The Future Fast Fourier Transform? *

Alan Edelman † Peter McCorquodale ‡ Sivan Toledo §

Abstract

It seems likely that improvements in arithmetic speed will continue to outpace advances in communications bandwidth. Furthermore, as more and more problems are working on huge datasets, it is becoming increasingly likely that data will be distributed across many processors because one processor does not have sufficient storage capacity. For these reasons, we propose that an inexact DFT such as an approximate matrix-vector approach based on singular values or a variation of the Dutt-Rokhlin fast-multipole-based algorithm [9] may outperform any exact parallel FFT. The speedup may be as large as a factor of three in situations where FFT run time is dominated by communication. For the multipole idea we further propose that a method of "virtual charges" may improve accuracy, and we provide an analysis of the singular values that are needed for the approximate matrix-vector approaches.

1 Introduction

In future high-performance parallel computers, improvements in floating-point performance are likely to continue to outpace improvements in communication bandwidth. Therefore important algorithms for the future may trade off arithmetic for reduced communication. Indeed, with the increasing popularity of networks of workstations and clusters of symmetric multiprocessors, even on present machines it may be worthwhile to make this tradeoff.

Traditional research into algorithmic design for the Fast Fourier Transform focuses on memory and cache management and organization. All such algorithms are in effect variations of the original algorithm of Cooley and Tukey [7]. A few important variants are the Stockham framework [5], the Bailey method [3], Swarztrauber's method [13] and the recent algorithm by Cormen and Nicol [6]. Also see Briggs and Henson [4], and Van Loan [14].

In our distributed-memory model, we assume that the input and output vectors are stored in natural order. In this model, the standard approach to the parallel FFT is known as the "six-step framework" [14, pages 173–174], consisting of: (1) a global bit reversal or shuffle, (2) local FFTs, (3) a global transpose, (4) multiplication by twiddle factors, (5) local FFTs, (6) a global shuffle or bit reversal. The global shuffles in steps (1) and (6) each require an amount of communication equivalent to the transpose in step (3). They may be saved if the order is not important. The communication pattern is as indicated in Figure 1.

*The first and second authors were supported by NSF grants 9501278-DMS and 9404326-CCR.

†Department of Mathematics Room 2-380, Massachusetts Institute of Technology, Cambridge, MA 02139-4307, edelman@math.mit.edu, WWW page http://www-math.mit.edu/~edelman.

‡Department of Mathematics Room 2-333, Massachusetts Institute of Technology, Cambridge, MA 02139-4307, petermc@math.mit.edu.

§Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304, toledo@parc.xerox.com. The work was done while this author was at the IBM T.J. Watson Research Center, Yorktown Heights, NY.

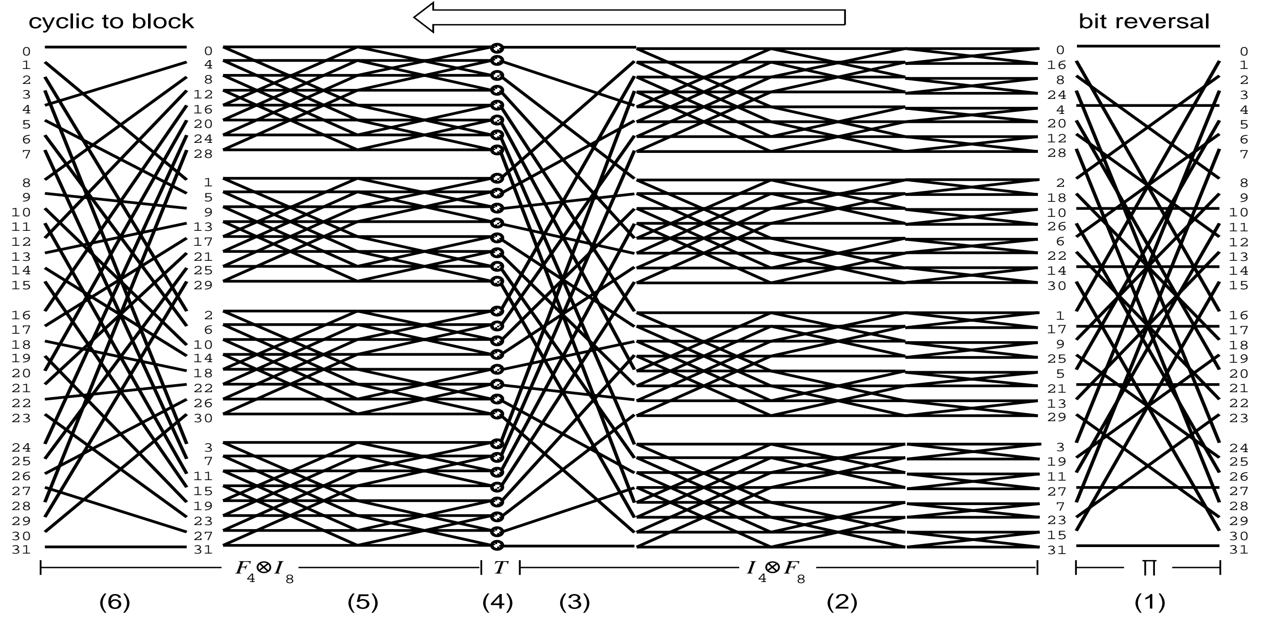

Fig. 1. *Communication pattern in parallel FFT of length 32 over 4 processors, using the six-step framework based on the factorization* $F_{32} = (F_4 \otimes I_8)T(I_4 \otimes F_8)\Pi$ *of equation (3) in section 4. The step numbers are indicated at the bottom of the figure.*

This paper presents a method which can save up to a factor of three in communication cost, by using an approximate algorithm that essentially combines the three global transposes into one. Accuracy can be extended to full machine precision with negligible effect on communication complexity.

The main contributions of this work are:

- the proposal that these algorithms in the parallel context may in fact be faster than the traditional algorithms;
- a mathematical analysis of the why these methods work in terms of singular values and their connection to the prolate matrix;
- a portable prototype MPI code that demonstrates the accuracy of the algorithm;
- an improvement of the Dutt-Rokhlin algorithm [9] that our experiments show can yield two additional digits of accuracy in the equispaced case.

2 Mathematical Insights

The operators that represent the relationship between the input on one processor and the output on another processor are nearly rank-deficient, thus allowing for speedups on parallel supercomputers.

The DFT of $x \in \mathcal{C}^n$ is $y = F_n x$, where

$$(F_n)_{jk} = \exp(-2\pi ijk/n) \qquad (0 \le j, k \le n-1).$$

The normalized matrix $\frac{1}{\sqrt{n}}F_n$ is unitary.

Let $F_{n|p}$ denote the top left $m \times m$ submatrix of the unitary matrix $\frac{1}{\sqrt{n}}F_n$, where $m = n/p$ is an integer. Then the singular values of $F_{n|p}$ are at most 1. These singular values

have an interesting property closely linked to the eigenvalues of the prolate matrix [15], and suggested by the plot in Figure 2:

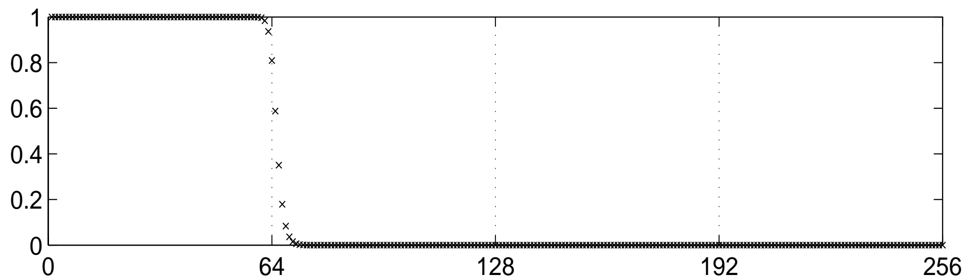

FIG. 2. *Singular values of $F_{1024|4}$, computed with* MATLAB.

THEOREM 2.1. *For fixed p and $0 < \epsilon < \frac{1}{2}$, let $S_n(a, b)$ represent the number of singular values of $F_{n|p}$ in the interval (a, b). Then asymptotically, with $m = n/p$:*

$$S_n(0, \epsilon) \sim \frac{1}{p}m, \; S_n(\epsilon, 1-\epsilon) \sim O(\log n), \; \text{and } S_n(1-\epsilon, 1) \sim (1 - \frac{1}{p})m.$$

3 Algorithm 1: A Matrix-Vector Algorithm

Our first algorithm for the DFT involves matrix-vector multiplication by SVD matrices. In the equation $y = F_n x$, if we write F_n as p^2 blocks of size $m = n/p$ and if p^2 divides n, then

$$\begin{pmatrix} y(0:m-1) \\ y(m:2m-1) \\ \vdots \\ y(n-m:n-1) \end{pmatrix} = \sqrt{n} \begin{pmatrix} F_{n|p} & DF_{n|p} & \cdots & D^{p-1}F_{n|p} \\ F_{n|p}D & DF_{n|p}D & \cdots & D^{p-1}F_{n|p}D \\ \vdots & \vdots & & \vdots \\ F_{n|p}D^{p-1} & DF_{n|p}D^{p-1} & \cdots & D^{p-1}F_{n|p}D^{p-1} \end{pmatrix} \begin{pmatrix} x(0:m-1) \\ x(m:2m-1) \\ \vdots \\ x(n-m:n-1) \end{pmatrix} \tag{1}$$

where $D = \text{diag}(1, \eta, \eta^2, \ldots, \eta^{m-1})$ and $\eta = \exp(-2\pi i/p)$. Since the number of significant singular values of $F_{n|p}$ is asymptotically only m/p, this suggests the idea of using compression to reduce communication.

The singular-value decomposition of $F_{n|p}$ is written $F_{n|p} = U\Sigma V^*$,
where U and V are $m \times m$ unitary matrices, and Σ is a diagonal matrix of singular values. Let Σ_k denote the matrix consisting of the first k rows of Σ, containing the k largest singular values. The value of k depends on the accuracy desired, but for fixed accuracy and fixed p, the results of the previous section tell us $k = m/p + O(\log m)$.

Let U_k be the first k columns of U. Then

$$F_{n|p} \approx U_k \Sigma_k V^*. \tag{2}$$

Our matrix-vector algorithm uses this approximation in equation (1).

The arithmetic complexity of the matrix-vector algorithm is $16mkp = 16m^2 + O(m \log m)$ flops per processor. Communication consists of an all-to-all personalized communication with each processor sending k scalars to each other processor. The total number of scalars sent by each processor is $(p-1)k = m(1-1/p) + O(\log m)$.

The Fast Fourier Transform, by comparison, has each processor sending $3m(1-1/p)$ scalars but uses only $5m \lg n$ flops. The matrix-vector multiplication algorithm using the

SVD saves as much as a factor of 3 in communication at the cost of greater arithmetic. In the next section, we show how a different algorithm using the fast multipole method can reduce the arithmetic but maintain this saving in communication.

4 Algorithm 2: Fast Multipole Approach

The fast multipole algorithm is based on the work of Dutt and Rokhlin [9]. They compute the DFT for nonequispaced points serially. We treat the equispaced case in parallel.

For parallel FFT computations on p processors, the standard "6-step framework" [14, pages 173–174] is based on the radix-p splitting [14, eqn. 2.1.5], factoring

$$F_n = (F_p \otimes I_m) T (I_p \otimes F_m) \Pi \tag{3}$$

where again $m = n/p$ and T is a diagonal matrix of twiddle factors,

$$T = \mathrm{diag}(I_m, \Omega, \Omega^2, \ldots, \Omega^{p-1}), \qquad \Omega = \mathrm{diag}(1, \omega, \omega^2, \ldots, \omega^{m-1}), \qquad \omega = \exp(-2\pi i/n),$$

and Π is a block-to-cyclic permutation.

Our algorithm uses the factorization $F_n = (I_p \otimes F_m)(F_p \otimes I_m) M \Pi$.
Using the fact that $I_p \otimes F_m$ and $F_p \otimes I_m$ commute, we obtain

$$(F_p \otimes I_m) T (I_p \otimes F_m) \Pi = F_n = (F_p \otimes I_m)(I_p \otimes F_m) M \Pi \tag{4}$$

which gives

$$M = (I_p \otimes F_m)^{-1} T (I_p \otimes F_m) = \mathrm{diag}(I_m, C^{(1)}, \ldots, C^{(p-1)}) \tag{5}$$

where the matrices $C^{(s)} = (c_{jk}^{(s)})$ have elements $c_{jk}^{(s)} = \rho^{(s)} [\cot(\frac{\pi}{m}(k - j + \frac{s}{p})) + i]$, with $\rho^{(s)} = \frac{1}{m} \exp(-i\pi s/p) \sin(\pi s/p)$.

For fast multiplication by $C^{(s)}$, we can use the one-dimensional fast multipole method of Dutt, Gu and Rokhlin [8]. We view each of these $p-1$ transformations as a mapping of m charges on a circle to values of the potential due to these charges, at points on the circle.

Dutt and Rokhlin [9] showed how the non-equispaced Fourier transform can be computed using the fast multipole method. In this article, we are restricted to an equispaced DFT but we use a different set of interpolating functions that offer greater accuracy in this restricted case. We also compute it in parallel, using the method of Greengard and Gropp [10] and Katzenelson [11].

The number of interpolation coefficients, t, must also be chosen high enough to obtain sufficient accuracy. In general, t will depend on the size of the problem and the number of particles in each box. Finite machine precision, however, will also provide an upper limit on t beyond which improvements in accuracy are not obtained.

Dutt and Rokhlin [9] use Chebyshev polynomials to interpolate potentials. We use an approximation by the potentials due to t "virtual charges" located at fixed positions within each box. In practice, the virtual-charge approximation is found to be more accurate.

Figure 3 shows the maximum relative 2-norm error in computation of $F_n x$ as a function of the number of coefficients used in expansions. Results are indicated for odd numbers only, because the error for even t is actually greater than that for $t-1$.

The authors have also found that when using double-precision arithmetic, accuracy is not improved for increasing t above 15, because of the effects of round-off error. In fact, with $t = 15$ for virtual charges or $t = 17$ for Chebyshev, the computed maximum error for a problem of size 32K is less than the error of 3×10^{-13} that one obtains using random data in MATLAB.

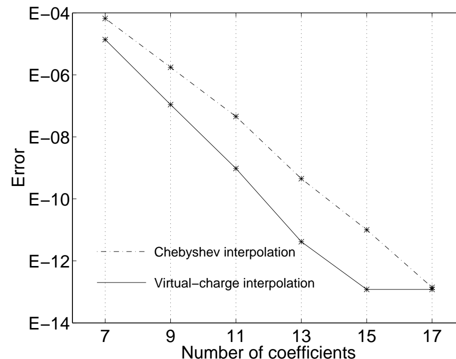

FIG. 3. *Maximum relative 2-norm error as a function of number of coefficients used, with Chebyshev polynomial interpolation (dashed line) and virtual-charge interpolation (solid line). The problem size is $n = 32768$, with $p = 4$ processors.*

In the fast-multipole-based DFT algorithm, the total number of flops per processor is bounded above by

$$\frac{n}{p}[5 \lg n + (1 - \frac{1}{p})(10 + 55t)] + 12t^2(2 \lg p - 7)(p - 1). \tag{6}$$

The total number of scalars sent by each processor is

$$(p - 1)[m/p + 64 + t(4 \lg m + \lg p - 29)]. \tag{7}$$

The number of messages required to be sent from each processor is at most $2p + 5 \lg p - 8$.

5 Experimental Results

We have implemented both our new algorithm and a conventional high-performance parallel FFT algorithm in order to assess the accuracy and performance of the new algorithm. We use our implementation to show below that the new algorithm is accurate and that it can outperform the performance of conventional FFT algorithms.

The experiments are intended to show that the performances of the two algorithms are within a small factor of each other, and that the relative speed of the two algorithms is determined by the communication-to-computation-rates ratio of the parallel computer on which they are executed. When the ratio is high, the conventional algorithm is faster. When the ratio is low, the new algorithm is faster. Our experiments are *not intended* to show that either of our implementations is a state-of-the-art code that is better than other parallel FFT codes. We do believe, however, that if both implementations are improved to a state-of-the-art level, our new algorithm would still prove faster on machines with fast processors and relatively slow communication network.

5.1 Performance Results

This section compares the performance of our implementations of the new algorithm and a conventional high-performance FFT algorithm. Both algorithms are coded in Fortran 77. We use a publicly available FFT package, FFTPACK [12], for local FFTs on individual processors, and MPI for interprocessor communication, thereby obtaining portable software.

Experiments were conducted on an IBM SP2 parallel computer [2]. The machine was configured with so-called thin nodes with 128 Mbytes of main memory. Thin nodes have a 66.7 MHz POWER2 processor [16], 64 Kbytes 4-way set associative level-1 data-cache, no level-2 cache, and a 64-bit-wide main memory bus. They have smaller data paths between the cache and the floating-point units than all other POWER2-based SP2 nodes.

The computation-to-communication balance of the SP2 can be summarized as follows. The peak floating-point performance of POWER2-based nodes is 266 million operations per seconds (Mflops). While many dense matrix operations run on these nodes at close to peak performance [1], FFT codes run at lower rates. Large power-of-two one-dimensional FFTs from FFTPACK run at 20–30 Mflops, and similar routines from IBM's Engineering and Scientific Subroutine Library (ESSL) run at 75–100 Mflops.

Table 1

A comparison of the performance of the two algorithms on an SP2 parallel computer using three communication mechanisms. The table compares the running time T_C of a conventional parallel FFT with the running time T_N of the new approximate DFT algorithm. Running times are in seconds. The three communication mechanisms that were used are user-space communication over the High-Performance Switch (US-HPS, 41 Mbytes/sec per node), internet protocol over the High-Performance Switch (IP-HPS, 31 Mbytes/sec per node), and internet protocol over ethernet (IP-EN, 1.25 Mbytes/sec for all nodes combined). The last two rows give the ratios of the actual timings to what one would expect from equation (8) for T_C, or (9) for T_N.

		US-HPS		IP-HPS		IP-EN	
p	n	T_C	T_N	T_C	T_N	T_C	T_N
2	32768	0.113	0.199	0.164	0.219	0.769	0.443
	65536	0.220	0.399	0.301	0.432	1.526	0.857
	131072	0.471	0.833	0.633	0.888	3.083	1.725
	262144	1.043	1.761	1.354	1.869	6.250	3.535
	524288	2.545	3.987	3.154	4.197	12.976	7.479
4	32768	0.059	0.128	0.109	0.152	1.213	0.602
	65536	0.116	0.268	0.199	0.302	2.368	1.198
	131072	0.220	0.563	0.355	0.614	5.928	2.528
	262144	0.469	1.171	0.719	1.264	11.474	4.902
	524288	1.033	2.441	1.504	2.605	18.540	8.726
	1048576	2.608	5.355	3.540	5.778	37.020	17.000
8	32768	0.031	0.077	0.061	0.101	1.708	1.263
	65536	0.070	0.150	0.114	0.179	3.166	2.117
	131072	0.140	0.296	0.266	0.358	7.225	3.196
	262144	0.265	0.593	0.446	0.681	12.983	5.691
	524288	0.556	1.288	0.866	1.410	22.165	10.097
	1048576	1.172	2.704	1.770	2.924	42.093	17.827
	2097152	2.823	5.926	3.926	6.320	85.428	33.783
	min ratio	5.503	4.984	3.482	4.597	1.193	1.356
	max ratio	10.060	6.890	5.405	6.087	1.639	1.918

Our results are summarized in Table 1 using $t = 16$ coefficients. We see that the conventional algorithm is faster with the two faster communication mechanisms, and that the new algorithm is faster with the slowest communication mechanism, IP over ethernet. The absolute running times using ethernet are very slow. Ethernet is also the only communication mechanism that does not allow additional processors to reduce the absolute running times, since it is a broadcast mechanism in which the total bandwidth does not grow

with the number of processors. The High-Performance Switch allows additional processors to decrease the absolute running times of both algorithms.

With a flop rate of FR (in flops per second) and a communications bandwidth of BW (in bytes per second), the times we would *expect* are:

$$(8) T_C = [5\frac{n}{p}\lg n + 6\frac{n}{p}]/FR + [3\frac{n}{p}(1 - \frac{1}{p})] \cdot (16 \text{ bytes})/BW$$

$$(9) T_N = [5\frac{n}{p}\lg\frac{n}{p} + 890\frac{n}{p}(1 - \frac{1}{p})]/FR + [(p-1)(\frac{n}{p^2} + 64\lg n - 48\lg p - 400)] \cdot (16 \text{ bytes})/BW$$

The last two rows of Table 1 show the minimum and maximum ratio of the actual times recorded to the times expected with $FR = 266$ Mflops and the maximum bandwidth for the particular communication mechanism.

5.2 Extrapolation to Other Machines

We saw that with Ethernet interconnect, our algorithm outperforms a conventional FFT. While Ethernet is not an appropriate communication medium for high-performance scientific computing, high-performance machines with similar communication-to-computation-rates ratio do exist and are likely to be popular platforms in the future.

Consider a cluster of symmetric multiprocessors connected with a fast commodity network. Sun Ultra Enterprise servers with 8 UltraSparc processors each, connected by an ATM switch are an example. The peak floating-point performance of each node is about 2.5 Gflops. Measurements by Bobby Blumofe with Sun Sparc workstations connected by a Fore ATM switch have shown that the application-to-application communication bandwidth of the switch is about 5 Mbytes per second per node in one direction (the nominal peak bandwidth of this network is 155 Mbits per second). Even if the network can support 5 Mbytes/sec in both directions, the communication-to-computation-rates ratio is only 0.002 bytes/flop.

The ratio in our SP2 experiments with ethernet is about 0.0022 bytes/flop when we use 2 nodes, 0.0010 with 4 nodes, and 0.0005 with 8 nodes. The peak performance of each node is 266 Mflops and the measured communication bandwidths are about 580, 270, and 132 Kbytes per second per node with 2, 4, and 8 nodes. Since the new algorithm outperformed the conventional FFT by a large margin even on two processors, when the ratio is 0.0022 bytes/flop, it seems safe to predict that the new algorithm would outperform a conventional FFT on the above-mentioned clusters whose ratios are even lower.

If we assume that tuning both algorithms would improve the performance of their local computations by a factor of 3, say, then the new algorithm would outperform a conventional FFT even if the networks of the clusters improved by a similar factor. This assumption is supported by the fact that a tuned high-performance local FFT routine (in ESSL) is about 3.75 times faster than the publicly available package that we used (FFTPACK).

6 Conclusions

The results of our experiments on the SP2 have shown that when the communication-to-computation-rates ratio is low, the new algorithm outperforms a conventional parallel FFT by more than a factor of 2. Quantitative performance extrapolation indicates that the new algorithm would also be faster on state-of-the art clusters of symmetric multiprocessors.

The new algorithm is faster when communication dominates the running time of conventional parallel FFTs. When communication is so expensive, both conventional and the new algorithms are not likely to be very efficient when compared to a uniprocessor FFT.

That is, their speedups are likely to be modest. There are at least two reasons to believe that the new algorithm would prove itself useful even when speedups are modest. First, in many applications the main motivation to use parallel machines is the availability of large memories, and not necessarily parallel speedups. In other words, it may be necessary to compute FFTs on multiple nodes because the data does not fit within the main memory of one node. Second, an FFT with a small or no speedup can be a part of a larger application which exhibits a good overall speedup. The application might include, for example, FFTs as well as grid computations, which require less communication per floating-point operation than the FFTs. In both cases, accelerating the parallel FFTs contributes to the performance of the application, whereas switching to a single-node FFT is not a viable option.

References

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair, *Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms*, IBM J. Res. Dev., 38 (1994), pp. 563–576.

[2] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir, *SP2 system architecture*, IBM Systems Journal, 34 (1995), pp. 152–184.

[3] D. H. Bailey, *FFTs in external or hierarchical memory*, Journal of Supercomputing, 4 (1990), pp. 23–35.

[4] W. L. Briggs and V. E. Henson, *The DFT: an owner's manual for the Discrete Fourier Transform*, SIAM, Philadelphia, 1995.

[5] W. T. Cochrane, J. W. Cooley, J. W. Favin, D. L. Helms, R. A. Kaenel, W. W. Lang, G. C. Maling, D. E. Nelson, C. M. Rader, and P. D. Welch, *What is the Fast Fourier Transform?*, IEEE Trans. Audio and Electroacoustics, AU-15 (1967), pp. 45–55.

[6] T. H. Cormen and D. M. Nicol, *Performing Out-of-Core FFTs on Parallel Disk Systems*, Tech. Report PCS-TR96-294, Dartmouth College, 1996.

[7] J. W. Cooley and J. W. Tukey, *An Algorithm for the Machine Calculation of Complex Fourier Series*, Math. Comput., 19 (1965), pp. 297–301.

[8] A. Dutt, M. Gu, and V. Rokhlin, *Fast algorithms for polynomial interpolation, integration and differentiation*, Research Report YALEU/DCS/RR-977, Yale University, 1993.

[9] A. Dutt and V. Rokhlin, *Fast Fourier transforms for nonequispaced data, II*, Applied and Computational Harmonic Analysis, 2 (1995), pp. 85–100.

[10] L. Greengard and W. D. Gropp, *A parallel version of the fast multipole method*, Computers Math. Applic., 20 (1990), pp. 63–71.

[11] J. Katzenelson *Computational structure of the N-body problem*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 787–815.

[12] P. N. Swarztrauber, *Vectorizing the FFT*, in Parallel Computations, Academic Press, New York, 1982.

[13] P. N. Swarztrauber, *Multiprocessor FFTs*, Parallel Computing, 5 (1987), pp. 197–210.

[14] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM, Philadelphia, 1992.

[15] J. M. Varah, *The Prolate Matrix*, Lin. Alg. Appl., 187 (1993), pp. 269–278.

[16] S. W. White and S. Dhawan, *POWER2: next generation of the RISC System/6000 family*, IBM J. Res. Dev., 38 (1994), pp. 493–502.