Fast Fourier Transform Algorithm Design and Tradeoffs

Ray A. Kamin III
George B. Adams III

December, 1988

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 88.18

NASA Cooperative Agreement Number NCC 2-387

RIACS

Research Institute for Advanced Computer Science

Fast Fourier Transform Algorithm Design and Tradeoffs

*Ray A. Kamin III**
*George B. Adams III**

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 88.18
December, 1988

The Fast Fourier Transform is a mainstay of certain numerical techniques for solving fluid dynamics problems. The Connection Machine CM-2 is the target for an investigation into the design of multidimensional SIMD parallel FFT algorithms for high performance. Critical algorithm design issues are discussed, necessary machine performance measurements are identified and made, and the performance of the developed FFT programs are measured. Our FFT programs are compared to the currently best Cray-2 FFT program.

The work reported herein was supported in part by Cooperative Agreement NCC2-387 between the National Aeronautics and Space Administration (NASA) and the Universities Space Research Association (USRA) and the Defense Advanced Research Projects Agency (DARPA). The contents of this document do not represent the official position of NASA or DARPA.

*The authors are with the School of Electrical Engineering, Purdue University, West Lafayette, Indiana. Work performed while visitors at RIACS and performed at the School of Electrical Engineering, Purdue University.

TABLE OF CONTENTS

Page

1 Introduction 1

2 Background 2

2.1 The Fast Fourier Transform 2

2.2 The Connection Machine 2 10

2.3 The Cray 2 15

3 Algorithm Design 16

3.1 Tradeoffs 17

3.2 FFTs on Conventional Vector Supercomputers 23

3.3 FFTs on the CM-2 24

4 Performance 30

4.1 Measurement Techniques on the CM-2 31

4.2 Experimental Results 36

5 Conclusion 40

6 Acknowledgement 41

7 References 42

8 Appendix: C/Paris Programs 44

1 Introduction

As advanced computer systems continue to increase in performance, computationally intensive problems that were not feasible to attack are becoming more practical. Included in this category are the compressible Reynolds averaged Navier-Stokes equations in the field of computational fluid dynamics (CFD). These equations model the flow of viscous, Newtonian fluids at subsonic, transonic, and supersonic speeds.

Extensive work in this area is being done in the Numerical Aerodynamic Simulation (NAS) Systems Division at the NASA Ames Research Center. For example, detailed transonic flow analysis has been performed on the airframe geometry of the McDonnell-Douglas F-16A jet fighter using a three dimensional grid of 129×33×33 points. The vast memory of Cray-2 supercomputers (256 MWords), currently employed by the NAS, allows implementation of finer grids to provide a more detailed analysis of the flow over complex geometries.

The Fast Fourier Transform (FFT) is a mainstay of certain CFD methods because of its use in solving partial differential equations. The FFT is computationally intensive for the problem sizes of interest, making efficient implementations of FFT algorithms essential. Many investigations have considered the FFT on conventional [BRI74] and vector [SWA84] architectures. Since the FFT algorithm admits to considerable parallelism, it is attractive for computation on a parallel computer. The fine grain, massively parallel nature of the Connection Machine[1] model CM-2 make it a logical choice for an FFT investigation. Preliminary FFT algorithm design for the CM-2 has been explored in [JHJ88]. Detailed analysis of an FFT algorithm for another parallel computer that supports various modes of parallelism has been done [BCJ89].

The work presented here is a design and tradeoff study taken through to algorithm implementation and algorithm performance measurement. In the supercomputing arena, where the highest performance algorithms are sought, an algorithm study without implementation and performance measurement is unlikely to result in the best algorithm design. The behavior of computers is complex, and readily measured quantities such as total number of instructions or processor utilization do not necessarily provide guidance for reaching the goal of a high performance algorithm. The research began on a CM-2 with 8192 processors running Version 4.3 of the system software, and continued onto the current configuration of 32768 processors with floating point hardware running Version 5.0β system software. The CM-2 is sited at the NAS division of the NASA Ames Research Center.

[1]Connection Machine is a trademark of Thinking Machines, Inc.

2 Background
2.1 The Fast Fourier Transform

The discrete Fourier Transform (DFT) was developed for digital computer evaluation of the continuous Fourier transform, which relates the time domain of a signal to its corresponding frequency domain. The periodic function $x(t)$ with period T can be represented by a Fourier series of the form

$$x(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} [a_n \cos(n\omega_0 t) + b_n \sin(n\omega_0 t)] \qquad (1)$$

where ω_0 is the fundamental frequency, equal to $2\pi/T$. By *Euler's Identity*,

$$e^{j\theta} = \cos\theta + j\sin\theta ,$$

where $j = \sqrt{-1}$, Equation (1) can be written as

$$x(t) = \frac{a_0}{2} + \frac{1}{2}\sum_{n=1}^{\infty}(a_n - jb_n)e^{jn\omega_0 t} + \frac{1}{2}\sum_{n=1}^{\infty}(a_n + jb_n)e^{-jn\omega_0 t}. \qquad (2)$$

Since

$$\sum_{n=1}^{\infty} a_n e^{-jn\omega_0 t} = \sum_{n=-1}^{-\infty} a_n e^{jn\omega_0 t}$$

and

$$\sum_{n=1}^{\infty} jb_n e^{-jn\omega_0 t} = -\sum_{n=-1}^{-\infty} jb_n e^{jn\omega_0 t} ,$$

Equation (2) simplifies to

$$x(t) = \frac{1}{2}\sum_{n=0}^{\infty}(a_n - jb_n)e^{jn\omega_0 t} + \frac{1}{2}\sum_{n=-1}^{-\infty}(a_n - jb_n)e^{jn\omega_0 t}$$

$$= \frac{1}{2}\sum_{n=-\infty}^{\infty}(a_n - jb_n)e^{jn\omega_0 t}.$$

Letting $c_n = \frac{1}{2}(a_n - jb_n)$ gives

$$x(t) = \sum_{n=-\infty}^{\infty} c_n e^{jn\omega_0 t} ,$$

which is the the Fourier series expressed in exponential form. The derivation of the Fourier transform comes from the limiting case of the Fourier series. From the definitions

$$a_n = \frac{2}{T}\int_{T/2}^{-T/2} x(t)\cos(n\omega_0 t)dt$$

and

$$b_n = \frac{2}{T} \int_{T/2}^{-T/2} x(t)\sin(n\omega_0 t)dt$$

we have

$$c_n = \frac{1}{T} \int_{T/2}^{-T/2} x(t)e^{-jn\omega_0 t}dt \quad .$$

As the period increases without limit, the frequency moves from being a discrete variable to becoming a continous variable, $n\omega_0 \longrightarrow \omega$. Also, $c_n T \longrightarrow \int_{-\infty}^{+\infty} x(t)e^{-j\omega t}dt$ [NIL83]. If the integral limits are confined over N time samples for which N independent frequency samples can be calculated, the DFT results. Thus, for $n = 0, 1, 2, \cdots, N-1$ and $k = 0, 1, 2, \cdots, N-1$

$$X(k\omega_0) = \sum_{n=0}^{N-1} x(nT)e^{\frac{-2\pi jnk}{N}} \quad .$$

For simplicity, the terms T and ω_0 are dropped from the indicies giving the common form of the DFT.

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{\frac{-2\pi jnk}{N}}$$

$$= \sum_{n=0}^{N-1} x(n)W^{nk} \quad . \tag{3}$$

The values

$$W^k = e^{\frac{-2\pi jk}{N}} \qquad k = 0, 1, \ldots, N-1$$

are referred to as *twiddle factors.*

Straight-forward evaluation of Equation (3) for all values of k has time complexity $O(N^2)$. However, because of inherent redundancies in the evaluation of the transform, an $O(Nlog_2 N)$ time algorithm can be constructed.

If N is factored as $N = L \times M$ then the 1-dimensional vector can be represented as a 2-dimensional array with L columns and M rows [RAG75]. To facilitate indexing of this array we define

$$n = Ml + m, \qquad k = Lr + s$$

where

$$x(n) = x(l, m) \qquad l = 0, \ldots, L-1, \quad m = 0, \ldots, M-1$$

$$X(k) = X(s, r) \qquad s = 0, \ldots, L-1, \quad r = 0, \ldots, M-1$$

Expressing (3) using this indexing scheme yields

$$X(k) = X(s,r) = \sum_{m=0}^{M-1} \sum_{l=0}^{L-1} x(l,m)\, W^{(Ml + m)(Lr + s)} \quad . \tag{4}$$

Simplifying and rearranging terms gives

$$\begin{aligned} X(s,r) &= \sum_{m=0}^{M-1} \sum_{l=0}^{L-1} x(l,m)\, W^{(MlLr + mLr + sMl + ms)} \\ &= \sum_{m=0}^{M-1} \sum_{l=0}^{L-1} x(l,m)\, W^{mLr}\, W^{sMl}\, W^{ms} \\ &= \sum_{m=0}^{M-1} W^{Lmr}\, W^{ms} \sum_{l=0}^{L-1} x(l,m)\, W^{Msl} \end{aligned} \tag{5}$$

since $W^{MlLr} = W^{Nlr} = 1$. Therefore, $X(s,r)$ can be computed as two multiple transforms. First, M-transforms of length L are computed. If we let $q(s,m)$ be this result multiplied by W^{ms},

$$q(s,m) = W^{ms} \sum_{l=0}^{L-1} x(l,m)\, W^{Msl} \quad ,$$

then Equation (5) reduces to

$$X(s,r) = \sum_{m=0}^{M-1} q(s,m)\, W^{Lmr} \quad . \tag{6}$$

Finally, compute the M-point DFT of each row of the $q(s,m)$ matrix using (6). Instead of computing $X(s,r)$ directly from Equation (6), it can be factored in the same way Equation (3) was factored. If $N = N_0 N_1 \cdots N_{n-1}$, then n such factorizations are possible, which yields the Cooley-Tukey Fast Fourier Transform (FFT) [COT65]. When $N_0 = N_1 = \cdots = N_{n-1} = r$, the algorithm is described as radix-r. As a result of the factorization, the output data vector is in *bit-reversed* order. Specifically, element $X[a_{n-1}\, a_{n-2} \cdots a_1\, a_0]$ must be swapped with element $X[a_0\, a_1 \cdots a_{n-2}\, a_{n-1}]$ where $N = 2^n$. For example if N=8, element X[1] must be swapped with element X[4] since $(1)_{10} = (001)_2$ and $(4)_{10} = (100)_2$. Similarly, the elements X[3] and X[6] must also be swapped.

Since the FFT algorithm described successively divides the original sequence into equal halves, the frequency resolution of the output of both of the smaller DFTs is decreased by a factor of two. This corresponds to dropping (decimating) alternate outputs of the original DFT and is called a *decimation-in-frequency* (DIF) FFT. By simply reordering the summations in (5), a *decimation-in-time* (DIT) FFT can be derived.

Each stage in an N-point, radix-r FFT consists of N/r independent *butterfly* operations. A butterfly operation takes r inputs and the corresponding twiddle factor(s) to compute each of the r outputs. The flow graphs in Figures 1 and 2 illustrate the DIF and DIT algorithms, respectively,

for radix-2 and N=8. As shown, the DIF FFT produces bit-reversed output from natural ordered input, while the DIT FFT produces ordered output from bit-reversed input. Preference for DIF versus DIT depends on data ordering requirements imposed by previous or subsequent computations or I/O activity.

Many factorization schemes have been implemented as an attempt to reduce the total number of complex operations required by the FFT [SWA87]. For example, if N is an integer power of four then a radix-4 algorithm can be used. This reduces the number of arithmetic operations required as compared to the radix-2 algorithm [BER68]. The same holds true for radix-8 and radix-16 algorithms. If N is a power of two but not four, then a mixed radix or "2+4" algorithm may be used. The first or last stage is computed using a radix-2 algorithm, while all other stages use the radix-4 algorithm.

Table 1 gives operation counts for the DFT and various radix FFTs. In counting the number of real operations, note that one complex multiplication requires four real multiplications and two real additions, except for the case where the twiddle factor is $W^0 = 1$. Similarly, one complex addition requires two real additions. The number of complex multiplications can further be reduced at the expense of determining those operations involving twiddle factors of $\pm j$.

The two-dimensional DFT is given by

$$X(k_1,k_2) = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x(n_1,n_2) e^{-j2\pi(n_1k_1/N_1 + n_2k_2/N_2)} \quad , \tag{7}$$

where N_1 and N_2 represent the sizes of the transform in each of the two dimensions. The two-dimensional DFT is generally evaluated by either the classical *row-column* method or more recently by a *direct* method [JMS86].

The *row-column* method computes the two-dimensional FFT as a series of one-dimensional FFTs. Given an $N_1 \times N_2$ array of data points to be transformed, N_1 one-dimensional N_2-point FFTs are computed on the rows of the array, followed by N_2 one-dimensional N_1-point FFTs on the columns of the partially transformed array. This can be shown by rearranging the terms in Equation (7) as

$$X(k_1,k_2) = \sum_{n_1=0}^{N_1-1} e^{-j2\pi(n_1k_1/N_1)} \sum_{n_2=0}^{N_2-1} x(n_1,n_2) e^{-j2\pi(n_2k_2/N_2)} \quad . \tag{8}$$

As in the derivation of the 1-dimensional FFT, repeated factorization across each dimension of Equation (8) gives the 2-dimensional FFT. Consequently, this is an $O(N^2 \log N)$ process on a sequential machine. However, on a parallel machine with $N_1 N_2/4$ processors, the time complexity reduces to $O(logN)$.

Alternatively, when $N_1 = N_2$ the *direct* method can be used. This method does not reduce the problem to a set of one-dimensional FFTs, instead it computes the 2-D FFT directly. This method is also $O(logN)$ on a parallel machine with $N^2/4$ processors.

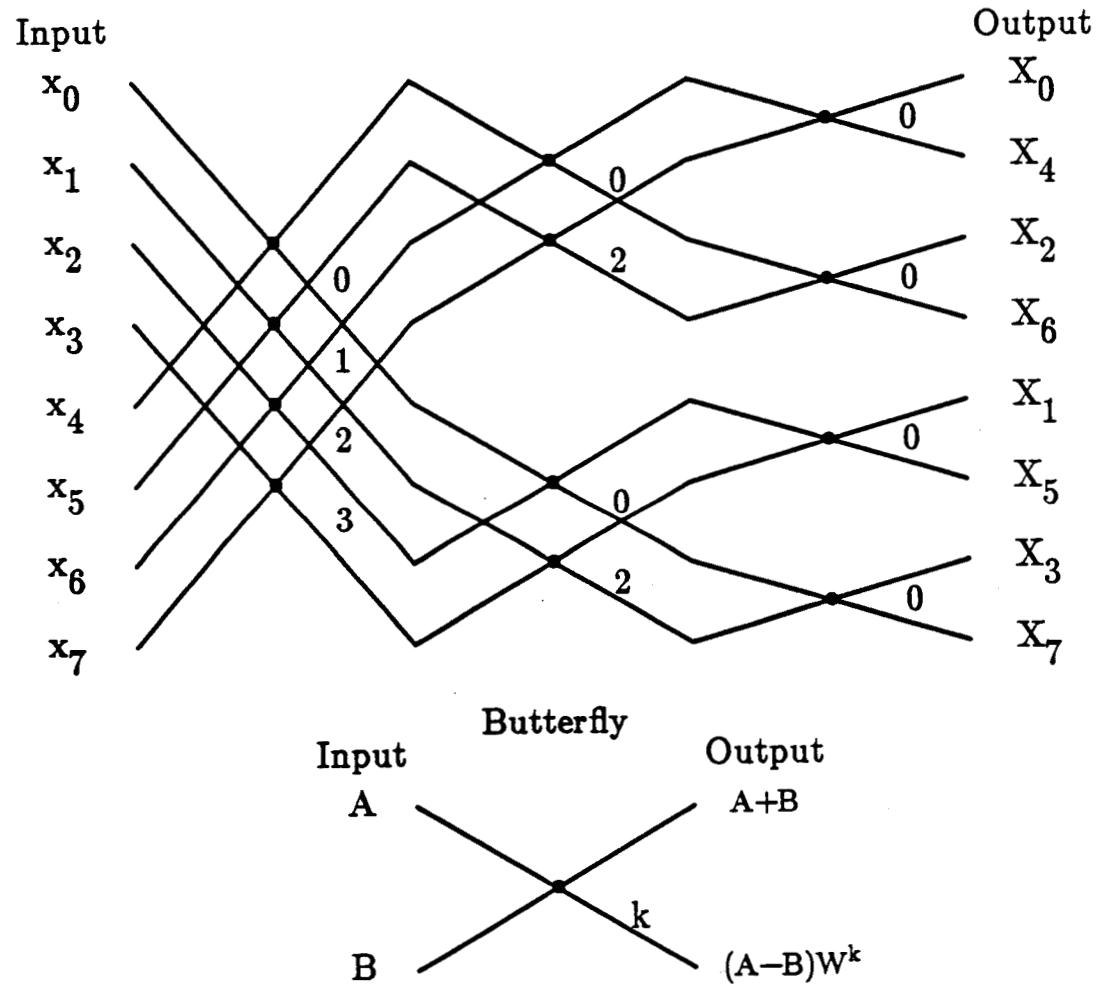


Figure 1: DIF flow graph (N=8) and detail of individual butterfly.

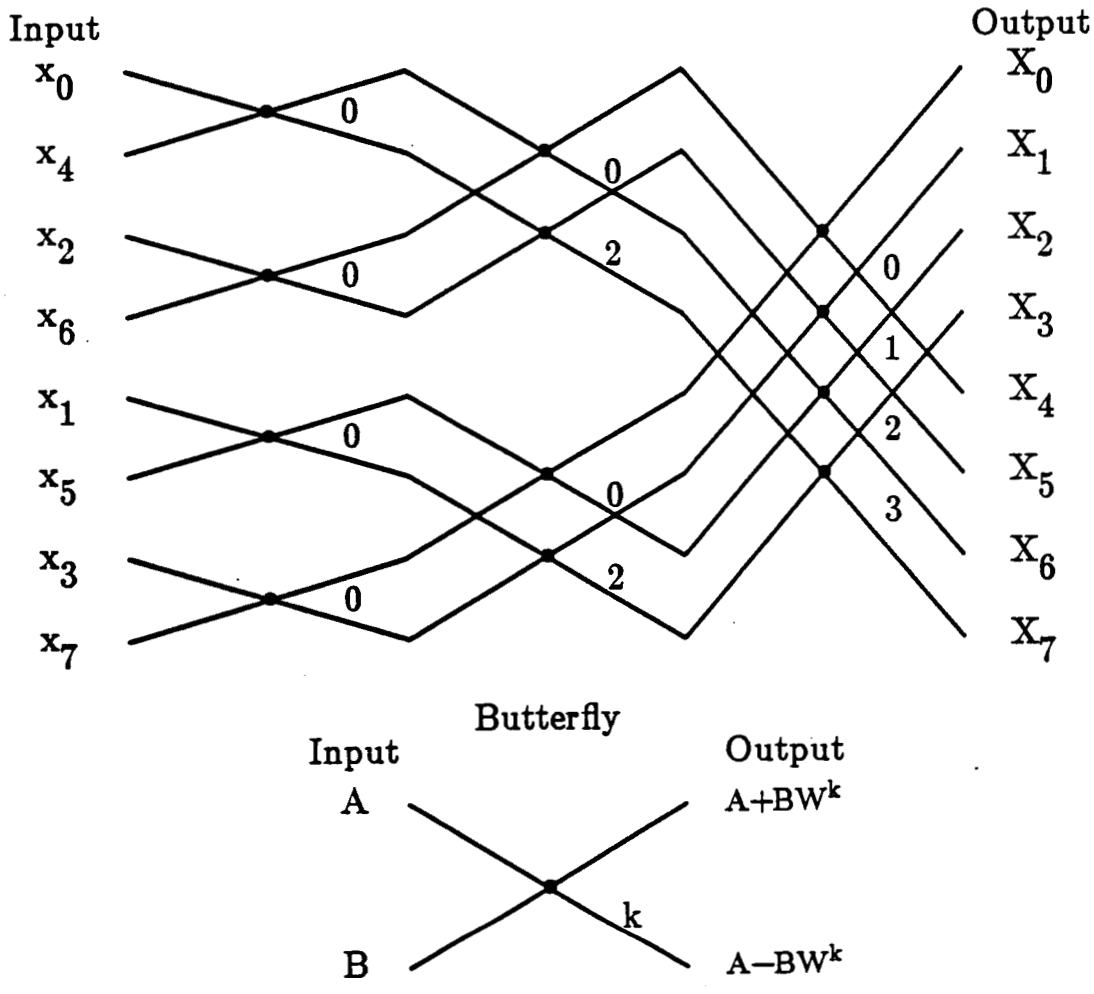


Figure 2: DIT flow graph (N=8) and detail of individual butterfly.

Table 1: Operation counts of DFT and FFT for N complex-valued data points.

Method	Number of Real Multiplications	Number of Real Additions
DFT	$4N^2 - 8N + 4$	$4N^2 - 6N + 2$
Radix-2 FFT	$2Nlog_2N - 4N + 4$	$3Nlog_2N - 2N + 2$
Radix-4 FFT	$\frac{3}{2}Nlog_2N - 4N + 4$	$\frac{11}{4}Nlog_2N - 2N + 2$
Radix-8 FFT	$\frac{4}{3}Nlog_2N - 4N + 4$	$\frac{11}{4}Nlog_2N - 2N + 2$

The multidimensional DFT is a direct extension of the one dimensional case. The D-dimensional DFT is defined for $N = N_1 N_2 \cdots N_D$ by

$$X(k_1, ..., k_D) = \sum_{n_1}^{N_1-1} \cdots \sum_{n_D}^{N_D-1} x(n_1, ..., n_D) \times \omega^{k_1 n_1 N/N_1} \cdots \omega^{k_D n_D N/N_D} \qquad (9)$$

A D-dimensional DFT can be evaluated with a 1-dimensional FFT by properly ordering the inputs and selecting the appropriate twiddle factors [ELR82].

2.2 The Connection Machine 2

A large class of parallel computers are SIMD (*S*ingle-*I*nstruction stream *M*ultiple-*D*ata stream) machines. As shown in Figure 3, an SIMD machine consists of a control unit, N processing elements (PEs), an interconnection network, and N memory modules. The control unit broadcasts a single instruction stream simultaneously to all N PEs. Depending on whether a PE is enabled or disabled, it either executes the instruction or waits. Since each processor operates in lock step with the control unit, synchronization is inherent in the design. The interconnection network provides a means for exchanging data among the PEs.

The CM-2 is an SIMD machine consisting of 64K (K=1024) PEs when fully configured [TMA87]. The actual hardware with the floating point option consists of 2048 chip-sets where a chip-set includes two processor chips (32 PEs), 256 Kbytes of memory, a Weitek 32-bit floating point unit, and a data transposer used to convert parallel data to serial and vice-versa. Control of the Connection Machine is handled by one or more front-end host processors. The host machine provides the user with a standard operating system, I/O facilities, and debugging tools. Currently, the CM-2 may be used with DEC VAX BI bus machines running Ultrix and/or with Symbolics 3600-series Lisp machines.

Programming can be done currently in two high-level languages: *Lisp (pronounced star-lisp) and C* (pronounced cee-star). These languages are supersets of Common Lisp and C, respectively. Additional language constructs provide the means for expressing parallelism in algorithms for the CM-2. Paris (*PA*Rallel *I*nstruction *S*et) is the *assembly language* of the CM-2. The Paris calls are made from within a high-level language such as C. These low-level instructions remove the degree of abstraction found in *Lisp and C* enabling the user to explicitly control the operation of the machine.

Physically, the 64K processors are divided into four groups of 16K PEs (see Figure 4). Each group is controlled by a microsequencer that receives macroinstructions from the host machine, then decodes and broadcasts the microinstructions to the individual PEs. These sequencers can operate independently to provide four users simultaneous access to one-fourth of the CM-2.

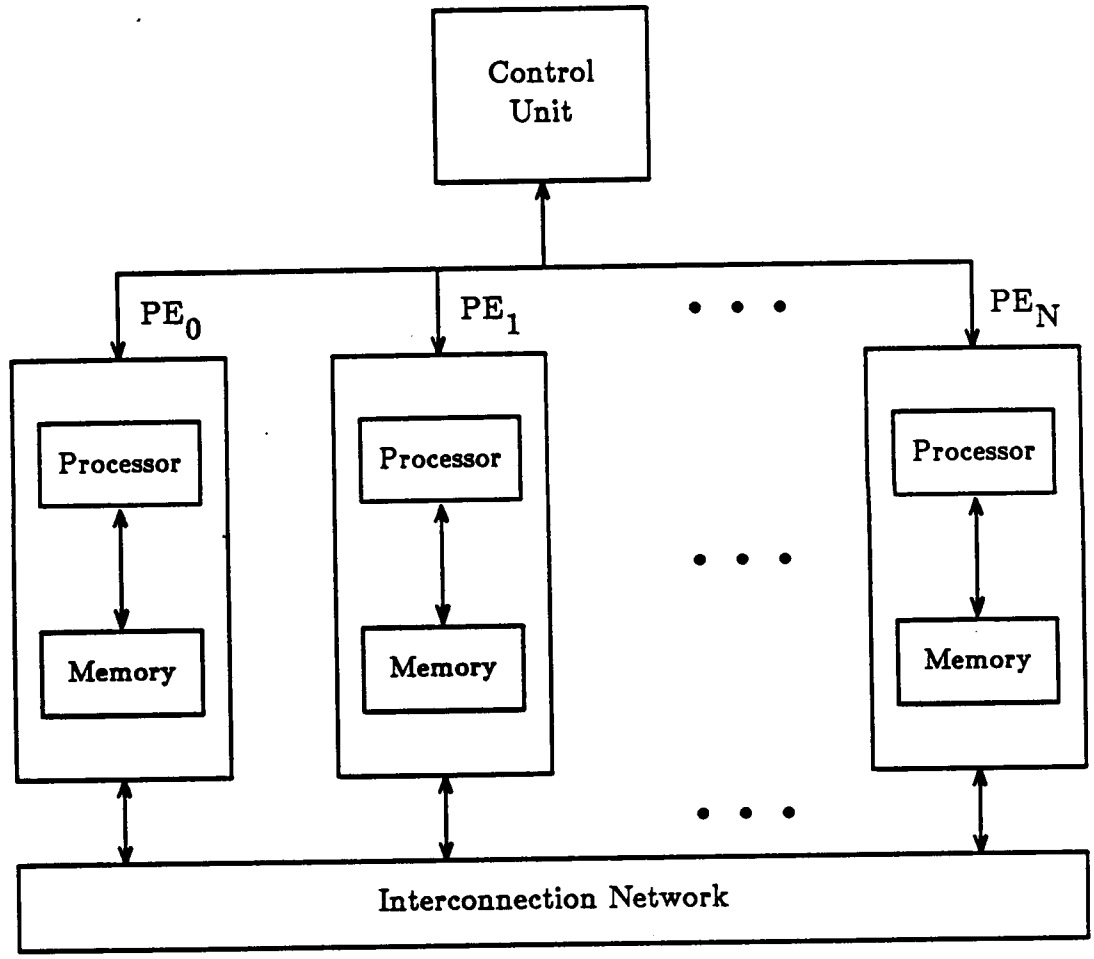


Figure 3: SIMD parallel computer structure.

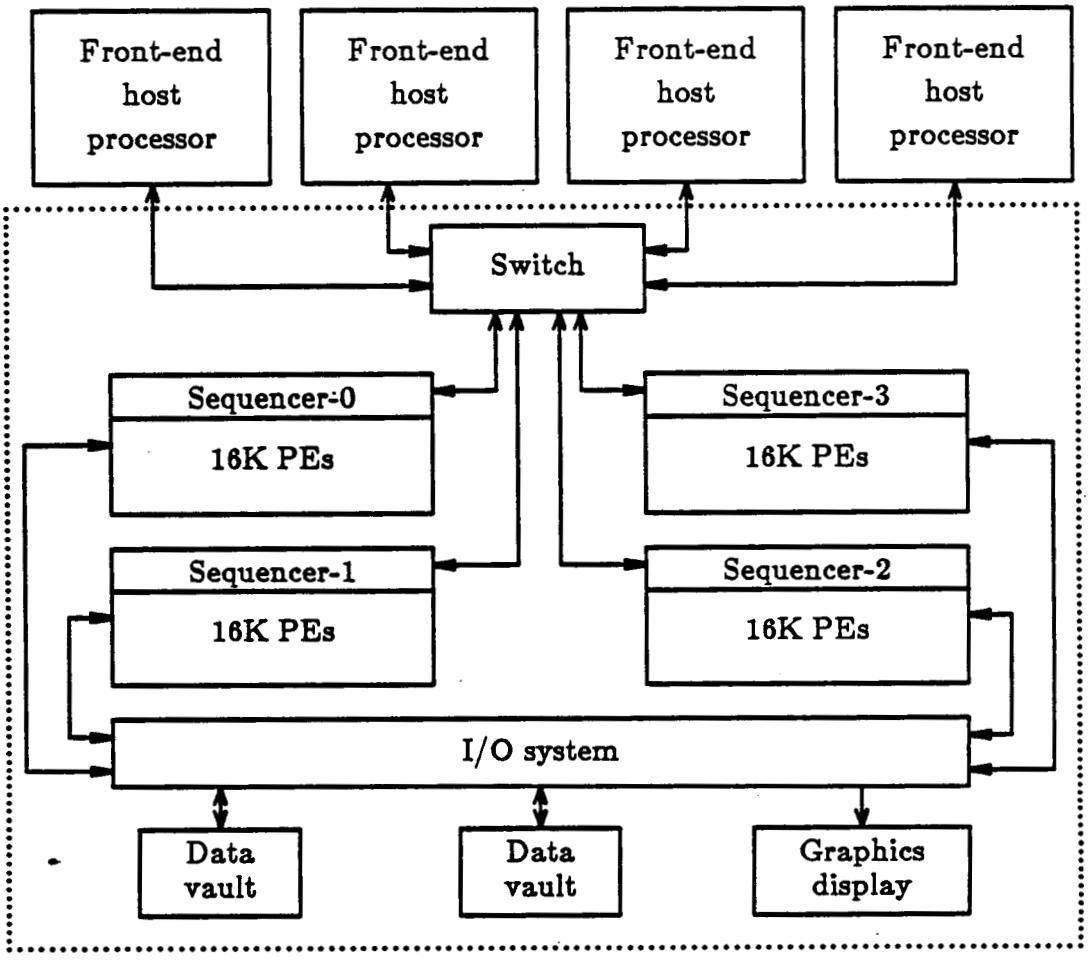


Figure 4: CM-2 organization.

The individual PEs include 3-input, bit-serial ALUs. This simplicity allows sixteen PEs to be implemented on a single chip. A 12-dimensional hypercube network handles the communication between PEs on different chips. Logically, PE $16i$ through PE $16i+15$, where $0 \leq i < 4096$ are on the same chip. Each physical PE is allotted 8 KBytes of bit-addressable memory. This yields a total of 512 MBytes of memory available on the CM-2.

The data vault is a mass storage system for the CM-2. Each data vault has a storage capacity of 5 GBytes, expandable to 10 GBytes. The graphics display consists of a framebuffer module and a high-resolution 19-inch color monitor. The framebuffer contains a large video memory and I/O circuitry to support graphical output.

Despite the seemingly large number of processors in the CM-2, many applications could benefit from additional processors. For this reason, the CM-2 software was designed to support *virtual processors*. The CM-2 uses time-slicing to simulate machine configurations of greater than the current number of available physical processors. For example, an algorithm that operates on 1024×1024 pixel images may be run on a 64K machine, allocating one pixel per processor, by using a virtual processor ratio of 16:1. In this case, each physical processor does the work of 16 virtual processors.

The current configuration of the CM-2 at NAS consists of 32K processors, floating-point hardware, three Symbolics front-ends, one VAX-8000 Series front-end, and Version 5.0β of the system software. Since the 32K processors are arranged as four sections of 8K processors each, the four sequencers and four front-ends enable up to four simultaneous users.

Paris has undergone major changes between Version 4.3 and Version 5.0β. An effort was made to establish a consistency in naming conventions among Paris instructions. Specifically, in Version 5.0β each instruction has a prefix of **CM_** and a suffix that specifies the number of field operands and the number of length operands, where the field is an integer specifying the starting location in local memory of the operand. For example, **CM_s_abs_1_1L**(*S,16*) is a Paris instruction with one field operand, *S*, and one length operand, *16*. This instruction directs each enabled processor to take the absolute value of the 16-bit signed integer in field *S* and store the result back into field *S*. Similarly, **CM_f_abs_2_1L**(*D,S,23,8*) has two field operands (source field *S* and destination field *D*), and one length operand. Each enabled processor takes the absolute value of the floating point number in field *S* and stores the result in field *D*. In this case, the length field indicates the floating point number has a 23-bit mantissa (with a hidden bit), and an 8-bit exponent. A 1-bit sign is implied. This is the IEEE standard format for a single precision 32-bit floating point number, however, the user may define an alternate format.

Another significant change in Version 5.0β is memory allocation. In Version 4.3, the user could directly specify absolute memory locations of the

various fields. However, in Version 5.0β, a memory management system has been implemented. The user indicates the length of a field and the system returns a *field-id* used to access the field. This permits system safety checking and more system control over memory.

Among the new instructions, there are many transcendental functions that have been implemented. Of specific interest to FFT applications, the sine and cosine functions are now available.

The floating point hardware consists of one Weitek 3132 floating point accelerator and one memory interface unit for every 32 processors. The memory interface unit is used to convert bit serial data to word parallel data and vice-versa. When a Paris floating point add, subtract, or multiply instruction is issued, each processor sends its first operand to its assigned memory interface chip. Simultaneously, these values are then transferred into the accelerator chip while the second operands are loaded into the memory interface unit. At this point the second operands are transferred into the accelerator and the floating point operation is executed. Finally, all 32 results are sent back to the memory interface unit and subsequently returned to the appropriate PE memory. The complete cycle requires five stages. However, if a virtual processor ratio of N is used, the process is pipelined as to only require $3N + 2$ stages instead of the expected $5N$ stages.

2.3 The Cray 2

In contrast to the CM-2, the Cray-2 is a tightly coupled MIMD (*M*ultiple *I*nstruction stream, *M*ultiple *D*ata stream) machine. Four high-speed vector processors share 256 MWords (4096 MBytes) of common memory. The operating system functions and I/O are handled by a dedicated foreground processor. Each of the four background processors include vector hardware pipelines. These pipelines reduce the overhead required for vector operations, where a vector operation is an arithmetic function applied to an entire array of data.

In addition to pipelined vector operations, the Cray-2 has the ability of exploiting multiprocessor parallelism at various levels including loops, subroutines, intraprogram tasks, and independent programs. The system software provides a library of Fortran subroutines to synchronize processor execution. Because of this incurred overhead, multitasking on the Cray-2 does not provide optimal speedup. For this reason, most applications are run on a single processor, allowing other applications to use the remaining processors.

3. Algorithm Design

There are many different, yet functionally equivalent, ways to implement an FFT algorithm. The current best implementations for vector supercomputers [BAI87, BAI88, SWA87] carefully tailor the calculation to have memory access patterns and vector operation lengths that minimize overhead and avoid idle time when possible. Algorithm designers also seek to reduce the number of operations required, by careful choice of FFT radix. When the utmost in performance is to be achieved, the precise details of a particular implementation are strongly dependent on the target computer.

Design for best performance in a parallel computing environment involves not only the machine-specific algorithm implementation tuning seen for sequential computers, but a change in design goals. For example, the design goal of reducing total operation count, which may lead to better sequential algorithms, is the wrong focus for use when the algorithm is to run on an SIMD computer. Given N PEs, N operations may take no more time than one operation, provided the N operations involve no sequential data dependencies. For SIMD computing, executed instruction count reduction serves the analogous purpose as operation count reduction does for sequential machines. There are many tradeoffs available to the parallel algorithm designer.

3.1 Tradeoffs

One important implementation decision concerns how to efficiently distribute the data over the available processors. In a distributed memory machine, such as the CM-2, a balance should be achieved between computation time and communication time in an attempt to best utilize the architecture to achieve high performance. A low data-point to processor ratio would be favored given a fast interconnection network and limited computational power per processor. If the number of available processors is large enough to allow this low ratio, significant speedup may be realized through the high degree of parallelism. This scenario characterizes the CM-2 without the floating-point hardware installed. Data sets with as many as 65536 points can be processed with a 1:1 data-point to processor ratio on a fully configured CM-2. However, with the optional floating-point hardware installed, it may be advantageous to increase the number of data points per processor.

A second important implementation decision is generation of twiddle factors. Three possible choices have been considered. The choice of which approach to take depends upon available memory, I/O bandwidth, and processing capabilities.

(1) **Table look-up.** Initially, available processors compute the $N/2$ twiddle factors and store these values in a table. For subsequent FFTs, the

twiddle factors would be available for use without recalculation. In a shared memory machine with fast access rates, this may be the best choice. However, in distributed memory systems multiple copies of this table have to be stored and therefore require large amounts of memory (see Table 2).

(2) **Direct calculation.** The twiddle factors can be computed directly from the equation $W^k = \cos[(2\pi/N)k] - j\sin[(2\pi/N)k]$. Calculation of the trigonometric functions on each iteration may be time consuming. However, for large values of N, there may not be enough additional memory to store a table of twiddle factors.

(3) **Permutation.** This method depends on a fast interconnection network. Initially, the twiddle factors are distributed among the processors according to the calculations required in the first stage. On subsequent stages, half of the twiddle factors are permuted each to two other processors. For example, using N PEs to compute a 1-dimensional N-point DIF FFT, the following permutation can be used. After iteration i, where i decrements by 1 from $\log_2 N-1$ to 0, only those PEs with address $A = (a_{n-1}\, a_{n-2} \ldots a_1\, a_0)$ matching the tag $(a_{n-1}\, a_{n-2} \ldots a_{i+1}\, 1\, a_{i-1} \ldots a_1\, 0)$ are enabled. Each of these PEs sends its twiddle factor to PE $A/2$ and to PE $(A+N)/2$.

During the evolution of FFT algorithms on conventional uniprocessor architectures particular attention was devoted to minimizing the number of required multiplications and additions. This lead to the use of *higher* radix algorithms.

The butterfly for the radix-2 DIF FFT is shown in Figure 5. An N-point 1-dimensional FFT requires N/2 butterfly operations at each of the $\log_2 N$ stages. The inputs, A_C and B_C, are complex numbers denoted by the subscript C. The real number representation of $A_C = A_R + jA_I$ where A_R is the real part and A_I is the imaginary part of the complex number ($j = \sqrt{-1}$). Using temporary registers, T1 and T2, Figure 6 lists the actual arithmetic operations required to perform a radix-2 butterfly. A count of the arithmetic operations yields 4 real multiplications and 6 real additions. Since there are $\frac{N}{2}\log_2 N$ butterfly operations, the exact count of operations for a complex-valued FFT can be computed and is shown in Table 1.

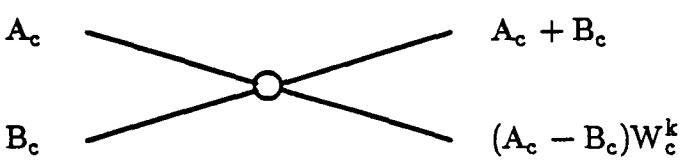
A similar analysis can be applied to radix-4 algorithms. The radix-4 butterfly is shown in Figure 7. In this case, four temporary registers are needed. Each radix-4 butterfly requires 12 real multiplications and 22 real additions.

In general, a radix-r algorithm for a 1-dimensional FFT requires N/r butterfly operations at each of the $\log_r N$ stages. Each of the N/r butterfly operations at stage i, where $0 \leq i < \log_r N$, are independent and therefore can be executed simultaneously. On an SIMD machine with N/2 processors, if

Table 2: Twiddle factor storage requirements.

Method	Memory (Bytes)	
	Single Precision	Double Precision
Table	$4N\log_2 N - 4N$	$8N\log_2 N - 8N$
Direct	0†	0†
Permutation	$4N$	$8N$

† No storage space required, assuming scratch area available.



$$A_C = A_R + jA_I$$
$$B_C = B_R + jB_I$$
$$W_C = W_R + jW_I$$

Figure 5: DIF complex, radix-2 butterfly.

$$
\begin{aligned}
T1 &\leftarrow A_R - B_R \\
T2 &\leftarrow A_I - B_I \\
A_R &\leftarrow A_R + B_R \\
A_I &\leftarrow A_I + B_I \\
B_R &\leftarrow T1 \times W_R^k \\
B_I &\leftarrow T2 \times W_I^k \\
B_R &\leftarrow B_R - B_I \\
B_I &\leftarrow T2 \times W_R^k \\
T2 &\leftarrow T1 \times W_I^k \\
B_I &\leftarrow T2 + B_I
\end{aligned}
$$

Figure 6: Radix-2 butterfly real-valued arithmetic operations

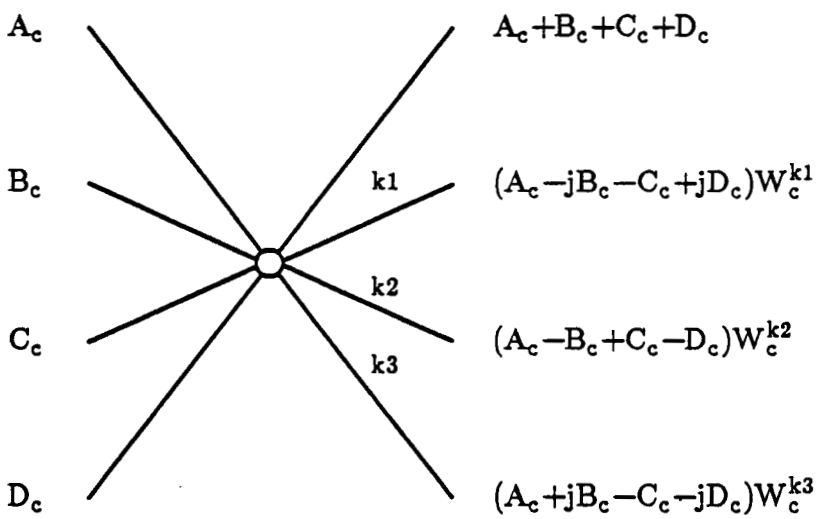


Figure 7: DIF complex, radix-4 butterfly.

each processor performs one radix-2 butterfly operation then the total number of arithmetic operations issued equals $10\log_2 N$. Similarly, a parallel radix-4 algorithm using N/4 processors would require $34\log_4 N$ or $17\log_2 N$ arithmetic operations. Consequently, when N/2 processors are available it would be advantageous to use the radix-2 algorithm since $10\log_2 N < 17\log_2 N$.

3.2 FFTs on Conventional Vector Supercomputers

As computer technology advances, algorithms intended to have very high performance must continually be analyzed to determine what characteristics are currently important. Previously, improving FFT algorithms meant reducing the number of multiplications required and performing the transform *in-place* to reduce storage requirements. However, with a clock cycle of 4.1 nanoseconds and 256 MWords (2048 MBytes) of memory, the Cray-2 illustrates that algorithm design priorities must be dynamic. Issues involved in algorithm design for the Cray-2 include minimizing memory contention through the use of unit-stride memory accesses and operating on long vectors [BAI87, SWA87]. Since the memory is interleaved into 128 banks, the processor is able to fetch a word (64-bits) from memory on every clock cycle provided there are no conflicts with other processors and each bank is accessed only once every 128 clock cycles. The worst case occurs when the data has a stride that is a multiple of 128, which means all the data is on the same bank.

3.3 FFTs on the CM-2

A radix-2 algorithm was implemented in C/Paris (see **CM_fft.c** in Appendix) that computes the 1-dimensional transform of a set of N complex data points. Due to available memory restrictions and the lack of transcendental functions in Version 4.3 of the CM-2 software, permutation was chosen as the method for twiddle factor generation (see Table 2). The twiddle factors are computed by the front-end processor and then loaded into the appropriate PEs. This program requires N processors and $6P + 2\log_2 N + 1$ bits of memory per processor, where P is the number of bits used to represent a floating point number. Figure 8 and Table 3 describe the memory map of each processor.

From the formula in Table 2, the largest one-dimensional, double precision FFT that can be done within the bounds of the CM-2 memory space (512 MBytes) is 8M points. The amount of memory required for each virtual processor is 431 bits. Therefore, with 65536 bits of memory available per processor, the largest possible virtual processor ratio (must be an integer power of 2) is 128:1. Using a fully configured CM-2 with 65536 processors, a total of 128×65536 or 8M (M=1,048,576) virtual processors are available. This size restriction only applies to this specific program. Larger FFTs may

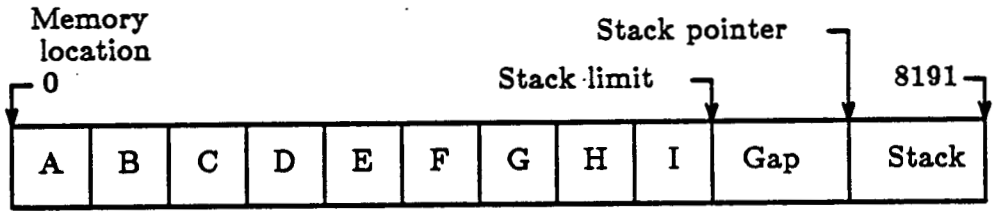


Figure 8: Memory map of each PE for program CM_fft.c

Table 3: PE memory space requirements.

Field	Contents	Bits*
A	Self Address	$\log_2 N$
B	Real Data Value	P
C	Complex Data Value	P
D	Real Data Value	P
E	Complex Data Value	P
F	Real W Value	P
G	Complex W Value	P
H	Temporary Flag	1
I	Destination Address	$\log_2 N$
Gap	Stack growth space	0†
Stack	Not used	0†
	TOTAL	$6P+2\log_2 N+1$

* Single Precision P=32, Double Precision P=64

† No space required; algorithm does not use stack

be performed by using more efficient storage techniques. For example, a program that allocates 256 points per processor with a virtual processor ratio of 1:1 can transform a 16M point array. This implementation requires only 512 KBytes of memory for twiddle factors, a savings of more than 63 MBytes. This savings is at the expense of throughput since each twiddle factor must be computed before its use.

The C/Paris program performs the 1-D FFT with $2N\log_2 N - 2N$ real multiplications and $3N\log_2 N - N$ real additions. Since these operations are done in parallel on N processors, the actual serial operation count is only $4\log_2 N - 4$ real multiplications and $6\log_2 N - 2$ real additions. Parallel algorithm design on SIMD machines requires that the number of serial operations be kept to a minimum, not so much the total number of operations. For example, in serial machines the total number of complex multiplications required in the first stage of a DIF FFT algorithm may be reduced from N/2 to $N/2 - 4$ by taking advantage of twiddle factors of ± 1 and $\pm j$. However, on an SIMD machine such as the CM-2, all the complex multiplications required in the first stage are executed simultaneously if N/2 processors are used. Therefore, it would be more efficient to actually perform the complex multiplications of ± 1 and $\pm j$ rather than take additional time to disable those processors and treat them as special cases. This holds as long as at least one nontrivial complex multiplication must be performed.

Figure 9 illustrates the total number of floating point instructions executed on the CM-2 for radix-2 and radix-4 algorithms. It is assumed that the radix-2 algorithm uses N/2 PEs for an N-point FFT, while the radix-4 algorithm uses N/4 PEs. Since the CM-2 has a maximum of 64K physical processors, FFTs requiring more PEs use virtual processors.

After the installation of Version 5.0β of the system software, two additional C/Paris programs were written, **CM_fft_t_2.c** and **CM_fft_t_1.c**. Both programs perform complex-valued multidimensional FFTs. However, **CM_fft_t_1.c** requires N PEs for an N-point FFT while **CM_fft_t_2.c** requires only N/2 PEs. For the timing measurements taken, the twiddle factors, W^k, are assumed precomputed and available in local memory. These values may also be computed directly in-line as shown in the variant programs **CM_fft_d_2.c** and **CM_fft_d_1.c** in the Appendix.

In the case of **CM_fft_t_1.c**, only half of the processors are active at any given time. The butterfly operation is divided between two PEs. Specifically, at stage i, where i decrements from $\log_2 N-1$ to 0, the $\frac{N}{2}$ PEs that have an address that matches the bit mask $X^{\log_2 N-i-1}0X^i$ are enabled first. The symbol X denotes a DON'T CARE in the mask. The other $\frac{N}{2}$ PEs remain idle while the enabled PEs compute $x(P) + x(P + \frac{N}{2})$, where P is the address of the active PE. Subsequently, the context flag in each PE is inverted thereby

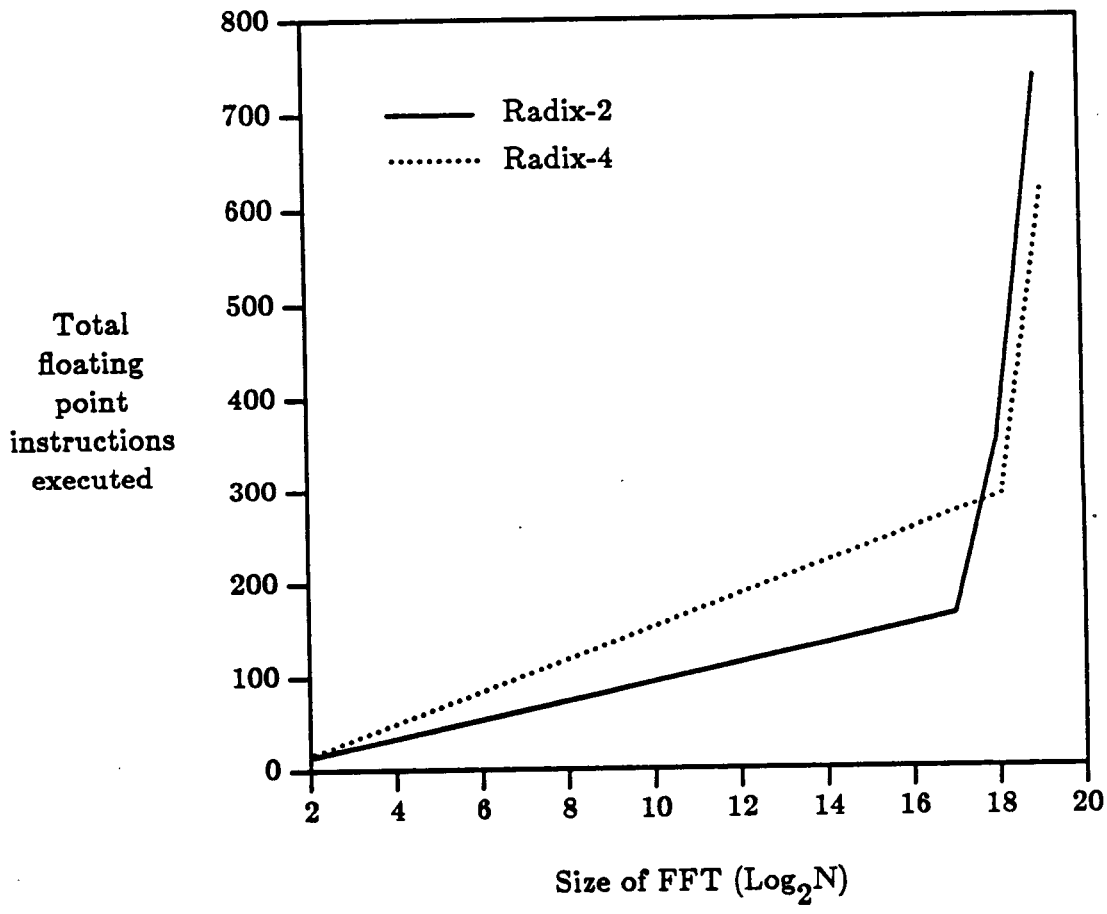


Figure 9: Floating point instruction count for radix-2 and radix-4 FFTs for instructions that are executed on the CM-2.

enabling those PEs that were previously idle and vice-versa. Then the remaining butterfly operation of $(x(P-\frac{N}{2}) - x(P))W^k$ is performed.

After a butterfly is completed, the data is realigned for the next stage of the FFT. This requires PE P exchange its newly computed data with PE $P \oplus i$. Since the data are complex values, each stage requires two transfers.

After reviewing **CM_fft_t_1.c**, it should be evident that the PE utilization is only about 50%. Therefore, it appears possible to construct a program that requires N/2 PEs and does not suffer any degradation in performance. In fact, this is exactly what was done in **CM_fft_t_2.c**. With this modification, we are now able to perform an FFT twice as large as previously possible using **CM_fft_t_1.c**. Unfortunately, for problem sizes less than this upper bound, the $\frac{N}{2}$ program suffers from increased overhead because of the problem of aligning the data between stages. Specifically, $\frac{N}{2}$ PEs need to transfer their lower butterfly operation result, while the remaining PEs need to transfer their upper butterfly result. These two results must be sent to two different memory locations in the receiving PEs. In order to perform this permutation in one (actually two for complex numbers) interprocessor exchanges, the data must be moved to a common source location. This problem can be alleviated with the implementation of the **CM_store_1L** Paris instruction. This instruction allows use of indirect addressing techniques. Currently, extra data shuffling must be done, resulting in slightly lower performance figures for **CM_fft_t_2.c**.

4. Performance

One of the fundamental problems researchers encounter with parallel machines is the difficulty in adequately comparing the performance of machines of different architectures [DEA86, MYA88]. Indeed, this is the case for the CM-2 and Cray-2. An effort was made to keep many of the variables constant. For example, all programs compared compute the complex-valued FFT. Whenever possible, the CM-2 programs operate on double precision (64-bit) floating point numbers to maintain consistency with the Cray-2 results. However, the floating point hardware installed in the CM-2 only has 32-bit capability. This should be kept in mind when evaluating these results.

The two and three dimensional FFT results are expressed in MFLOPS (*M*illions of *F*loating *P*oint *O*perations per *S*econd). Unfortunately, this unit of measure may lead to different interpretations of the results. Since it has been shown that a complex-valued radix-2 FFT can be performed using only $5N\log_2 N$ floating point operations, this number will serve as the basis for comparisons. That is, no matter how many floating point operations a specific complex-valued FFT program actually executes, the MFLOPS rate is computed using

$$\text{rate} = \frac{5N\log_2 N}{\text{execution time in microseconds}} .$$

The MFLOPS rate may be misinterpreted if this guideline is not followed. Consider a program that uses more than the required $5N\log_2 N$ floating point operations. While this program might require more actual execution time than a radix-2 version, its MFLOPS rate could potentially be higher, giving a false impression of better performance. By following the imposed guideline, both programs would be compared based on the given formula, thereby producing a higher MFLOPS rate for the radix-2 program as expected.

4.1 Measurement Techniques on the CM-2

Few tools have been developed to allow the user of the CM-2 to effectively and accurately collect performance information. A tool that can be used is **gprof**, a Unix utility that produces an execution profile of C programs. The profile data is taken from a file that is created by programs compiled with the **-pg** option. That option also links in versions of the library routines that have been compiled to generate profiling information.

First, a profile gives the total execution times and call counts for each of the functions in the program, sorted by decreasing time. A second listing shows the functions sorted according to execution time, including the execution time of their descendents. Associated with each function entry are its children, and how their times are propagated to this function. An advantage of **gprof** is that CM-2 programs do not have to be modified in

order to gather performance information. However, the results of **gprof** should only be used to compare relative performance, since **gprof** only profiles host activities and not those of the CM-2. For example, this utility might be used when trying to speed up a program; in this case, **gprof** can establish a basis for performance measurements.

The CM-2 software includes a timing facility for reporting both real time and CM-2 active time. This facility consists of three Paris instructions:

CM_start_timer(1) - Begins accumulating timing information
CM_stop_timer(0) - Stops accumulating timing information
CM_reset_timer() - Clears timing information

These timing instructions can be inserted around any block of code in either a C/Paris program or C* program. If **CM_stop_timer(1)** is used, the timing information is automatically printed out. However, in Version 4.3, this function gives times to only two significant figures. Therefore, the programmer can use **CM_stop_timer(0)** which returns a pointer of type *CM_timeval_t* that points to where the full precision timing data is stored. Use of these functions is shown in Figure 10.

Initially **CM_start_timer** clears out all the sequencer queues and makes sure all the previous Paris instructions have finished, then it reads the system time from the front end and resets the idle timer in the CM-2. The idle timer increments a counter whenever the microsequencer is waiting for an instruction from the front end. After initialization, the program executes normally until **CM_stop_timer** is encountered. At this point, the system time and idle timer are read. The elapsed front end time, CM-2 time, and CM-2 utilization can then be calculated.

For accurate results, the duration of time being measured must be much greater than the initialization latencies. Typically, program segments taking on the order of 10 seconds produce consistent results. Iterative loops can be used to measure the time of short segments or even single instructions.

4.2 Experimental Results

High performance FFT algorithm design requires a knowledge of the time to perform various elementary instructions and data transmissions. Table 4 lists execution times of various Paris instructions performed on 8192 PEs (based on 100,000 iterations of each instruction).

Figure 11 shows the time required for various communication tasks in the CM-2, and can be understood as follows. The pipeline through the router is 29 bits (12 cube wires x 2 bits per dimension + 1 for overhead + 4 for

```
#include <stdio.h>
#include <cm/cm.h>

main ()
{
  CM_timeval_t *cmtvp;
  CM_init();
    /*
     *      Program Segment
     */
  CM_start_timer(1);
    /*
     *      Program Segment to be timed
     */
  cmtvp = CM_stop_timer(0);
  printf("Real time: %g sec; CM time: %g sec; CM utilization: %g%%",
        cmtvp->cmtv_real, cmtvp->cmtv_cm,
        cmtvp->cmtv_real > 0.0 ?
          100.0 * cmtvp->cmtv_cm / cmtvp->cmtv_real : 0.0);
  CM_reset_timer();
    /*
     *      Program Segment
     */
}
```

Figure 10: Use of CM_start_timer, CM_stop_timer, and CM_reset_timer.

Table 4: CM-2 Version 4.3 Paris instruction timings.

Operation	Execution time (μsec)	
	32-bit	64-bit
CM_f_multiply †	532.8	1704.7
CM_f_divide †	1224.7	4893.6
CM_f_add †	555.1	1042.9
CM_f_subtract †	557.1	1045.2
CM_multiply2 ††	1117.9	3739.9
CM_add2 ††	32.2	45.2
CM_move	32.5	43.1
CM_move_constant	51.8	105.7
CM_iszero	27.1	33.3
CM_send ($P_i \rightarrow P_{i+16}$)	1353.6	2137.9
CM_send ($P_i \rightarrow P_{i+1}$)	365.4	543.7
$Cube_i$ ($i<4$)	366.3	545.3
$Cube_i$ ($4 \leq i < 13$)	1165.1	1825.4

† floating point operation, no floating point hardware
†† integer operation

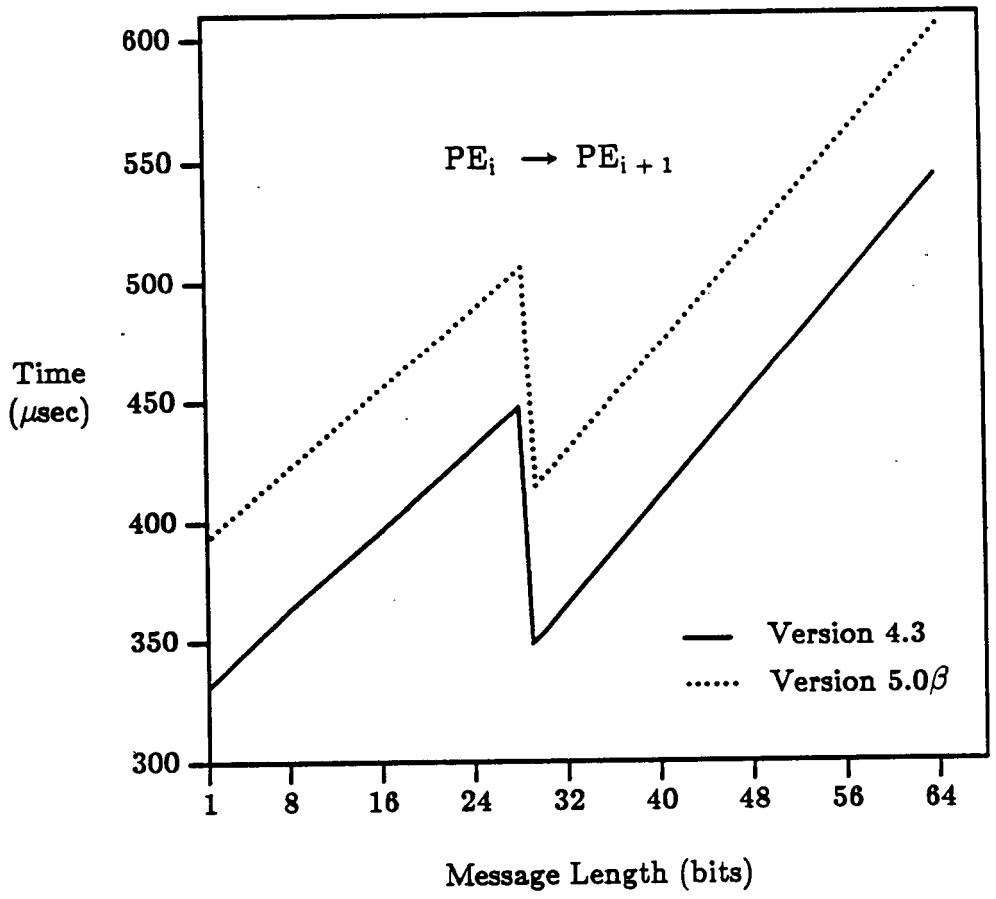


Figure 11: CM-2 communication timings.

combining functions), therefore, the natural message length is 29 bits. Rather than collapse each pipeline down as the message length decreases, the router microcode copies the message into a temporary location if it is less than 29 bits long, pads the message to 29 bits, then sends the message. This copying of shorter messages accounts for the slight increase in delivery times with message length increasing from one bit and then sudden decrease at a message length of 29 bits.

Figure 12 illustrates an inherent difference that exists between parallel and vector architectures, specifically the CM-2 and Cray-2. On this semilog graph of FFT performance figures, the time required on a SIMD machine, the CM-2, follows a straight line. However, the time required by the Cray-2 [BAI88] follows an exponential curve. The reason for this difference is that as the size of the 1-dimensional FFT increases, the number of processors in the CM-2 is also allowed to increase. At the same time, the amount of hardware represented by the Cray-2 remains constant.

Table 5 includes times realized from executing various size one-dimensional FFTs on the CM-2 Version 4.3 with 8K physical processors. Times do not include down-loading the initial data or unscrambling the results (bit-reversal).

Additional execution timing measurements were taken based on the two Version 5.0β C/Paris programs, **CM_fft_t_2.c** and **CM_fft_t_1.c**. Table 6 contains these performance measurements for both the Cray-2 and CM-2. As expected, the performance figures for the Cray-2 reflect the advantage of using longer vector length operations. For example, a 4096 point FFT arranged as a 128×32 two-dimensional transform results in a higher MFLOPS rate than the same size FFT arranged as 16×16×16 in three dimensions. However, this is not the case for the CM-2. The performance figures for the CM-2 depend on the size of the FFT rather than the dimensions of the transform.

The results in Table 6 demonstrate that the CM-2 is most effective on large FFTs, that is, when all of the available processors are utilized.

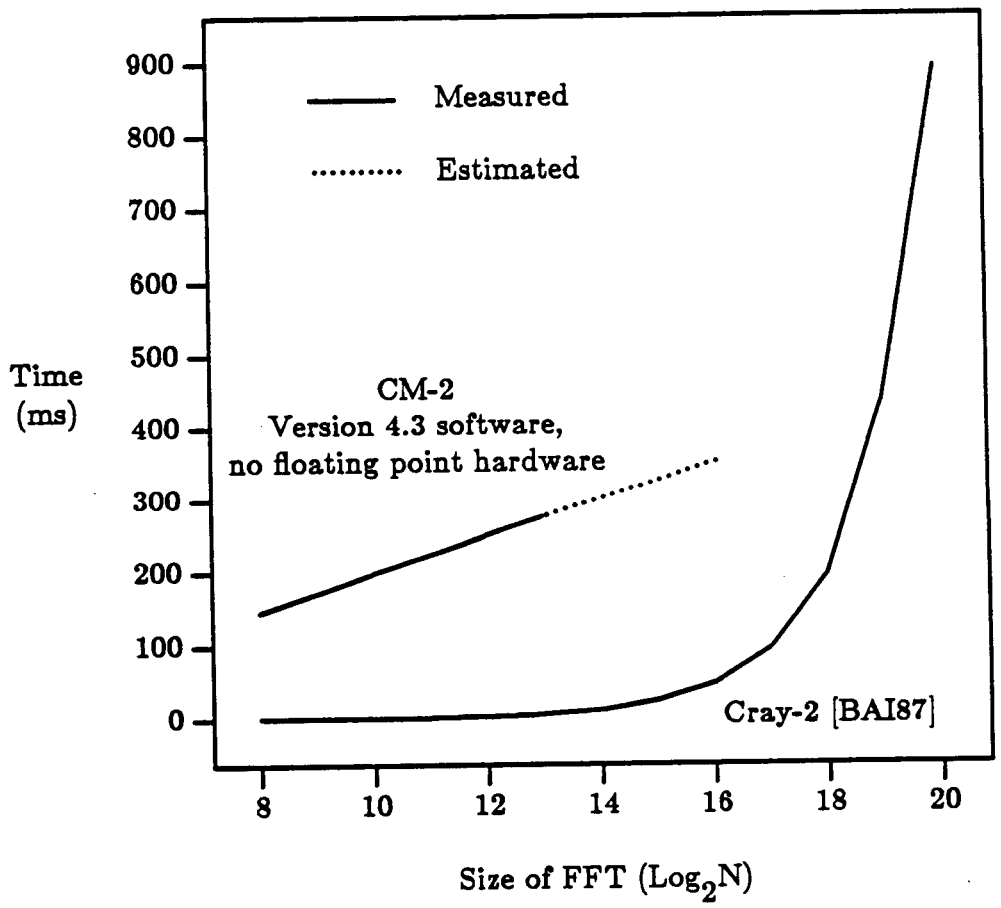


Figure 12: 1-D complex FFT execution time using **CM_fft.c** in the Appendix.

Table 5: 1-D complex-valued FFT timings using **CM_fft.c** in the Appendix, running under Version 4.3.

Size	Log(Size)	Virtual Processor Ratio	Execution time (ms)	
			Double Precision	Single Precision
131072	17	16:1	7229.0	3806.0
65536	16	8:1	3431.0	1807.0
32768	15	4:1	1345.4	704.1
16384	14	2:1	595.3	306.3
8192	13	1:1	275.2	140.9
4096	12	1:1	249.4	127.4
2048	11	1:1	223.1	113.5
1024	10	1:1	198.0	100.4
512	9	1:1	171.0	86.2

Table 6: Multidimensional complex-valued FFT MFLOPS

FFT Size	Cray-2	CM-2 (Version 5.0β)	
	1 CPU	CM_fft_t_2.c N/2 PEs	CM_fft_t_1.c N PEs
32×32	118.58	1.95	2.04
32×64	172.37	3.77	4.00
32×128	168.79	7.30	7.90
32×256	181.19	14.45	15.66
32×512	189.14	28.50	30.51
32×1024	194.01	55.25	61.32
64×32	167.26	3.77	4.00
64×64	220.58	7.40	7.85
64×128	236.36	14.43	15.60
64×256	255.82	28.02	30.70
64×512	230.15	55.55	60.97
64×1024	249.32	111.17	N/A
128×32	177.66	7.30	7.90
128×64	231.82	14.43	15.60
128×128	243.00	28.27	30.84
128×256	244.98	54.72	59.26
128×512	243.84	111.17	N/A
256×32	194.82	14.45	15.66
256×64	245.24	28.02	30.70
256×128	246.94	54.72	59.26
256×256	242.02	111.17	N/A
512×32	189.76	28.50	30.51
512×64	242.54	55.55	60.97
512×128	248.78	111.17	N/A
1024×32	195.19	55.25	61.32
1024×64	242.11	111.17	N/A
16×16×16	45.26	7.40	8.03
32×32×32	78.84	56.04	61.47

5. Conclusion

As the computational demands placed on scientific computers continues to escalate, massively parallel computers, such as the CM-2, will play a vital role in meeting this demand. The various algorithm design tradeoffs presented and analyzed in this report are intended to provide a better understanding of those characteristics significant to parallel execution.

Because of the highly parallel nature of FFT algorithms, it is advantageous to utilize the maximum number of available processors by evenly distributing the data among them. When $N/2$ PEs are available the radix-2 algorithm yields the best results. However, additional work is being done to analyze what effect increasing the virtual processor ratio will have on performance.

In applications that require computation of a large number of successive FFTs, precalculation of the twiddle factors will result in a substantial savings in overall execution time. However, as the size of the FFT grows, the memory requirements of this table look-up may not be feasible. In this case, the permutation scheme outlined helps to alleviate this problem. For applications requiring only a limited number of FFTs, the direct calculation scheme becomes the logical choice.

The C/Paris programs written use the general hypercube router of the CM-2 for interprocessor communications. However, since the interconnection patterns required for the FFT use only nearest-neighbor communication along the hypercube, the full functionality of the general router creates unnecessary overhead. For this reason, an investigation is also being made into the possible use of more primitive routing instructions.

6. Acknowledgement

We thank David H. Bailey for discussions and providing data on FFT performance on the Cray-2.

7. References

[BAI87] Bailey, D. H., "A High-Performance Fast Fourier Transform Algorithm for the Cray-2", *The Journal of Supercomputing*, vol. 1, 1987, pp. 43-60.

[BAI88] Bailey, D. H., "A High-Performance Fast Fourier Transform Algorithm for Vector Supercomputers", *The Journal of Supercomputing*, to appear.

[BCJ89] Bronson, E. C., Casavant, T. L., and Jamieson, L. H., "Experimental Analysis of Multi-Mode Fast Fourier Transforms", *1989 IEEE International Conference on Acoustics, Speech, and Signal Processing*, May 1989, to appear.

[BER68] Bergland, G. D., "A Fast Fourier Transform Algorithm Using Base 8 Iterations", *Math. Comp.*, vol. 22, 1968, pp. 275-279.

[BRI74] Brigham O. E., *The Fast Fourier Transform*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1974.

[COT65] Cooley, J. M., and Tukey, J. W., "An Algorithm for the Machine Calculation of Complex Fourier Series", *Math. Comp.*, vol. 19, 1965, pp. 279-301.

[DEA86] Denning, P. J., and Adams, G. B. III, "Research Questions for Performance Analysis of Supercomputers", Research Institute for Advanced Computer Science, RIACS Technical Report TR86.27, December, 1986.

[ELR82] Elliot, D. F. and Rao, K. R., *Fast Transforms: Algorithms, Analyses, Applications*, Academic Press, New York, 1982.

[JHJ88] Johnsson, L., Ho, C. T., Jacquemin, M., and Ruttenberg, A., "Systolic FFT Algorithms on Boolean Cube Networks", *International Conference on Systolic Arrays*, May 1988, pp. 151-162.

[JMS86] Jamieson, L. H., Mueller, P. T., and Siegel, H. J., "FFT Algorithms for SIMD Parallel Processing Systems", *Journal of Parallel and Distributed Computing*, vol. 3, 1986, pp. 48-71.

[MYA88] Myers, D. W., and Adams, G. B. III, "Benchmarking and Performance Analysis of the CM-2", Research Institute for Advanced Computer Science, RIACS Technical Report TR88.19, December, 1988.

[NIL83] Nilsson, J. W., *Electric Circuits*, Addison-Wesley Publishing Company, 1983, pp. 688-691.

[RAG75] Rabiner, L. R., and Gold, B., *Theory and Application of Digital Signal Processing*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975.

[SWA84] Swarztrauber, P. N., "FFT algorithms for Vector Computers", *Parallel Computing*, vol. 1, 1984, pp. 45-63.

[SWA87] Swarztrauber, P. N., "Multiprocessor FFTs", *Parallel Computing*, vol. 5, 1987, pp. 197-210.

[TMA87] "The Connection Machine System Model CM-2 Technical Summary", Technical Report HA87-4, Thinking Machines Corp., Cambridge, Massachusetts, April, 1987.

APPENDIX:

C/Paris Programs

CM_fft.c

```
/*
 * CM_fft.c - Ray Kamin - June 2, 1988
 * Paris Version 4.3
 *
 *
 * This program performs a 1-D DIF FFT on NP complex
 * points using NP processors. The data is loaded in
 * by the front end processor in normal order and read
 * back in bit-reversed order. The twiddle factors
 * are computed by the front-end and down loaded into
 * the appropriate PEs of the CM-2. The twiddle
 * factors are permuted for subsequent iterations.
 *
 *
 * Use <cc CM_fft.c -lparis -lm -o fft> to compile
 *
 *
 * Note: When changing NP, you must also change K to
 *       its corresponding value (i.e. NP=2^K).
 */

#include <math.h>
#include <stdio.h>
#include <cm/cm.h>

#define PI 3.14159265358979
#define NP     2048      /* Number of processors       */
#define K       11      /* 2**K processors            */
#define BASE     0      /* Base address of memory in PE */
#define MAN     52      /* Length of mantissa S=23 D=52 */
#define EXP     11      /* Length of exponent S=8  D=11 */


main ()
{
  double arg, real[NP], complex[NP], omega_r[NP/2], omega_c[NP/2];
  int reverse, reg[10], wdsize, i, k;

  CM_init();

  k=K;
  arg=2*PI/NP;    /* Used to compute twiddle factors */

  /* Establish some meaningless data */

  for (i=0; i<NP; i++)
    {
      real[i]=cos((double) arg*i);
      complex[i]=sin((double) arg*i);
    }
```

```
    /* Set up the complex omega values */

for (i=0; i<NP/2; i++)
 {
    omega_r[i]=cos((double) (arg*i));
    omega_c[i]=0-(sin((double) (arg*i)));
 }

    /*      Set up the static memory in each PE
     *
     * reg[0] - CM_cube_address_length-bit self-address
     * reg[1] - Real part of first data element
     * reg[2] - Complex part of first data element
     * reg[3] - Real part of second data element
     * reg[4] - Complex part of second data element
     * reg[5] - Real part of twiddle factor
     * reg[6] - Complex part of twiddle factor
     * reg[7] - 1-bit context flag for PEs < NP
     * reg[8] - Temporary address register
     */

reg[0]=BASE;
wdsize=MAN+EXP+1;

for (i=1; i<=7; i++)
 reg[i]=reg[0]+CM_cube_address_length+(i-1)*wdsize;

reg[8]=reg[7]+1;

CM_set_stack_limit(reg[8]+CM_cube_address_length);
CM_set_stack_upper_bound(CM_user_memory_address_limit);
CM_reset_stack_pointer;
CM_move_constant_always(CM_context_flag,1,1);
CM_my_cube_address(reg[0]);
CM_u_le_constant(reg[0],NP-1,CM_cube_address_length);
CM_move(reg[7],CM_test_flag,1);
CM_move(CM_context_flag,CM_test_flag, 1);

/* Load PEs with data in normal order */

for (i=0; i<NP/2; i++)
 {
    CM_f_write_to_processor(i,reg[1],real[i],MAN,EXP);
    CM_f_write_to_processor(i,reg[2],complex[i],MAN,EXP);
    CM_f_write_to_processor(i+NP/2,reg[1],real[i+NP/2],MAN,EXP);
    CM_f_write_to_processor(i+NP/2,reg[2],complex[i+NP/2],MAN,EXP);
    CM_f_write_to_processor(i+NP/2,reg[5],omega_r[i],MAN,EXP);
    CM_f_write_to_processor(i+NP/2,reg[6],omega_c[i],MAN,EXP);
 }

fft(k,reg);       /* Execute the FFT */

    /* Read data from all PEs in bit-reversed order */
```

```
 for (i=0; i<NP; i++)
  {
     reverse=ibitr(i,k);
     real[reverse]=CM_f_read_from_processor(i,reg[1],MAN,EXP);
     complex[reverse]=CM_f_read_from_processor(i,reg[2],MAN,EXP);
  }

 if (NP<=16)
  {
     for (i=0; i<NP; i++)
      {
        printf("PE %d = %g + j%g\n",i,real[i],complex[i]);
      }
  }
 else
  {
     printf("PE %d = %g + j%g\n",0,real[0],complex[0]);
     printf("PE %d = %g + j%g\n",NP-1,real[NP-1],complex[NP-1]);
  }
}

/*  1-D DIF FFT Function
 *
 *  This function executes the DIF radix-2 CTA on the data in
 *  the PEs.  The data is NOT unscrambled before returning to
 *  the main program.
 *
 */


fft(k,reg)
int k, reg[10];

{
 int i, xp;
 CM_timeval_t *cmtvp;

 CM_start_timer(1);

 xp=1;
 xp<<=(k-1);

 for (i=k-1; i>=0; i--)
  {
          /* Exchange data between PEs */
          /* actually doing CUBE(i-1)  */
    CM_move_always(CM_context_flag,reg[7],1);
    CM_u_move_constant(reg[8],xp,CM_cube_address_length);
    CM_logxor(reg[8],reg[0],CM_cube_address_length);
    CM_send(reg[3],reg[8],reg[1],MAN+EXP+1,CM_NOLIMIT);
    CM_send(reg[4],reg[8],reg[2],MAN+EXP+1,CM_NOLIMIT);

    xp>>=1;
       /* Select PEs for TOP btrfly*/
```

```
        CM_iszero(reg[0]+i,1);
        CM_move(CM_context_flag,CM_test_flag,1);
            /* DO X=A+B                */
        CM_f_add(reg[1],reg[3],MAN,EXP);
        CM_f_add(reg[2],reg[4],MAN,EXP);

            /* Select PEs for BOTTOM btrfly*/
        CM_move_always(CM_context_flag,reg[7],1);
        CM_move(CM_context_flag,reg[0]+i,1);
            /* DO X=(A-B)*twiddle factor   */
        CM_f_subtract(reg[3],reg[1],MAN,EXP);
        CM_f_subtract(reg[4],reg[2],MAN,EXP);
        CM_f_move(reg[1],reg[3],MAN,EXP);
        CM_f_move(reg[2],reg[4],MAN,EXP);
            /* Not necessary to do mult.   */
            /* if i==0 since twid. f ==1   */
        if (i>0)
         {
          CM_f_multiply(reg[3],reg[5],MAN,EXP);
          CM_f_multiply(reg[4],reg[6],MAN,EXP);
          CM_f_subtract(reg[3],reg[4],MAN,EXP);
          CM_f_multiply(reg[1],reg[6],MAN,EXP);
          CM_f_multiply(reg[2],reg[5],MAN,EXP);
          CM_f_add(reg[2],reg[1],MAN,EXP);
          CM_f_move(reg[1],reg[3],MAN,EXP);

          if (i>1)
           {
                  /* Permute twiddle factors */
            CM_iszero(reg[0],1);
            CM_move(CM_context_flag,CM_test_flag,1);
            CM_move(CM_context_flag,reg[0]+i,1);

            CM_move_constant(reg[3],-1,MAN+EXP+1);
            CM_u_move(reg[8],reg[0],CM_cube_address_length);
            CM_shift(reg[8],reg[3],CM_cube_address_length,MAN+EXP+1);
            CM_send(reg[3],reg[8],reg[5],MAN+EXP+1,CM_NOLIMIT);
            CM_send(reg[4],reg[8],reg[6],MAN+EXP+1,CM_NOLIMIT);
            CM_u_add_constant(reg[8],NP/2,CM_cube_address_length);
            CM_send(reg[3],reg[8],reg[5],MAN+EXP+1,CM_NOLIMIT);
            CM_send(reg[4],reg[8],reg[6],MAN+EXP+1,CM_NOLIMIT);

            CM_move_always(CM_context_flag,reg[7],1);
            CM_f_move(reg[5],reg[3],MAN,EXP);
            CM_f_move(reg[6],reg[4],MAN,EXP);
           }
         }
       }
      cmtvp = CM_stop_timer(0);
      printf("Real time: %g sec; CM time: %g sec; CM utilization: %g%%\n",
             cmtvp->cmtv_cm, cmtvp->cmtv_real,
             cmtvp->cmtv_cm > 0.0 ?
               100.0 * cmtvp->cmtv_real / cmtvp->cmtv_cm :
               0.0);
```

```
}

/* Bit Reversal Fuction
 *
 * This function takes an input j of length k and returns the
 * bit-reversed value.  For example ibitr(5,4)=10 since if 0101
 * is reversed, 1010 results.
 *
 */

int ibitr(j, k)
int j,k;

{
  int rev, j1, j2, i;

  j1=j;
  rev=0;
  for (i=1; i<=k; i++)
    {
      j2=j1/2;
      rev=rev*2+(j1-2*j2);
      j1=j2;
    }
  return(rev);
}
```

CM_cube.c

```
/*
 * CM_cube.c - Ray Kamin - June 16, 1988
 * Paris Version 4.3
 *
 * This program performs cube functions
 * on NP processors using the router.
 *
 * Use <cc CM_cube.c -lparis -lm -o cube> to compile
 *
 */

#include <stdio.h>
#include <cm/cm.h>

#define NP  8192          /* Number of proc
#define BASE  0           /* Base address of
#define MAN  52           /* Precision; S=2
#define EXP  11           /* Precision; S= 8


main ()
{
 int  reg[4], wdsize;

 CM_init();


      /*       Set up the static memory in ea
       *
       * reg[0] - CM_cube_address_length-bi
       * reg[1] - CM_cube_address_length-bi
       * reg[2] - Data Register
       * reg[3] - Data Register
       */

 wdsize=MAN+EXP+1;

 reg[0]=BASE;
 reg[1]=reg[0]+CM_cube_address_length;
 reg[2]=reg[1]+CM_cube_address_length;
 reg[3]=reg[2]+wdsize;

 CM_set_stack_limit(reg[3]+wdsize+10);
 CM_set_stack_upper_bound(CM_user_memo
 CM_reset_stack_pointer;
 CM_move_constant_always(CM_context_flag,
 CM_my_cube_address(reg[0]);
 CM_u_lt_constant(reg[0],NP,CM_cube_addre
 CM_move(CM_context_flag,CM_test_flag, 1);

 cube(reg);
```

```
}

/************ Execute Cube sub i Functions ****************/

cube(reg)
int  reg[4];

{
 CM_timeval_t *cmtvp;
 int size, xp, t;

 size=0;

 while (size<13)
  {
     printf("Cube sub ->");
     scanf("%d",&size);

     xp=1;
     xp<<=size;
     CM_move_constant_always(CM_context_flag,1,1);
     CM_u_move_constant(reg[1],xp,CM_cube_address_length);
     CM_logxor(reg[1],reg[0],CM_cube_address_length);

     CM_start_timer(1);
     for (t=1; t<=100000; t++)
      CM_send(reg[3],reg[1],reg[2],MAN+EXP+1,CM_NOLIMIT);
     cmtvp = CM_stop_timer(0);
     printf( "Real time: %g sec; CM time: %g sec;
       CM utilization: %g%%\n",
       cmtvp->cmtv_real, cmtvp->cmtv_cm,
       cmtvp->cmtv_real > 0.0 ?
         100.0 * cmtvp->cmtv_cm / cmtvp->cmtv_real :
         0.0);
     CM_reset_timer();
  }
}
```

CM_send_message .c

```
/*
 * CM_send_message.c - Ray Kamin - June 9, 1988
 * Paris Version 4.3
 *
 * This program sends various length messages
 * though the router in an effort to establish
 * a feel for the performance of the network.
 *
 *
 * Use <cc CM_send_message.c -lparis -o send> to compile
 *
 */

#include <stdio.h>
#include <cm/cm.h>

#define BASE 0          /* Base address of memory in PE */
#define MSGLEN 128      /* Maximum message length       */


main ()
{
 int reg[4], i;

 CM_init();


      /*       Set up the static memory in each PE
       *
       * reg[0] - CM_cube_address_length-bit self-address
       * reg[1] - CM_cube_address_length-bit dest. address
       * reg[2] - <MSGLEN> length register
       * reg[3] - <MSGLEN> length register
       */

 reg[0]=BASE;
 reg[1]=reg[0]+CM_cube_address_length;
 reg[2]=reg[1]+CM_cube_address_length;
 reg[3]=reg[2]+MSGLEN;

 CM_set_stack_limit(reg[3]+MSGLEN);
 CM_set_stack_upper_bound(CM_user_memory_address_limit);
 CM_reset_stack_pointer;
 CM_move_constant_always(CM_context_flag,1,1);
 CM_my_cube_address(reg[0]);

 /* Permutation : PE i -> PE i+16 */

 CM_move_constant(reg[1],16,CM_cube_address_length);
 CM_add2(reg[1],reg[0],CM_cube_address_length);
```

```
  printf("Sending . . . \n");
  send(reg);
}

/************** transfer message routine ******************/

send(reg)
int reg[4];

{
  int t, size;
  CM_timeval_t *cmtvp;

  size=1;

  while (size != 0) {
    printf("Enter the message length-->");
    scanf("%d",&size);
    CM_start_timer(1);
    for (t=1; t<=100000; t++)
       CM_send(reg[3],reg[1],reg[2],size);
    cmtvp = CM_stop_timer(0);
    printf( "Real time: %g sec; CM time: %g sec;
            CM utilization: %g%%\n",
            cmtvp->cmtv_real, cmtvp->cmtv_cm,
            cmtvp->cmtv_real > 0.0 ?
              100.0 * cmtvp->cmtv_cm / cmtvp->cmtv_real :
              0.0);
    CM_reset_timer();
    }
}
```

CM_fft_t_1.c

```
/*
 *      CM_fft_t_1.c - R. Kamin - Sept. 1, 1988
 *      Paris Version 5.0B
 *
 *    This program performs a multidimensional DIF FFT on N
 *    complex points using N processors.  The data is loaded
 *    in by the front end processor in normal order and read
 *    back in bit-reversed order.  The twiddle factors
 *    are assumed precomputed by the CM-2 processors.
 *
 *
 *    Use <cc XXX.c -lparis -lm -o fft> to compile
 *
 *
 *   Note: Must use 32-bit floating point to use FPU
 */

#include <math.h>
#include <stdio.h>
#include <cm/paris.h>

#define PI  3.14159265358979323846
#define MAX_N      32769
#define MAN          23    /* Length of mantissa; D=52 S=23    */
#define EXP          8     /* Length of exponent; D=11 S=8     */
#define INT_SIZE   MAN+EXP+1    /* Length of integer             */
#define MAX(A, B)  ((A) > (B) ? (A) : (B))


main ()
{
 int max_dim, N, reg[10], wdsize, i, n, base[15], size[15];

 CM_init();

 printf("Enter the number of dimensions ->");
 scanf("%d",&max_dim);

 base[0]=0;

 for (i=1; i<=max_dim; i++)
  {
    printf("Size of dimension %d (in log base 2) ->",i);
    scanf("%d",&size[i]);
    base[i]=base[i-1]+size[i];
  }

 n=base[max_dim];
 N=(1<<n);
```

```
    /*      Set up the static memory in each PE
     *
     * reg[0] - Self-address
     * reg[1] - First data register (REAL PART)
     * reg[2] - First data register (COMPLEX PART)
     * reg[3] - Second data register (REAL PART)
     * reg[4] - Second data register (COMPLEX PART)
     * reg[5] - 1-bit context flag for PEs < N
     * reg[6] - Destination address register
     * reg[7] - Twiddle factor register (REAL PART)
     * reg[8] - Twiddle factor register (COMPLEX PART)
     */

    wdsize=MAN+EXP+1;

    reg[0]=CM_allocate_stack_field(INT_SIZE);
    reg[1]=CM_allocate_stack_field(wdsize);
    reg[2]=CM_allocate_stack_field(wdsize);
    reg[3]=CM_allocate_stack_field(wdsize);
    reg[4]=CM_allocate_stack_field(wdsize);
    reg[5]=CM_allocate_stack_field(1);
    reg[6]=CM_allocate_stack_field(INT_SIZE);
    reg[7]=CM_allocate_stack_field(wdsize);
    reg[8]=CM_allocate_stack_field(wdsize);

    CM_set_context;                                  /* Activate all PEs  */
    CM_u_move_constant_1L(reg[5],0,1);
    printf("Setting addresses of %d processors . . .",MAX(N,8192));
    for (i=0; i<MAX(N,8192); i++)
      CM_u_write_to_processor_1L(i,reg[0],i,INT_SIZE);
    printf("done!\n");

    printf("Activating %d processors . . .",N);
    CM_u_lt_constant_1L(reg[0],N,INT_SIZE);          /* Select low N PEs */
    CM_load_context(CM_test_flag);
    CM_u_move_constant_1L(reg[5],1,1);
    printf("done!\n");

        /*  Load PEs with data in normal order */

    printf("Loading data . . .");
    CM_f_move_constant_1L(reg[1],0.0,MAN,EXP);
    CM_f_move_constant_1L(reg[2],0.0,MAN,EXP);
    CM_f_move_constant_1L(reg[3],0.0,MAN,EXP);
    CM_f_move_constant_1L(reg[4],0.0,MAN,EXP);
    CM_f_move_constant_1L(reg[7],0.0,MAN,EXP);
    CM_f_move_constant_1L(reg[8],0.0,MAN,EXP);
    printf("done!\n");

    fft(n,reg,base,max_dim);                         /* Execute the FFT */

}

/********************** 1-D DIF FFT Function ******************************/
```

```
fft(n,reg,base,dimension)
int n, reg[10], base[15], dimension;

{
 int lfactor, q, t, i, two_i, current_dim, previous_dim;
 double arg;
 CM_timeval_t *cmtvp;

 printf("Enter loop factor ->");
 scanf("%d",&lfactor);
 printf("\n");
 q=dimension;
 CM_start_timer(1);

 for (t=0; t<lfactor; t++)
 {
 dimension=q;

 for (i=n-1; i>=0; i--)
  {
   if (i<base[dimension])
    dimension=dimension-1;
   current_dim=base[dimension];
   previous_dim=base[dimension+1];

                /* Exchange data between PEs    */
                /* actually doing CUBE(i)    */
   two_i=(1<<i);

   CM_u_move_always_1L(CM_context_flag,reg[5],1);
   CM_logxor_constant_3_1L(reg[6],reg[0],two_i,INT_SIZE);
   CM_send_1L(reg[3],reg[6],reg[1],MAN+EXP+1);
   CM_send_1L(reg[4],reg[6],reg[2],MAN+EXP+1);

            /* Select PEs for TOP butterfly        */
   CM_u_eq_zero_1L(reg[0]+i,1);
   CM_load_context(CM_test_flag);
            /* DO X=A+B          */
   CM_f_add_2_1L(reg[1],reg[3],MAN,EXP);
   CM_f_add_2_1L(reg[2],reg[4],MAN,EXP);

            /* Select PEs for BOTTOM butterfly     */
   CM_u_move_always_1L(CM_context_flag,reg[5],1);
   CM_load_test(reg[0]+i);
   CM_load_context(CM_test_flag);
            /* DO X=(A-B)          */
   CM_f_subtract_2_1L(reg[1],reg[3],MAN,EXP);
   CM_f_subtract_2_1L(reg[2],reg[4],MAN,EXP);

            /* Not necessary to do mult.    */
            /* if i=current_dim since w=1    */
   if (i>current_dim)
    {
            /* Multiply by twiddle fact.    */
```

```
        CM_f_multiply_3_1L(reg[4],reg[2],reg[8],MAN,EXP);
        CM_f_multiply_2_1L(reg[2],reg[7],MAN,EXP);
        CM_f_multiply_2_1L(reg[7],reg[1],MAN,EXP);
        CM_f_multiply_2_1L(reg[1],reg[8],MAN,EXP);
        CM_f_add_2_1L(reg[2],reg[1],MAN,EXP);
        CM_f_subtract_3_1L(reg[1],reg[7],reg[4],MAN,EXP);
      }
    }
  }
  cmtvp=CM_stop_timer(0);
  i=(1<<n);
  printf("==========================================================
  printf("                CM_fft_t_1.c      Paris Version 5.0B\n");
  printf("\nTimes computed for a single %d-point %d-dimension complex FFT\n",i,q);
  printf("Times based on a loop factor of %d and using %d PEs\n",lfactor,i);
  printf("Twiddle factors are precomputed and looked-up.\n");
  printf("Floating point hardware is ACTIVE!\n");
  printf("----------------------------------------------------------------\n");
  printf("Real time: %g sec; CM time: %g sec; CM utilization %g%%\n",
       cmtvp->cmtv_real/lfactor, cmtvp->cmtv_cm/lfactor,
       cmtvp->cmtv_real > 0.0 ?
       100.0 * cmtvp->cmtv_cm / cmtvp->cmtv_real:0.0);
  printf("Using a 5NlogN counter, %g MFLOPS\n",.000005*i*n/cmtvp->cmtv_cm*lfactor);
  printf("==========================================================
  CM_reset_timer();
}
```

CM_fft_t_2.c

```
/*
 *     CM_fft_t_2.c - R. Kamin - October 22, 1988
 *     Paris Version 5.0B
 *
 *    This program performs a multidimensional DIF FFT on
 *    N complex points using N/2 processors.  The data is
 *    loaded in by the front end processor in normal order
 *    and read back in bit-reversed order.  The twiddle
 *    factors are assumed precomputed by the CM-2 PEs.
 *
 *
 *    Use <cc XXX.c -lparis -lm -o fft> to compile
 *
 *
 *   Note: Must use 32-bit format to take advantage of FPU
 */

#include <math.h>
#include <stdio.h>
#include <cm/paris.h>

#define PI  3.14159265358979323846
#define MAX_N      32769
#define MAN         23    /* Length of mantissa; D=52 S=23    */
#define EXP         8    /* Length of exponent; D=11 S=8     */
#define INT_SIZE   MAN+EXP+1    /* Length of integer           */
#define MAX(A, B)  ((A) > (B) ? (A) : (B))


main ()
{
 int max_dim, N, reg[17], wdsize, i, n, base[15], size[15];

 CM_init();

 printf("Enter the number of dimensions ->");
 scanf("%d",&max_dim);

 base[0]=0;

 for (i=1; i<=max_dim; i++)
  {
    printf("Size of dimension %d (in log base 2) ->",i);
    scanf("%d",&size[i]);
    base[i]=base[i-1]+size[i];
  }

 n=base[max_dim];
 N=(1<<n);
```

```
    /*      Set up the static memory in each PE
     *
     * reg[0]    -  SR = Self-address Register
     * reg[1&2]  -  R1 = Complex Register #1
     * reg[3&4]  -  R2 = Complex Register #2
     * reg[5&6]  -  T1 = Temporary Complex Register #1
     * reg[7&8]  -  T2 = Temporary Complex Register #2
     * reg[9&10] -   W = Complex Twiddle Factor Register
     * reg[11&12] - DTR1 = Complex Data Transfer Register #1
     * reg[13&14] - DTR2 = Complex Data Transfer Register #2
     * reg[15]   -  DR = Destination address Register
     * reg[16]   -  CFS = Context Flag storage register
     */

wdsize=MAN+EXP+1;       /* Floating point number length      */

reg[0]=CM_allocate_stack_field(INT_SIZE);
for (i=1; i<15; i++)
  reg[i]=CM_allocate_stack_field(wdsize);
reg[15]=CM_allocate_stack_field(INT_SIZE);
reg[16]=CM_allocate_stack_field(1);

CM_set_context;             /* Activate all PEs   */
CM_u_move_constant_1L(reg[16],0,1);
printf("Setting addresses of %d processors . . .",MAX(N/2,8192));
for (i=0; i<MAX(N/2,8192); i++)
  CM_u_write_to_processor_1L(i,reg[0],i,INT_SIZE);
printf("done!\n");

printf("Activating %d processors . . .",N/2);
CM_u_lt_constant_1L(reg[0],N/2,INT_SIZE);    /* Select low N/2 PEs   */
CM_load_context(CM_test_flag);
CM_u_move_constant_1L(reg[16],1,1);
printf("done!\n");

        /*  Load PEs with data in normal order  */

printf("Loading data . . .");

CM_f_move_constant_1L(reg[1],0.0,MAN,EXP);
CM_f_move_constant_1L(reg[2],0.0,MAN,EXP);
CM_f_move_constant_1L(reg[3],0.0,MAN,EXP);
CM_f_move_constant_1L(reg[4],0.0,MAN,EXP);
CM_f_move_constant_1L(reg[9],0.0,MAN,EXP);
CM_f_move_constant_1L(reg[10],0.0,MAN,EXP);

printf("done!\n");

fft(n,reg,base,max_dim);      /* Execute the FFT      */

}

/******************** M-D DIF FFT Function ****************************/
```

```
fft(n,reg,base,dimension)
int n, reg[17], base[15], dimension;

{
 int lfactor, q, t, i, half_two_i, current_dim, previous_dim;
 double arg;
 CM_timeval_t *cmtvp;

 printf("Enter loop factor ->");
 scanf("%d",&lfactor);
 printf("\n");

 q=dimension;
 CM_start_timer(1);

 for (t=0; t<lfactor; t++)
 {
 dimension=q;

 for (i=n-1; i>=0; i--)
  {
   if (i<base[dimension])          /* Compute which dimension    */
     dimension=dimension-1;
   current_dim=base[dimension];
   previous_dim=base[dimension+1];

   half_two_i=(1<<(i-1));

   CM_load_context(reg[16]);       /* Enable N/2 PEs   */

   CM_f_add_3_1L(reg[5],reg[1],reg[3],MAN,EXP);       /* T1=R1+R2    */
   CM_f_add_3_1L(reg[6],reg[2],reg[4],MAN,EXP);
   CM_f_subtract_3_1L(reg[7],reg[1],reg[3],MAN,EXP);  /* T2=R1-R2    */
   CM_f_subtract_3_1L(reg[8],reg[2],reg[4],MAN,EXP);

   if (i>0)
    {
     if (i==current_dim)
      {
       CM_f_move_1L(reg[11],reg[7],MAN,EXP);
       CM_f_move_1L(reg[12],reg[8],MAN,EXP);
      }
     else
      {
                    /* DTR1=T2*W    */
       CM_f_multiply_3_1L(reg[11],reg[7],reg[9],MAN,EXP);
       CM_f_multiply_3_1L(reg[12],reg[8],reg[10],MAN,EXP);
       CM_f_subtract_2_1L(reg[11],reg[12],MAN,EXP);
       CM_f_multiply_3_1L(reg[12],reg[8],reg[9],MAN,EXP);
       CM_f_multiply_3_1L(reg[13],reg[7],reg[10],MAN,EXP);
       CM_f_add_2_1L(reg[12],reg[13],MAN,EXP);
      }
          /* Select PEs for BOTTOM butterfly    */
     CM_load_test(reg[0]+i-1);
```

```
        CM_load_context(CM_test_flag);

        CM_f_move_1L(reg[7],reg[11],MAN,EXP);          /* T2=DTR1   */
        CM_f_move_1L(reg[8],reg[12],MAN,EXP);
        CM_f_move_1L(reg[11],reg[5],MAN,EXP);          /* DTR1=T1   */
        CM_f_move_1L(reg[12],reg[6],MAN,EXP);

              /* Select all N/2 PEs          */
        CM_load_context(reg[16]);
        CM_logxor_constant_3_1L(reg[15],reg[0],half_two_i,INT_SIZE);
        CM_send_1L(reg[13],reg[15],reg[11],MAN+EXP+1);   /* DTR2 <- DTR1 */
        CM_send_1L(reg[14],reg[15],reg[12],MAN+EXP+1);

        CM_f_move_1L(reg[1],reg[5],MAN,EXP);           /* R1=T1   */
        CM_f_move_1L(reg[2],reg[6],MAN,EXP);

        CM_f_move_1L(reg[3],reg[13],MAN,EXP);          /* R2=DTR2   */
        CM_f_move_1L(reg[4],reg[14],MAN,EXP);

              /* Select PEs for BOTTOM butterfly   */
        CM_load_test(reg[0]+i-1);
        CM_load_context(CM_test_flag);

        CM_f_move_1L(reg[1],reg[13],MAN,EXP);          /* R1=DTR2   */
        CM_f_move_1L(reg[2],reg[14],MAN,EXP);
        CM_f_move_1L(reg[3],reg[7],MAN,EXP);           /* R2=T2   */
        CM_f_move_1L(reg[4],reg[8],MAN,EXP);
      }
     else
      {
        CM_f_move_1L(reg[1],reg[5],MAN,EXP);           /* R1=T1   */
        CM_f_move_1L(reg[2],reg[6],MAN,EXP);
        CM_f_move_1L(reg[3],reg[7],MAN,EXP);           /* R2=T2   */
        CM_f_move_1L(reg[4],reg[8],MAN,EXP);
      }
   }
 }
 cmtvp=CM_stop_timer(0);
 i=(1<<n);
 printf("==========================================================
 printf("          CM_fft_t_2.c          Paris Version 5.0B\n");
 printf("\nTimes computed for a single %d-point %d-dimension complex FFT.\n",i,q);
 printf("Times based on a loop factor of %d and using %d PEs.\n",lfactor,i/2);
 printf("Program uses table look-up for twiddle factor generation.\n");
 printf("Floating point hardware is ACTIVE!\n");
 printf("----------------------------------------------------------\n");
 printf("Real time: %g sec; CM time: %g sec; CM utilization %g%%\n",
      cmtvp->cmtv_real/lfactor, cmtvp->cmtv_cm/lfactor,
      cmtvp->cmtv_real > 0.0 ?
      100.0 * cmtvp->cmtv_cm / cmtvp->cmtv_real:0.0);
 printf("Using a 5NlogN counter, %g MFLOPS\n",.000005*i*n/cmtvp->cmtv_cm*lfactor);
 printf("==========================================================
 CM_reset_timer();
}
```

CM_fft_d_1.c

```
/*
 *       CM_fft_d_1.c - R. Kamin - Sept. 1, 1988
 *       Paris Version 5.0B
 *
 *    This program performs a multidimensional DIF FFT on N
 *    complex points using N processors.  The data is loaded
 *    in by the front end processor in normal order and read
 *    back in bit-reversed order.  The twiddle factors
 *    are computed in-line by the CM-2 processors.
 *
 *
 *    Use <cc XXX.c -lparis -lm -o fft> to compile
 *
 *
 *   Note: Must use 32-bit floating point to use FPU
 */

#include <math.h>
#include <stdio.h>
#include <cm/paris.h>

#define PI  3.14159265358979323846
#define MAX_N      32769
#define MAN          23    /* Length of mantissa; D=52 S=23    */
#define EXP          8     /* Length of exponent; D=11 S=8     */
#define INT_SIZE   MAN+EXP+1    /* Length of integer             */
#define MAX(A, B)  ((A) > (B) ? (A) : (B))


main ()
{
 int max_dim, N, reg[10], wdsize, i, n, base[15], size[15];

 CM_init();

 printf("Enter the number of dimensions ->");
 scanf("%d",&max_dim);

 base[0]=0;

 for (i=1; i<=max_dim; i++)
  {
    printf("Size of dimension %d (in log base 2) ->",i);
    scanf("%d",&size[i]);
    base[i]=base[i-1]+size[i];
  }

 n=base[max_dim];
 N=(1<<n);
```

```
    /*      Set up the static memory in each PE
     *
     * reg[0] - Self-address
     * reg[1] - First data register (REAL PART)
     * reg[2] - First data register (COMPLEX PART)
     * reg[3] - Second data register (REAL PART)
     * reg[4] - Second data register (COMPLEX PART)
     * reg[5] - 1-bit context flag for PEs < N
     * reg[6] - Destination address register
     * reg[7] - Temp register
     */

wdsize=MAN+EXP+1;

reg[0]=CM_allocate_stack_field(INT_SIZE);
reg[1]=CM_allocate_stack_field(wdsize);
reg[2]=CM_allocate_stack_field(wdsize);
reg[3]=CM_allocate_stack_field(wdsize);
reg[4]=CM_allocate_stack_field(wdsize);
reg[5]=CM_allocate_stack_field(1);
reg[6]=CM_allocate_stack_field(INT_SIZE);
reg[7]=CM_allocate_stack_field(wdsize);

CM_set_context;                                  /* Activate all PEs  */
CM_u_move_constant_1L(reg[5],0,1);
printf("Setting addresses of %d processors . . .",MAX(N,8192));
for (i=0; i<MAX(N,8192); i++)
  CM_u_write_to_processor_1L(i,reg[0],i,INT_SIZE);
printf("done!\n");

printf("Activating %d processors . . .",N);
CM_u_lt_constant_1L(reg[0],N,INT_SIZE);          /* Select low N PEs */
CM_load_context(CM_test_flag);
CM_u_move_constant_1L(reg[5],1,1);
printf("done!\n");

    /*  Load PEs with data in normal order */

printf("Loading data . . .");
CM_f_move_constant_1L(reg[1],0.0,MAN,EXP);
CM_f_move_constant_1L(reg[2],0.0,MAN,EXP);
CM_f_move_constant_1L(reg[3],0.0,MAN,EXP);
CM_f_move_constant_1L(reg[4],0.0,MAN,EXP);
CM_f_move_constant_1L(reg[7],0.0,MAN,EXP);
printf("done!\n");

fft(n,reg,base,max_dim);                         /* Execute the FFT */

}

/********************** M-D DIF FFT Function *******************************/

fft(n,reg,base,dimension)
int n, reg[10], base[15], dimension;
```

```
{
 int lfactor, q, t, i, two_i, mask, current_dim, previous_dim;
 double arg;
 CM_timeval_t *cmtvp;

 printf("Enter loop factor ->");
 scanf("%d",&lfactor);
 printf("\n");
 q=dimension;
 CM_start_timer(1);

 for (t=0; t<lfactor; t++)
 {
 dimension=q;

 for (i=n-1; i>=0; i--)
  {
   if (i<base[dimension])
    dimension=dimension-1;
   current_dim=base[dimension];
   previous_dim=base[dimension+1];

   mask=(1<<(previous_dim-current_dim-1))-1;

                /* Exchange data between PEs    */
                /* actually doing CUBE(i)    */
   two_i=(1<<i);

   CM_u_move_always_1L(CM_context_flag,reg[5],1);
   CM_logxor_constant_3_1L(reg[6],reg[0],two_i,INT_SIZE);
   CM_send_1L(reg[3],reg[6],reg[1],MAN+EXP+1);
   CM_send_1L(reg[4],reg[6],reg[2],MAN+EXP+1);

            /* Select PEs for TOP butterfly       */
   CM_u_eq_zero_1L(reg[0]+i,1);
   CM_load_context(CM_test_flag);
            /* DO X=A+B          */
   CM_f_add_2_1L(reg[1],reg[3],MAN,EXP);
   CM_f_add_2_1L(reg[2],reg[4],MAN,EXP);

            /* Select PEs for BOTTOM butterfly     */
   CM_u_move_always_1L(CM_context_flag,reg[5],1);
   CM_load_test(reg[0]+i);
   CM_load_context(CM_test_flag);
            /* DO X=(A-B)              */
   CM_f_subtract_2_1L(reg[1],reg[3],MAN,EXP);
   CM_f_subtract_2_1L(reg[2],reg[4],MAN,EXP);

   if (i>current_dim)
    {
     arg= -2*PI/(1<<(previous_dim-current_dim));

     CM_u_multiply_constant_3_1L(reg[6],reg[0],(1<<(previous_dim-i-1)),INT_SIZE);
     CM_logand_constant_2_1L(reg[6],mask,INT_SIZE);
```

```
        CM_f_u_float_2_2L(reg[3],reg[6],INT_SIZE,MAN,EXP);
        CM_f_multiply_constant_2_1L(reg[3],arg,MAN,EXP);
        CM_f_sin_2_1L(reg[4],reg[3],MAN,EXP);
        CM_f_cos_1_1L(reg[3],MAN,EXP);
                                /* Multiply by twiddle fact.*/
        CM_f_multiply_3_1L(reg[7],reg[2],reg[4],MAN,EXP);
        CM_f_multiply_2_1L(reg[2],reg[3],MAN,EXP);
        CM_f_multiply_2_1L(reg[3],reg[1],MAN,EXP);
        CM_f_multiply_2_1L(reg[1],reg[4],MAN,EXP);
        CM_f_add_2_1L(reg[2],reg[1],MAN,EXP);
        CM_f_subtract_3_1L(reg[1],reg[3],reg[7],MAN,EXP);
      }
    }
  }
  cmtvp=CM_stop_timer(0);
  i=(1<<n);
  printf("================================================================
  printf("                CM_fft_d_1.c      Paris Version 5.0B\n");
  printf("\nTimes computed for a single %d-point %d-dimension complex FFT\n",i,q);
  printf("Times based on a loop factor of %d and using %d PEs\n",lfactor,i);
  printf("Twiddle factors are computed in-line.\n");
  printf("Floating point hardware is ACTIVE!\n");
  printf("----------------------------------------------------------\n");
  printf("Real time: %g sec; CM time: %g sec; CM utilization %g%%\n",
       cmtvp->cmtv_real/lfactor, cmtvp->cmtv_cm/lfactor,
       cmtvp->cmtv_real > 0.0 ?
       100.0 * cmtvp->cmtv_cm / cmtvp->cmtv_real:0.0);
  printf("Using a 5NlogN counter, %g MFLOPS\n",.000005*i*n/cmtvp->cmtv_cm*lfactor);
  printf("================================================================
  CM_reset_timer();
}
```

CM_fft_d_2.c

```
/*
*     CM_fft_d_2.c - R. Kamin - October 22, 1988
*     Paris Version 5.0B
*
*    This program performs a multidimensional DIF FFT on N
*    complex points using N/2 processors.  The data is loaded
*    in by the front end processor in normal order and read
*    back in bit-reversed order.  The twiddle factors
*    are computed in-line by the CM-2 processors.
*
*
*    Use <cc XXX.c -lparis -lm -o fft> to compile
*
*
*   Note: Must use 32-bit format to take advantage of FPU
*/

#include <math.h>
#include <stdio.h>
#include <cm/paris.h>

#define PI  3.14159265358979323846
#define MAX_N      32769
#define MAN          23    /* Length of mantissa; D=52 S=23    */
#define EXP           8    /* Length of exponent; D=11 S=8     */
#define INT_SIZE   MAN+EXP+1    /* Length of integer             */
#define MAX(A, B)  ((A) > (B) ? (A) : (B))


main ()
{
 int max_dim, N, reg[18], wdsize, i, n, base[15], size[15];

 CM_init();

 printf("Enter the number of dimensions ->");
 scanf("%d",&max_dim);

 base[0]=0;

 for (i=1; i<=max_dim; i++)
  {
    printf("Size of dimension %d (in log base 2) ->",i);
    scanf("%d",&size[i]);
    base[i]=base[i-1]+size[i];
  }

 n=base[max_dim];
 N=(1<<n);
```

```
    /*      Set up the static memory in each PE
     *
     * reg[0]    -  SR = Self-address Register
     * reg[1&2]  -  R1 = Complex Register #1
     * reg[3&4]  -  R2 = Complex Register #2
     * reg[5&6]  -  T1 = Temporary Complex Register #1
     * reg[7&8]  -  T2 = Temporary Complex Register #2
     * reg[9&10] -   W = Complex Twiddle Factor Register
     * reg[11&12] - DTR1 = Complex Data Transfer Register #1
     * reg[13&14] - DTR2 = Complex Data Transfer Register #2
     * reg[15]   -  DR = Destination address Register
     * reg[16]   - CFS = Context Flag storage register
     * reg[17]   - TMP = Temporary integer register
     */

  wdsize=MAN+EXP+1;        /* Floating point number length      */

  reg[0]=CM_allocate_stack_field(INT_SIZE);
  for (i=1; i<15; i++)
    reg[i]=CM_allocate_stack_field(wdsize);
  reg[15]=CM_allocate_stack_field(INT_SIZE);
  reg[16]=CM_allocate_stack_field(1);
  reg[17]=CM_allocate_stack_field(INT_SIZE);

  CM_set_context;              /* Activate all PEs   */
  CM_u_move_constant_1L(reg[16],0,1);
  printf("Setting addresses of %d processors . . .",MAX(N/2,8192));
  for (i=0; i<MAX(N/2,8192); i++)
    CM_u_write_to_processor_1L(i,reg[0],i,INT_SIZE);
  printf("done!\n");

  printf("Activating %d processors . . .",N/2);
  CM_u_lt_constant_1L(reg[0],N/2,INT_SIZE);    /* Select low N/2 PEs   */
  CM_load_context(CM_test_flag);
  CM_u_move_constant_1L(reg[16],1,1);
  printf("done!\n");

          /*  Load PEs with data in normal order  */

  printf("Loading data . . .");

  CM_f_move_constant_1L(reg[1],0.0,MAN,EXP);
  CM_f_move_constant_1L(reg[2],0.0,MAN,EXP);
  CM_f_move_constant_1L(reg[3],0.0,MAN,EXP);
  CM_f_move_constant_1L(reg[4],0.0,MAN,EXP);
  CM_f_move_constant_1L(reg[9],0.0,MAN,EXP);
  CM_f_move_constant_1L(reg[10],0.0,MAN,EXP);

  printf("done!\n");

  fft(n,reg,base,max_dim);       /* Execute the FFT       */

}
```

```
/******************** M-D DIF FFT Function ******************************/

fft(n,reg,base,dimension)
int n, reg[18], base[15], dimension;

{
 int lfactor, q, t, i, half_two_i, mask, current_dim, previous_dim;
 double arg;
 CM_timeval_t *cmtvp;

 printf("Enter loop factor ->");
 scanf("%d",&lfactor);
 printf("\n");

 q=dimension;
 CM_start_timer(1);

 for (t=0; t<lfactor; t++)
 {
 dimension=q;

 for (i=n-1; i>=0; i--)
  {
   if (i<base[dimension])          /* Compute which dimension     */
    dimension=dimension-1;
   current_dim=base[dimension];
   previous_dim=base[dimension+1];

   mask=(1<<(previous_dim-current_dim-1))-1;

   half_two_i=(1<<(i-1));

   CM_load_context(reg[16]);                    /* Enable N/2 PEs    */

   CM_f_add_3_1L(reg[5],reg[1],reg[3],MAN,EXP);     /* T1=R1+R2     */
   CM_f_add_3_1L(reg[6],reg[2],reg[4],MAN,EXP);
   CM_f_subtract_3_1L(reg[7],reg[1],reg[3],MAN,EXP); /* T2=R1-R2     */
   CM_f_subtract_3_1L(reg[8],reg[2],reg[4],MAN,EXP);

   if (i>0)
    {
     if (i==current_dim)
      {
       CM_f_move_1L(reg[11],reg[7],MAN,EXP);
       CM_f_move_1L(reg[12],reg[8],MAN,EXP);
      }
     else
      {
       arg= -2*PI/(1<<(previous_dim-current_dim));
                    /* Compute W     */
       CM_logand_constant_3_1L(reg[17],reg[0],mask,INT_SIZE);
       CM_u_add_constant_3_1L(reg[15],reg[17],1<<(i-current_dim),INT_SIZE);
       CM_logior_2_1L(reg[15],reg[17],INT_SIZE);
       CM_u_multiply_constant_2_1L(reg[15],(1<<(previous_dim-i-1)),INT_SIZE);
```

```
        CM_logand_constant_2_1L(reg[15],mask,INT_SIZE);
        CM_f_u_float_2_2L(reg[9],reg[15],INT_SIZE,MAN,EXP);
        CM_f_multiply_constant_2_1L(reg[9],arg,MAN,EXP);
        CM_f_sin_2_1L(reg[10],reg[9],MAN,EXP);
        CM_f_cos_1_1L(reg[9],MAN,EXP);
                /* DTR1=T2*W    */
        CM_f_multiply_3_1L(reg[11],reg[7],reg[9],MAN,EXP);
        CM_f_multiply_3_1L(reg[12],reg[8],reg[10],MAN,EXP);
        CM_f_subtract_2_1L(reg[11],reg[12],MAN,EXP);
        CM_f_multiply_3_1L(reg[12],reg[8],reg[9],MAN,EXP);
        CM_f_multiply_3_1L(reg[13],reg[7],reg[10],MAN,EXP);
        CM_f_add_2_1L(reg[12],reg[13],MAN,EXP);
       }
          /* Select PEs for BOTTOM butterfly    */
      CM_load_test(reg[0]+i-1);
      CM_load_context(CM_test_flag);

      CM_f_move_1L(reg[7],reg[11],MAN,EXP);          /* T2=DTR1    */
      CM_f_move_1L(reg[8],reg[12],MAN,EXP);
      CM_f_move_1L(reg[11],reg[5],MAN,EXP);          /* DTR1=T1    */
      CM_f_move_1L(reg[12],reg[6],MAN,EXP);

          /* Select all N/2 PEs          */
      CM_load_context(reg[16]);
      CM_logxor_constant_3_1L(reg[15],reg[0],half_two_i,INT_SIZE);
      CM_send_1L(reg[13],reg[15],reg[11],MAN+EXP+1);   /* DTR2 <- DTR1 */
      CM_send_1L(reg[14],reg[15],reg[12],MAN+EXP+1);

      CM_f_move_1L(reg[1],reg[5],MAN,EXP);          /* R1=T1    */
      CM_f_move_1L(reg[2],reg[6],MAN,EXP);

      CM_f_move_1L(reg[3],reg[13],MAN,EXP);          /* R2=DTR2    */
      CM_f_move_1L(reg[4],reg[14],MAN,EXP);

          /* Select PEs for BOTTOM butterfly    */
      CM_load_test(reg[0]+i-1);
      CM_load_context(CM_test_flag);

      CM_f_move_1L(reg[1],reg[13],MAN,EXP);          /* R1=DTR2    */
      CM_f_move_1L(reg[2],reg[14],MAN,EXP);
      CM_f_move_1L(reg[3],reg[7],MAN,EXP);           /* R2=T2    */
      CM_f_move_1L(reg[4],reg[8],MAN,EXP);
     }
    else
     {
      CM_f_move_1L(reg[1],reg[5],MAN,EXP);          /* R1=T1    */
      CM_f_move_1L(reg[2],reg[6],MAN,EXP);
      CM_f_move_1L(reg[3],reg[7],MAN,EXP);          /* R2=T2    */
      CM_f_move_1L(reg[4],reg[8],MAN,EXP);
     }
   }
}
cmtvp=CM_stop_timer(0);
i=(1<<n);
```

```
  printf("==========================================================\n");
  printf("                CM_fft_d_2.c      Paris Version 5.0B\n");
  printf("\nTimes computed for a single %d-point %d-dimension complex FFT\n",i,q);
  printf("Times based on a loop factor of %d and using %d PEs\n",lfactor,i/2);
  printf("Twiddle factors are computed in-line.\n");
  printf("Floating point hardware is ACTIVE!\n");
  printf("----------------------------------------------------------\n");
  printf("Real time: %g sec; CM time: %g sec; CM utilization %g%%\n",
        cmtvp->cmtv_real/lfactor, cmtvp->cmtv_cm/lfactor,
        cmtvp->cmtv_real > 0.0 ?
        100.0 * cmtvp->cmtv_cm / cmtvp->cmtv_real:0.0);
  printf("Using a 5NlogN counter, %g MFLOPS\n",.000005*i*n/cmtvp->cmtv_cm*lfactor);
  printf("==========================================================\n");
  CM_reset_timer();
}
```