High Radix FFT on Boolean Cube Networks

S. Lennart Johnsson, Michel Jacquemin and Ching-Tien Ho

YALEU/DCS/TR-751
November 1989

This work was supported in part by the Air Force Office of Scientific Research, contract number AFOSR-89-0382, and Thinking Machines Corporation.

High Radix FFT on Boolean Cube Networks

S. Lennart Johnsson[1], Michel Jacquemin[2] and Ching-Tien Ho[3]

Thinking Machines Corp.
245 First Street,
Cambridge, MA 02142

Abstract

A high radix FFT requires fewer arithmetic operations than a radix-2 FFT and has a reduced need for memory bandwidth in systems with a storage hierarchy. We show two ways in which high radix FFT can be used on Boolean cube networks. With several elements per node in a Boolean n-cube, multi-sectioning of the data set can be used for full utilization of the communications bandwidth. With 2^r-way sectioning the number of complex element transfers in sequence is $\frac{P}{2N} + (\frac{n}{r} - 1)2^{r-1}$ for an FFT on P complex points. Multi-sectioning with a high radix FFT performed locally in each processor can benefit fully from a local storage hierarchy with respect to storage bandwidth requirements. Pipelining successive stages of a high radix FFT for the stages requiring inter-processor communication yields a communication complexity approximately twice that of multi-sectioning. For the interprocessor communication stages a pipelined FFT cannot fully utilize a local storage hierarchy to reduce the need for storage bandwidth.

The storage required for twiddle factors is minimized for the combinations of normal order input, consecutive storage, and decimation-in-time in-place FFT, or cyclic storage and decimation-in-frequency FFT. For bit-reversed order input and consecutive storage decimation-in-frequency FFT, or cyclic storage and decimation-in-time FFT minimizes the storage needs. For these combinations the maximum storage required per processor is $\frac{P}{N} + \frac{n}{r} - 2$ with no communication or computation of twiddle factors. For a radix of 2 and 4 a reduction is possible by computing 90-degree rotations on-the-fly, or by using symmetries.

For local FFT of size 64 or greater the local radix-4/radix-8 FFT achieves a performance in the range of 7 – 9 Gflops/s on a 64k processor CM – 2. The performance of these kernels is 2.5 – 3.5 times higher than that of the radix-2, local FFT.

1 Introduction

This paper describes the data mapping and control structures for high radix and multi-sectioned Fast Fourier Transforms (FFT) on Boolean cube networks. We first review the

[1]Also affiliated with the Department of Computer Science, Yale University.
[2]Present address Department of Computer Science, Yale University.
[3]Present address, IBM Almaden Research Center.

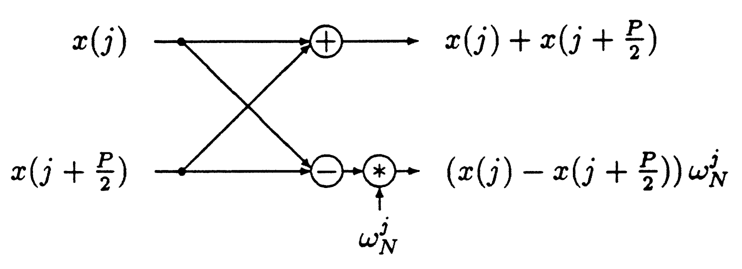

Figure 1: A butterfly.

Cooley-Tukey FFT, then discuss the mapping of such FFT computations to Boolean cube networks with particular emphasis on utilization of the communications bandwidth, storage bandwidth, and coefficient storage [12]. The parallel computation of twiddle factors is also considered. The implementation of local radix-4 and radix-8 FFT on the Connection Machine are described last.

The Discrete Fourier Transform (DFT) is defined by

$$X(l) = \sum_{j=0}^{P-1} \omega_P^{lj} x(j), \quad \forall l \in [0, P-1], \quad \omega_P = e^{-\frac{2\pi i}{P}}.$$

The Cooley-Tukey Fast Fourier Transform [1] is obtained by a factoring of $P = P_0 P_1 \ldots P_{p-1}$ and by using the following properties of the "twiddle factors": $\omega_P^{\frac{P}{2}} = -1$, $\omega_P^{k\frac{P}{4}} = (-i)^k$, and $\omega_P^{k\frac{P}{8}} = (\frac{(1-i)}{\sqrt{2}})^k$. In a radix-R FFT, $P_m = R, \forall m \in [0, u-1]$, where $u = \log_R P$. For the special case of $R = 2$, $P = 2^p$ and $u = p$. We first show the derivation of a radix-2 FFT, then derive a radix-4 FFT.

The sum defining the Discrete Fourier Transform can be decomposed by distinguishing between even and odd indices of X:

$$\begin{aligned} X(2l') &= \sum_{j=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{jl'} (x(j) + x(j + \frac{P}{2})) \\ X(2l'+1) &= \sum_{j=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{jl'} (\omega_P^j (x(j) - x(j + \frac{P}{2}))) \end{aligned}$$

The two expressions consist in discrete Fourier transforms *of size* $\frac{P}{2}$ of two different combinations of pairs of input data, as shown in Figure 1. This combination of inputs defines a *butterfly*. It is the basic computation in the FFT algorithm.

Figure 2 shows the decomposition into even and odd components through the butterfly computations. The "•'s" represent complex multiplies by twiddle factors.

By applying the decomposition recursively the Discrete Fourier Transform of size $\frac{P}{2}$ is decomposed into Fourier Transforms of size $\frac{P}{4}$, etc, until the Fourier Transforms are of size

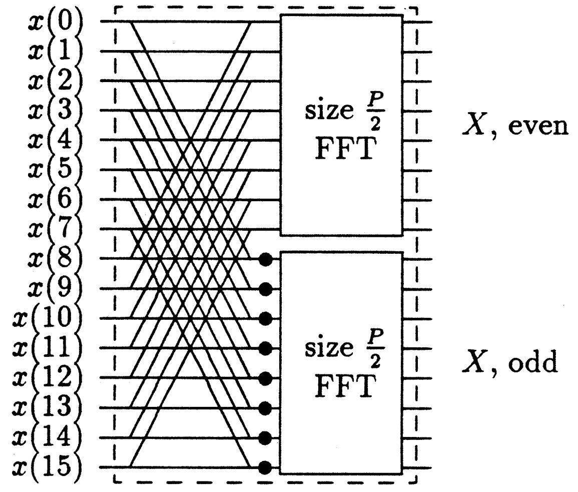

Figure 2: Decomposition of the discrete Fourier Transform.

2 for $P = 2^p$. A Fourier Transform of size 2 is exactly a butterfly computation with the twiddle factor equal to 1. A complete FFT of size 16 is shown in Figure 3.

Note that the results are in bit-reversed order. The FFT algorithm derived above is called the *decimation-in-frequency* (*DIF*) FFT. Another derivation produces the *decimation-in-time* (*DIT*) FFT algorithm. The sum

$$X(l) = \sum_{j=0}^{N-1} \omega_P^{jl} x(j)$$

is then decomposed into:

$$\begin{aligned} X(l) &= \sum_{j'=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{j'l} x(2j') + \omega_N^l \sum_{j'=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{j'l} x(2j'+1) \\ X(l+\frac{P}{2}) &= \sum_{j'=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{j'l} x(2j') - \omega_N^l \sum_{j'=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{j'l} x(2j'+1) \end{aligned}$$

In this decomposition the complex multiplication is performed before the addition and subtraction of arguments, and the twiddle factors are different from those in the decimation-in-frequency butterfly. The decimation-in-time butterfly is shown in Figure 4. A complete decimation-in-time FFT is illustrated in Figure 5 for a Fourier Transform of size 16.

The order in which the decimation-in-frequency FFT uses the twiddle factors is the opposite from the order the decimation-in-time FFT uses them. If the stages of the FFT are numbered from 0 to $\log_2 N - 1$, then

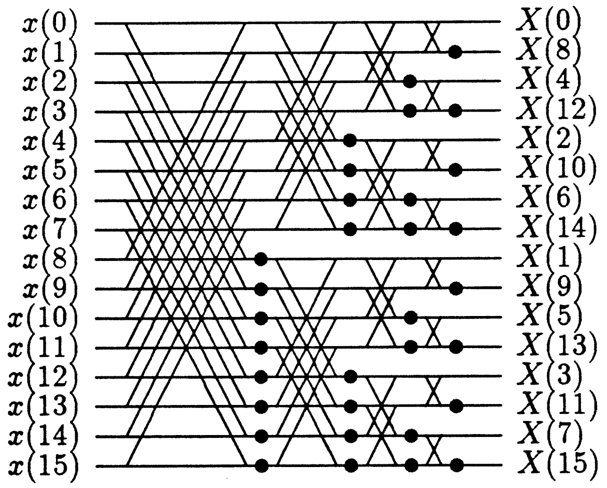

Figure 3: Decimation-in-frequency FFT.

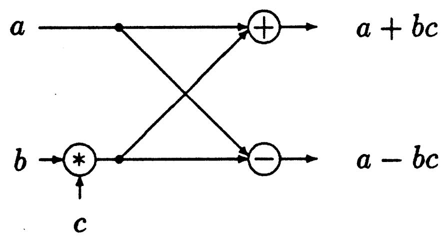

Figure 4: Decimation-in-time butterfly.

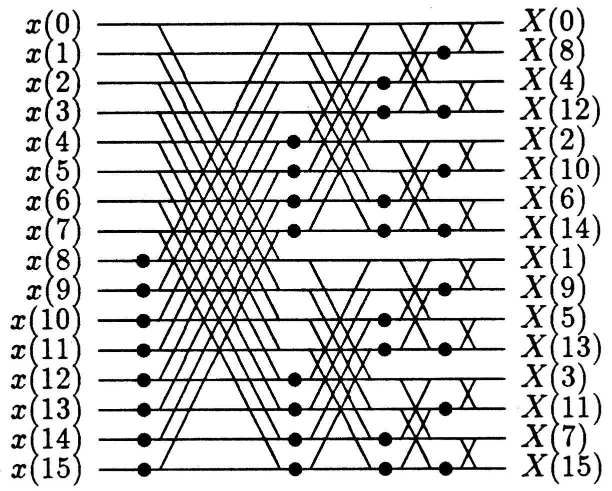

Figure 5: Decimation-in-time FFT.

- in stage q of the DIF FFT, twiddle factors $\omega^k_{\frac{P}{2^q}}, k = \{0, \ldots, \frac{P}{2^{q+1}} - 1\}$ are used, each twiddle factor being used 2^q times.
- in stage q of the DIT FFT, twiddle factors $\omega^k_{2^{q+1}}, k = \{0, \ldots, 2^q - 1\}$ are used, each twiddle factor being used $\frac{P}{2^{q+1}}$ times.

The twiddle factors used in the last stage of the DIF FFT and the first stage of the DIT FFT are all equal to $\omega^0_2 = 1$.

A radix-R FFT algorithm is a simple generalization of the radix-2 FFT algorithm. Instead of decomposing the computation of the Discrete Fourier Transform of size P into 2 subproblems of size $\frac{P}{2}$, the computation is decomposed into R subproblems of size $\frac{P}{R}$. The integer R is the radix. For Cooley-Tukey type FFT it is usually a small power of 2, such as 4 or 8. For the derivation of a radix-4 DIF FFT $j = j' + k\frac{P}{4}$, $j' \in [0, \frac{P}{4} - 1]$, $k \in [0, 3]$.

$$\begin{aligned} X(l) &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_P^{lj'} x(j') + \sum_{j'=0}^{\frac{P}{4}-1} \omega_P^{l(j'+\frac{P}{4})} x(j' + \frac{P}{4}), + \sum_{j'=0}^{\frac{P}{4}-1} \omega_P^{l(j'+2\frac{P}{4})} x(j' + 2\frac{P}{4}) \\ &+ \sum_{j'=0}^{\frac{P}{4}-1} \omega_P^{l(j'+3\frac{P}{4})} x(j' + 3\frac{P}{4}) \qquad \forall l \in [0, P-1] \end{aligned}$$

Rewriting this expression with $l = r + 4l', r \in [0,3], l' \in [0, \frac{P}{4} - 1]$ yields

$$\begin{aligned}
X(4l') &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'}(x(j') + x(j'+\frac{P}{4}) + x(j'+2\frac{P}{4}) + x(j'+3\frac{P}{4})) \\
X(4l'+1) &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'}\omega_P^{j'}(x(j') - ix(j'+\frac{P}{4}) - x(j'+2\frac{P}{4}) + ix(j'+3\frac{P}{4})) \\
X(4l'+2) &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'}\omega_P^{2j'}(x(j') - x(j'+\frac{P}{4}) + x(j'+2\frac{P}{4}) - x(j'+3\frac{P}{4})) \\
X(4l'+3) &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'}\omega_P^{3j'}(x(j') + ix(j'+\frac{P}{4}) - x(j'+2\frac{P}{4}) - ix(j'+3\frac{P}{4})).
\end{aligned}$$

These expressions can be reorganized as follows [15, 17].

$$\begin{aligned}
X(4l') &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'}([x(j') + x(j'+2\frac{P}{4})] + [x(j'+\frac{P}{4}) + x(j'+3\frac{P}{4})]) \\
X(4l'+1) &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'}\omega_P^{j'}([x(j') - x(j'+2\frac{P}{4})] - i[x(j'+\frac{P}{4}) - x(j'+3\frac{P}{4})]) \\
X(4l'+2) &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'}\omega_P^{2j'}([x(j') + x(j'+2\frac{P}{4})] - [x(j'+\frac{P}{4}) + x(j'+3\frac{P}{4})]) \\
X(4l'+3) &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'}\omega_P^{3j'}([x(j') - x(j'+2\frac{P}{4})] + i[x(j'+\frac{P}{4}) - x(j'+3\frac{P}{4})]).
\end{aligned}$$

The latter set of equations shows that a radix-4 stage can be organized as two radix-2 stages, as shown in Figure 6. The derived algorithm is a radix-4 *decimation-in-frequency* (DIF) FFT.

A *decimation-in-time* (DIT) FFT is derived by expressing the data index j as $j = 4j' + k$, $j' \in [0, \frac{P}{4} - 1]$, $k \in [0, 3]$. Then,

$$\begin{aligned}
X(l) = & \sum_{j'=0}^{\frac{P}{4}-1} \omega_P^{l(4j')}x(4j') + \sum_{j'=0}^{\frac{P}{4}-1} \omega_P^{l(4j'+1)}x(4j'+1), + \sum_{j'=0}^{\frac{P}{4}-1} \omega_P^{l(4j'+2)}x(4j'+2) \\
& + \sum_{j'=0}^{\frac{P}{4}-1} \omega_P^{l(4j'+3)}x(4j'+3) \qquad \forall l \in [0, P-1].
\end{aligned}$$

Rewriting this expression with $l = r\frac{P}{4} + l', r \in [0,3], l' \in [0, \frac{P}{4} - 1]$ yields

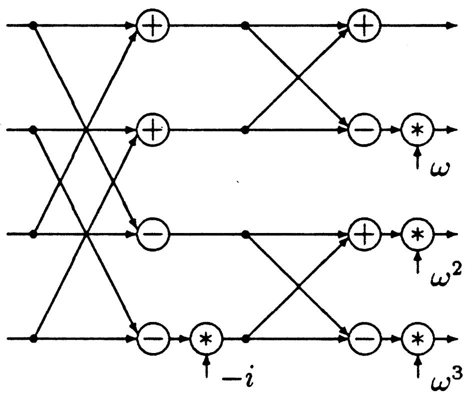

Figure 6: Factoring of a radix-4 DIF butterfly.

$$
\begin{aligned}
X(l') &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j') + \omega_P^{l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+1) \\
&\quad + \omega_P^{2l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+2) + \omega_P^{3l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+3) \\
X(\frac{P}{4}+l') &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j') - i\omega_P^{l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+1) \\
&\quad - \omega_P^{2l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+2) + i\omega_P^{3l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+3) \\
X(2\frac{P}{4}+l') &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j') - \omega_P^{l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+1) \\
&\quad + \omega_P^{2l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+2) - \omega_P^{3l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+3) \\
X(3\frac{P}{4}+l') &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} (x(4j') + i\omega_P^{l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+1) \\
&\quad - \omega_P^{2l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+2) - i\omega_P^{3l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+3)
\end{aligned}
$$

As in the case of decimation-in-frequency FFT this expression can be rewritten such that each radix-4 stage is implemented as two radix-2 stages.

$$\begin{aligned}
X(l') &= \left[\sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j') + \omega_P^{2l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+2)\right] \\
&+ \left[\omega_P^{l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+1) + \omega_P^{3l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+3)\right] \\
X(\frac{P}{4}+l') &= \left[\sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j') - \omega_P^{2l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+2)\right] \\
&- i\left[\omega_P^{l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+1) - \omega_P^{3l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+3)\right] \\
X(2\frac{P}{4}+l') &= \left[\sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j') + \omega_P^{2l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+2)\right] \\
&- \left[\omega_P^{l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+1) + \omega_P^{3l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+3)\right] \\
X(3\frac{P}{4}+l') &= \left[\sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j') - \omega_P^{2l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+2)\right] \\
&+ i\left[\omega_P^{l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+1) - \omega_P^{3l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+3)\right].
\end{aligned}$$

Figure 6 shows the factored form of a radix-4 decimation-in-time butterfly computation. The multiplication by i requires no arithmetic operations, only an interchange of real and imaginary parts with the appropriate sign change. From Figures 6 and 7 it is clear that complex multiplications are only required for every other radix-2 butterfly stage in a radix-4 FFT.

For a radix-8 DIF FFT the derivation is made by the factoring $j = j' + k\frac{P}{8}$, $j' \in [0, \frac{P}{8}-1]$, $k \in [0, 7]$, and $l = r + 8l'$, $r \in [0, 7]$, $l' \in [0, \frac{P}{8}-1]$. A radix-8 DIT FFT can be derived similarly. In a radix-8 FFT, complex multiplications are only required for every three radix-2 butterfly stage, as seen in Figures 1 and 1.

Different radix FFT are merely reorganizations of the arithmetic operations. The required number of complex multiplications are reduced by combining the operations from several stages. The number of complex multiplications is $(p-1)\frac{P}{2}$ for a radix-2 algorithm, and $(\frac{p}{2}-1)\frac{3P}{4}$ for a radix-4 algorithm. A radix-8 algorithm requires internal complex multiplications corresponding to 45-degree rotations. The number of real operations and memory accesses for radix-2, -4, and -8 butterflies are given in Table 1. The total number of operations

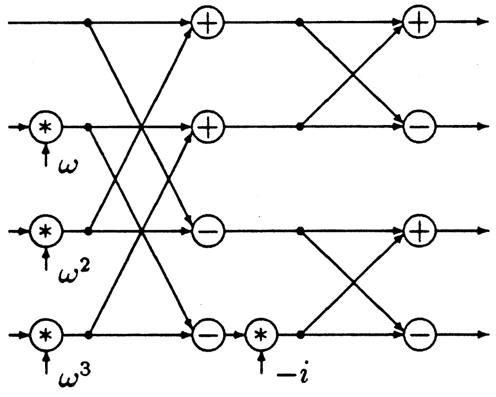

Figure 7: Factoring of a radix-4 DIT butterfly.

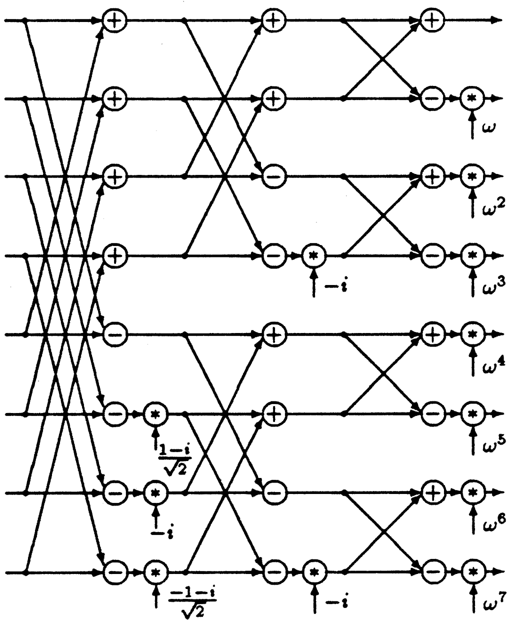

Figure 8: A Decimation in Frequency Radix 8 Kernel

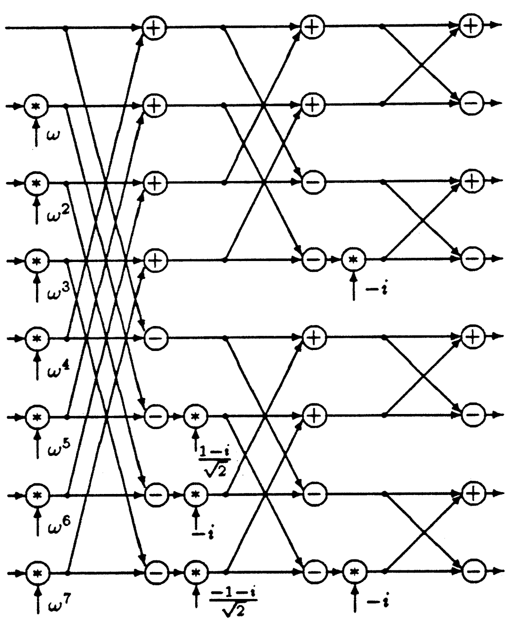

Figure 9: A Decimation in Time Radix 8 Kernel

(considering higher order terms only) are summarized in Table 2. The ratios of the number of real arithmetic operations normalized to the radix-8 FFT are $\frac{60}{49}$: $\frac{51}{49}$: 1. The total number of arithmetic operations for the radix-8 algorithm is approximately 20% less than that of the radix-2 algorithm. The exact number of multiplications and additions can be found, for instance, in [17].

In most architectures the effective use of the memory bandwidth is more critical, with respect to performance, than minimizing the number of arithmetic operations. With a local memory of size M, a radix M FFT offers a reduction in memory bandwidth requirement by a factor of $\log_M$, which is optimum [3]. The ratios for memory operations are $\frac{60}{23}$: $\frac{33}{23}$: 1. However, it should be noticed that if $R = 2^r$ registers (complex) are used for twiddle factors, then r ranks can be computed with a single load of $R-1$ twiddle factors by successively computing all butterflies in a given rank requiring those coefficients. The adjacent set of r radix-2 butterfly ranks require R loads of $R-1$ twiddle factors, etc. The total number of storage references for twiddle factors is $P-1$. If the ordering of the computations is such that $P-1$ loadings of twiddle factors suffice, then the storage references for data dominates. The ratios of the number of memory references become 3 : $\frac{3}{2}$: 1 for radix-2, -4, and -8 FFT. The number of registers needed for twiddle factors are 2, 6, and 14, respectively.

FFT	Arithmetic Operations			Storage References		
	Add/Sub	Mult	Total	Data	Twiddles	Total
Radix-2	6	4	10	8	2	10
Radix-4	22	12	34	16	6	22
Radix-8	66	32	98	32	14	46

Table 1: Arithmetic and memory operations for radix-2, -4, and -8 butterflies.

FFT	Arithmetic Operations			Storage References		
	Add	Mult	Total	Data	Twiddles	Total
Radix-2	$3Pp$	$2Pp$	$5Pp$	$4Pp$	Pp	$5Pp$
Radix-4	$\frac{22}{8}Pp$	$\frac{12}{8}Pp$	$\frac{17}{4}Pp$	$\frac{16}{8}$	$\frac{6}{8}Pp$	$\frac{11}{4}$
Radix-8	$\frac{66}{24}Pp$	$\frac{32}{24}Pp$	$\frac{49}{12}Pp$	$\frac{32}{24}Pp$	$\frac{14}{24}Pp$	$\frac{23}{12}Pp$

Table 2: Arithmetic and memory operations for radix-2, -4, and -8 FFTs.

2 Data Allocation

Data motion often has a significant impact on performance in distributed memory architectures. With appropriate data allocation the need for data motion can be minimized. A good (optimum) choice of data allocation requires knowledge about the data interaction in the algorithm, the topology, channel widths, and channel rates of the network. In a Boolean cube of $N = 2^n$ nodes every node $u = (u_{n-1}u_{n-2} \ldots u_m \ldots u_0)$ is connected to nodes $v = (u_{n-1}u_{n-2} \ldots \overline{u}_m \ldots u_0), \forall m \in [0, n-1]$. Every node has n neighbors. The distance between a pair of nodes u and v is $Hamming(u, v) = \sum_{m=0}^{n-1}(u_m \oplus v_m)$. The maximum distance between any pair of nodes is n.

Letting the n highest order bits encode processor addresses, and the lower order bits encode memory addresses in each processor yields a *consecutive* assignment [5]:

Consecutive assignment:

$$(\underbrace{x_m x_{m-1} \ldots x_{m-n+1}}_{rp}\underbrace{x_{m-n}x_{m-n-1} \ldots x_0}_{vp}).$$

The field denoted *rp* encodes *real processor* addresses as opposed to *memory* addresses. For a data set of m complex points $m+1$ address bits are required, n of which are processor address bits. There are $m-n+1$ local storage address bits. In *cyclic* assignment the lowest order address bits determine the *real processor* address.

Cyclic assignment:

$$(\underbrace{x_m x_{m-1} \ldots x_n}_{vp} \underbrace{x_{n-1} x_{n-2} \ldots x_0}_{rp}).$$

All data elements with the same n low order bits reside in the same processor. In the consecutive assignment the elements in a processor have the same n high order bits. The cyclic assignment results in better load balance than consecutive allocation for certain computations, but results in higher communication requirements for some [7]. We consider both forms of data allocation for the FFT.

For multi-dimensional arrays each axis is often encoded separately, as for instance is the case in the Connection Machine programming systems [20]. If each axis is encoded in a unique set of address bits, then effectively each axis is extended to a length that is equal to some power of two. The encoding of an axis length P requires $\lceil \log_2 P \rceil$ address bits. For the FFT computations we consider here $P = R^p$, where $R = 2^r$. In this case the address space is used with 100% efficiency.

3 Cooley-Tukey FFTs on Boolean Cubes

3.1 Mapping Butterfly Networks to Boolean Cubes

A radix-2 butterfly network for P inputs and outputs has $P(p+1)$ nodes. These can be uniquely encoded with a total of $p + \lceil \log_2(p+1) \rceil$ bits. Let the address be partitioned as follows $(y_{p-1}y_{p-2} \ldots y_0 | z_{t-1}z_{t-2} \ldots z_0)$, where $t = \lceil \log_2(p+1) \rceil$. Then, the butterfly network is defined by connecting node $(y|z)$ to the nodes $(y \oplus 2^{p-1-z} | z+1)$ and $(y|z+1)$, $z \in [0, p-1]$, where $\oplus$ denotes the bit-wise exclusive-or operation. For the computation of the radix-2 FFT the last t bits can also be interpreted as time. The network utilization defined as the fraction of the total number of nodes that are active at any given time is $\frac{1}{t}$. By identifying all nodes with the same y value and different z values in the butterfly network node y becomes connected to nodes $y \oplus 2^z$, $\forall z \in [0, p-1]$, which defines a Boolean p-cube. All nodes participate in every step in computing an FFT on P elements on a p-cube. In step z all processors communicate in dimension z. Only $\frac{1}{p}th$ of the total communications bandwidth of the p-cube is used.

In computing an FFT on $P = 2^p$ complex elements on $N = N < P$ processors there are $\frac{P}{N}$ elements per *real processor*. If the cyclic assignment is used, then regardless of whether a decimation-in-frequency or decimation-in-time FFT is used, the first $p - n$ ranks of butterfly computations are local to a processor. The last n ranks require inter-processor communication. For consecutive assignment the first n steps require inter-processor communication, and the last $p - n$ steps are local to a processor. If the data is allocated in a bit-reversed order, then the order of the inter-processor communication and the local reference phases are reversed. With a cyclic assignment the first $p - n$ local stages decomposes the FFT computation on P elements into $\frac{P}{N}$ independent FFT, each with one element per processor.

Similarly, with consecutive allocation the lower order $p-n$ bits define $\frac{P}{N}$ independent FFT, each with one element per processor.

With multiple elements per processor, the communication efficiency can be improved from $\frac{1}{\min(p,n)}$ to $\frac{\min(p,n)}{n}$, which for $p > n$ is one. Multi-sectioning and pipelining independent FFT computations [12, 13] are techniques that can be used for increased communication efficiency through concurrent communication in as many dimensions as possible. Both techniques also improve the load balance. In a pipelined radix-2 algorithm for an FFT with one element per processor the complex multiplication is performed by one of the processors in a pair, if a single exchange of data is performed. Splitting the complex multiplication between a pair of processors such that each processor performs 5 real operations per butterfly computation requires one more communication. The multi-sectioning technique achieves 100% arithmetic load balance. In a radix-R FFT performed across the Boolean n-cube $\frac{R-1}{R}$ processors perform complex multiplications concurrently.

The embedding defined above is the binary encoding of array indices. Every index is directly identified by an address in the address space. For arrays embedded by a binary-reflected Gray code [19, 14] array elements that differ by a power of two greater than zero are at a distance of two, i.e., $G(i) \oplus G(i+2^j) = 2, j \neq 0$ [4]. Even though the elements to be used in a butterfly computation are at a Hamming distance of two it is still possible to perform an FFT with $\min(p,n)$ nearest neighbor communications [10].

3.2 Bi-section

By recursively partitioning the set of $\frac{P}{N}$ FFTs during the n inter-processor communication stages, such that one half of the FFTs are computed in a half-sized Boolean cube, and the other half of the FFTs in the other half sized cube perfect arithmetic load balance is achieved, and the communication time reduced by a factor of up to two [12]. The recursive partitioning doubles the number of elements per processor of a given FFT in every step. The number of FFTs serviced by a processor is reduced by a factor of two for every recursion step. The number of complex element transfers in sequence for this pipelined, recursive partitioning FFT is $\frac{P}{2N} + n - 1$, which is approximately half of the number of element transfers of the straightforward pipelined algorithm. The recursive partitioning technique was used in [6, 8] for *Balanced Cyclic Reduction* on Boolean cube networks.

The recursive partitioning strategy for computing a radix-2 FFT with two complex data elements per real processor, $p - n = 1$, and *cyclic* allocation of elements to processors is illustrated in Table 3, and below. The numbers in Table 3 denote the initial data indices. The indices of the computed frequency components are obtained in bit-reversed order with respect to the input ordering. The bit-reversal operation applies to the entire data set.

The n steps with inter-processor communication can be illustrated in terms of the address space as follows, where the most significant bit is the leftmost bit.

Proc. id	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
initial	0	1	2	3	4	5	6	7
alloc.	8	9	10	11	12	13	14	15
after	0	1	2	3	8	9	10	11
1st exch.	4	5	6	7	12	13	14	15
after	0	1	4	5	8	9	12	13
2nd exch.	2	3	6	7	10	11	14	15
after	0	2	4	6	8	10	12	14
3rd exch.	1	3	5	7	9	11	13	15

Table 3: The data distribution for the recursive partitioning, radix-2, FFT for two *virtual processors.*

Initial allocation: $(\underbrace{x_n}_{vp}\underbrace{x_{n-1}x_{n-2}\cdots x_0}_{rp})$ $\quad (p-1=n)$.

Step 1: $(\underbrace{\overline{x_{n-1}}}_{vp}\underbrace{\overline{x_n}x_{n-2}\cdots x_0}_{rp})$.

Step 2: $(\underbrace{\overline{x_{n-2}}}_{vp}\underbrace{x_n\overline{x_{n-1}}x_{n-3}\cdots x_0}_{rp})$

$\vdots \qquad \vdots$

Step n: $(\underbrace{\overline{x_0}}_{vp}\underbrace{x_{n-1}\cdots\overline{x_1}}_{rp})$.

The dimension representing the virtual processors is successively moved to the lowest order bit position. The bits of the address space on which the exchange in each step takes place are marked by a bar. Since one of the dimensions is a local memory address the exchange is always an exchange between adjacent processors. The exchange sequence in the illustration converts the cyclic allocation to consecutive allocation. It is also an *unshuffle.* For $p-n>1$ performing the recursive subdivision on the highest order local address bit yields a final data ordering in which the processor address field is defined by $(a_{p-1}a_{n-1}\ldots a_1)$, and the local memory order is $(a_0a_{p-2}a_{p-3}\ldots a_n)$. By using successively lower order address bits the processor addresses for $p-n\geq n$ become $(a_{p-1}a_{p-2}\ldots a_{p-n})$, and the local memory addresses are $(a_{n-1}a_{n-2}\ldots a_0a_{p-n-1}\ldots a_n)$. This ordering is of the consecutive type with respect to processor addresses, but requires a local unshuffle of order $p-2n-1$, or shuffle of order n to establish a consecutive data allocation. In any step only one dimension is used, and successive steps can be pipelined.

Initial allocation: $(\underbrace{x_{p-1}x_{p-2}\dots x_n}_{vp}\underbrace{x_{n-1}\dots x_0}_{rp})$.

1st exch. $(\underbrace{x_{n-1}x_{p-2}x_{p-3}\dots x_n}_{vp}\underbrace{x_{p-1}x_{n-2}\dots x_0}_{rp})$.

2nd exch. $(\underbrace{x_{n-1}x_{n-2}x_{p-3}x_{p-4}\dots x_n}_{vp}\underbrace{x_{p-1}x_{p-2}x_{n-3}\dots x_0}_{rp})$.

$\vdots$ $\vdots$

nth exch. $(\underbrace{x_{n-1}x_{n-2}\dots x_0x_{p-n-1}\dots x_n}_{vp}\underbrace{x_{p-1}x_{p-2}\dots x_{p-n}}_{rp})$.

If the initial data ordering is consecutive, then the recursive partitioning of the data set moves lower order, memory address bits into the real processor address field. If the same memory address bit is used for the splitting in each step, say the lowest order bit, then the processor address field after the first n steps is defined by $(a_0a_{p-1}a_{p-2}\dots a_{p-n+1})$, and the local memory address field is defined by $(a_{p-n-1}a_{p-n-2}\dots a_1a_{p-n})$. The remaining butterfly computations are local, except for the last on bit a_0, which requires interprocessor communication. If again the lowest order address bit is used for the splitting, then the final processor address field is given by $(a_{p-n}a_{p-1}a_{p-2}\dots a_{p-n+1})$, and the local memory address field is $(a_{p-n-1}a_{p-n-2}\dots a_1a_0)$.

3.3 Multi-section

The recursive partitioning idea can be generalized to multi-way partitioning. A $R = 2^r$ way partitioning implies *all-to-all personalized communication* [11] in r-dimensional cubes. After each such partitioning step a radix-2^r FFT can be performed locally. Table 4 illustrates the idea for the inter-processor communication steps for $p - n = 2$ and $n = 4$. The numbers in the table are the initial indices. The first partitioning step is an *all-to-all personalized communication* [11] within each subcube of dimension 2 with respect to the 2 highest order real processor dimensions. For instance, processors 0, 4, 8 and 12 are in the same subcube. For the radix-4 algorithm, two bits are involved in every step. For a 2^r-way partitioning algorithm, r bits are involved. Successive steps involve consecutive blocks of r dimensions, and the steps can be pipelined. The number of element transfers in sequence is $\frac{P}{2N} + (\lceil\frac{n}{r}\rceil - 1)\frac{R}{2}$. An N-way partitioning minimizes the number of element transfers in sequence. Next to N-way partitioning a 4-way partitioning algorithm is the best choice with respect to element transfers. A radix-2 algorithm is insignificantly inferior. The multi-way recursive partitioning is illustrated below. As in the bi-section case there exist many ways in which partitioning of the local data can be performed throughout the algorithm resulting in different final orderings.

Proc. id.	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Initially	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
	0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51
1st	4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55
part.	8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59
	12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63
	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
2nd	1	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61
part.	2	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62
	3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63

Table 4: The data distribution for the recursive partitioning, radix-4 FFT for 16 processors and 4 elements per processor.

Initial allocation: $(\underbrace{x_{p-1}x_{p-2}\dots x_n}_{vp}\underbrace{x_{n-1}\dots x_0}_{rp})$.

1st part. $(\underbrace{x_{n-1}x_{n-2}\dots x_{n-r}x_{p-r-1}x_{p-r-2}\dots x_n}_{vp}\underbrace{x_{p-1}x_{p-2}\dots x_{p-r}x_{n-r-1}\dots x_0}_{rp})$.

2nd part. $(\underbrace{x_{n-1}x_{n-2}\dots x_{n-2r}x_{p-2r-1}x_{p-2r-2}\dots x_n}_{vp}\underbrace{x_{p-1}x_{p-2}\dots x_{p-2r}x_{n-2r-1}\dots x_0}_{rp})$.

$\vdots$ $\vdots$

Step $\frac{n}{r}$: $(\underbrace{x_{n-1}x_{n-2}\dots x_0x_{p-n-1}\dots x_n}_{vp}\underbrace{x_{p-1}x_{p-2}\dots x_{p-n}}_{rp})$.

3.4 Communication for high radix FFT.

Performing a radix-2^r FFT involves r inter-processor dimensions. In unfactored form the high radix butterfly computation requires *all-to-all broadcasting* [11] in r-dimensional cubes, followed by a reduction during the butterfly computation. The all-to-all broadcasting requires a time of $\frac{R-1}{r}$ for one radix-R butterfly, and a total of $(\frac{P}{N}+\frac{n}{r}-1)\frac{R-1}{r}$ for the complete FFT. This communication time is clearly inferior to the multi-sectioning technique. However, with the factored form of a high radix FFT, a butterfly computation and reduction is associated with each dimension. Pipelining can be applied as in the radix-2 algorithm. The pipelining of radix-2 stages is illustrated in Figure 10. The number of element transfers in sequence for a pipelined high radix FFT is the same as for a radix-2, pipelined FFT.

Multiplication with twiddle factors is only associated with every r inter-processor communications in a radix-2^r FFT, ignoring internal multiplications (of which there are none for

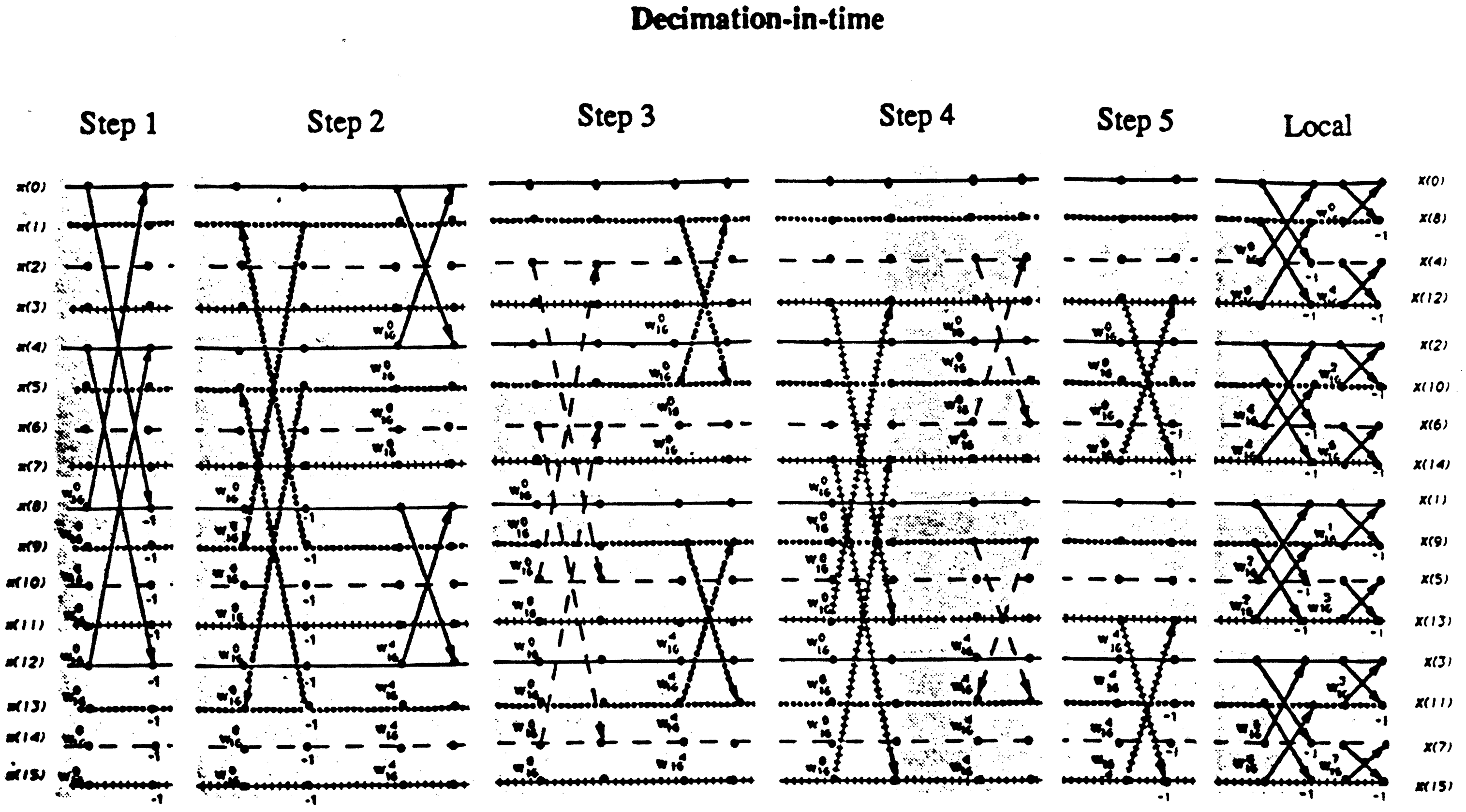

Figure 10: Pipelining of radix-2 butterfly stages.

radix-4 FFT). A radix-4 algorithm only requires half as many complex multiplication steps as a radix-2 algorithm. In the inter-processor communication phase $\frac{3}{4}$ of all processors perform a multiplication, whereas only half of the processors perform a complex multiplication in the pipelined radix-2 case. For the factored radix-8 FFT $\frac{7}{8}$ processors perform a complex multiplication every third step. In addition, in one out of every three steps $\frac{3}{8}$ processors need to perform a rotation by 45-degrees, or a multiple thereof. Higher radix FFT improves the arithmetic load balance, and arithmetic efficiency.

3.5 Twiddle Factors

The total number of twiddle factors needed for a radix-R FFT of size P is $(R-1)\frac{P}{R}$. For the computation of an FFT on a distributed memory machine, it is important to minimize the need for either redundant storage of twiddle factors, or communication of twiddle factors should they be required in a processor different from the one in which they are stored. The order in which the set of twiddle factors are used is different for decimation-in-time and decimation-in-frequency FFT.

3.5.1 Decimation-in-frequency

A radix-2 FFT performed by a decimation-in-frequency algorithm on data in normal order allocated cyclically requires $\frac{P}{N} + n - 2$ twiddle factors per processor [13]. Of these twiddle factors $\frac{P}{N} - 1$ are required for the local FFT. By computing 90-degree rotations on-the-fly the number of twiddle factors for local computations can be reduced by a factor of two. Consecutive data allocation, input data in normal order, and decimation-in-time FFT has the same property with respect to twiddle factor storage. Consecutive data allocation and decimation-in-frequency FFT, and cyclic data allocation and decimation-in-time FFT requires $(n-1)\frac{P}{N}$ twiddle factors per processor. Here, we only consider cyclic data allocation and decimation-in-frequency FFT, and consecutive data allocation and decimation-in-time FFT.

For a radix-2 DIF FFT the set of twiddle factor exponents required after the first rank is defined by $(a_{p-1}) \times (a_{p-2}a_{p-2} \ldots a_0)$, where $(a_{p-1}a_{p-2} \ldots a_0)$ is the address of a data location. Data is assumed to be in normal order, and the algorithm an *in-place* FFT [18], as depicted previously. The twiddle factors associated with an address can be derived from the recursive decomposition, and may be intuitively justified by Figure 3. The twiddle factors may also be directly derived from the following iterative formulation of the decimation-in-frequency FFT. The radix-2, in-place, DIF FFT can be formulated as

$$\begin{aligned}
\tilde{x}_{-1}(a_{p-1},\ldots,a_0) &= x(a_{p-1},\ldots,a_0)\\
\tilde{x}_0(a_{p-1},\ldots,a_0) &= (\tilde{x}_{-1}(0,a_{p-2},\ldots,a_0) + \omega_2^{a_{p-1}}\,\tilde{x}_0(1,a_{p-2},\ldots,a_0))\,\omega_P^{\langle a_{p-2},\ldots,a_0\rangle a_{p-1}}\\
\tilde{x}_1(a_{p-1},\ldots,a_0) &= (\tilde{x}_0(a_{p-1},0,a_{p-3},\ldots,a_0) + \omega_2^{a_{p-2}}\,\tilde{x}_0(a_{p-1},1,a_{p-3},\ldots,a_0))\,\omega_{\frac{P}{2}}^{\langle a_{p-3},\ldots,a_0\rangle a_{p-2}}\\
&\vdots\\
\tilde{x}_q(a_{p-1},\ldots,a_0) &= (\tilde{x}_{q-1}(a_{p-1},\ldots,0_{p-q-1},\ldots,a_0) + \omega_2^{a_{p-q-1}}\,\tilde{x}_{q-1}(a_{p-1},\ldots,1_{p-q-1},\ldots,a_0))\,\omega_{\frac{P}{2^q}}^{\langle a_{p-q-2},\ldots,a_0\rangle a_{p-q-1}}\\
&\vdots\\
\tilde{x}_{p-1}(a_{p-1},\ldots,a_0) &= (\tilde{x}_{p-2}(a_{p-1},\ldots,a_1,0) + \omega_2^{a_0}\,\tilde{x}_{p-2}(a_{p-1},\ldots,a_1,1))\,\omega_2^{0a_0}\\
X(a_{p-1},\ldots,a_0) &= \tilde{x}_{p-1}(a_0,\ldots,a_{p-1})
\end{aligned}$$

The iterative formulation can be generalized to radix-R FFT. With $s \in [0, u-1]$ ($u = \log_R$), and $(d_{u-1}d_{u-2} \ldots d_0)$ the addresses expressed in base R, the radix-R, in-place, DIF FFT can be written as

$$\begin{aligned}
\tilde{x}_{-1}(d_{u-1},\ldots,d_0) &= x(d_{u-1},\ldots,d_0)\\
\tilde{x}_s(d_{u-1},\ldots,d_0) &= \omega_{\frac{P}{R^s}}^{\langle d_{u-s-2},\ldots,d_0\rangle \widehat{d_{u-s-1}}} \sum_{j=0}^{R-1} \tilde{x}_{s-1}(d_{u-1},\ldots,d_{u-s},j,d_{u-s-2},\ldots,d_0)\,\omega_R^{\widehat{d_{u-s-1}}j}\\
\tilde{x}_{u-1}(d_{u-1},\ldots,d_0) &= \omega_{\frac{P}{R^{u-1}}}^{\langle 0\rangle \widehat{d_0}} \sum_{j=0}^{R-1} \tilde{x}_{u-2}(d_{u-1},\ldots,d_1,j)\,\omega_R^{\widehat{d_0}j}\\
X(\widehat{d_{u-1}},\ldots,\widehat{d_0}) &= \tilde{x}_{u-1}(d_0,\ldots,d_{u-1})
\end{aligned}$$

where the bit-reversed value of a digit d_i is $\widehat{d_i}$.

For a radix-2^r, in-place, DIF FFT with normal order input the required twiddle factor exponent for data in location $(a_{p-1}a_{p-2}\ldots a_0) = (d_{u-1}d_{u-2}\ldots u_0)$ is $\widehat{d_{u-1}} \times (d_{u-2}d_{u-3}\ldots d_0)$ after the first radix-R stage. For the second radix-R stage the set of twiddle factor exponents are $\widehat{d_{u-2}} \times (d_{u-3}d_{u-4}\ldots d_0)2^r$. The first factor in a twiddle factor index is the *bit-reversed* value of a radix-R digit defined by r consecutive address bits. For the first radix-R stage the digit is defined by the r highest order address bits, for the second radix-R stage the next r address bits, etc. The second factor in the twiddle factor index is defined by all address bits of lower order than the radix-R digit used for the first factor. The twiddle factor index is obtained by multiplying the product of the two factors obtained from the address by 2^r as many times as corresponds to the radix-R stage, with the first stage being stage zero. In general, for a radix-R *in-place* DIF FFT algorithm the twiddle factor index required for the data item in location $(a_{p-1}a_{p-2}\ldots a_0)$ after the sth radix R stage is $\widehat{d_{u-s-1}} \times (d_{u-s-2}d_{u-s-3}\ldots d_0)2^{sr}$.

If the FFT of size P is computed on a Boolean n-cube, the data allocation is cyclic, and the input is in normal order, then the computations corresponding to the first $p - n$ radix-2 stages, or $\frac{p-n}{r}$ radix-2^r stages, only involves local data. The indices of the twiddle factors required for radix-2^r stage s in processor $(a_{n-1}a_{n-2}\ldots a_0)$ are $\{\widehat{d_{u-s-1}}\} \times (\{d_{u-s-2}d_{u-s-3}\ldots d_{\frac{n}{r}}\}d_{\frac{n}{r}-1}\ldots d_0)2^{sr}$. The notation $\{\ldots\ldots\}$ denotes the set of all values that can be assumed by the digit string within the braces. When $\frac{P}{N}$ is a multiple of R, then $(\frac{P}{N} - 1)$ twiddle factors are needed for the local stages.

After the local stages the remaining computation corresponds to $\frac{P}{N}$ independent FFTs of size N, each with one element per processor. All $\frac{P}{N}$ FFTs require the same set of twiddle factors. A total of $\lceil\frac{n}{r}\rceil - 1$ twiddle factors are needed maximally per processor for the interprocessor communication stages, one for each radix-R butterfly stage, except the last stage. Hence, for cyclic data allocation, normal input order, and a radix-2^r DIF FFT of size P computed on N processors, $N < P$, the maximum number of distinct twiddle factors needed in a processor is $\frac{P}{N} + \lceil\frac{n}{r}\rceil - 2$. Allocating twiddle factor storage uniformly across all processors yield a total twiddle factor storage of $P + (\lceil\frac{n}{r}\rceil - 2)N$, which for $P \gg N$ is about twice the storage required on a sequential computer. For $P = N$ the uniform twiddle factor storage across processors yields a total storage of $(\lceil\frac{n}{r}\rceil - 1)N$, which exceeds the sequential storage by a factor of approximately $(\lceil\frac{n}{r}\rceil - 1)\frac{R}{R-1}$.

3.5.2 Decimation-in-time

Higher radix FFT of the decimation-in-time type requires complex multiplications for the input to stages q mod r, where q is the radix-2 stage. As in the decimation-in-frequency case we consider an in-place algorithm, and express the twiddle factors needed for the data in a given location $(a_{n-1}a_{n-2}\ldots a_0)$ in terms of the address, and stage number. Using the iterative formulation for the radix-2 case yields

$$\begin{aligned}
\tilde{x}_{-1}(a_{p-1},\ldots,a_0) &= x(a_{p-1},\ldots,a_0)\\
\tilde{x}_0(a_{p-1},\ldots,a_0) &= \tilde{x}_{-1}(0,a_{p-2},\ldots,a_0) + \omega_2^{a_{p-1}}\,\omega_2^0\tilde{x}_{-1}(1,a_{p-2},\ldots,a_0)\\
\tilde{x}_2(a_{p-1},\ldots,a_0) &= \tilde{x}_1(a_{p-1},0,a_{p-3},\ldots,a_0) + \omega_2^{a_{p-2}}\,\omega_4^{\langle a_{p-1}\rangle}\tilde{x}_1(a_{p-1},1,a_{p-3},\ldots,a_0)\\
&\vdots\\
\tilde{x}_s(a_{p-1},\ldots,a_0) &= \tilde{x}_{s-1}(a_{p-1},\ldots,0_{p-s-1},\ldots,a_0) + \omega_2^{a_{p-s-1}}\,\omega_{2^{s+1}}^{\langle a_{p-s},\ldots,a_{p-1}\rangle}\tilde{x}_{s-1}(a_{p-1},\ldots,1_{p-s-1},\ldots,a_0)\\
&\vdots\\
\tilde{x}_{p-1}(a_{p-1},\ldots,a_0) &= \tilde{x}_{p-2}(a_{p-1},\ldots,a_1,0) + \omega_2^{a_0}\,\omega_P^{\langle a_1,\ldots,a_{p-1}\rangle}\tilde{x}_{p-2}(a_{p-1},\ldots,a_1,1)\\
X(a_{p-1},\ldots,a_0) &= \tilde{x}_{p-1}(a_0,\ldots,a_{p-1})
\end{aligned}$$

A radix-R, in-place, DIT FFT can be written as

$$\begin{aligned}
\tilde{x}_{-1}(d_{u-1},\ldots,d_0) &= x(d_{u-1},\ldots,d_0)\\
\tilde{x}_s(d_{u-1},\ldots,u_0) &= \sum_{j=0}^{R-1}\omega_R^{\widehat{d_{u-s-1}}j}\,\omega_{R^{s+1}}^{\langle\widehat{d_{u-s}},\ldots,\widehat{d_{u-1}}\rangle j}\,\tilde{x}_{s-1}(d_{u-1},\ldots,d_{u-s},j,d_{u-s-2},\ldots,d_0)\\
\tilde{x}_{u-1}(d_{u-1},\ldots,u_0) &= \sum_{j=0}^{R-1}\omega_R^{\widehat{d_0}j}\,\omega_N^{\langle\widehat{d_1},\ldots,\widehat{d_{u-1}}\rangle j}\,\tilde{x}_{u-2}(d_{u-1},\ldots,d_1,j)\\
X(\widehat{d_{u-1}},\ldots,\widehat{d_0}) &= \tilde{x}_{u-1}(d_0,\ldots,d_{u-1})
\end{aligned}$$

The indices of the twiddle factors for normal order input are all one for the first stage, $j \times \widehat{d_{u-1}}2^{p-2r}$ for the second radix-R stage, and $j \times (\widehat{d_{u-s}}\ldots\widehat{d_{u-1}})2^{p-(s+1)r}$ for an arbitrary stage s. Note, that the address is bit-reversed and shifted for the proper exponent. If the P complex data points are allocated consecutively and are in normal order, then the data in address location $\underline{(a_{p-1}a_{p-2}\ldots a_0)} = (d_{u-1}d_{u-2}\ldots d_0)$ requires twiddle factors with indices $\{j\}\times(\{\widehat{d_{u-s}}\ldots\widehat{d_{u-\frac{n}{r}-1}}\}\widehat{d_{u-\frac{n}{r}}}\ldots\widehat{d_{u-1}})2^{p-(s+1)r}$ for stage s of an in-place DIT algorithm. With a *consecutive* data allocation the processor address bits form the high order bits of the element index. The first $\frac{n}{r}$ radix-R butterfly stages correspond to $\frac{P}{N}$ independent FFTs of size N. All these FFT require the same set of twiddle factors. The local addresses do not enter into the index computation. Moreover, the first stage does not require any twiddle factor. The last $u-\frac{n}{r}$ radix-R stages are local to a processor. The maximum total number of twiddle factors required in a processor is $\frac{P}{N}+\lceil\frac{n}{r}\rceil-2$, the same as for cyclic data allocation, normal input order, and in-place decimation-in-frequency FFT. The set of twiddle factors required in a processor is the same as for cyclic data allocation, bit-reversed input order and a decimation-in-frequency, in-place FFT. The load balance and arithmetic efficiency for a radix-R in-place decimation-in-time FFT is the same as for the in-place radix-R decimation-in-frequency algorithm.

3.5.3 Bit-reversed input

With the input in normal order the FFT computation proceeds from the highest order bit to the lowest order bit with respect to data being paired for a butterfly computation. With the input in bit-reversed order the traversal of the bits in the address field is from the lowest order to the highest order bit. With the data indices being bit-reversed with respect to the addresses the decimation-in-frequency FFT requires addresses in bit-reversed order instead of normal order for the twiddle index computation. Similarly, the decimation-in-time FFT requires addresses in normal order instead of in bit-reversed order for normal order inputs. With these differences the consecutive ordering yields the smallest requirements for twiddle factor storage for the decimation-in-frequency FFT, and cyclic storage for the decimation-in-time FFT. The preferred combinations of data allocation and FFT type is the opposite compared to normal order input.

3.5.4 Inverse FFT

The Inverse Discrete Fourier Transform (IDFT) is defined by

$$\tilde{x}(j) = \sum_{l=0}^{P-1} \omega_P^{-lj} X(l), \quad \forall j \in [0, P-1], \quad \omega_P = e^{-\frac{2\pi i}{P}}.$$

It is easy to show that $\tilde{x}(j) = Px(j)$. For the computation of the IDFT we notice that $\omega_P^{-lj} = \omega_P^{(P-l)j}$. Hence, the IDFT can be computed by using $P-l$ as the index of the twiddle factors used for a DFT. The scaling can either be made by $\sqrt{P}$ during both the DFT and the IDFT, or by P during either the DFT, or the IDFT. With exception of the twiddle factor index the computations are identical.

3.5.5 Multi-dimensional FFT

In general, each axis has its set of twiddle factors. The twiddle factors are a function of the axis length. The twiddle factor for an axis is a subset of the twiddles for the longest axis. With axes of length $P_1 \times P_2 \times \ldots P_k$ the minmum number of twiddle factors is $\max_\ell (R-1)\frac{P_\ell}{R}$. With separate storage of the twiddle factors for each axis the total storage is $\sum_\ell (R-1)\frac{P_\ell}{R}$, which is less than the required storage for a one-dimensional FFT of size $\Pi_\ell P_\ell$.

3.5.6 Reduced twiddle factor storage

For radix-2 FFT a reduction in twiddle factor storage needs by a factor of two is possible by performing 90-degree rotations "on-the-fly" [13]. The reduction is based on the observation that for consecutive data allocation, normal order input, and decimation-in-time radix-2 FFT, the set of twiddle factor indices in the last stage is $\{a_1 a_2 \ldots a_{n-1}\}|a_n \ldots a_{p-1}$. The highest order bit a_1 corresponds to bit position $p-2$. Hence, $\{1 a_2 \ldots a_{n-1}\}|a_n \ldots a_{p-1} =$

$\frac{P}{4} + \{0a_2 \ldots a_{n-1}\}|a_n \ldots a_{p-1}$. But, $\omega_P^{\frac{P}{4}} = -i$. In the radix-2 case half of the twiddle factors can be obtained from the other half without any arithmetic. This property is true for all on-processor stages. In the case of a radix-4 FFT the highest order bit position corresponds to position $p-3$ representing a difference between the indices $\{0a_3 \ldots a_{n-1}\}|a_n \ldots a_{p-1}$ and $\{1a_3 \ldots a_{n-1}\}|a_n \ldots a_{p-1}$ of $\frac{P}{8}$. The relationship $\cos(\alpha) = \sin(\frac{\pi}{2} - \alpha)$ can be used to save storage in this case (as well as an additional savings in the radix-2 case). For a higher radix than four additional compaction of the table for $\{a_r a_{r+1} \ldots a_{n-1}\}|a_n \ldots a_{p-1}$ requires arithmetic operations.

3.6 Summary of algorithmic and data layout issues

The communication efficiency can be improved from $\frac{1}{n}$ in the naive implementation of Cooley-Tukey FFT on a Boolean n-cube to 1, if $\frac{P}{N} \geq 1$. N-way multi-sectioning yields full utilization of the communications bandwidth. R-way multi-sectioning for $R < N$ can be pipelined to achieve full utilization of the communication bandwidth. The second best value of R with respect to communication time is 4, and the third best value is 2, which only requires one more communication compared to 4-way sectioning. All multi-sectioning algorithms yield perfect load balance. With cyclic data allocation multi-sectioning only requires communication once in each of the n processor dimensions. With consecutive data allocation at least one additional inter-processor communication is necessary. For bit-reversed input order the data allocation shall be consecutive for optimum effectiveness of the multi-sectioning technique.

A higher radix FFT can be pipelined in the same way as a radix-2 FFT, if the high radix FFTs are factored into radix-2 butterfly computations. A high radix FFT performed in this way yields no reduction in the communication complexity compared to a radix-2 FFT. Performing a radix-2^r FFT in unfactored form by all-to-all broadcasting in r dimensional subcubes results in a higher communication complexity. For $\frac{P}{N} \gg 1$ the communication complexity of the pipelined algorithm is approximately twice that of the multi-sectioning algorithm.

The number of complex multiplications is reduced by a factor of two for a radix-4 FFT, and by a factor of approximately three for a radix-8 FFT. Multi-sectioning yields perfect arithmetic load balance. The arithmetic load balance improves as $\frac{R-1}{R}$ with increasing radix for an FFT across processors. The demand for memory bandwidth is reduced in proportion to $\log_2 R$. The multi-sectioning technique benefits fully from the reduced need of memory bandwidth of a high radix FFT.

The twiddle factor indices required in a processor and the total storage requirements are summarized in Tables 5 and 6. The storage requirements for on-processor twiddles can be reduced by a factor of two for radix-2 FFT by computing half of the twiddles by performing 90-degree rotations "on-the-fly". An additional factor of two reduction in twiddle factor storage is possible for radix-2 FFT, and a factor of two for radix-4 FFT by using the relation $\cos(\alpha) = \sin(\frac{\pi}{2} - \alpha)$.

The inverse Discrete Fourier Transform can be computed as a Discrete Fourier transform

FFT	Data	Twiddle	Max. number
	alloc.	index	of twiddles
		stage s	per proc.
DIT	consec.	$\{j\} \times (\{\widehat{d_{u-s}} \dots \widehat{d_{u-\frac{n}{r}-1}}\}\widehat{d_{u-\frac{n}{r}}} \dots \widehat{d_{u-1}})2^{p-(s+1)r}$	$\frac{P}{N} + \frac{n}{r} - 2$
DIF	cyclic.	$\{\widehat{d_{u-s-1}}\} \times (\{d_{u-s-2}d_{u-s-3} \dots d_{\frac{n}{r}}\}d_{\frac{n}{r}-1} \dots d_0)2^{sr}$	$\frac{P}{N} + \frac{n}{r} - 2$

Table 5: Radix-2^r twiddle factor storage, normal input order.

FFT	Data	Twiddle	Max. number
	alloc.	index	of twiddles
		stage s	per proc.
DIT	cyclic.	$\{\widehat{d_{u-s-1}}\} \times (\{d_{u-s-2}d_{u-s-3} \dots d_{\frac{n}{r}}\}d_{\frac{n}{r}-1} \dots d_0)2^{sr}$	$\frac{P}{N} + \frac{n}{r} - 2$
DIF	consec.	$\{j\} \times (\{\widehat{d_{u-s}} \dots \widehat{d_{u-\frac{n}{r}-1}}\}\widehat{d_{u-\frac{n}{r}}} \dots \widehat{d_{u-1}})2^{p-(s+1)r}$	$\frac{P}{N} + \frac{n}{r} - 2$

Table 6: Radix-2^r twiddle factor storage, bit-reversed input order.

by replacing the twiddle factor index l by $P - l$ for a transform of size P, or by using conjugated twiddle factors.

With the exception of the twiddle factors there is no essential difference between a multi-dimensional FFT and a one-dimensional FFT. Pipelining can be extended across axes of the array on which the FFT is performed, as long as there are no on-processor dimensions inter-mixed with inter-processor dimensions.

4 A Connection Machine implementation of local high-radix FFT.

Consecutive data allocation is used by all programming systems on the Connection Machine. A decimation-in-time FFT is used for data in normal input order, and a decimation-in-frequency FFT for bit-reversed input order. This combination of data input order and FFT minimizes the requirements for twiddle factor storage. The inverse Discrete Fourier Transform is computed using the conjugated twiddle factors.

The FFT routine described below is a complex-to-complex local FFT with input and output data in local memory. The inter-processor communication part can be accomplished though multi-sectioning, or pipelining of a high radix FFT, or a radix-2 as described in [13]. The data is assumed to be mapped into the global address space by a binary encoding. For normal order input the output is in bit-reversed order. If the output is desired in normal order a reordering is made after the FFT is computed.

The standard form of data storage on the Connection Machine is *field-wise* storage. The bits of a word are stored in successive memory locations of a processor. But, groups of 32 processors share a floating-point unit that can access the memories of the 32 processors in parallel, *slice-wise*. The FFT routines are developed for data stored in this form. In the "slice-wise" view of the Connection Machine there are up to 2048 floating-point processors, each with a 32-bit wide data path to memory. Each such unit has 64k 4 byte words in the 512 Mbyte total memory option, and 256k 4 byte words in the 2 Gbyte memory option. The floating-point units are interconnected as an 11-dimensional Boolean cube with two communication channels between every pair of units.

The conversion from field-wise to slice-wise storage is performed before the actual FFT computation starts. The change of data storage form affects the stride along any axis. The set of strides for an axis depends on how it is mapped to the memory, on-chip, and the lowest order bit of the off-chip address fields. For programming convenience a reordering of the local memory is performed after the change of storage form such that the stride for the first axis is one, the stride for the second axis is equal to the length of the local part of the first axis, etc. The strides for each axis are therefore constant. The memory reordering is a k-shuffle, which can be performed using the techniques in [2, 9, 16].

4.1 Organization of the local kernels.

For a data set of $P = 2^p$ complex data points it is necessary to use kernels of different radices. The current floating-point unit in the Connection Machine has a register bank of 32 registers, and three temporary registers. All twiddle factors for a radix-4 kernel can be stored in these registers, but for a radix-8 kernel there are too few registers for temporary variables and all twiddle factors. A higher radix than eight is not feasible. The following is a list of dichotomies that classify all cases that might occur:

1. radix-2/radix-4/radix-8
2. decimation-in-time/decimation-in-frequency
3. direct FFT/inverse FFT
4. normal order input/bit-reversed order input
5. load twiddle factors from memory/twiddle factors already in the register file/no twiddle factors

The following observations reduce the number of required radix-4/radix-8 kernels from 48 to 12.

- Due to the global organization of the FFT computations, only 2 subcases of the 4 defined by items number 2 and 4 will occur: normal input order and decimation-in-time FFT, and bit-reversed input order and decimation-in-frequency FFT.

m mod 3	number of radix-8 kernels	number of radix-4 kernels
0	$\frac{P}{3N}$	0
1	$\lfloor\frac{P}{3N}\rfloor$ -1	2
2	$\lfloor\frac{P}{3N}\rfloor$	1

Table 7: Decomposition of a local FFT of size $\frac{P}{N}$ into radix-4 and radix-8 kernels.

- For a radix-2 FFT the inverse FFT is the same as the direct FFT, except that the conjugates of the twiddle factors are used. For the high radix FFT algorithms the factoring of the kernels must account for the fact that the twiddles for the inverse FFT are the conjugates of the forward twiddle factors. But, the direct and inverse kernels can still be merged together, with conditionals handling the differences.
- Most operations are in common to the kernels enumerated in point five. The first and second subcase are the same, with the exception that no twiddle factors need to be loaded into the floating-point unit in the second case. The third subcase in addition avoid the complex multiplication with twiddle factors.

An FFT algorithm is typically expressed in terms of three nested loops. The outermost loop ranges over the stages of the FFT. The two inner loops are spelling out the number of times a given butterfly kernel (with twiddle factors) is used in a stage, and the number of different kernels in a stage. The decomposition of the FFT into stages of radix-4 and radix-8 kernels is such that as many radix-8 kernels as possible are used. For example, a local FFT of size 128 is decomposed into 1 stage of radix-8 kernels followed by 2 stages of radix-4 kernels; an FFT of size 4096 is decomposed into 4 stages of radix-8 kernels. The number of stages of radix-8 and radix-4 kernels used to perform a size 2^m local FFT, for any m greater than 1, is given in Table 7.

The outermost loop is called the "stage loop". The set of kernels using the same twiddle factors form a "group". The kernels in a group are executed consecutively in order to minimize the number of twiddle factor loads. The middle loop ranges over groups, the innermost loop ranges over kernels in a group. The number of groups (and the number of kernels in a group) changes from stage to stage. The product of the number of groups and the number of kernels in a group is the total number of kernels in a stage, which is equal to the FFT size divided by the size of the current kernel. Table 8 gives the number of groups and the number of kernels of size R per group, for a given radix-2 butterfly stage q. The loop structure is as follows:

```
for q:=0 to $p-n-1$ step $r$
  for g:=0 to nb_groups(q,$r$)
    for k:=0 to nb_kernels(q,$r$)
      call kernel of size $2^r$ on the appropriate data
```

FFT type	radix-2^r stage	number of groups	number of kernels per group
DIT, normal order input	s	2^{sr}	$\frac{P}{2^{(s+1)r}N}$
DIF, bit-rev. input	s	$\frac{P}{2^{(s+1)r}N}$	2^{sr}

Table 8: Number of groups and kernels per group.

For example, an FFT of size 128 computed by decimation-in-frequency, consists of 3 stages:

- a first stage of 16 groups each with 1 radix-8 kernel
- a second stage of 4 groups each with 8 radix-4 kernels
- a third stage of 1 group with 32 radix-4 kernels

In the last stage, only the first radix-4 kernel needs to load the twiddle factors from memory to the register file; the other 31 kernels will use these twiddle factors already in the register file. Performing the FFT by decimation-in-time instead yields the same stages, but in reverse order.

Each kernel requires a starting address and stride for data and twiddle factors. For instance, if we call a radix-4 kernel with pointer value `p` and stride value `st` for the data array, then the kernel will operate on the complex data points `p`, `p+st`, `p+2*st`, and `p+3*st`. The pointer in the twiddle factor table indicates where the kernel will find the twiddle factors it needs ($R-1$ of them at once for a radix R kernel). Each stage has its own twiddle factor storage, so the stride for the twiddle factors is always one. Moreover, since the twiddle factors used by all the kernels in a group are the same, the twiddle factor pointer needs to be updated only once for each group. The storage of the twiddle factors in the table is contiguous, so the pointer is incremented by $R-1$ for each group.

For the data array and normal order input (DIT), the stride in stage s is $\frac{P}{2^{(s+1)r}}$ (with radix R kernels). For the bit-reversed input order (DIF), the stride to use is 2^{sr}. The management of the pointer into the data array is more complicated. It is reset to 0 at the beginning of each "stage" iteration, then incremented by a certain quantity at each "group" iteration. Each "kernel" iteration also increments it (by a different quantity), but at the end of the last kernel iteration in a group, it is restored to the value it had before the first kernel iteration. The loop structure is

```
twiddle_pointer:=0
for q:=0 to $p-n-1$ step $r$
  data_pointer:=0
  data_stride:=stride(q,$r$)
  for g:=0 to nb_groups(q,$r$)
```

```
data_pointer_2 := data_pointer
for k:=0 to nb_kernels(q,r)
  kernel($2^r$,data_pointer_2,data_stride,twiddle_pointer)
  twiddle_pointer := twiddle_pointer + $2^r - 1$
  data_pointer_2 := data_pointer_2 + dp_inc_2
data_pointer := data_pointer + dp_inc
```

dp_inc has the value $\frac{P}{2^{sr}}$ in the normal order case (DIT), and the value $2^{(s+1)r}$ in the bit-reversed order case (DIF). In both cases the value of dp_inc_2 is 1.

4.1.1 Description of a Kernel

As an example we describe the radix-8 decimation-in-frequency kernel, which assumes the input data to be in normal order. The kernel can be used for both direct and inverse FFT (but the direct case is only of interest if the input is in bit-reversed order). The kernel exists in three versions: one that loads the twiddle factors from memory to the register file in the floating-point unit, one that assumes the twiddle factor are already in the register file (the "no load" kernel), and one that does not multiply by any twiddle factor (the "no multiplication" kernel).

The kernel is decomposed into 5 logical steps:

1. Load the data into the register file of the floating-point unit
2. Perform the internal butterflies (3 stages: 2.1, 2.2, 2.3)
3. Load twiddle factors into the register file
4. Multiply outputs of step 2 by the twiddle factors
5. Store the results in place of the data in memory

Remarks:

- Steps 4 and 5 are merged together, since store and multiplication operations can be overlapped.
- Direct and inverse versions differ in the middle stage of the internal butterflies, step (2.2), and in the multiplication step.
- The "no load" kernel actually does load some twiddle factors, in step 3, because of lack of space in the register file. Three twiddle factors out of seven do not need to be reloaded, but four do.
- The "no multiplication" kernel skips steps 3 and 4.

The kernel microcode is called with 4 arguments (3 for "no multiplication" kernel):

- `data-start`: the address in CM memory of the first of the 8 input data needed by the kernel.
- `data-stride`: the difference between the addresses of two consecutive input data of the kernel.
- `coeff-start`: the address in CM memory of the first of the 7 twiddle factors needed by the kernel; not used in the "no multiplication" kernel. The 7 twiddle factors are stored in consecutive memory locations.
- `inverse`: an integer that is non-zero if an inverse FFT is required.

All the memory addresses are physical slice-wise addresses. The data and twiddle factors being complex single-precision numbers require two consecutive memory locations: real part followed by imaginary part.

4.2 The Twiddle Factors

In stage s (as defined above), a radix-R FFT ($R = 2^r$) needs $R-1$ twiddle factors per kernel. Since all the kernels in one group use the same set of twiddle factors, the number of twiddle factors used in one stage is the number of groups multiplied by $R-1$. This means that in stage s, the DIT FFT needs $(R-1)2^{sr}$ different twiddle factors, and the DIF FFT needs $\frac{(R-1)}{2^{(s+1)r}}\frac{P}{N}$ twiddle factors.

The DIT FFT is performed on data stored in normal order, with consecutive processor assignment. At stage s of the FFT, processor N_i needs the twiddle factors

$$\omega_{2^{(s+1)r}}^{j\times\widehat{N_i\|g}},\quad j\in[1,R-1],$$

for the kernels in group g, where $x\|y$ is the concatenation of x and y. For the DIF FFT with cyclic data allocation and the input in bit-reversed order (that is, consecutive assignment of the bit-reversed array of elements), the twiddle factors used by the kernels in group g in processor N_i at stage s are

$$\omega_{2^{p-sr}}^{j\times\widehat{N_i\|g}},\quad j\in[1,R-1].$$

The values of g in the DIF and the DIT FFT's are such that the two expressions above are exactly the same, if the substitution $s \leftarrow u-s-1$ is made in one of them. The DIF and the DIT FFT's are using the same set of twiddle factors, but are using them in reverse orders. The implementation is only using one table of twiddle factors for the DIT and the DIF FFT.

The following pseudo-code reflects how exactly the table of twiddle factors is stored in the processor memory. It generates them in the order used by the decimation-in-time FFT.

```
twiddle_pointer:=0
for q:=0 to $p-n-1$ step $r$
  for g:=0 to nb_groups(q,$r$)
```

```
    for d:=1 to $2^r - 1$
        twiddle[twiddle_pointer] := $\omega_{2^{(s+1)r}}^{j \times \widehat{N_i \| g}}$
        twiddle_pointer := twiddle_pointer + 1
```

The twiddle factors are computed in the field-wise representation, such that the twiddles are in the correct place after transposition to the slice-wise representation. The computation is subdivided among the 32 processors sharing a floating-point unit, such that each Connection Machine processor computes $\frac{1}{32}$ of the table. Each Connection Machine processor is represented by its binary expansion $\langle a_{n+5}, \ldots, a_0\rangle$ and the address within the part of the table located in the processor by $\langle e_{p-n-6}, \ldots, e_0\rangle$. All processors execute the following code:

```
index := $\langle a_{n+4}, \ldots, a_5\rangle \| \langle e_{p-n-6}, \ldots, e_0\rangle \| \langle a_4, \ldots, a_0\rangle$
for $q$:=0 to $p - n - 1$ step $r$
    if $(2^q - 1) \leq$ index $< (2^{q+r} - 1)$
        then $j$ := (index - $2^q$ + 1) mod $(2^r - 1)$ + 1
             $g$ := $\lfloor \frac{\text{index - } 2^q \text{ + 1}}{2^r - 1} \rfloor$
             $t$ := $q$
endfor
$N_i$ := $\langle a_{n+4}, \ldots, a_5\rangle$
twiddle := $\omega_{2^{t+n+1}}^{\widehat{N_i \| g} \times j}$
```

4.3 Performance measurements

The implementation of the local kernels as described above have been performs as presented in Table 9 and Figures 11 and 12. The measured timings do not include the time for twiddle factor computation. The timings only include the computation of the data local to the processors, and assumes that the data is already in the slice-wise representation. The result is in slice-wise form. The stride for the data is assumed to be one, i.e., any need for memory reordering due to the change of representation from field-wise to slice-wise is not included. The execution rates are scaled to a 64k processor Connection Machine model CM - 2. The number of floating-point operations for an FFT of size P is counted as $5P \log_2 P$. The expected performance gain through tuning of the kernels is at most 10%.

5 Summary

Multi-sectioning (all-to-all personalized communication) can be used to achieve full bandwidth utilization in FFT computations on Boolean cubes. On an n-cube 2^n sectioning is optimal. Pipelining the butterfly stages in a factored high radix FFT also achieves full bandwidth utilization for at least n data points per processor. However, the multi-sectioning only requires half the data motion. The number of element transfers for an FFT on P points is $\frac{P}{2N} + (\frac{n}{r} - 1)\frac{R}{2}$ for r-way sectioning compared to $\frac{P}{N} + n - 1$ for a pipelined high radix FFT. With consecutive data allocation, normal order input and decimation-in-time FFT,

FFT Size	*Direct, Norm* *Time (Gflops)*	*Direct, Bit-rev* *Time (Gflops)*	*Inverse, Norm* *Time (Gflops)*	*Inverse, Bit-rev* *Time (Gflops)*
4	.000053 (1.5)	.000044 (1.9)	.000050 (1.6)	.000046 (1.8)
8	.000060 (4.1)	.000064 (3.9)	.000058 (4.2)	.000055 (4.5)
16	.000148 (4.4)	.000149 (4.4)	.000157 (4.2)	.000138 (4.7)
32	.000259 (6.3)	.000243 (6.8)	.000256 (6.4)	.000253 (6.5)
64	.000461 (8.5)	.000489 (8.0)	.000499 (7.9)	.000490 (8.0)
128	.001244 (7.4)	.001241 (7.4)	.001290 (7.1)	.001243 (7.4)
256	.002539 (8.3)	.002543 (8.2)	.002643 (7.9)	.002551 (8.2)
512	.005195 (9.1)	.005199 (9.1)	.005428 (8.7)	.005196 (9.1)
1024	.013409 (7.8)	.013442 (7.8)	.013938 (7.5)	.013420 (7.8)
2048	.027621 (8.4)	.026535 (8.7)	.028667 (8.0)	.027580 (8.4)
4096	.057607 (8.7)	.053574 (9.4)	.059617 (8.4)	.055651 (9.0)
8192	.134013 (8.1)	.134192 (8.1)	.134090 (8.1)	.134363 (8.1)

Table 9: CM time (in seconds) and execution rates for a 64k processor CM – 2.

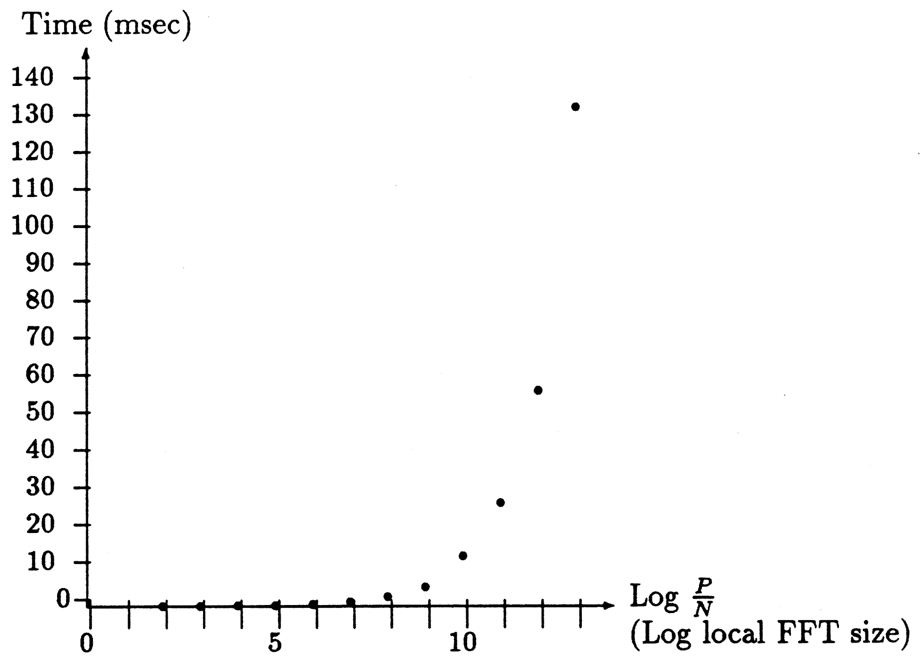

Figure 11: Execution time for 2048 direct DIT FFT (normal order input) on a 64k CM – 2.

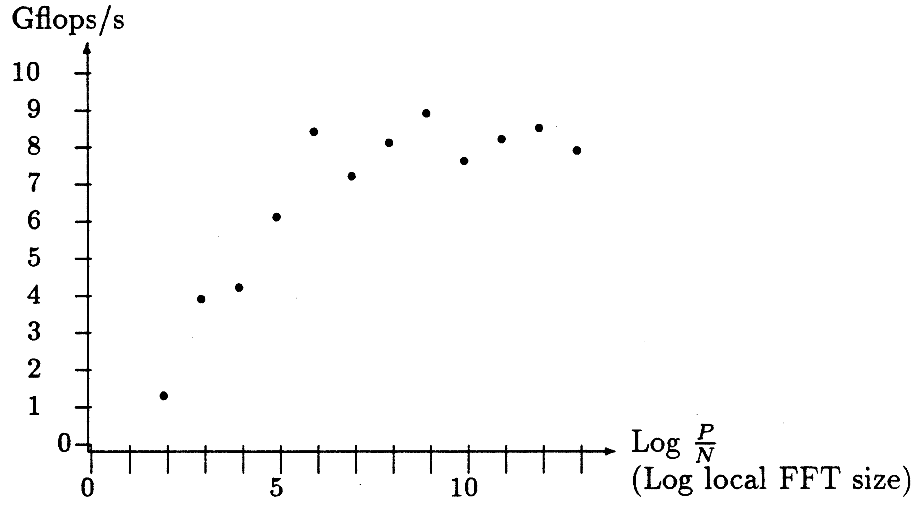

Figure 12: Execution rate for 2048 direct DIT FFT (normal order input) on a 64k CM – 2.

and bit-reversed order input and decimation-in-frequency FFT a twiddle factor storage per node of $\frac{P}{N} + \frac{n}{r} - 2$ suffice. No computation, or communication of twiddle factors is necessary with this amount of storage. For radix-2 FFT a reduction in storage by a factor of two is possible by computing 90-degree rotations "on-the-fly". For a radix-4 FFT, as well as for radix-2 FFT, the symmetry relation $\cos(\alpha) = \sin(\frac{\pi}{2} - \alpha)$ can be used to reduce the storage requirements.

An FFT algorithm based on multi-sectioning of the data set fully benefits from the reduced memory bandwidth requirement of a high radix FFT. Such an FFT algorithm also achieves 100% arithmetic load balance. A high radix, pipelined FFT fully benefits from the reduced memory bandwidth requirements for the local stages, but will not require any lower memory bandwidth than a radix-2 FFT for the inter-processor communication stages. The arithmetic load balance for the local stages is 100%, and $\frac{R-1}{R}$ for a radix-R pipelined, inter-processor FFT.

The performance of the current Connection Machine implementation of the local mixed radix-4/radix-8 FFT is in the range of 7 – 9 Gflops/s for local kernels of size 64 or greater. The performance advantage of the radix-8 kernels over the radix-4 kernels is significant. Compared to a local radix-2 FFT [13] the performance gain is approximately a factor of 2.5 – 3.5.

Acknowledgement

This work was in part supported by the U.S. Air Force under grant AFOSR-89-0382.

References

[1] Jim C. Cooley, P.A.W. Tukey, and P.D. Welch. *J. Sound Vibrations*, 12(3):315–337, 1970.

[2] Ching-Tien Ho and S. Lennart Johnsson. Stable dimension permutations on Boolean cubes. Technical Report YALEU/DCS/RR-617, Department of Computer Science, Yale University, October 1988.

[3] J.W. Hong and H.T. Kung. I/O complexity: The red-blue pebble game. In *Proc. of the 13th ACM Symposium on the Theory of Computation*, pages 326–333. ACM, 1981.

[4] S. Lennart Johnsson. Odd-even cyclic reduction on ensemble architectures and the solution tridiagonal systems of equations. Technical Report YALE/DCS/RR-339, Dept. of Computer Science, Yale University, October 1984.

[5] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *J. Parallel Distributed Comput.*, 4(2):133–172, April 1987.

[6] S. Lennart Johnsson. Solving tridiagonal systems on ensemble architectures. *SIAM J. Sci. Statist. Comput.*, 8(3):354–392, May 1987.

[7] S. Lennart Johnsson. *Optimal Communication in Distributed and Shared Memory Models of Computation on Network Architectures.* Morgan Kaufman, 1989.

[8] S. Lennart Johnsson and Ching-Tien Ho. Multiple tridiagonal systems, the alternating direction method, and Boolean cube configured multiprocessors. Technical Report YALEU/DCS/RR-532, Dept. of Computer Science, Yale University, New Haven, CT, June 1987.

[9] S. Lennart Johnsson and Ching-Tien Ho. Shuffle permutations on Boolean cubes. Technical Report YALEU/DCS/RR-653, Department of Computer Science, Yale University, October 1988.

[10] S. Lennart Johnsson and Ching-Tien Ho. Emulating butterfly networks on gray code encoded data in Boolean cubes. Technical report, Department of Computer Science, Yale University, 1989. in Preparation.

[11] S. Lennart Johnsson and Ching-Tien Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computers*, 38(9):1249–1268, September 1989.

[12] S. Lennart Johnsson, Ching-Tien Ho, Michel Jacquemin, and Alan Ruttenberg. Computing fast Fourier transforms on Boolean cubes and related networks. In *Advanced Algorithms and Architectures for Signal Processing II*, volume 826, pages 223–231. Society of Photo-Optical Instrumentation Engineers, 1987.

[13] S. Lennart Johnsson, Robert L. Krawitz, Douglas MacDonald, and Roger Frye. A radix-2 FFT on the Connection Machine. In *Supercomputing 89.* ACM, November 1989. Department of Computer Science, Yale University, Technical Report YALEU/DCS/RR-734, Technical Report NA 89-2, Thinking Machines Corp., September 1989.

[14] S. Lennart Johnsson and Peggy Li. Solutionset for AMA/CS 146. Technical Report 5085:DF:83, California Institute of Technology, May 1983.

[15] S. Lennart Johnsson, Uri Weiser, Danny Cohen, and Al Davis. Towards a formal treatment of VLSI arrays. In *Proceedings of the Second Caltech Conference on VLSI*, pages 375 – 398. Caltech Computer Science Dept., January 1981.

[16] Michael McKenna. *The Shuffle and Transpose Micorcode Routines*, 1989.

[17] Henri J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms.* Springer-Verlag, 1982.

[18] Alan V. Oppenheimer and Ronald W. Schafer. *Digital Signal Processing.* Prentice-Hall, Englewood Cliffs. NJ, 1975.

[19] E M. Reingold, J Nievergelt, and N Deo. *Combinatorial Algorithms.* Prentice-Hall, Englewood Cliffs. NJ, 1977.

[20] Thinking Machines Corp. *CM-Fortran Release Notes*, 1989.