

**CM Fortran**

```
n1 =8
n = nvec
n2 = nvec-1
k2 = log2(n)
do j=2,6
qa(j,2:n2) = za(j+1,2:n2) * zr(j,2:n2)
+ za(j-1,2:n2) * zb(j,2:n2) +
za(j,3:n2+1) * zu(j,2:n2) + zz(kf,2:n2)
- za(j,2:n2)
enddo

b = za + .175 * qa
a = .175 * zv
za = b
do k=1,k2 - 1
za=za+a*eoshift(za,2,-(2**(k-1)))
a=a*eoshift(a,2,-(2**(k-1)))
enddo

za=za+a*eoshift(za,2,-(2**(k2-1)))
```

*Kernel 24 (Location of First Minimum)*

**Fortran 77**

```
m = 1
do k=2,n
if (x(k) .lt. x(m)) m = k
```

**CM Fortran**

```
integer index(nvec)
index = [1:nvec]
m = minval(index,mask= x .eq. minval(x))
```