The Fast Fourier Transform on Hypercube Parallel Computers

Clare Yung-lei Chu

87-882

November 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

THE FAST FOURIER TRANSFORM ON HYPERCUBE PARALLEL COMPUTERS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Clare Yung-lei Chu

January 1988

The Fast Fourier Transform on Hypercube Parallel Computers

Clare Yung-lei Chu, Ph.D.
Cornell University 1988

The Fast Fourier Transform appears frequently in scientific computing. Therefore it is desirable to implement it efficiently on parallel computers. In this thesis, we investigate several different aspects of parallel Fast Fourier Transform implementation techniques for distributed-memory message-passing systems such as hypercube multiprocessors. We describe various Fast Fourier Transform algorithms using a matrix notation. An error analysis is presented that considers the effect of different methods used in the computation of the Fourier Transform coefficients as well as accumulated roundoff. New implementations of one and two-dimensional Fast Fourier Transforms are presented along with comparisons with existing methods. New algorithms for symmetric transforms are also developed and the results show excellent speedup when implemented on the Intel iPSC hypercube.

Biographical Sketch

The author was born on January 13, 1960 to Dr. Chong-Wei Chu and Mrs. Charlotte C. Chu in Minneapolis, MN where her father finished up his Ph.D. in Fluid Dynamics and her mother an M.S. in Biostatistics. Her family then moved to Southern California when she was three and she has lived there since, first graduating from Occidental College, Los Angeles with an A.B. degree in 1980 and then receiving a M.A. in Applied Mathematics from the University of California, San Diego, La Jolla. Her majors were sun-bathing and jogging, with a minor in windsurfing. And her summers were radically fun since she worked at T.R.W. Defense Systems Group, Redondo Beach, as a Programmer/Analyst.

The author's career at Cornell began when she decided to "experience" the East Coast. So she reviewed the Eastern universities that offer Applied Mathematics programs and chose Cornell both for its rural, scenic location and the quality of the faculty. Seasons changed, leaves fell, and wintry snow blew in a short spurt of spring, followed by sweltering summer heat. Still she vegged out at the Center for Applied Mathematics, studying matrix computations, partying at the International Living Center, jogging around the Plantations, and working out at the Ithaca Fitness Center. True she had lost her tan, but gained a world of knowledge.

After graduation, she will work as a Senior Engineer at Northrop Aircraft in Hawthorne, California; regain her bronze complexion, continue to pursue Ultimate Fitness, and enjoy the Liberal Arts in sunny Southern California. Her East Coast experience was, as they say, quite character-building. Only thing, she was already a Mega-Character before she got here, so one can surmise that these five years have made her a bit of a Hyper-Character. SoCal Rocks!

In the memory of my father,
Dr. Chong-Wei Chu

Acknowledgements

My sincere gratitude to Professor Charles Van Loan, my thesis advisor, for his kindness and understanding during these trying years of dissertation creation. He was instrumental in directing me into this avenue of research and gave me many hours of advice. The personal attention he gave me these past few years means a lot to me. I am also grateful for the generous financial support provided by Professor Van Loan through the following grants, ONR Contract N00014-83-K-640 and NSF Contract DCR 86-2310. I also wish to thank Professor Lars Wahlbin for his friendly help and interest in my well-being, and Professor Tom Coleman for serving on my committee and making himself available for an occasional chat. Finally I thank Professor Al Schatz for his willingness to substitute for Professor Wahlbin on my B-exam and his interest in my work.

The Center for Applied Mathematics went through four Center Directors during my stay here at Cornell and I would like to thank Professor Phil Holmes, Professor Lars Wahlbin, Professor Mike Todd, and Professor Sid Liebovich for the advice and help they gave me. I also sincerely appreciate Mrs. Dolores Pendell for lending me a helping hand whenever I needed it and for cutting through some of the administrative matters that a graduate student has to face.

My parents deserve appreciation for their love and care as well as providing me with an academic attitude, as do my brother Dale, and two sisters Constance and Charleen. I also appreciate the friendship of my former supervisor Mr. Richard Maitlen of T.R.W. for his interest in my career, as well as Professor Stanley "Gil" Williamson of U.C. San Diego for his confidence in my ability. My gratitude goes out to Professor Joan Moschovakis and Professor Benedict Freedman at Occidental College because if it weren't for their encouragement, I'd probably have been a Diplomacy and World Affairs major!

I wish to thank all my friends here and in California for all the fun I had these last five years. I also appreciate my cronies here at Applied Mathematics for putting up with my talkative nature. And the fact that my physical condition survived Ithaca is due to the great staff and facilities at the Ithaca Fitness Center. I will also miss the ambience and people at the International Living Center where

I lived during my years at Cornell. My friends have all helped me immensely over the past five years and I would like to thank them: Daniel Medina, Tim Waag, Ann Monzon, Jada Valderrama, Amelia Wu, Irene Duffy, Richard Koo, Sherease Pratt, Jeter Iseley, John Dailey, Cindy Anfinson, Chris Hempel, Glen Swindle, Ernie DeMarie, Alex Espalin, Gabrielle Tayac, and Kym Fraser. And lastly I am grateful to my boyfriend, José A. Ayala for his love and concern.

Table of Contents

List of Tables

List of Figures

Chapter 1

Preliminaries

1.1 Introduction

The parallel computation of numerical problems holds a tremendous amount of promise for speeding up large compute-intensive procedures. The hope is to be able to perform real-time computations on important applications such as image processing, the processing of seismic data, signal processing, and the solutions of partial differential equations. Buried inside a lot of scientific code is the discrete Fourier transform (DFT). The realization of a "fast" method for its computation in the 1960's [Cooley and Tukey (1965)] allows scientists to solve problems that were previously considered beyond the realm of practical computability. Many variations of the fast Fourier transform (FFT) were subsequently developed to cater to the parameters of specific problems and machine architectures. Having reached the limit with which a single processor can perform FFT computations, one is naturally led to consider the implementation of FFTs on parallel computers. This dissertation focuses on issues dealing with the design of new algorithms and procedures for the implementation of the FFT and fast symmetric transforms on distributed memory ensemble architectures, in particular the hypercube interconnected parallel machine.

The complex FFT is one of the first algorithms to be implemented on the first experimental hypercubes where inefficiencies due to load imbalance and communication overhead were noted. [Fox (1985)] While on the surface, the radix-2 FFT implementation appears straightforward and easy, further research reveals that there are many details with room for improvement. Furthermore the efficient implementation of symmetric transforms is more complicated and had not been extensively studied before. The purpose here is to increase the understanding of parallel Fourier transforms by presenting new methods and techniques for improving their execution rates on parallel message-passing computers. New heuristics for conceptualizing concurrent numerical computations are introduced in the form of a matrix language representing algorithm design, task allocation, data partitioning and inter-processor communication.

The first chapter provides the reader with an introduction to the Fast Fourier Transform algorithm interpreted according to matrix factorizations. The use of matrix language is well-suited for the description of numerical algorithms especially for distributed computations. By getting away from the *i-j* level, replete with multiple summations and backwards running indices, an overall view of the big picture can be obtained where blocks of data are partitioned among distributed processors and operated upon. The use of permutation matrices to describe communication is especially useful and provides a higher-level abstraction for algorithmic description and development. Some basic permutations and notational conventions are introduced in Chapter 1. The fast Fourier transform is also conveniently denoted as a sequence of sparse matrix multiplications. Four canonical in-place FFT algorithms are presented along with some of their properties. Facility with the matrix notation allows us to transcend the tangled web of *i-j* indices and derive different, but equivalent, versions of the FFT by rearranging the matrix products.

Chapter 2 presents an error analysis for the radix-2 FFT. A different approach is used where the FFT coefficients (or multipliers) are assumed to contain noise due to the method with which they are computed. This is a non-random, determinable error and dependent on the method that is used to calculate these trigonometric values. Several methods are highlighted, with their error properties examined. These errors are then incorporated into the error analysis of the FFT and are shown to contribute multiplicatively to the total FFT error.

Chapter 3 describes a new method of implementing one-dimensional complex FFTs on the hypercube. Issues such as load balancing and the mapping of data to the nodes are examined. A new load balanced method is presented and shown to execute faster than the naive method. This method also requires no extra buffers for communication and therefore is more storage efficient. This method is used subsequent implementations of symmetric transforms.

In Chapter 4, two-dimensional FFTs are discussed and implemented using four different methods that vary in degrees of concurrency, communication costs, and data transposition requirements. A model is derived that accounts for vector computers at the nodes. Some conclusions are drawn as to the effect of architecture and hardware parameters on algorithm choice.

Real symmetric transforms are the topic of both Chapters 5 and 6. The difficulties of implementing them by the usual sequential methods involving pre- and post- processing of the data is highlighted. Two new methods are presented that remedy this. The Chapter 5 method involves a modified pre- and post- processing scheme that computes the sine transform on data both in natural and bit-reversed order so that all post- processing is processor local. The method of Chapter 6 computes the sine, cosine, quarter-wave sine and quarter-wave cosine transforms of a single sequence in one pass through the complex FFT subroutine. This new method is hence extremely simple to implement and fast, as four symmetric transforms are obtained in one FFT with additional processing limited to multiplication by diagonal matrices. It is also a numerically stable way of computing symmetric transforms since all of the multiplications applied to the data vector are roots of unity.

Finally, Chapter 7 describes a particular implementation of a distributed mixed radix FFT for hypercube multiprocessors, demonstrating that hypercube FFTs are not limited solely to radix-2 methods.

1.2 Parallel Scientific Computing

The area of scientific computing is being revolutionized by the development of supercomputers and parallel machines. Large problems can now be speedily solved so that engineers can redesign and test problem parameters with faster turnaround times. More design parameters can be incorporated into a simulation to better model the physical properties of a particular problem. However as promising as these new developments are, the problem-solving methods must be carefully thought through to ensure numerical stability and accuracy as well as computational efficiency. Therefore using parallel processing architectures requires the rethinking of solution algorithms.

Efficient implementation of numerical algorithms on parallel architectures depends on a host of system parameters, such as memory organization, processor power, communication interconnection between the processors and processor load balancing. A shared-memory machine consists of a number of processors that can address a single global memory. Memory access conflicts occur when two processors try to read or write to the same memory address. Therefore some sort of protection is utilized. This overhead can degrade any potential parallel speedup.

A message-passing system consists of independent processors each possessing its own private memory with limited memory access. Communication between the processors is relegated to message passing. Here an interconnection scheme is very important. The ideal situation is a "crossbar" interconnection where each processor possesses a direct link to every other processor. However if the goal is to create a system with a large number of processors, such a communication pattern becomes prohibitively complex. Nearest neighbor meshes and generalized hypercube structures [Bhuyan and Agrawal (1984)] are usually implemented so that processors communicate directly with a few neighbors and indirectly through message forwarding.

The interconnection schemes and degrees of system coupling will affect the implementation of numerical algorithms. For example, a fully implicit finite difference algorithm that is efficiently parallelized on a shared memory machine may be difficult to implement on message-passing architectures. [George *et al.* (1987)] Algorithms that work well on serial computers such as the preconditioned conjugate gradient methods for solving sparse systems of linear equations become prohibitively slow on parallel computers because global information is required at each iteration to check for convergence. Consequently local relaxation methods whose convergence rates may be slower can become viable candidates on parallel architectures. [Kuo, Levy and Musicus (1987)]

Thus, the fast computation of large-scale scientific problems requires investigation into algorithmic design, architectural parameters, and the development and application of expert systems to integrate and aid in the interpretation of the results obtained. In this dissertation we concentrate on the study of implementing a set of FFT algorithms onto a loosely coupled parallel system with the hypercube interconnection scheme.

1.3 The Hypercube Architecture

The d-dimensional hypercube is a parallel architecture composed of $P = 2^d$ node processors with local memories and an interconnection scheme where two nodes are joined if and only if their binary node labels differ at exactly one bit. Therefore nodes whose binary labels have Hamming distance exactly one are connected. This setting allows the number of nodes to grow linearly with the interconnection complexity only growing logarithmically. The hypercube connection scheme is a popular one because many other topologies such as meshes, trees, rings, and butterflies can be embedded in the binary d-cube graph.

There are several parameters for evaluating an interconnection scheme for a network of computers. We can represent a computer network as a graph $G = (V, E)$, a set of vertices V and edges E. The vertices represent the node processors. and the edges are defined between nodes where there exists a direct link between them. Each node should have a reasonably small degree, so that each processor has to only handle a few links. The maximum distance (diameter) from one node to another should be small. And there should be a large number of alternative paths from one node to another for fault tolerance. The binary d-cube possesses the following properties that make it attractive.

- Each and every vertex has degree d;
- G is a connected graph of diameter d;
- If A and B are any two nodes in a d-cube, then there are $H(A, B)$ parallel paths of length $H(A, B)$ between the nodes A and B.

Parallel paths denote two paths from A to B such that there are no common nodes on the paths except for A and B. $H(A, B)$ is the Hamming distance between A and B. These and other properties of the hypercube can be found in Saad and Schultz (June, 1985).

Parallel computers configured with the hypercube topology are usually message-passing systems. The system is a multiple-instruction multiple-data (MIMD) machine where communication between concurrent processes occur only in the context of messages rather than the use of shared variables. Each processor has a fast local memory without any shared memory among the processors.

Several commercial machines based on the hypercube architecture have been built. The first hypercube computer, the Cosmic Cube, was constructed in 1983 at the California Institute of Technology. [Seitz (1985)] Since then, vendors such as Ametek, Inc., Floating Point Systems, Intel Scientific Computers, and NCUBE have developed hypercube computers, some with vector processor nodes and some with transputer communication co-processors. [Wiley (1987)]

The Cornell Theory Center's 16 processor Intel iPSC/D4MX is the system used for the results gathered in this dissertation. The hardware consists of a Multibus-based System 286/310 microcomputer (cube manager) and computational units of sixteen high performance microcomputers with extended memory. The cube manager is connected to each node processor by an Ethernet communication link. Each node processor has its own numeric processing unit and local memory. Communication between nodes is achieved entirely by queued message passing. Messages can be sent from any node to any other node. The message system automatically routes messages through the system, forwarding them if necessary. One can think of "virtual" communications channels that connect all the processors.

All of the algorithms in this thesis are coded in FORTRAN, compiled by the Ryan McFarland compiler, and executed under the XENIX 286 release 3.4 Update 1 of the host operating system and the iPSC release 3.1.1 of the node operating system with the Exelan R3.3 networking software. Although the programs are written for the Intel iPSC System 286 hypercube and our timings results are derived from this particular system, other hypercube are discussed. By considering their configurations and properties, recommendations on algorithm characteristics relating to specific properties are given. The current hypercubes by Intel, Ametek, NCUBE and Floating-Point Systems all have slow communication rates relative to computational speed. Therefore the implementations strive to both minimize the amount of necessary communication and also the distance that they have to travel. Some hypercubes are equipped with vector boards on the nodes and therefore algorithms are evaluated with respect to vector length. Other hypercubes, such as the Floating-Point T-Series, can interleave communication and computation simultaneously, making it possible for the programmer to "hide" the communication overhead within the computational workload. In general there is no specific algorithm or method that is the best for all situations. Furthermore the state-of-the-art hypercubes do not come close to supercomputer performance. Therefore these machines are used primarily as research tools in the study of parallel algorithms with consideration of architectural constraints and hardware properties. The absolute running times of the results obtained should not be the criteria for judging the algorithms. Rather, the timings for one method should be considered only as relative to the timings obtained from another method. Models are created for the various algorithms to provide general guidelines concerning communication overhead, vector length, and storage requirements. The timing results are used roughly to check the model and also to identify the proportion of communication overhead relative to computational effort.

1.4 Distributed Multiprocessor Computing Issues

Programming paradigms on message passing multiprocessor systems requires a whole new set of heuristics defining such factors as task allocation, message passing protocols, computational complexity, communication requirements, processor mapping and topology, storage space, and synchronization. Conventional rules of measuring execution time primarily with computational complexity analysis are superseded by new standards encompassing all aspects of distributed concurrent programming techniques. [Adams and Crockett (1984), Gannon and Van Rosendale (1984)] Given the proliferation of new parallel machines spanning a whole spectrum of architectural configurations [Haynes *et al.* (1982)], the task of creating and implementing algorithms to effectively exploit parallel processing capabilities is a complicated endeavor.

Communication is a very important topic in distributed memory machines. A processor can access data in its local memory much faster than data resident in another processor. In fact general hypercubes are designed to compute at a rate that is 10 times faster than the rate of communication. [Wiley (1987)] Recall that each node can communicate directly with its nearest neighbors. Sending a message to a more remote node requires intermediate processors to relay the message. Currently communication between neighboring processors is more efficient than communication between more distant nodes. Hence an important consideration in hypercube programming is to define the problem so as to minimize communication.

Two message passing primitives are used: *sendw* and *recvw*. A processor executes a *sendw* to initiate transmission of a message to another processor. The calling process is blocked until the message has been sent. A processor uses a *recvw* to receive information. The execution of *recvw* causes the calling process to be blocked until the message has arrived. Therefore, if the receiving process invokes *recvw* prior to message availability, execution of the routine is delayed. The processor is then *blocked.*

Processors do not need to be directly connected in order to communicate. The networking software is designed to automatically forward messages from node to node until the message reaches its destination. Hence the particular path that a message will traverse is entirely transparent to the programmer. The invocation of a communication request involves a latency period, or communication start-up time that is an order of magnitude longer than the actual data transfer rate. Moving a vector of length m from one node to a neighbor takes the time

$$\tau + mt_{comm}$$

where τ is the communication start-up or latency and the constant t_{comm} is the elemental transfer time. It is usually the case that $\tau >> t_{comm}$. [Saad and Schultz (Sept. 1985)] Therefore it is much more efficient to send one long message than several short ones. This is especially true on the Intel iPSC.

A measure of how well a procedure is utilizing its resources on a multiprocessor is the ratio between the cumulative active time and the total duration of the computation. A processor is active if it is either doing arithmetic, data loading or rearranging, and actively communicating. It is inactive, or idle, while it is blocked awaiting data that it must receive before it can proceed with further computation. The utilization percentage determines the effectiveness of a particular procedure on the multiprocessor. Several concepts directly affect processor utilization. First of all there is the allocation of tasks to processors. Tasks that can be performed in parallel are ideally suited to multiprocessor applications. However these tasks should be evenly distributed among the processors so that they all have approximately an equal amount of work to do. This is called *load balancing* and is important in that processors with a higher workload may cause other processors to block and wait for critical portions of the total procedure. Processor utilization of the blocked processors decreases and the total time for the whole procedure increases. If workload is approximately equal, blocking time would be reduced along with the total execution time.

Efficient use of the limited local memories of the node processors is another important consideration. Some computations require the data in the memories of more than one processor. Therefore message organization is important to minimize the need for buffering and additional storage space. Overwriting is possible for the straight exchange of two sets of data

$$\mathbf{y} \leftrightarrow \mathbf{x}$$

However, if the data requirements were

$$\begin{aligned} \mathbf{y} &\leftarrow f(\mathbf{x}, \mathbf{y}) \\ \mathbf{x} &\leftarrow f(\mathbf{x}, \mathbf{y}) \end{aligned}$$

overwriting an existing array, say $\mathbf{x}$ by the incoming message $\mathbf{y}$ is clearly an algorithmic mistake. Hence an extra buffer is required to store the incoming message. Therefore, rearrangements in the way of algorithm implementation and data mapping into the processor nodes may be called for.

The mapping of node processors into various topologies such as meshes, rings, trees, etc. is a distributed computing issue as well as an algorithmic abstraction. Problems modeling two- or three- dimensional physical phenomena such as heat transfer on plates or gas dynamics work well on mesh topologies. Other applications which require global broadcast or search algorithms find better matches on tree structures. Much algorithmic implementation work is therefore concerned with finding the best match between topology and data movement and permutation patterns inherent in the algorithm.

Finally it should be noted that the hypercube, being a message-passing computer system, achieves synchronization primarily by the availability and *type* of message being sent. A processor cannot request a message, but must wait for messages that it needs to arrive. Messages are tagged with an integer constant *type* that is used to synchronize the total computational effort. For example, iteration numbers can be used as a message *type* to keep processors using the correct data for the correct iteration. In any case, careful design of algorithms is required to make sure that the entire system does not become deadlocked due to processes waiting for nonexistent messages.

In this dissertation we review all of the above issues and considerations with respect to the implementation of Fast Fourier Transforms.

1.5 Definitions and Matrix Notation

The Kronecker Product notation is a way of expressing block matrices as tensor products (direct products) of lower order matrices. It gives a convenient way of describing matrices which have a certain repeating structure.

Definition 1.5.1 *Kronecker Product: If* $\mathbf{A} \in \mathcal{C}^{p \times q}$ *and* $\mathbf{B} \in \mathcal{C}^{m \times n}$ *then the Kronecker product* $\mathbf{A} \otimes \mathbf{B}$ *is the p-by-q block matrix*

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{00}\mathbf{B} & \cdots & a_{0,q-1}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{p-1,0}\mathbf{B} & \cdots & a_{p-1,q-1}\mathbf{B} \end{bmatrix} \in \mathcal{C}^{pm \times qm}$$

$(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = (\mathbf{AC}) \otimes (\mathbf{BD})$ *if the matrix multiplications* $\mathbf{AC}$ *and* $\mathbf{BD}$ *are defined.*

Under certain circumstances, a commutativity relation is useful and the following lemma is a useful tool to have.

Lemma 1.5.1 *Let* p, q, r, s *be integers with the property that*

i) $pq = rs$

ii) p *divides* r

then if $\mathbf{A} \in \mathcal{C}^{p \times p}$ *and* $\mathbf{B} \in \mathcal{C}^{s \times s}$

$$(\mathbf{A} \otimes \mathbf{I}_q)(\mathbf{I}_r \otimes \mathbf{B}) = (\mathbf{I}_r \otimes \mathbf{B})(\mathbf{A} \otimes \mathbf{I}_q)$$

Proof See Van Loan (1987) Lemma 6.3.

——*——

An easy method for designating submatrices is useful for the description of matrix algorithms or procedures. The MATLAB "colon" method is described here. [Coleman and Van Loan, (1987), Chapter 5]

There are two forms of colon arguments

$$\{\text{starting value}\} : \{\text{terminating value}\}$$

being shorthand for all the indices from the starting value to the terminating value, with an increment of unit value. For example $1 : 5$ designates the indices $\{1, 2, 3, 4, 5\}$. When the increment if any other nonzero integer, either positive or negative, the colon format is

$$\{\text{starting value}\} : \{\text{increment}\} : \{\text{terminating value}\}$$

An example is $1 : 2 : 5$ denoting the indices $\{1, 3, 5\}$, whereas $8 : -3 : 1$ represents the set of indices $\{8, 5, 2\}$.

A submatrix can now be designated by use of the colon notation for row and/or column indices. A couple of examples suffice to illustrate.

$$\mathbf{B} = \mathbf{A}(i:j, p:q) \rightarrow \mathbf{B} = \begin{bmatrix} a_{ip} & \cdots & a_{iq} \\ \vdots & \ddots & \vdots \\ a_{jp} & \cdots & a_{jq} \end{bmatrix}$$

Letting $a^{(k)}$ denote the kth column of a matrix $\mathbf{A}$ the submatrix consisting of the odd columns of $\mathbf{A}$ is denoted as follows:

$$\mathbf{B} = \mathbf{A}(:, 1:2:n) = [a^{(1)}, a^{(3)}, a^{(5)}, \ldots]$$

Finally the *diag* notation if used to designate a diagonal matrix. The argument is usually a vector or an algebraic expression so that

$$\mathbf{D} = diag(\mathbf{x}) \rightarrow \mathbf{D} = \begin{bmatrix} x_0 & & \\ & \ddots & \\ & & x_{n-1} \end{bmatrix}$$

and

$$\mathbf{D} = diag(\alpha^j), \qquad j = 0, \ldots, n-1$$

$$\rightarrow \mathbf{D} = \begin{bmatrix} a^0 & & & \\ & a^1 & & \\ & & \ddots & \\ & & & a^{n-1} \end{bmatrix}$$

1.6 Permutations

A permutation is a function defined on a finite set $\mathbf{X}$ that maps $\mathbf{X}$ *onto* itself in a *one-to-one* manner. Permutations occur in the discussion and modeling of switching networks for loosely coupled distributed computing systems. Data transfers which are one-to-one are actually permutations that map data from one processor to another processor. Permutations are also an important part of the FFT algorithm, manifested in the recursive splitting algorithm that describes and characterizes the FFT.

Permutations can be described in several intuitive ways.

function: If $\pi : \{0, 1, \ldots, n-1\} \rightarrow \{0, 1, \ldots, n-1\}$ is a permutation, we denote it by

$$\begin{pmatrix} 0 & 1 & \cdots & n-1 \\ \pi(0) & \pi(1) & \cdots & \pi(n-1) \end{pmatrix}$$

The set of all available permutations $\mathcal{P}(\mathbf{X})$ forms a group $(\mathcal{P}(\mathbf{X}), \circ)$ under composition. The theory of groups can be applied to their study. [Gilbert (1976)]

base-p function: Let π be a function on a set of integers that can be represented by an arbitrary base p. Some permutations are more conveniently described in this manner. Let

$$\begin{aligned} j &= \{b_{t-1}, b_{t-2}, \ldots, b_0\}_p \\ &= b_{t-1}p^{t-1} + b_{t-2}p^{t-2} + \cdots + b_1 p + b_0, \\ & \quad 0 \leq j, \pi(j) \leq n-1 \end{aligned}$$

represent the base-p address of an element in the set $\mathbf{X}$. Then permutations of the set of inputs can be defined by operations or permutations on the digits (or bits) of the base-p address. [Hockney and Jesshope (1981)] Hence

$$\pi(j) \leftrightarrow f(\{b_{t-1}, b_{t-2}, \ldots, b_0\}_p)$$

with f one-to-one and onto.

matrix: A permutation can also be described as a matrix. Let

$$\mathbf{I}_n = [\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_{n-1}]$$

be the identity matrix of order n and $\mathbf{e}_j$ the jth unit vector $\mathbf{e}_j(k) = 1$, if $k = j$ and 0 otherwise. Then the permutation π can be defined as the matrix

$$\mathbf{P}_n = [\mathbf{e}_{\pi(0)}, \mathbf{e}_{\pi(1)}, \ldots, \mathbf{e}_{\pi(n-1)}]$$

Base-2 Permutations and Kronecker Products

Let π be a permutation defined on a set $\{0, 1, \ldots, n-1\}$ with the indices written in binary. π is defined by its action on $\{b_{t-1}, b_{t-2}, \ldots, b_1, b_0\}$ for each address. Denote its corresponding permutation matrix by $\mathbf{P}_n$. The kth sub-π permutation $\pi_{(k)}$ is defined on the k lowest order bits $\{b_{k-1}, b_{k-2}, \ldots, b_1, b_0\}$ in the same manner as π is on the whole set. The kth super-π permutation $\pi^{(k)}$ is defined on the k highest order bits $\{b_{t-1}, b_{t-2}, \ldots, b_{t-k}\}$ in the same manner as π.

Theorem 1.6.1 *Let $\mathbf{P}_r$, $r = 2^k$ be the permutation matrix defining π over the set $\{0, \ldots, 2^k - 1\}$ and assume that $n = 2^t$, with $k \leq t$. If*

$$\pi \leftrightarrow \mathbf{P}_n$$

then

$$\begin{aligned}\pi_{(k)} &\leftrightarrow \mathbf{I}_{n/r} \otimes \mathbf{P}_r \\ \pi^{(k)} &\leftrightarrow \mathbf{P}_r \otimes \mathbf{I}_{n/r}\end{aligned}$$

Proof The corresponding permutation matrices are:

$$\begin{aligned}\mathbf{P}_n &= [\mathbf{e}_{\pi(0)}, \ldots, \mathbf{e}_{\pi(n-1)}] \\ \mathbf{P}_r &= [\mathbf{e}_{\pi(0)}, \ldots, \mathbf{e}_{\pi(r-1)}] \\ & r = 2^k\end{aligned}$$

$\pi_{(k)}(j)$ leaves the $n-k$ highest order bits of j alone so that

$$\pi_{(k)}(j) = \pi_{(k)}(j \bmod 2^k).$$

Since $\pi_{(k)}$ is a bijection, this implies that $\pi_{(k)}$ operates disjointly on the n/r sets $\{0, \ldots, r-1\}$, $\{r, \ldots, 2r-1\}, \ldots, \{(\frac{n}{r}-1)r, \ldots, n-1\}$. Each corresponds to an independent application of $\mathbf{P}_r$. Hence the definition of the Kronecker product gives

$$\pi_{(k)} \leftrightarrow \mathbf{I}_{n/r} \otimes \mathbf{P}_r$$

Similarly $\pi^{(k)}(j)$ leaves the $n-k$ lowest order bits alone so that

$$\pi^{(k)}(j) = \pi^{(k)}(j/2^k).$$

$\pi^{(k)}$, being a bijection, now operates disjointly on the $\frac{n}{r}$ sets $\{0 : r : (\frac{n}{r}-1)r\}$, $\{1 : r : (\frac{n}{r}-1)r+1\}, \ldots, \{r-1 : r : n-1\}$, corresponding to independent applications of $\mathbf{P}_r$. Therefore the definition of Kronecker Products gives

$$\pi^{(k)} \leftrightarrow \mathbf{P}_r \otimes \mathbf{I}_{n/r}$$

——*——

Theorem 1.6.1 allows us to conveniently discuss permutations both in terms of binary function notation and matrix notation.

Exchange Permutation

The exchange permutation $\mathbf{E}_n$ is described by

$$\mathbf{E}_n = [\mathbf{e}_{n-1}, \mathbf{e}_{n-2}, \ldots, \mathbf{e}_1, \mathbf{e}_0]$$

If $\mathbf{y} = \mathbf{E}_n\mathbf{x}$, then $\mathbf{y}$ consists of the elements of $\mathbf{x}$ flipped upside-down.

The kth sub-exchange permutation is defined over the binary representation of j by flipping b_0 through b_{k-1}, and the kth super-exchange involves flipping all the bits b_{t-k} through b_{t-1}. Let

$$\overline{b}_i = \begin{cases} 0 & \text{if } b_i = 1 \\ 1 & \text{if } b_i = 0 \end{cases}$$

then the kth sub- and super- exchange permutations are described as follows:

$$\epsilon_{(k)}(j) = (b_{t-1}, \ldots, \overline{b}_{k-1}, \overline{b}_{k-2}, \ldots, \overline{b}_1, \overline{b}_0)_2$$

$$\epsilon^{(k)}(j) = (\overline{b}_{t-1}, \ldots, \overline{b}_{t-k}, b_{t-k-1}, \ldots, b_0)_2$$

The exchange permutation can also be defined in terms of the Binary Reflected Gray Code representation of j. (See Section 1.9).

$$\epsilon_{(k)}(j) = (g_{t-1}, \ldots, \overline{g}_{k-1}, \ldots, g_0)_2$$

where $G_t(j) = (g_{t-1}, \ldots, g_0)_2$ (with $n = 2^t$) and the bar denoting the complement of a given bit. The kth exchange permutation is simply the complementing of the $k-1$th bit of the BRGC representation of j. In matrix terms,

$$\epsilon_{(k)} \leftrightarrow (\mathbf{I}_{n/r} \otimes \mathbf{E}_r), \quad r = 2^k$$

Butterfly Permutation

The butterfly permutation $\beta(j)$ is defined over the binary representation of an integer j by exchanging the first and last bits:

$$\beta(j) = (b_0, b_{t-2}, \ldots, b_1, b_{t-1})_2$$

with $j = (b_{t-1}, b_{t-2}, \ldots, b_1, b_0)_2$ The kth sub-butterfly $\beta_{(k)}(j)$ exchanges b_{k-1} and b_0, whereas the kth super-butterfly exchange b_{t-1} with b_{t-k}:

$$\begin{aligned} \beta_{(k)}(j) &= \{b_{t-1}, \ldots, b_k, b_0, b_{k-2}, \ldots, b_1, b_{k-1}\} \\ \beta^{(k)}(j) &= \{b_{t-k}, b_{t-2}, \ldots, b_{t-k+1}, b_{t-1}, b_{t-k-1}, \ldots, b_0\} \end{aligned}$$

The butterfly permutation is not conveniently described by matrix notation, however we shall designate the matrix $\mathbf{B}_n \leftrightarrow \beta(j)$.

Perfect Shuffle and Inverse Shuffle Permutations

The base-p perfect shuffle $\sigma(j)$ corresponds to a circular left shift of the t-digit base-p integer which represents the address j.

$$\sigma(j) = \{b_{t-2}, b_{t-3}, \ldots, b_1, b_0, b_{t-1}\}$$

The kth sub-shuffle $\sigma_{(k)}$ and the kth super-shuffle $\sigma^{(k)}$, are defined by cyclic left shifts on the least and most significant k bits respectively.

$$\begin{aligned}
\sigma_{(k)}(j) &= \{b_{t-1}, \ldots, b_k, b_{k-2}, \ldots, b_0, b_{k-1}\} \\
\sigma^{(k)}(j) &= \{b_{t-2}, \ldots, b_{t-k}, b_{t-1}, b_{t-k-1}, \ldots, b_0\}
\end{aligned}$$

The base-p inverse perfect shuffle $\mu(j)$ is defined as a cyclic right shift of the t-digit base-p integer which represents j.

$$\mu(j) = \{b_0, b_{t-1}, \ldots, b_1\}$$

The kth sub-inverse shuffle $\mu_{(k)}$ and kth super-inverse shuffle are analogously defined.

$$\begin{aligned}
\mu_{(k)}(j) &= \{b_{t-1}, \ldots, b_k, b_0, b_{k-1}, \ldots, b_1\} \\
\mu^{(k)}(j) &= \{b_{t-k}, b_{t-1}, \ldots, b_{t-k+1}, b_{t-k-1}, \ldots, b_0\}
\end{aligned}$$

If the perfect-shuffle permutation, and likewise the inverse-perfect shuffle were applied t times, the original bit-sequence will be restored.

Theorem 1.6.2 *The perfect shuffle operator*

$$\sigma : \{0, 1, \ldots, n-1\} \rightarrow \{0, 1, \ldots, n-1\}, \quad n = p^t$$

and the inverse perfect shuffle operator

$$\mu : \{0, 1, \ldots, n-1\} \rightarrow \{0, 1, \ldots, n-1\}, \quad n = p^t$$

each form cyclic groups of order t under composition.

Now we define the matrix notation for perfect shuffle and inverse perfect shuffle operators. Define the base-p perfect shuffle matrix $\mathbf{\Pi}_n^{(p)}$ by its effect on a vector if indices $(0 : n-1)^T$. If $n = pm$, then

$$\mathbf{\Pi}_n^{(p)}(0 : n-1)^T = (0 : m : n-1, 1 : m : n-1, \ldots, m-1 : m : n-1)^T$$

Example

$$\mathbf{\Pi}_8^{(2)}\begin{bmatrix}0\\1\\2\\3\\4\\5\\6\\7\end{bmatrix}=\begin{bmatrix}0\\4\\1\\5\\2\\6\\3\\7\end{bmatrix}$$

The base-p inverse perfect shuffle

$$\mathbf{M}_n^{(p)}=[\mathbf{\Pi}_n^{(p)}]^{-1}=[\mathbf{\Pi}_n^{(p)}]^T$$

sorts the indices $(0:n-1)^T$ by

$$\mathbf{M}_n^{(p)}(0:n-1)^T=(0:p:n-1,1:p:n-1,\ldots,p-1:p:n-1)^T$$

Example

$$\mathbf{M}_8^{(2)}\begin{bmatrix}0\\1\\2\\3\\4\\5\\6\\7\end{bmatrix}=\begin{bmatrix}0\\2\\4\\6\\1\\3\\5\\7\end{bmatrix}$$

Application of Theorem 1.6.1 for the base-2 perfect shuffle and inverse perfect shuffle permutation gives following correspondences,

$$\begin{aligned}\sigma(j)&\leftrightarrow\mathbf{\Pi}_n^{(2)}\\ \mu(j)&\leftrightarrow\mathbf{M}_n^{(2)}\\ \sigma_{(k)}(j)&\leftrightarrow\mathbf{I}_{n/r}\otimes\mathbf{\Pi}_r^{(2)}\\ \sigma^{(k)}(j)&\leftrightarrow\mathbf{\Pi}_r^{(2)}\otimes\mathbf{I}_{n/r}\\ \mu_{(k)}(j)&\leftrightarrow\mathbf{I}_{n/r}\otimes\mathbf{M}_r^{(2)}\\ \mu^{(k)}(j)&\leftrightarrow\mathbf{M}_r^{(2)}\otimes\mathbf{I}_{n/r}\\ &\quad\;0\le j\le n-1\\ &\quad\;n=2^t\\ &\quad\;r=2^k\end{aligned}$$

The following theorem allows us to change the order of Kronecker Products.

Theorem 1.6.3

$$\mathbf{M}_n^{(p)}(\mathbf{A} \otimes \mathbf{I}_p)\mathbf{\Pi}_n^{(p)} = \mathbf{I}_p \otimes \mathbf{A}$$

for $\mathbf{A}$ *an arbitrary square matrix of dimension* $m = n/p$.

Proof Let

$$\mathbf{B} = (\mathbf{A} \otimes \mathbf{I}_p) = [a_{ij}\mathbf{I}_p], \quad i,j = 0,\ldots,m$$

Premultiplication by $\mathbf{M}_n^{(p)}$ picks out every pth row of $(\mathbf{A} \otimes \mathbf{I}_p)$ and post-multiplication by $\mathbf{\Pi}_n^{(p)}$ picks out every pth column. Hence $\mathbf{B}(i+k:p:n-1, j+k:p:n-1) = \mathbf{A}$ for $k = 0:p-1$. Thus $[\mathbf{M}_n^{(p)}\mathbf{B}\mathbf{\Pi}_n^{(p)}](kp+i, kp+j) = \mathbf{B}(i+k:p:n-1, j+k:p:n-1)$, $k = 0:p-1$, which is exactly $(\mathbf{I}_p \otimes \mathbf{A})$.

——*——

Corollary 1.6.1

$$\mathbf{\Pi}_n^{(p)}(\mathbf{I}_p \otimes \mathbf{A})\mathbf{M}_n^{(p)} = \mathbf{A} \otimes \mathbf{I}_p$$

Theorem 1.6.4 *The powers of* $\mathbf{\Pi}_n^{(p)}$ *form a cyclic group* (mod t) *for* $n = p^t$ *on the product of matrix multiplication.*

Theorem 1.6.5 *The powers of* $\mathbf{M}_n^{(p)}$ *form a cyclic group* (mod t) *for* $n = p^t$ *on the product of matrix multiplication.*

The following theorem relates the base-2 inverse perfect shuffle permutation to the butterfly permutation:

Theorem 1.6.6 *Let* $\mathbf{B}_n$ *denote the butterfly permutation matrix. Then*

$$\mathbf{\Pi}_n^{(2)} = \prod_{i=t:-1:2} (\mathbf{I}_{n/r} \otimes \mathbf{B}_r), \quad r = 2^i$$

and

$$\mathbf{M}_n^{(2)} = \prod_{i=t:-1:2} (\mathbf{B}_r \otimes \mathbf{I}_{n/r}), \quad r = 2^i$$

Proof Using theorem 1.6.1 we can translate the permutation matrices into sub- and super- butterflies and use induction. First we easily see that $\sigma_{(1)} = \beta_{(1)} = \mu_{(1)} = i$ the identity. We can also easily check that $\sigma_{(2)} = \beta_{(2)}\beta_{(1)}$ and $\mu_{(2)} = \beta^{(2)}\beta^{(1)}$. (Notice here that the permutations are applied from right to left, just like in matrix multiplication). Let the induction hypothesis suppose that $\sigma_{(k)} = \beta_{(k)}\cdots\beta_{(1)}$ is true. Looking at

$$\sigma_{(k)} = b_{t-1}\ldots b_k b_{k-2}\ldots b_0 b_{k-1}$$

and

$$\sigma_{(k+1)} = b_{t-1} \dots b_{k-1} b_{k-2} \dots b_0 b_k$$

we see that $\sigma_{(k+1)} = \beta_{(k+1)}\sigma_{(k)}$. Similarly for $\mu_{(k)} = \beta^{(k)} \dots \beta^{(1)}$ we see by comparing

$$\mu_{(k)} = b_{t-k} b_{t-1} \dots b_{t-k+1} b_{t-k-1} \dots b_0$$

and

$$\mu_{(k+1)} = b_{t-k-1} b_{t-1} \dots b_{t-k+1} b_{t-k} \dots b_0$$

we see that $\mu_{(k+1)} = \beta^{(k+1)}\mu_{(k)}$

——*——

Bit-Reversal Permutation

The bit-reversal (or digit-reversal) permutation is naturally described by its action on the bits of an index address. Let $\{b_{t-1}, \dots, b_1, b_0\}$ be the base-p representation of j, then $\rho(j) = \{b_0, b_1, \dots, b_{t-1}\}$ is a reversal of the ordering of the bits of j. As with the previous permutations, the kth sub bit-reversal and the kth super bit-reversal can also be defined over the least and most significant k bits of j:

$$\begin{aligned} \rho_{(k)}(j) &= \{b_{t-1}, \dots, b_k, b_0, b_1, \dots, b_{k-1}\} \\ \rho^{(k)}(j) &= \{b_{t-k}, b_{t-k+1}, \dots, b_{t-1}, b_{t-k-1}, \dots, b_0\} \end{aligned}$$

A base p bit-reversal permutation $\mathbf{y} = \mathbf{P}_n^{(p)}\mathbf{x}$ is defined by numbering the indices of $\mathbf{x}$ in base p notation and then reversing the digits of each index. The vector $\mathbf{y}$ is $\mathbf{x}$ reordered according to this recipe.

Example (n=8, r=2):

$$\mathbf{P}_8^{(2)} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \\ 2 \\ 6 \\ 1 \\ 5 \\ 3 \\ 7 \end{bmatrix}$$

Lemma 1.6.1

$$\mathbf{P}_n^{(p)}\mathbf{P}_n^{(p)} = \mathbf{I}_n \quad or \quad [\mathbf{P}_n^{(p)}]^{-1} = \mathbf{P}_n^{(p)}$$

Proof Reversing the order of the base p digits of an index twice gives back the original index.

——*——

Corollary 1.6.2

$$[\mathbf{P}_n^{(p)}]^T = \mathbf{P}_n^{(p)}$$

There are several ways that $\mathbf{P}_n^{(p)}$ can be factored in terms of Kronecker Products and the Shuffle and Inverse Shuffle matrices [Sloate (1974)]. In the following we assume that $n = p^t$.

Theorem 1.6.7

$$\mathbf{P}_n^{(p)} = \mathbf{\Pi}_n^{(p)}(\mathbf{I}_p \otimes \mathbf{P}_{n/p}^{(p)}) \tag{1.6-1}$$

Proof Translating this into bit function notation we basically have to show that

$$\rho_{(k+1)} = \sigma_{(k+1)}\rho_{(k)}$$

This can be easily seen by comparing

$$\rho_{(k)}(j) = b_{t-1}\ldots b_k b_{k-1}\ldots b_1 b_0$$

and

$$\rho_{(k+1)}(j) = b_{t-1}\ldots b_k b_0 b_1 \ldots b_{k-1}$$

——*——

Lemma 1.6.2

$$\mathbf{P}_n^{(p)} = \mathbf{\Pi}_n^{(p)}(\mathbf{I}_p \otimes \mathbf{\Pi}_{n/p}^{(p)})(\mathbf{I}_{p^2} \otimes \mathbf{\Pi}_{n/p^2}^{(p)})\cdots(\mathbf{I}_{n/p^2} \otimes \mathbf{\Pi}_{p^2}^{(p)}) \tag{1.6-2}$$

Proof Recursively plugging in 1.6-1 of Theorem 1.6.7.

——*——

Lemma 1.6.3

$$\mathbf{P}_n^{(p)} = (\mathbf{\Pi}_{p^2}^{(p)} \otimes \mathbf{I}_{n/p^2})(\mathbf{\Pi}_{p^3}^{(p)} \otimes \mathbf{I}_{n/p^3})\cdots(\mathbf{\Pi}_{n/p}^{(p)} \otimes \mathbf{I}_p)\mathbf{\Pi}_n^{(p)} \tag{1.6-3}$$

Proof Lemma 1.6.2 and Theorem 1.6.3.

——*——

This is the sorting scheme described by Singleton (1967) for $p = 2$. Let $\mathbf{M}_k^{(p)} = [\mathbf{\Pi}_k^{(p)}]^{-1}$.

Lemma 1.6.4

$$\mathbf{P}_n^{(p)} = (\mathbf{I}_{n/p^2} \otimes \mathbf{M}_{p^2}^{(p)})(\mathbf{I}_{n/p^3} \otimes \mathbf{M}_{p^3}^{(p)}) \cdots (\mathbf{I}_p \otimes \mathbf{M}_{n/p}^{(p)})\mathbf{M}_n^{(p)} \tag{1.6-4}$$

Proof Taking the transpose of 1.6-2.

——*——

Lemma 1.6.5

$$\mathbf{P}_n^{(p)} = \mathbf{M}_n^{(p)}(\mathbf{M}_{n/p}^{(p)} \otimes \mathbf{I}_p) \cdots (\mathbf{M}_{p^3}^{(p)} \otimes \mathbf{I}_{n/p^3})(\mathbf{M}_{p^2}^{(p)} \otimes \mathbf{I}_{n/p^2}) \tag{1.6-5}$$

Proof Taking the transpose of 1.6-3.

——*——

Flip Permutation

The kth flip permutation is defined by flipping the $(k-1)$th bit of the binary representation of j.

$$\phi_{(k)}(j) = \{b_{n-1}, \ldots, \bar{b}_{k-1}, \ldots, b_0\}$$

where the bar denotes the complementation of a given bit. This permutation corresponds to the data-transfer pattern of the radix-2 in-place FFT algorithms.

Shift Permutations

The forward shift permutation takes

$$\{0, 1, \ldots, n-1\} \rightarrow \{1, 2, \ldots, n-1, 0\}$$

and is represented by the permutation matrix

$$\mathbf{R}_n = [\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_{n-1}, \mathbf{e}_0].$$

The backward shift permutation is

$$\mathbf{R}_n^T = [\mathbf{e}_{n-1}, \mathbf{e}_0, \mathbf{e}_1, \ldots, \mathbf{e}_{n-2}].$$

Reflection Permutation

The reflection permutation matrix $\mathbf{T}_n$ is defined as follows

$$\mathbf{T}_n = \mathbf{E}_n \mathbf{R}_n^T.$$

It can be shown that

$$\mathbf{T}_n = \begin{bmatrix} 1 & 0 \\ 0 & \mathbf{E}_{n-1} \end{bmatrix}.$$

Let n be a positive integer, then

$$\tau(j) = n - j \bmod n$$

This permutation occurs in the discussion of symmetric FFT's.

1.7 The Fast Fourier Transform

The discrete Fourier Transform (DFT) of a complex vector

$$(x_0, x_1, \ldots, x_{n-1})^T$$

is defined by

$$y(k) = \frac{1}{n} \sum_{j=0}^{n-1} x_j \exp(-i\frac{2\pi jk}{n}), \quad k = 0, 1, \ldots, n-1.$$

This expression describes the computation of n equations. Defining

$$\omega_n = \exp(-i\frac{2\pi}{n}),$$

we can compose the n-by-n matrix $\mathbf{F}_n$ by setting the j,k element of $\mathbf{F}_n$ to be $\omega^{jk \bmod n}$. The discrete Fourier transform can then be written in the form of the following matrix-vector multiplication:

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix} = \frac{1}{n} \begin{bmatrix} \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^0 & \omega^1 & \cdots & \omega^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{n-1} & \cdots & \omega^1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

or more compactly as

$$\mathbf{y} = \frac{1}{n} \mathbf{F}_n \mathbf{x}.$$

The inverse discrete Fourier Transform (IDFT) is defined as

$$\mathbf{x} = \mathbf{F}_n^H \mathbf{y}$$

where $\mathbf{F}_n^H$ is the Hermitian transpose of $\mathbf{F}_n$.

Direct computation of the DFT by matrix-vector multiplication requires $O(n^2)$ operations. The FFT is a "divide-and-conquer" algorithm that requires only $O(n \log_p n)$ operations ($n = p^t$). This is because the DFT matrix $\mathbf{F}_n$ can be factored into a product of $O(t)$ sparse matrices consisting of t diagonal matrices requiring multiplications and a host of permutation matrices. The FFT algorithm is commonly explained in terms of finite sums and signal flow graphs. [Bergland (1969), Bertram (1970), Brigham and Morrow (1967), Cochran *et al.* (1967), Cooley and Tukey (1965), Cooley, Lewis and Welch (1969a, 1969b), Gentleman and Sande (1966)]

1.8 Matrix Description of the FFT

Since the FFT is a sequence of sparse matrix-vector multiplications, it is more convenient to express it in terms of matrices. Several different methods of representations have been described by Corinthios (1971), Drubin (1971), Glassman (1970), Kahaner (1970), Pease (1968), and Theilheimer (1969) for different versions of the FFT. Sloate (1974) uses Kronecker product notation and the perfect shuffle base p permutation operator $\mathbf{\Pi}_n^{(p)}$ to provide a unifying approach to expression FFT's in terms of matrix factors. He also generates eight different algorithms used for the sorting of FFT data. Van Loan (1987) provides an in depth derivation of the matrix factorization that leads to the Cooley-Tukey radix-two algorithm (CT2) described by Swarztrauber (1982). We shall use Sloate's approach to generalize the derivation of four canonical in-place FFT algorithms for general radix p. Afterwards we will describe the matrix factorization for the four radix-two canonical in-place FFT algorithms summarized on p. 181 of Brigham (1974).

General Radix FFT Factorization

Let $n = p^t$, $\omega_n = \exp(-i\frac{2\pi}{n})$. Define a diagonal matrix of weights

Definition 1.8.1

$$\mathbf{\Delta}_{n/pk} = diag(1, \omega_{n/k}, \ldots, \omega_{n/k}^{n/pk-1}) \quad k = p^j, j = 0, \ldots, t-2$$

and a block diagonal matrix of weights,

Definition 1.8.2

$$\mathbf{D}^{(p)}_{n/k} = diag(\mathbf{I}_{n/pk}, \mathbf{\Delta}_{n/pk}, \mathbf{\Delta}^2_{n/pk}, \ldots, \mathbf{\Delta}^{p-1}_{n/pk})$$

The basic splitting equation for the decimation in time FFT is introduced in the following theorem.

Theorem 1.8.1

$$\mathbf{F}_n \mathbf{\Pi}^{(p)}_n = (\mathbf{F}_p \otimes \mathbf{I}_{n/p})\mathbf{D}^{(p)}_n(\mathbf{I}_p \otimes \mathbf{F}_{n/p}) \tag{1.8-1}$$

Proof Theorem 3.4 in Van Loan [1987].

———*———

Corollary 1.8.1

$$\begin{aligned} \mathbf{F}_{n/k}\mathbf{\Pi}^{(p)}_{n/k} &= (\mathbf{F}_p \otimes \mathbf{I}_{n/pk})\mathbf{D}^{(p)}_{n/k}(\mathbf{I}_p \otimes \mathbf{F}_{n/pk}) \\ & \quad k = p^j, j = 0, 1, \ldots, t-2 \end{aligned} \tag{1.8-2}$$

Equations 1.8-1 and 1.8-2 can be applied repeatedly to obtain the radix-p decimation in time algorithm that takes data in digit-reversed order and utilizes the trigonometric weights in natural order. The use of 1.6-2 combines the shuffle matrices applied during the recursion into the digit-reversal matrix $\mathbf{P}^{(p)}_n$.

Definition 1.8.3 *Decimation in Time (DIT2): If $n = p^t$, then*

$$\mathbf{F}_n\mathbf{P}^{(p)}_n = \mathbf{A}_t \cdots \mathbf{A}_1$$

letting $L = p^q$,

$$\mathbf{A}_q = (\mathbf{I}_{n/L} \otimes (\mathbf{F}_p \otimes \mathbf{I}_{L/p}))(\mathbf{I}_{n/L} \otimes \mathbf{D}^{(p)}_L)$$

The radix-2 Cooley-Tukey algorithm (CT2) is gotten by replacing p by 2.

The transpose of Theorem 1.8.1 gives,

Theorem 1.8.2

$$\mathbf{M}^{(p)}_n\mathbf{F}_n = (\mathbf{I}_p \otimes \mathbf{F}_{n/p})\mathbf{D}^{(p)}_n(\mathbf{F}_p \otimes \mathbf{I}_{n/p}) \tag{1.8-3}$$

Corollary 1.8.2

$$\begin{aligned} \mathbf{F}_{n/k}\mathbf{\Pi}^{(p)}_{n/k} &= (\mathbf{F}_p \otimes \mathbf{I}_{n/pk})\mathbf{D}^{(p)}_{n/k}(\mathbf{I}_p \otimes \mathbf{F}_{n/pk}) \\ & \quad k = p^j, j = 0, 1, \ldots, t-2 \end{aligned} \tag{1.8-4}$$

The corresponding decimation in frequency radix-p FFT algorithm that post-sorts the data and uses trigonometric weights in natural order can be derived from recursing on equations 1.8-3 and 1.8-4 and using equation 1.6-4 to combine the inverse shuffle matrices into $\mathbf{P}_n^{(p)}$.

Definition 1.8.4 *Decimation in Frequency (DIF1): If $n = p^t$, then*

$$\mathbf{P}_n^{(p)}\mathbf{F}_n = \mathbf{A}_1^T \cdots \mathbf{A}_t^T$$

where letting $L = 2^q$,

$$\mathbf{A}_q^T = (\mathbf{I}_{n/L} \otimes \mathbf{D}_L^{(p)})(\mathbf{I}_{n/L} \otimes (\mathbf{F}_p \otimes \mathbf{I}_{L/p}))$$

By taking $p = 2$, we get the Gentleman-Sande (GS1) algorithm.

There are two more in-place algorithms that require the trigonometric weights in digit-reversed order, DIT1 and DIF2. The radix-2 DIT1 algorithm is the original Cooley-Tukey algorithm (CT1) where the data is input in natural order, trigonometric weights are used in bit-reversed order and post-sorting of data is required. The DIF2 radix-2 algorithm is the Gentleman-Sande algorithm that requires pre-sorting of data with bit-reversed ordered weights. Each pair of algorithms (DIF1 and DIT2) and (DIT1 and DIF2) can be used together to perform a forward transform followed by an inverse transform without the necessity of bit-reversal of data items.

The following theorem allows us to establish the factorization for the two canonical forms in which the multipliers are applied in digit-reversed order.

Theorem 1.8.3 (Sloate (1974))

$$\begin{aligned}\mathbf{F}_{n/k} &= \mathbf{M}_{n/k}^{(p)}(\mathbf{I}_{n/pk} \otimes \mathbf{F}_p)(\boldsymbol{\Pi}_{n/k}^{(p)}\mathbf{D}_{n/k}^{(p)}\mathbf{M}_{n/k}^{(p)})(\mathbf{F}_{n/pk} \otimes \mathbf{I}_p) \\ & \quad k = p^j, \quad j = 0, 1, \ldots, t-2\end{aligned}$$

Proof Starting with equation 1.8-2 and using theorem 1.6.3 we can commute the next to the last term. Hence,

$$\begin{aligned}\mathbf{F}_{n/k} &= (\mathbf{F}_p \otimes \mathbf{I}_{n/pk})\mathbf{D}_{n/k}^{(p)}(\mathbf{I}_p \otimes \mathbf{F}_{n/pk})\mathbf{M}_{n/k}^{(p)} \\ &= (\mathbf{F}_p \otimes \mathbf{I}_{n/pk})\mathbf{D}_{n/k}^{(p)}\mathbf{M}_{n/k}^{(p)}(\mathbf{F}_{n/pk} \otimes \mathbf{I}_p)\end{aligned}$$

Next we commute the term $(\mathbf{F}_p \otimes \mathbf{I}_{n/pk})$. Let $v = \frac{n}{pk}$ and $v = p^{\gamma-1}$ since $\frac{n}{k} = p^{\gamma}$. Then

$$(\mathbf{F}_p \otimes \mathbf{I}_v) = \boldsymbol{\Pi}_{n/k}^{(v)}(\mathbf{I}_v \otimes \mathbf{F}_p)\mathbf{M}_{n/k}^{(v)}$$

The $\mathbf{\Pi}^{(p)}_{n/k}$ forms a group (mod γ) under matrix multiplication, so that

$$\begin{aligned}(\mathbf{F}_p \otimes \mathbf{I}_{n/pk}) &= [\mathbf{\Pi}^{(p)}_{n/k}]^{\gamma-1}(\mathbf{I}_{n/pk} \otimes \mathbf{F}_p)[\mathbf{M}^{(p)}_{n/k}]^{\gamma-1} \\ &= \mathbf{M}^{(p)}_{n/k}(\mathbf{I}_{n/pk} \otimes \mathbf{F}_p)\mathbf{\Pi}^{(p)}_{n/k}\end{aligned}$$

——*——

Next we use equations 1.6-2, 1.6-3, 1.6-4, 1.6-5, theorem 1.8.3, and lemma 1.5.1 to recursively develop a factorization for the DIT1 FFT ($\mathbf{P}^{(p)}_n\mathbf{F}_n$). We show the first step in detail. Initially we have

$$\mathbf{\Pi}^{(p)}_n\mathbf{F}_n = (\mathbf{I}_{n/p} \otimes \mathbf{F}_p)(\mathbf{\Pi}^{(p)}_n\mathbf{D}^{(p)}_n\mathbf{M}^{(p)}_n)(\mathbf{F}_{n/p} \otimes \mathbf{I}_p)$$

Now we apply $\mathbf{\Pi}^{(p)}_{n/p} \otimes \mathbf{I}_p$ to the left of the equation and recurse on $\mathbf{F}_{n/p}$.

$$\begin{aligned}&(\mathbf{\Pi}^{(p)}_{n/p} \otimes \mathbf{I}_p)\mathbf{\Pi}^{(p)}_n\mathbf{F}_n = \\ &\quad(\mathbf{\Pi}^{(p)}_{n/p} \otimes \mathbf{I}_p)(\mathbf{I}_{n/p} \otimes \mathbf{F}_p)\mathbf{\Pi}^{(p)}_n\mathbf{D}^{(p)}_n\mathbf{M}^{(p)}_n(\mathbf{M}^{(p)}_{n/p} \otimes \mathbf{I}_p) \\ &\quad(\mathbf{I}_{n/p^2} \otimes \mathbf{F}_p \otimes \mathbf{I}_p)(\mathbf{\Pi}^{(p)}_{n/p} \otimes \mathbf{I}_p)(\mathbf{D}^{(p)}_{n/p} \otimes \mathbf{I}_p)(\mathbf{M}^{(p)}_{n/p} \otimes \mathbf{I}_p)(\mathbf{F}_{n/p^2} \otimes \mathbf{I}_{p^2})\end{aligned}$$

Using lemma 1.5.1, we can push the permutation $(\mathbf{\Pi}^{(p)}_{n/p} \otimes \mathbf{I}_p)$ past the term $(\mathbf{I}_{n/p} \otimes \mathbf{F}_p)$.

$$\begin{aligned}&= (\mathbf{I}_{n/p} \otimes \mathbf{F}_p)[(\mathbf{\Pi}^{(p)}_{n/p} \otimes \mathbf{I}_p)\mathbf{\Pi}^{(p)}_n\mathbf{D}^{(p)}_n\mathbf{M}^{(p)}_n(\mathbf{M}^{(p)}_{n/p} \otimes \mathbf{I}_p)] \\ &\quad(\mathbf{I}_{n/p^2} \otimes \mathbf{F}_p \otimes \mathbf{I}_p)[(\mathbf{\Pi}^{(p)}_{n/p}\mathbf{D}^{(p)}_{n/p}\mathbf{M}^{(p)}_{n/p}) \otimes \mathbf{I}_p](\mathbf{F}_{n/p^2} \otimes \mathbf{I}_{n/p^2})\end{aligned}$$

Next theorem 1.6.3 allows us to commute the diagonal portion,

$$[(\mathbf{\Pi}^{(p)}_{n/p}\mathbf{D}^{(p)}_{n/p}\mathbf{M}^{(p)}_{n/p}) \otimes \mathbf{I}_p] = \mathbf{\Pi}^{(p)}_n[\mathbf{I}_p \otimes \mathbf{\Pi}^{(p)}_{n/p}\mathbf{D}^{(p)}_{n/p}\mathbf{M}^{(p)}_{n/p}]\mathbf{M}^{(p)}_n$$

Continuing in this manner we get obtain a factorization for the DIT1 algorithm.

Definition 1.8.5 *Decimation in Time (DIT1): If $n = p^t$, then*

$$\mathbf{P}^{(p)}_n\mathbf{F}_n = \mathbf{C}_t \cdots \mathbf{C}_1$$

Let $L = p^q$,

$$\mathbf{C}_q = (\mathbf{I}_{L/p} \otimes \mathbf{F}_p \otimes \mathbf{I}_{n/L})[\mathbf{P}^{(p)}_n(\mathbf{I}_{n/L} \otimes \mathbf{D}^{(p)}_L)\mathbf{P}^{(p)}_n]$$

The radix-2 Cooley-Tukey algorithm (CT1) comes from $p = 2$.

The analogous Decimation in Frequency algorithm (DIF2) is now easy to come by. We simply transpose the DIT1 factorization.

Definition 1.8.6 *Decimation in Frequency (DIF2): If $n = p^t$, then*

$$\mathbf{F}_n \mathbf{P}_n^{(p)} = \mathbf{C}_1^T \cdots \mathbf{C}_t^T$$

where letting $L = p^q$,

$$\mathbf{C}_q^T = [\mathbf{P}_n^{(p)}(\mathbf{I}_{n/L} \otimes \mathbf{D}_L^{(p)})\mathbf{P}_n^{(p)}](\mathbf{I}_{L/p} \otimes \mathbf{F}_p \otimes \mathbf{I}_{n/L})$$

The radix-2 Gentleman-Sande algorithm (GS2) is the special case $p = 2$.

We next discuss some algorithmic details of the four in-place canonic forms of the FFT, including stride length, multiplier application, and sorting. Notice that at each step there is a data-permutation matrix and a coefficient matrix. The coefficient matrix is a direct sum of matrices $\mathbf{D}_L^{(p)}$. Recall that

$$\mathbf{D}_L^{(p)} = \text{diag}(\mathbf{I}_{L/p}, \mathbf{\Delta}_{L/p}, \mathbf{\Delta}_{L/p}^2, \ldots, \mathbf{\Delta}_{L/p}^{p-1})$$

and

$$\mathbf{\Delta}_{L/p} = \text{diag}(1, \omega_L, \ldots, \omega_L^{L/p-1}).$$

Following are the algorithms for each canonic form and a listing of certain properties.

Algorithm 1.8.1 DIT1:

```
/* compute $\mathbf{x} \leftarrow \mathbf{P}_n^{(p)}\mathbf{F}_n\mathbf{x}$ */
  for $k = 1 : t$
    $q \leftarrow k,\ L \leftarrow p^q$
    $\mathbf{x} \leftarrow (\mathbf{I}_{L/p} \otimes \mathbf{F}_p \otimes \mathbf{I}_{n/L})[\mathbf{P}_n^{(p)}(\mathbf{I}_{n/L} \otimes \mathbf{D}_L^{(p)})\mathbf{P}_n^{(p)}]\mathbf{x}$
  end
```

- post-sorting of data;
- multipliers applied in digit-reversed order;
- at step k in the algorithm:
  - stride length $n/L = p^{t-k}$;
  - powers of $\omega_L = \omega_n^{n/L} = \omega_n^r$, where $r = p^{t-k}$.

Algorithm 1.8.2 DIT2:

```
/* compute $x \leftarrow \mathbf{F}_n \mathbf{P}_n^{(p)} \mathbf{x}$ */
  for $k = 1 : t$
    $q \leftarrow k,\ L \leftarrow p^q$
    $\mathbf{x} \leftarrow (\mathbf{I}_{n/L} \otimes \mathbf{F}_p \otimes \mathbf{I}_{L/p})(\mathbf{I}_{n/L} \otimes \mathbf{D}_L^{(p)})\mathbf{x}$
  end
```

- pre-sorting of data;
- multipliers applied in the natural order;
- at step k in the algorithm:
  - stride length $L/p = p^{k-1}$;
  - powers of $\omega_L = \omega_n^{n/L} = \omega_n^r$ where $r = p^{t-k}$;
  - multiplier applied to x_j is $\omega_L^{j \bmod L/p}$.

Algorithm 1.8.3 DIF1:

```
/* compute $x \leftarrow \mathbf{P}_n^{(p)} \mathbf{F}_n \mathbf{x}$ */
  for $k = 1 : t$
    $q \leftarrow (t - k + 1);\ L \leftarrow p^q$
    $\mathbf{x} \leftarrow (\mathbf{I}_{n/L} \otimes \mathbf{D}_L^{(p)})(\mathbf{I}_{n/L} \otimes \mathbf{F}_p \otimes \mathbf{I}_{L/p})\mathbf{x}$
  end
```

- post-sorting of data;
- multipliers applied in the natural order;
- at step k in the algorithm:
  - stride length $L/p = p^{t-k}$;
  - powers of $\omega_L = \omega_n^{n/L} = \omega_n^r$ where $r = p^{k-1}$;
  - multiplier applied to x_j is $\omega_L^{j \bmod L/p}$.

Algorithm 1.8.4 DIF2:

```
/* compute $\mathbf{x} \leftarrow \mathbf{F}_n \mathbf{P}_n^{(p)} \mathbf{x}$ */
  for $k = 1 : t$
    $q \leftarrow (t - k + 1)$; $L = p^q$
    $\mathbf{x} \leftarrow [\mathbf{P}_n^{(p)}(\mathbf{I}_{n/L} \otimes \mathbf{D}_L^{(p)})\mathbf{P}_n^{(p)}](\mathbf{I}_{L/p} \otimes \mathbf{F}_p \otimes \mathbf{I}_{n/L})\mathbf{x}$
  end
```

- pre-sorting of data;
- multipliers applied in digit-reversed order;
- at step k in the algorithm:
  - stride length $n/L = p^{k-1}$;
  - powers of $\omega_L = \omega_n^{n/L} = \omega_n^r$ where $r = p^{k-1}$

The version DIT2 has the property that the partial transforms which are produced at the intermediate stages possess the same properties as the full transform. [Briggs (1987)] This property makes the DIT2 FFT the most convenient to describe.

1.9 The Binary Reflected Gray Code

The Binary Reflected Gray Code (BRGC) can be defined recursively as follows:

$$\begin{aligned} G_1 &= [0, 1] \\ G_{d+1} &= [0G_d, 1G_d^R] \end{aligned}$$

where $0G_d$ denotes the sequence obtained by prefixing each member of G_d by 0 and G_d^R is the sequence obtained by reversing the order of G_d.

This particular Gray code has several important properties which we shall summarize below.

Nearest Neighbor. The Hamming distance between any two consecutive (cyclically) members of the sequence is equal to one. *Example:* For $d = 4$, 0100 and 1100 are the 7th and 8th elements, respectively of G_4.

Exchange. The Hamming distance between elements $G_d(i)$ and $G_d(2^j - 1 - i)$ $(1 \leq j \leq d)$, where all the subscripts are taken modulo $2^{(j+1)}$, is equal to one. *Example:* For $d = 4$, $j = 3$, 1111 and 1011 are the 9th and 14th elements, respectively of G_4.

Proof By induction. $G_1 = [0, 1]$ where $G_1(0) = 0$ and $G_1(2^1 - 1 - 0) = 1$ so it is trivially true for $d = 1$. Suppose that the sequence G_d also has this property. Then by induction the Hamming distance between any pair $G_{d+1}(i)$ and $G_{d+1}(2^j - 1 - i)$ $(1 \leq j \leq d)$ is equal to one because both of them are in the same half of the sequence since the indices are taken modulo $2^{(j+1)}$. For $j = d + 1$ we have $G_{d+1}(i)$ and $G_{d+1}(2^{d+1} - 1 - i)$ which differ at exactly one bit, the $(d+1)$st bit by construction.

$$\begin{aligned}
G_{d+1}(i) &= 0G_d(i) \\
G_{d+1}(2^{d+1} - 1 - i) &= 1G_d(i) \\
& \text{ for } i = 0, \ldots, 2^d - 1 \\
G_{d+1}(i) &= 1G_d(2^{d+1} - 1 - i) \\
G_{d+1}(2^{d+1} - 1 - i) &= 0G_d(2^{d+1} - 1 - i) \\
& \text{ for } i = 2^d, \ldots, 2^{d+1} - 1
\end{aligned}$$

—— * ——

Global Connectivity. The Hamming distance between elements $G_d(i)$ and $G_d(i + 2^j)$ $(j > 0)$ is precisely 2. The indices are taken modulo 2^d.

Proof [See Johnsson (1985)]

—— * ——

The first property shows that the BRGC is indeed a Gray code as claimed. Each element differs from its neighbors at exactly one bit. The second property is the exchange property of the BRGC where elements which differ only at the jth bit are exchanges of each other around the jth dimension. And the last is a global connectivity property since elements which differ by strides 2^j $(j > 0)$, no matter how large, remain at a constant Hamming distance of 2. This distance also does not increase with the dimension of the Gray code.

Let B_d be the binary encoding of the numbers $0, 1, \ldots, 2^d - 1$. We now show that the binary encoding does not possess some of the important properties of the BRGC.

Claim 1. The binary encoding B_d does not possess the nearest neighbor property. *Example:* For $d = 4$, 0111 differs from 1000 in all 4 bits.

Claim 2. The binary encoding B_d has Hamming distance j for elements that are exchanges around the jth dimension. *Example:* For $d = 4$ the elements 0000 and 1111 that are exchanges of each other in all 4 dimensions are at a Hamming distance of 4.

Claim 3. The binary encoding B_d possesses the property such that elements that differ by 2^j when i and $i + 2^j$ are in the same $(j + 1)$-dimensional sub-cube have Hamming distance exactly one. This can be seen from the fact that flipping the jth bit results in a binary number that differs from the original one by a stride length of a power of 2. *Example:* For $d = 4$, 0100 and 0110 differ at only one bit.

Therefore one can conclude that while the binary encoding is superior in terms of the particular powers-of-two connections needed for the FFT butterfly, it falls short with regard to nearest neighbor connections and exchange connections, with a maximum Hamming distance that increases with the dimension of the code. The BRGC, however incorporates the best compromise between these three properties.

Chapter 2

Multiplier Computation, Error Analysis

2.1 Introduction

There is no point in having an efficient computational procedure if the results produced are not to the precision required. Errors in finite precision arithmetic are unavoidable. The representation of numbers by a finite number of digits means that most quantities cannot be represented exactly on the computer. If a number x is represented by the nearest floating point number with precision t and machine base b, then

$$fl(x) \;=\; x(1+\epsilon) \qquad |\epsilon| \leq \mathbf{u}$$

where $\mathbf{u}$ is the *unit roundoff* defined by

$$\mathbf{u} = \frac{1}{2}b^{1-t}$$

When arithmetic operations are done on floating point numbers, roundoff errors occur. These errors can then be magnified by a particular numerical method. The amount of magnification of errors is related to the stability of the numerical method and its error propagation properties. Sometimes, two different ways of computing a quantity, while equivalent mathematically, can produce drastically different results when implemented in finite precision arithmetic. See Golub and Van Loan (1983, pp. 32–38), for details on rounding errors.

Mathematically, the FFT is a sequence of sparse unitary matrix-vector multiplications based on the recursion

$$\mathbf{x}^{(k)} \leftarrow \frac{1}{\sqrt{p}}\mathbf{A}_k\mathbf{x}^{(k-1)}$$

where $\mathbf{A}_k$ is defined in Chapter 1. Here we have $n = p^t$. Since each $\frac{1}{\sqrt{p}}\mathbf{A}_k$ is unitary, we have at each stage

$$\|\mathbf{x}^{(k)}\| = \|\mathbf{x}^{(k-1)}\| = \cdots = \|\mathbf{x}^{(0)}\|$$

and

$$\|\mathbf{y}\| = \|\mathbf{F}_n\mathbf{x}\| = \|\mathbf{x}\|$$

letting $\mathbf{x} = \mathbf{x}^{(0)}$ and $\mathbf{y} = \mathbf{x}^{(t)}$. In what follows we denote $\mathbf{A}_k = \frac{1}{\sqrt{p}}\mathbf{A}_k$ to simplify notation.

The use of finite precision arithmetic, however, causes the matrix-vector product to generate an error vector $\mathbf{d}_k$ such that the computed recursion is

$$\hat{\mathbf{x}}^{(k)} \leftarrow \mathbf{A}_k\hat{\mathbf{x}}^{(k-1)} + \mathbf{d}^{(k)}$$

Letting $\mathbf{e}^{(k)} = \hat{\mathbf{x}}^{(k)} - \mathbf{x}^{(k)}$, we see that the error vector is also built up according to a recursion,

$$\mathbf{e}^{(k)} \quad \leftarrow \quad \mathbf{A}_k\mathbf{e}^{(k-1)} + \mathbf{d}^{(k)}, \qquad \mathbf{e}^{(0)} = 0$$

or

$$\|\mathbf{e}^{(k)}\| \quad \leq \quad \|\mathbf{A}_k\| \cdot \|\mathbf{e}^{(k-1)}\| + \|\mathbf{d}^{(k)}\|, \qquad \|\mathbf{e}^{(0)}\| = 0$$

$\mathbf{A}_k$ is unitary so it does not magnify errors, hence

$$\|\mathbf{e}^{(k)}\| \quad \leq \quad \|\mathbf{e}^{(k-1)}\| + \|\mathbf{d}^{(k)}\|, \qquad \|\mathbf{e}^{(0)}\| = 0$$

where

$$\frac{\|\mathbf{d}^{(k)}\|}{\|\mathbf{x}^{(k)}\|} = O(\mathbf{u}).$$

Because the FFT is performed in $t = \log_2 n$ stages, the final roundoff error should be of the form

$$\frac{\|\mathbf{e}^{(t)}\|}{\|\mathbf{y}\|} \approx ct\epsilon, \qquad \epsilon \leq \mathbf{u}$$

with c a constant of order unity, or

$$\|\hat{\mathbf{y}} - \mathbf{y}\| = O(\log_2 n)\mathbf{u}\|\mathbf{y}\|$$

Therefore if the $\mathbf{A}_k$ and $\mathbf{x}^{(0)}$ are exact to full machine precision, we can expect the final error to be proportional to the number of arithmetic operations.

In practice the matrices $\mathbf{A}_k$ are not exact. The trigonometric elements making up $\mathbf{A}_k$ are computed values, and are therefore inexact. For example, if the sine is computed by series expansion, two sources of error will appear, the error from the computational arithmetic and that caused by truncation of the series. Therefore careful computation of the entries of the DFT matrix (which we alternately call multipliers or weights) is of prime importance if we want accurate results.

The purpose of this chapter is to bring into FFT error analyses, the question of how various methods of computing the entries in $\mathbf{A}_k$ alter the accuracy of the FFT. Instead of assuming random noise in the computation of trigonometric functions, we detail in a deterministic manner the errors incurred by various computational methods and their transference into the FFT algorithm. We first focus on several methods for computing these trigonometric functions. Properties such as stability, number of arithmetic operations needed per multiplier, the effect of previously computed weights on subsequent weights, error propagation and magnification are examined. Experimental data is used to confirm our analysis and general bounds are given for the errors generated by each procedure. The error bounds are then incorporated into an overall error analysis of the FFT where the norm used is the infinity norm $\|\cdot\|_\infty$.

2.2 Previous Work

The roundoff error analysis of the FFT has been studied from several different viewpoints. Gentleman and Sande (1966) show that the use of the FFT reduces the upper bound for accumulated roundoff error by a factor of $\frac{k}{m^{3/2}(k-1)}$ (where $n = m^k$) from that of computing the DFT directly by matrix-vector multiplication. This established the FFT as both a more efficient and a more accurate procedure than the DFT.

Fixed point error analyses were considered by the several researchers. Welch (1969) analyzes the accuracy of using rounded sign-magnitude binary arithmetic on a floating-block Decimation-In-Time (DIT) radix-2 FFT where the computations between successive stages are statistically independent. Upper and lower bounds were derived for the root-mean-square error, with the upper bound increasing as $\sqrt{n}$ or $\frac{1}{2}$ bit per stage, and the lower bound increasing as $\frac{1}{2}\log_2 n$. Oppenheim and Weinstein (1972) analyze the case where rounded binary arithmetic is used on a DIT radix-2 FFT with white noise as data, while Thong and Liu (1976) treat all cases generated by rounded or truncated 2-complement binary arithmetic on DIT and Decimation-In-Frequency (DIF) radix-2 FFT's. Knight and Kaiser's study (1979) cover truncated, rounded, complement, sign-magnitude arithmetic for any number base using DIT or DIF, mixed-radix FFT algorithms with data requiring any possible number of scaling shifts and give worse-case, rather than probabilistic, bounds. They also discuss the possibility of trigonometric errors affecting the results of the FFT.

A statistical approach for floating-point FFT error analysis is used by several authors who assumed random errors with zero mean, uniformly distributed in the interval $(-2^{-t}, 2^{-t})$, with t the number of bits of precision. Weinstein (1969) predicts the output noise-to-signal ratio in floating-point FFT computations for the case of white noise and rounded arithmetic. The ratio is found to be proportional to $\log_2 n$. The author surmises that if chopped arithmetic were used, this ratio would be proportional to $(\log_2 n)^2$. This is verified by Alt (1978). Bois and Vignes (1980) then present a FORTRAN program that is used to estimate the local accuracy of the FFT by computing the DIT algorithm three times, with different random perturbations done on each run. The mean of the three runs is used to estimate the result with comparison to the results obtained by double precision. Their computed error estimations are in line with those suggested by the theory.

A series of papers by Liu and several co-authors deal with the statistical approach for finding bounds for the total relative mean square error (MSE) of floating-point FFT's. The DIF algorithm is first analyzed in Kaneko and Liu (1970). They derive upper and lower bounds for the total relative MSE and find that it is bounded by $v \log_2 n$ and $3v \log_2 n$ (where $v = 2^{-2t}/3$ is the variance of a random variable uniformly distributed in $(-2^{-t}, 2^{-t})$). Liu and Kaneko (1975) then extend the method to the DIT FFT and find that while the analysis is different from that of the DIF approach for the *individual* Fourier coefficients, the upper and lower bounds for the total relative MSE is identical. Thong and Liu (1977b) next present a unified approach to DIT and DIF algorithms using their statistical approach. Throughout their analyses they assume that the sines and cosines are generated exactly. The same authors (1977a) also experiment with the use of a double precision floating point accumulator for single precision FFT's and find that the roundoff for an N-point transform $(N = a^m)$, is $O(m)$ independent of the radix a. Finally Munson and Liu (1981) derive a rather complicated expression for the floating point MSE of the prime factor FFT.

Prakash and Rao (1982) derive error variances for a multidimensional Vector Radix FFT algorithm using fixed-point arithmetic They find that the error performance of the vector radix approach is superior to the conventional method of successively applying one-dimensional FFTs when no scaling is done between the stages, otherwise the vector radix approach is only marginally better. Pitas and Strintzis (1983) analyze floating point Vector Radix-2 FFTs as well as Nussbaumer's (1979) Polynomial transform and prove that the Vector Radix FFT has better error characteristics for a two-dimensional FFT than either the Polynomial transform or the regular row-column approach. Multidimensional transforms were also considered by Chan and Jury (1974) who extend the method of FFT error analysis to general orthogonal transforms that can be computed with fast algorithms similar to the FFT.

Ramos (1971) takes a different tack and presents the roundoff analysis of the FFT as a sequence of sparse matrix-vector multiplications. In the bounds the effect of roundoff in the computation of the sines and cosines are modeled, although just by an absolute error constant. The bounds show that the contribution of the absolute error term for the computed sines and cosines is of the same proportion as that of the total roundoff error incurred by the additions and multiplications. Tsao (unpublished) compares the radix-2 DIT and DIF FFT algorithms and concludes that they are "equivalent" in terms of error complexity measures.

Our approach is similar to that of Ramos and is defined in terms of the Cooley-Tukey (CT2) algorithm in Chapter 1. The results for the Cooley-Tukey FFT are then extended to the Gentleman-Sande FFT algorithm. The errors in cosine and sine computation are incorporated as a function of how they are computed. As we shall see, in some unstable cases, these errors can actually overwhelm the roundoff errors incurred in the actual butterfly calculations.

2.3 Trigonometric Function Generation

The computation of a complex DFT $\mathbf{F}_n\mathbf{x}$ requires the computation of multipliers,

$$\omega_n^j \quad j = 0, \ldots, n/2 - 1$$

where ω_n is the nth root of unity, $\exp(-\frac{2\pi}{n})$.

$$\omega_n^j = \cos(j\frac{\pi}{n/2}) - i\sin(j\frac{\pi}{n/2})$$

The generation of the cosines and sines of the angles $j\theta = j\frac{2\pi}{n}$, with j ranging from 0 to $n/8$, i.e. the angles between 0 and $\frac{\pi}{4}$, is required. This is because the following symmetries can be used to acquire the rest of the sines and cosines.

$$\cos(\frac{\pi}{2} - \alpha) = \sin(\alpha) \tag{2.3-1}$$

$$\sin(\frac{\pi}{2} - \alpha) = \cos(\alpha) \tag{2.3-2}$$

$$\cos(\frac{\pi}{2} + \alpha) = -\sin(\alpha) \tag{2.3-3}$$

$$\sin(\frac{\pi}{2} + \alpha) = \cos(\alpha) \tag{2.3-4}$$

$$\cos(\pi - \alpha) = -\cos(\alpha) \tag{2.3-5}$$

$$\sin(\pi - \alpha) = \sin(\alpha) \tag{2.3-6}$$

There are many methods for computing these values [Oliver (1975)], most of which are based on recurrence relations. A few methods are based on angle bisection. Oliver finds that the bisection methods exhibit error properties superior to that of the various recurrence methods. But in any case, the error analysis involved demonstrates how mathematically equivalent recurrence relations can have very different computational properties. In this section we consider the numerical accuracy of several methods. Details on implementation requirements for the the FFT are discussed in the following section, while implementation on vector processors and distributed memory processors is discussed in Chapter 3.

Let

$$\theta = \frac{2\pi}{n},$$

and set

$$\begin{aligned} c_j &= \cos(j\theta) \\ s_j &= \sin(j\theta). \end{aligned}$$

The computed c_j and s_j are assumed to satisfy

$$|\hat{c}_j - c_j| \leq h_j^{method}\mathbf{u}$$
$$|\hat{s}_j - s_j| \leq h_j^{method}\mathbf{u}$$

where h_j^{method} varies with j and is dependent on the computational method used. On a system with good COS and SIN library functions, it is not unreasonable to assume that cosines and sines can be computed to machine precision, i.e. $h_j^{DC} = 1$, (DC stands for the Direct Call method of Algorithm 2.3.2.)

Different methods for the computation of c_j and s_j and equivalently $w_j = c_j - is_j$ are presented and the error coefficient h_j^{method} developed for each method. To standardize our discussion we define the problem as:

> *Given:* n, and $\theta = \frac{2\pi}{n}$. *Find:* c_j and s_j for $j = 1, \ldots, n/8$ such that $c_j = \cos(j\theta)$ and $s_j = \sin(j\theta)$.

The following algorithm creates a table of the $n/2$ cosines and sines of the angle $j\theta$, $j = 0, \ldots, n/2$, i.e. the angles from 0 to π.

Algorithm 2.3.1 Create Table

```
$\theta = \frac{2\pi}{n}$
for $j = 0 : n/8 - 1$
  c(j) ← $COS(j\theta)$; s(j) ← $SIN(j\theta)$
  c(n/4 − j) ← s(j); s(n/4 − j) ← c(j)
  c(n/4 + j) ← −s(j); s(n/4 + j) ← c(j)
  c(n/2 − j) ← −c(j); s(n/2 − j) ← s(j)
end
```

Library Functions

The most straightforward method is to use direct calls to the COS and SIN library functions.

A simple procedure for generating a table of cosines and sines can be written for any n.

Algorithm 2.3.2 Direct Call

```
    c(0) ← 1; s(0) ← 0
    for $j$ = $1{:}n/8$
      $\alpha = \frac{j}{n/2} \cdot \pi$
      c(j) ← $COS(\alpha)$; s(j) ← $SIN(\alpha)$
    end
```

Theorem 2.3.1 *If $fl(y) = COS(x) = \cos(x)(1 + \epsilon_c)$, and $fl(y) = SIN(x) = \sin(x)(1 + \epsilon_s)$, with $|\epsilon| \leq \mathbf{u}$, then computing c_j and s_j by direct calls to the COS and SIN routines produces $\hat{c}_j$ and $\hat{s}_j$ such that*

$$|\hat{c}_j - c_j| \leq h_j^{DC}\mathbf{u}$$
$$|\hat{s}_j - s_j| \leq h_j^{DC}\mathbf{u}$$

where $h_j^{DC} = 1$.

This method gives the most numerical accuracy and we use it as the "correct" solution for comparison with the other methods.

Forward Recursion

Forward recursion is based on the relations,

$$\begin{aligned} \cos(j\theta) &= 2\cos(\theta)\cos([j-1]\theta) - \cos([j-2]\theta) \\ \sin(j\theta) &= 2\cos(\theta)\sin([j-1]\theta) - \sin([j-2]\theta) \end{aligned}$$

A simple procedure for computing the cosines and sines of successive angles follows.

Algorithm 2.3.3 Forward Recursion

```
$\theta = \left(\frac{2\pi}{n}\right)$
c(0) $\leftarrow$ 1; s(0) $\leftarrow$ 0
c(1) $\leftarrow$ $COS(\theta)$, s(1) $\leftarrow$ $SIN(\theta)$
tc1 $\leftarrow$ 2 $\cdot$ c(1)
for $j = 2 : n/8 - 1$
  c(j) $\leftarrow$ tc1 $\cdot$ c(j - 1) - c(j - 2); s(j) $\leftarrow$ tc1 $\cdot$ s(j - 1) - s(j - 2)
end
```

Each step of the recursion involves one multiplication and one addition. Therefore roundoff error $\approx 2\mathbf{u}$ is incurred at each step. How these errors are magnified can be clarified by looking at the matrix which represents the relevant computations at each step. We focus on the cosines. However the same analysis applies to the sines since the recursion formulas are similar. The transition from step $j-1$ to j can be written as the following matrix-vector multiplication:

$$\begin{bmatrix} c_j \\ c_{j-1} \end{bmatrix} = \begin{bmatrix} 2c_1 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} c_{j-1} \\ c_{j-2} \end{bmatrix}$$

Let $\mathbf{A}$ be the forward recursion matrix.

$$\mathbf{A} = \begin{bmatrix} 2c_1 & -1 \\ 1 & 0 \end{bmatrix}$$

Let the initial error vector be

$$\mathbf{e} = \begin{bmatrix} \epsilon \\ 0 \end{bmatrix}, \quad |\epsilon| \leq \mathbf{u}$$

assuming the only errors are in c_1. We have

$$\|\mathbf{Ae}\|_2 \leq \|\mathbf{A}\|_2 \|\mathbf{e}\|_2$$

therefore all we have to do is to find the two-norm of $\mathbf{A}$ to get a bound for the magnification factor.

Theorem 2.3.2

$$\|\mathbf{A}\|_2 = \sqrt{2c_1^2 + 1 + 2c_1\sqrt{c_1^2+1}}$$

Proof $\|\mathbf{A}\|_2 = [\lambda_{max}\mathbf{A}^T\mathbf{A}]^{1/2}$. Working through the algebra, we find that

$$\lambda = 2c_1^2 + 1 \pm 2c_1\sqrt{c_1^2+1}$$

Theorem 2.3.3 *If $\widehat{c}_j$ and $\widehat{s}_j$ are computed via Algorithm 2.3.3 then*

$$\begin{aligned} |\widehat{c}_j - c_j| &\leq h_j^{FR}\mathbf{u} + O(\mathbf{u}^2) \\ |\widehat{s}_j - s_j| &\leq h_j^{FR}\mathbf{u} + O(\mathbf{u}^2) \end{aligned}$$

with $h_j^{FR} = (|c_1| + \sqrt{|c_1|^2 + 1})^j$

Proof $fl(c_1) = COS(fl(\theta))$ so:

$$\begin{aligned} |\widehat{c}_1 - c_1| &\leq h_1^{DC}\mathbf{u} \\ &= [2(\widehat{c}_1\widehat{c}_{j-1})(1+\delta_j) - \widehat{c}_{j-2}](1+\epsilon_j) \end{aligned}$$

From the forward recursion formula:

$$\widehat{c}_j = fl(c_j) = fl[2fl(\widehat{c}_1\widehat{c}_{j-1}) - \widehat{c}_{j-2}]$$

So that

$$\begin{aligned} |\widehat{c}_j - c_j| &\leq |\widehat{c}_j - (2\widehat{c}_1\widehat{c}_{j-1} - \widehat{c}_{j-2})] \\ &\quad + |2\widehat{c}_1\widehat{c}_{j-1} - \widehat{c}_{j-2} - c_j| \\ &\leq |2\widehat{c}_1\widehat{c}_{j-1}(\delta_j + \epsilon_j) - \widehat{c}_{j-2}\epsilon_j| \\ &\quad + |2\widehat{c}_1\widehat{c}_{j-1} - 2c_1c_{j-1}| + |\widehat{c}_{j-2} - c_{j-2}| \\ &\leq 2\mathbf{u}|\widehat{c}_j| + 2|\widehat{c}_1||\widehat{c}_{j-1} - c_{j-1}| \\ &\quad + |2c_{j-1}||\widehat{c}_1 - c_1| + |\widehat{c}_{j-2} - c_{j-2}| \\ &\leq 2\mathbf{u}|c_j| + 2|c_1|h_{j-1}^{FR}\mathbf{u} + 2|c_{j-1}|h_1^{DC}\mathbf{u} + h_{j-2}^{FR}\mathbf{u} + O(\mathbf{u}^2) \end{aligned}$$

The roundoff error for each step if $2\mathbf{u}$. The magnification of the error from the previous cosine c_{j-1} is reflected in the term

$$2|c_1|h_{j-1}^{FR}\mathbf{u}.$$

The error in the $j-2$ cosine c_{j-2} is represented in the term

$$h_{j-2}^{FR}\mathbf{u}$$

Solving the recurrence

$$h_j^{FR} = x^j = 2|c_1|x^{j-1} + x_{j-2}$$

gives us

$$h_j^{FR} = (|c_1| + \sqrt{|c_1|^2 + 1})^j$$

The same analysis can be carried out for the sines.

———*———

This algorithm is unstable because it magnifies roundoff error at each stage by roughly a factor of $|c_1|+\sqrt{|c_1|^2+1}$, a quantity greater than $2|c_1|$. Furthermore, the roundoff error introduced at each step propagates exponentially into subsequent steps.

Repeated Multiplication

Another way to generate FFT weights is to observe that $\exp(-i(j\theta)) = \cos(j\theta) - i\sin(j\theta)$. Let $\omega^j = \exp(-i(j\theta))$. Then $\omega^{j+1} = \omega \cdot \omega^j$.

Algorithm 2.3.4 Repeated Multiplication

```
    $\theta = \left(\frac{2\pi}{n}\right)$
    $c(0) \leftarrow 1;\ s(0) \leftarrow 0$
    $c(1) \leftarrow COS(\theta);\ s(1) \leftarrow SIN(\theta)$
    for $j = 2 : n/8 - 1$
        $c(j) \leftarrow c(1) \cdot c(j-1) + s(1) \cdot s(j-1)$
        $s(j) \leftarrow -s(1) \cdot c(j-1) + c(1) \cdot s(j-1)$
    end
```

Each step involves one complex multiplication which breaks down to two multiplications and one addition for each trigonometric function. The multiplication $\omega \cdot \omega^j$ is a Given's rotation;

$$\begin{bmatrix} c_j \\ s_j \end{bmatrix} \leftarrow \begin{bmatrix} c_1 & s_1 \\ -s_1 & c_1 \end{bmatrix} \begin{bmatrix} c_{j-1} \\ s_{j-1} \end{bmatrix}$$

Given's rotations are orthonormal and therefore do not magnify roundoff errors. Their properties are presented in Golub and Van Loan (1983, pp. 43–47).

Theorem 2.3.4 *If $\hat{c}_j$ and $\hat{s}_j$ are computed via Algorithm 2.3.4 then*

$$\begin{aligned} |\hat{c}_j - c_j| &\leq h_j^{RM}\mathbf{u} + O(\mathbf{u}^2) + O(\mathbf{u}^2) \\ |\hat{s}_j - s_j| &\leq h_j^{RM}\mathbf{u} + O(\mathbf{u}^2) + O(\mathbf{u}^2) \end{aligned}$$

with $h_j^{RM} = 2j$

Proof Since c_1 and s_1 are computed by calls to the cosine and sine function we have:

$$\begin{aligned} |\hat{c}_1 - c_1| &\leq h_1^{DC}\mathbf{u} \\ |\hat{s}_1 - s_1| &\leq h_1^{DC}\mathbf{u} \end{aligned}$$

Let $\mathbf{y}_j = [c_j, s_j]^T$ and $\mathbf{G}$ be the Given's rotation. Then

$$fl(\widehat{\mathbf{G}\mathbf{y}}_{j-1}) = \mathbf{G}\mathbf{y}_j + \mathbf{E},$$

where

$$\|\mathbf{E}\|_2 \leq \|\mathbf{y}_{j-1}\|_2 O(\mathbf{u})$$

We have

$$\begin{aligned} \hat{c}_j &= fl[fl(\hat{c}_1\hat{c}_{j-1}) + fl(\hat{s}_1\hat{s}_{j-1})] \\ &= [(\hat{c}_1\hat{c}_{j-1})(1+\delta_1) + (\hat{s}_1\hat{s}_{j-1})(1+\delta_2)](1+\epsilon_1) \end{aligned}$$

so that

$$\begin{aligned} |\hat{c}_j - c_j| &\leq |\hat{c}_j - \hat{c}_1\hat{c}_{j-1} + \hat{s}_1\hat{s}_{j-1}| \\ &\quad + |\hat{c}_1\hat{c}_{j-1} - c_1 c_{j-1}| + |\hat{s}_1\hat{s}_{j-1} - s_1 s_{j-1}| \\ &\leq 2\mathbf{u}|\hat{c}_j| + |\hat{c}_1||\hat{c}_{j-1} - c_{j-1}| + |c_{j-1}||\hat{c}_1 - c_1| \\ &\quad + |\hat{s}_1||\hat{s}_{j-1} - s_{j-1}| + |s_{j-1}||\hat{s}_1 - s_1| \\ &\leq 2\mathbf{u}|\hat{c}_j| + |\hat{c}_1|h_{j-1}^{RM}\mathbf{u} \\ &\quad + |c_{j-1}|h_1^{DC}\mathbf{u} + |\hat{s}_1|h_{j-1}^{RM}\mathbf{u} + |s_{j-1}|h_1^{DC}\mathbf{u} + O(\mathbf{u}^2) \end{aligned}$$

Since $|c_1| \leq 1$ and $|s_1| \leq 1$, the error from the previous cosine h_{j-1}^{RM} is not magnified and the absolute roundoff error, $2\mathbf{u}|c_j|$, decreases as $c_j \to 0$. Therefore $h_j^{RM} = 2j$. this is quite a liberal upper bound for the error. The analogous computations give the same result for the sines except that the roundoff error $2\mathbf{u}|s_j|$ increases from near 0 to $2\mathbf{u}$ as $s_j \to 1$.

——*——

Logarithmic Recursion

The methods in the previous two sections call COS and SIN only once, namely to compute c_1 and s_1, and thus suffered degeneracy of accuracy as j increases. To garner more accuracy throughout the range of j, we could use the approach of computing a logarithmic number of cosines and sines exactly and derive the remaining ones. A natural distribution is to compute

$$\{c_j : j = 2^k, \quad k = 0, 1, \ldots\}$$

$$\{s_j : j = 2^k, \quad k = 0, 1, \ldots\}$$

by calls to the COS and SIN routine. Using the recurrence formula:

$$\begin{aligned} \cos(A+B) &= (2\cos(B))\cos(A) - \cos(A-B) \\ \sin(A+B) &= (2\cos(B))\sin(A) - \sin(A-B). \end{aligned}$$

we derive a method of computing the rest of the c_j's and s_j's. Of course, this necessitates the storage of all the previously computed weights. In forward recursion, the angle B was always θ and A ranged from θ, 2θ,...,etc. In logarithmic recursion, the angle B is θ at first, then 2θ, then 4θ, etc. Meanwhile for each of the B's $= 2^k\theta$, A will range from θ to $(2^k - 1)\theta$. The angle B represents the cosines that were computed by calls to COS. The following procedure generates c_j and s_j, $j = 1, \ldots, \frac{n}{8}$.

Algorithm 2.3.5 Logarithmic Recursion

```
$n = 2^t$, $\theta = \frac{\pi}{n/2}$
c(1) $\leftarrow$ $COS(\theta)$; s(1) $\leftarrow$ $SIN(\theta)$
for $k = 1 : t - 4$
  $j = 2^k$
  c(j) $\leftarrow$ $COS(j\theta)$
  s(j) $\leftarrow$ $SIN(j\theta)$
  for $m = 1 : j - 1$
    c(j + m) $\leftarrow$ 2c(j) $\cdot$ c(m) $-$ c(j $-$ m);
    s(j + m) $\leftarrow$ 2c(j) $\cdot$ s(m) + s(j $-$ m)
  end
end
$j = 2^{t-3}$
c(j) $\leftarrow$ $COS(j\theta)$; s(j) $\leftarrow$ $SIN(j\theta)$
```

Notice that we stay in the first quadrant so that $\cos(A - B) = \cos(B - A)$. This explains the subtraction by c_{j-m} in the main loop. For the sines, we have $\sin(A - B) = -\sin(B - A)$ explaining the addition by s_{j-m}.

If $n = 2^t$, then j ranges from 0 to $n/8$. Let $j = (b_{t-2} \ldots b_1 b_0)_2$ and define

$$\beta_j = \sum_{i=0}^{t-2} b_i.$$

β_j is the number of bits in j's binary expansion that are equal to 1. Since all $\{j : j = 2^k\}$ were computed by direct calls to COS or SIN, $\widehat{c}_{2^k} = c_{2^k} + \epsilon$ and $\widehat{s}_{s^k} = s_{2^k} + \epsilon$, i.e. $h_{2^k}^{LR} = h_{2^k}^{DC} = 1$.

Theorem 2.3.5 *If $\widehat{c}_j$ and $\widehat{s}_j$ are computed via Algorithm 2.3.5 then*

$$\begin{aligned} |\widehat{c}_j - c_j| &\leq h_j^{LR}\mathbf{u} \\ |\widehat{s}_j - s_j| &\leq h_j^{LR}\mathbf{u} \end{aligned}$$

with $|h_j^{LR}| = \prod_{b_i=1}(|c_{2^i}| + \sqrt{|c_{2^i}|^2 + 1})$

Proof First we have

$$\begin{aligned} \widehat{c}_j &= fl[2fl(\widehat{c}_{2^k}\widehat{c}_m) - \widehat{c}_{2^k-m}] \\ &= [2\widehat{c}_{2^k}\widehat{c}_m(1+\delta_1) - \widehat{c}_{2^k-m}](1+\epsilon_1) \end{aligned}$$

and thus

$$\begin{aligned} |\widehat{c}_j - c_j| &\leq |\widehat{c}_j - 2\widehat{c}_{2^k}\widehat{c}_m - \widehat{c}_{2^k-m}| \\ &\quad +|2\widehat{c}_{2^k}\widehat{c}_m - 2c_{2^k}c_m| + |\widehat{c}_{2^k-m} - c_{2^k-m}| \\ &\leq 2\mathbf{u}|c_j| + 2|\widehat{c}_{2^k}||\widehat{c}_m - c_m| \\ &\quad +2|c_m||\widehat{c}_{2^k} - c_{2^k}| + |\widehat{c}_{2^k-m} - c_{2^k-m}| \\ &\leq 2\mathbf{u}|c_j| + 2|c_{2^k}|h_m^{LR}\mathbf{u} + 2|c_m|h_{2^k-m}^{DC}\mathbf{u} + h_{2^k-m}^{LR}\mathbf{u} + O(\mathbf{u}^2) \end{aligned}$$

$$|fl(c_j) - c_j| \leq 4\mathbf{u}|c_j| + 2|c_{2^k}|h_m^{LR}\mathbf{u} + 2|c_m|h_{2^k}^{DC}\mathbf{u} + h_{2^k-m}^{LR}\mathbf{u} + O(\mathbf{u}^2)$$

There are a logarithmic number of multiplications and additions for each j so that roundoff is of order β_j. However, this method also suffers from the instability analogous to that of forward recursion, although not to such a great extent. The error coefficient for c_m is multiplied by $|c_{2^k}| + \sqrt{|c_{2^k}|^2 + 1}$ where $\beta_{2^k} = 1$. We have $\beta_j = 1 + \beta_m$ due to this $b_{k-1} = 1$ term. Since the error coefficient for c_m includes all the other $|c_{2^i}| + \sqrt{|c_{2^i}|^2 + 1}$ where $b_i = 1$, the error for c_j is simply $|c_{2^k}| + \sqrt{|c_{2^k}|^2 + 1}$ times the error for c_m and is thus a product of the $|c_i| + \sqrt{|c_i|^2 + 1}$ for $b_i = 1$. Cosines decrease as the argument goes from 0 to 1 and therefore error magnification is not as bad as forward recursion. There are only β_j steps to contaminate. Letting the binary representation of j be $b_{t-3} \ldots, b_0$, an approximate bound for the magnification factor for c_j would be

$$\prod_{i:b_i=1} |c_{2^i}| + \sqrt{|c_{2^i}|^2 + 1}.$$

This says that the only factors $|c_{2^i}| + \sqrt{|c_{2^i}|^2 + 1}$'s that enter into the error of j are those where j's ith bit is equal to 1. The same bound holds analogously for the sines, although sines increase from 0 to 1.

——*——

Subvector Scaling

This method also computes the c_j's and s_j's where $j = 2^k$, $k = 0, 1, \ldots$ by directly calling the COS and SIN routines The rest of the weights are computed by noting

$$\omega_n^j = \omega_n^{\lfloor \log_2 j \rfloor} \cdot \omega_n^{j - \lfloor \log_2 j \rfloor}$$

At each stage k we have the weights $\mathbf{w}(0 : 2^k - 1)$ and we can compute

$$\mathbf{w}(2^k : 2^{k+1} - 1) = \omega_n^{2^k} \mathbf{w}(0 : 2^k - 1)$$

by scaling our previous vector of weights by $\omega_n^{2^k}$. The following algorithms computes the weight vector by this method.

Algorithm 2.3.6 Subvector Scaling

```
$n = 2^t$
$n_o \leftarrow n/2$
$\theta = \left(\frac{\pi}{n_o}\right)$
$c(0) \leftarrow 1$; $s(0) \leftarrow 0$
$c \leftarrow COS(\theta)$; $s \leftarrow SIN(\theta)$
$c(1) \leftarrow c$; $s(1) \leftarrow s$
for $q = 1 : t - 4$
  $L \leftarrow 2^q; L_o \leftarrow L/2$
  for $j = 0 : L_o - 1$
    $c(L_o+j) \leftarrow c \cdot c(j) + s \cdot s(j)$; $s(L_o+j) \leftarrow -s \cdot c(j) + c \cdot s(j)$
  end
  $n_o \leftarrow n_o/2$
  $\theta = \left(\frac{\pi}{n_o}\right)$
  $c \leftarrow COS(\theta)$; $s \leftarrow SIN(\theta)$
  $c(L) \leftarrow c$, $s(L) \leftarrow s$
end
```

Each ω^j needs only a maximum of $t-4$ multiplications. The multiplications are all Given's rotations and hence do not magnify errors. The following theorem proves that there are a logarithmic number of multiplications. Here we let the complex weight vector $\mathtt{w(0:n/8-1)}$ represent $\mathtt{c(0:n/8-1)} + i\mathtt{s(0:n/8-1)}$.

Theorem 2.3.6 *Let $n = 2^t$ and $j = (b_{t-3} \dots b_0)_2$, where b_i is the ith bit in the binary expansion of j. Then computing the weight vector $\mathtt{w(0:n/4-1)}$ by Algorithm 2.3.6 gives $\mathtt{w(j)}$ as a result of*

$$\left(\sum_{i=0}^{t-3} b_i \right) - 1$$

complex multiplications.

Proof This is based on induction with q = number of bits involved.

$$\begin{aligned} q=1 \quad j &= 0 \quad \mathtt{w(0)} = 1, \\ &\quad\ \text{no multiplications} \\ j &= 1 \quad \mathtt{w(1)} = \omega_n, \\ &\quad\ \text{direct computation, no multiplications} \end{aligned}$$

Now assume this is true for $q = k$, i.e. $j = (b_{k-1} \dots b_1 b_0)_2$. $\mathtt{w(j)}$ is a consequence of $(b_{k-1} + \dots + b_0 - 1)$ multiplications by the induction hypothesis. There are two cases.

$$\begin{aligned} \mathtt{w(0\hat{}j)} &= \mathtt{w(j)} \\ &\quad \text{result carries over} \end{aligned}$$

since $b_j = 0$ here, hence true for $q = k+1$.

$$\begin{aligned} \mathtt{w(1\hat{}j)} &= \omega_{n/2^k} \cdot \mathtt{w(j)} \\ &\quad \text{one multiplication + what we had previously} \end{aligned}$$

since $b_k = 1$ here, we have a total of $(b_k + b_{k-1} + \cdots b_0 - 1)$ multiplications, and hence the result follows for $q = k+1$.

____*____

Theorem 2.3.7 *If $\hat{c}_j$ and $\hat{s}_j$ are computed via Algorithm 2.3.6 then*

$$\begin{aligned} |\hat{c}_j - c_j| &\leq h_j^{SS}\mathbf{u} \\ |\hat{s}_j - s_j| &\leq h_j^{SS}\mathbf{u} \end{aligned}$$

with $h_j^{SS} = 2\beta_j$, with β_j the number of bits in j's binary expansion that is equal to 1.

Proof Each

$$c_j \leftarrow c_{\lfloor\log_2 j\rfloor} \cdot c_{j-\lfloor\log_2 j\rfloor} + c_{\lfloor\log_2 j\rfloor} \cdot s_{j-\lfloor\log_2 j\rfloor}.$$

Therefore the errors for each c_j are those of $c_{j-\lfloor\log_2 j\rfloor}$ and $s_{j-\lfloor\log_2 j\rfloor}$ plus the roundoff for this step. Each Given's rotation adds roughly $2\mathbf{u}$ to the roundoff, but does not magnify them. From the proof of Theorem 2.3.4, we see that

$$\begin{aligned}|\hat{c}_j - c_j| \;\le\; & 2\mathbf{u}|c_j| + |c_{\lfloor\log_2 j\rfloor}|h^{SS}_{j-\lfloor\log_2 j\rfloor}\mathbf{u} + |c_{j-\lfloor\log_2 j\rfloor}|h^{DC}_{\lfloor\log_2 j\rfloor}\mathbf{u} \\ & +|s_{\lfloor\log_2 j\rfloor}|h^{SS}_{j-\lfloor\log_2 j\rfloor}\mathbf{u}|s_{j-\lfloor\log_2 j\rfloor}|h^{DC}_{\lfloor\log_2 j\rfloor}\mathbf{u} + O(\mathbf{u}^2)\end{aligned}$$

Recall that β_j is the number of ones in the binary representation of j, hence $h^{SS}_j = 2\beta_j$ The sines are computed in the same manner so that a similar accounting of errors gives the same bound.

———*———

The error properties of this method are extremely good. The stability is that of the Given's rotations combined with only a logarithmic number of multiplications.

Recursive Bisection

A stable method introduced by Hopgood and Litherland in the 1960's involves interpolation. [Oliver(1975)] The underlying relations are specified by the trigonometric identities:

$$\cos A \cos B = \frac{1}{2}\{\cos(A-B) + \cos(A+B)\} \qquad (2.3\text{-}7)$$

$$\sin A \cos B = \frac{1}{2}\{\sin(A-B) + \sin(A+B)\} \qquad (2.3\text{-}8)$$

Notice that A is an angle that is half-way between the angles $A-B$ and $A+B$. This means that if we knew the cosines or sines for $A-B$ and $A+B$, we can then find the cosine or sine of A by one multiplication, one addition and the computation of $\cos B$.

Suppose $n = 2^t$ and $\theta = \frac{\pi}{n/2}$ and we wish to compute the cosines $c_j = \cos j\frac{\pi}{n/2}$ and sines $s_j = \sin j\frac{\pi}{n/2}$, $j = 1, \ldots, \frac{n}{8} - 1$.

Algorithm 2.3.7 Recursive Bisection

```
c(0) ← 1; s(0) ← 0
c(n/8) ← $\sqrt{2}/2$; s(n/8) ← $\sqrt{2}/2$
for $L = 4{:}t$
```

```
    $i \leftarrow n/2^L$; $j \leftarrow n/2^L$
    $c(n/2^L) \leftarrow COS(2\pi/2^L)$
    $s(n/2^L) \leftarrow SIN(2\pi/2^L)$
    $h \leftarrow 1/2c(n/2^L)$
    for $m=1{:}2^L/8-1$
      $j \leftarrow j+2i$
      $c(j) \leftarrow h\cdot(c(j-i)+c(j+i))$; $s(j) \leftarrow h\cdot(s(j-i)+c(j+i))$
    end
  end
```

Again the powers-of-two multipliers are computed by direct calls to the COS and SIN functions. Therefore we have $\{c_j,\ s_j :\ j = 2^k\}$. To get the rest of the c_j's and s_j's we must compute them via Equations 2.3-7 and 2.3-8. How many steps it takes to get to a particular j depends on the number search steps S_j it takes to "find" j when initially only the powers of two are available. For example, if we had $\{1,2,4,8,16,32\}$, $S_3 = 1$, since it comes right in between 2 and 4, which we have. $S_6 = 1$ also, since it comes right in between 4 and 8. $S_{14} = 2$, since we have to first get 12 between 8 and 16, and then 14 between 12 and 16. If we write out these numbers in binary, we see that S_j is equal to the position of the highest order 1 minus the position of the lowest order 1. Let $j = (b_t \ldots b_1 b_0)_2$

Lemma 2.3.1

(2.3-9) $$S_j = max\{k : b_k = 1\} - min\{k : b_k = 1\}$$

Proof Since we initially have all the powers of two, the number of search steps needed to find them is 0. Each binary representation of 2^k has a 1 at the kth position and zeros everywhere else. Thus $S_{2^k} = k - k = 0$. $\{j : j = 2^k + 2^{k-1}\}$ can now be found by averaging between 2^k and 2^{k+1} for each k. These numbers have a 1 at the kth position and another 1 at the $k-1$th position, thus $S_{2^k+2^{k-1}} = k-(k-1) = 1$. Next all the numbers $\{j : j = 2^k + 2^{k-2}, j = 2^k + 2^{k-1} + 2^{k-2}\}$ can be found by interpolating between $\{j : j = 2^k, j = 2^k + 2^{k-1}\}$. These numbers all have a 1 at the kth position and another 1 at the $k-2$th position. Some of them also have a 1 at the $k-1$th position and others have a 0 there, but for our purposes, it does not matter. We can now see a pattern emerge where all numbers $\{j : j = 2^k + (\sum_{i=k-s+1}^{k-1} b_i 2^i) + 2^{k-s}\}$ have search number $S_j = s$, representing s bisection steps.

———*———

Lemma 2.3.2

$$S_j = \lfloor \log_2 m_j \rfloor$$
$$where\ m_j = j/(j.AND.-j)$$

Proof In two's complement arithmetic, $j.AND. - j$ determines the number which is represented by the trailing 1. For example,

$$\begin{aligned} 1010.AND. - 1010 &= 1010.AND.0110 \\ &= 0010 \end{aligned}$$

Dividing j by $j.AND. - j$ has the effect of stripping away all the trailing zeros. Hence for our example, 1010/0010 = 0101. Now taking the floor of the logarithm base two of the result gives us the spread between the highest order 1 and the lowest order 1.

——*——

Let $l_j = \log_2(j.AND. - j)$, then l_j gives us the number of trailing zeros in j. c_{l_j} is the cosine of the angle which is half of the spread between c_{j-l_j} and c_{j+l_j}, and similarly s_{l_j} is the sine of the angle which is halfway between s_{j-l_j} and s_{j+l_j}.

Theorem 2.3.8 *If $\hat{c}_j$ and $\hat{s}_j$ are computed via Algorithm 2.3.7 then*

$$\begin{aligned} |\hat{c}_j - c_j| &\leq h_j^{RB}\mathbf{u} \\ |\hat{s}_j - s_j| &\leq h_j^{RB}\mathbf{u} \end{aligned}$$

with $h_j^{RB} = 2S_j$ where S_j is described by Equation 2.3-9.

Proof We have

$$\begin{aligned} |\hat{c}_j - c_j| &\leq |\hat{c}_j - \frac{1}{2\hat{c}_{l_j}}(\hat{c}_{j-l_j} + \hat{c}_{j+l_j})| \\ &\quad + |\frac{1}{2\hat{c}_{l_j}}\hat{c}_{j-l_j} - \frac{1}{2c_{l_j}}c_{j-l_j}| + |\frac{1}{2\hat{c}_{l_j}}\hat{c}_{j+l_j} - \frac{1}{2c_{l_j}}c_{j+l_j}| \\ &\leq \frac{2}{|c_{l_j}|}|c_j| + \frac{1}{2|c_{l_j}|}h_{j-l_j}^{RB} + \frac{1}{2|c_{l_j}|}h_{j+l_j}^{RB} + O(\mathbf{u}^2) \end{aligned}$$

Hence

$$|\hat{c}_j - c_j| \leq 2\mathbf{u}\frac{|c_j|}{|c_{l_j}|} + \frac{\max(S_{j-l_j}, S_{j+l_j})\mathbf{u}}{|c_{l_j}|} + O(\mathbf{u}^2)$$

and thus $h_j^{RB} = 2S_j$. The sines are computed by the same formula so analogous results hold.

——*——

Experimental Results

The results using single-precision arithmetic on a VAX shows that subvector scaling with the direct computation of the powers-of-two weights fares the best overall, with recursive bisection a close second. Table 2.1 shows the magnification factor for the maximum component error of a vector **c** consisting of cosines. This is computed as the maximum error divided by the single precision machine epsilon, which on the VAX is $2.98023e-08$. Table 2.2 shows the root mean square error of the vector **c** and thus takes an average error of all the components. Again, this error is divided by the single precision machine epsilon.

When Forward Recursion and Repeated Multiplication is implemented in double precision, we do not get magnification of errors until n is quite large with respect to $\frac{1}{\mathbf{u}}$. This is, of course, a machine dependent parameter. For Forward Recursion, this starts at $n = 32768$, and for Repeated Multiplication, $n = 2097152$. Table 2.3 compares the magnification factors of these two method with respect to single precision machine epsilon. The use of double precision for Forward Recursion delays the appearance of significant amounts of error until about $n = 10^7$ since the single precision unit roundoff for the VAX is about 10^{-8}. See Table 2.3.

2.4 Multipliers for the FFT

We do not have to worry about the error coefficient and its ordering and placement in the FFT algorithm for the method of Direct Call to Library Functions since it is assumed to be constant for each c_j and s_j. However for the other methods, each multiplier and its error coefficient h_j^{method} must be linked explicitly to the components of **x** that it is applied to. We must therefore be able to pinpoint the exact multiplier that is applied to each component at each stage.

The radix-two FFT algorithm for a vector **x** of length $n = 2^t$ runs through t stages ($k = 1, \ldots, t$). Denote $x_j^{(k)}$ to be the jth component of **x** at the completion of the kth stage. At the beginning of stage k, the multipliers are applied to $\mathbf{x}^{(k-1)}$. The particular multiplier used in the CT2 butterfly involving x_j is

$$\omega_r^{j \bmod 2^{(r/2)}}, \quad r = 2^k.$$

The angle used at stage k is $\theta_k = 2\pi/2^k$, hence ω_r is directly computed. Since $\omega_r^{j \bmod 2^{(r/2)}}$ is used for transforming $x_j^{(k-1)}$ to $x_j^{(k)}$, the error coefficient associated with $x_j^{(k)}$ is

$$h_{j \bmod 2^{(r/2)}}, \quad r = 2^k.$$

Table 2.1: Maximum Error Magnitude = $\|\mathbf{c}\|_\infty/\mathbf{u}$

n	Forward Recursion	Repeated Multiplication	Logarithmic Recursion	Subvector Scaling	Recursive Bisection
8	5.0	1.0	1.0	2.0	0.0
16	47.0	2.0	5.0	3.0	2.0
32	34.5	2.0	11.5	3.0	2.0
64	321.0	2.0	16.0	3.0	4.0
128	50.3	4.0	21.0	6.0	4.0
256	6353.4	14.4	57.0	4.0	4.0
512	14705.1	17.5	95.5	6.0	6.0
1024	18818.1	8.0	151.4	6.0	8.0
2048	373315.0	21.0	172.0	6.0	8.0
4096	682278.0	12.0	519.5	6.0	10.0
8192	1.87937e+06	35.0	717.5	6.0	12.0
16384	3.35480e+07	34.0	1920.9	6.0	12.0
32768	3.35512e+07	38.0	5137.7	8.0	12.0
65536	3.35528e+07	24.0	6746.0	8.0	12.0
131072	3.35536e+07	47.5	7550.2	8.0	12.0
262144	3.35540e+07	136.0	7952.3	8.0	14.0
524288	3.55542e+07	492.0	8153.3	8.0	14.0
1048576	3.35543e+07	5300.7	8253.8	12.0	14.0

Table 2.2: Root Mean Square Error Magnitude = $\|\mathbf{c}\|_{rms}/\mathbf{u}$

n	Forward Recursion	Repeated Multiplication	Logarithmic Recursion	Subvector Scaling	Recursive Bisection
8	4.8	1.5	0.5	1.5	0.0
16	31.3	1.5	1.9	2.1	1.0
32	19.2	1.1	5.3	1.9	1.1
64	146.7	1.0	6.5	2.0	1.4
128	31.0	1.6	7.3	2.0	1.4
256	3238.7	8.2	19.4	1.8	1.8
512	7981.9	10.0	31.5	2.1	2.1
1024	10514.6	3.2	50.6	2.0	2.1
2048	198593.0	11.9	56.4	2.2	3.3
4096	357597.0	6.2	181.3	2.2	4.2
8192	1.34489e+06	20.8	235.7	1.9	5.0
16384	1.59806e+07	20.3	661.4	1.9	4.9
32768	1.59795e+07	16.0	1793.2	1.9	5.0
65536	1.59789e+07	14.6	2291.6	1.9	5.2
131072	1.59787e+07	26.5	2536.0	1.9	5.4
262144	1.59785e+07	55.3	2657.3	1.9	5.6
524288	1.59785e+07	264.8	2717.8	1.9	5.8
1048576	1.59784e+07	2781.9	2748.0	1.9	6.1

Table 2.3: Double Precision Implementations

	Forward Recursion		Repeated Multiplication	
n	MAX	RMS	MAX	RMS
32768	2.0	0.1	–	–
65536	2.0	0.2	–	–
131072	2.0	0.5	–	–
262144	2.0	0.6	–	–
524288	4.3	2.3	–	–
1048576	16.0	8.3	–	–
2097152	65.6	34.0	–	–
4194304	66.4	34.6	–	–
8388608	1230.2	635.3	3.1e-02	1.5e-05
16777216	6388.3	3299.3	6.1e-05	–
33554432	14912.5	7799.4	1.0	2.9e-04
67108864	18841.0	10058.6	2.0	5.2e-04

(– means it is $<$ **u**.)

This means that we are using the (j mod 2^{k-1})th cosine and sine generated by a particular method. Let

$$m_j^{(k)^{CT2}} = j \bmod 2^{k-1}$$

denote the order of creation that the multiplier applied to $x_j^{(k-1)}$ to produce $x_j^{(k)}$ at the kth stage of the CT2 FFT algorithm. This notation serves as a shorthand in our multiplier-connected FFT error analysis.

The multiplier used on $x_j^{(k-1)}$ to get $x_j^{(k)}$ at stage k in the Gentleman-Sande (GS1) algorithm is x_j at stage k is

$$\omega_s^{j \bmod (s/2)}, \quad s = 2^{t-k+1}.$$

Therefore, the corresponding shorthand is,

$$m_j^{(k)^{GS1}} = j \bmod 2^{t-k}.$$

The above indices for the weights are applicable to the two methods, Forward Recursion, and Repeated Multiplication, where at each stage, a different $\theta = 2\pi/2^k$ is used as the initial angle. This is because these two methods, when used efficiently and intelligently, do *not* store a table of cosines and sines, and thus to capture the best possible error properties, i.e not let j get too large, we generate the multipliers at each stage with the largest initial angle possible.

The methods requiring the direct calculation of a logarithmic number of weights are usually implemented in an FFT subroutine by initializing a table first. This table is of length $O(n)$ and hence includes all the cosines and sines of angles which are a multiple of $\frac{2\pi}{n}$, $n = 2^t$. At each stage in the FFT algorithm one would perform a table lookup for the correct weight. The correct error coefficient would be corresponding to the index in the table that the weights are placed, since it has something to do with the order with which the weights are computed. In addition. recall that not all the weights are actually computed. The symmetries of the cosine and sine function (eqns. 2.3-1– 2.3-6) are all that are needed to determine the rest. Hence for methods requiring table lookup (Logarithmic Recursion, Subvector Scaling and Recursive Bisection),

$$\begin{aligned} {m_j^{(k)}}^{CT2} &= G(2^{t-k}(j \bmod 2^{k-1})) \bmod (2^{t-3}) \\ {m_j^{(k)}}^{GS1} &= G(2^{k-1}(j \bmod 2^{t-k})) \bmod (2^{t-3}) \end{aligned}$$

where $G()$ is the Binary Reflected Gray Code of Chapter 1.

The method of directly calling the Library Functions is most sensible for general purpose FFT computations where a large number of FFTs are done. Here a $O(n)$ table is created once at the initialization phase. Swarztrauber's FFTPACK pretabulates the weights during the initialization phase. The Digital Signal Processing Committee (1979) FFT subroutines also use direct call, but without storing the weights in table. They simply compute the cosines and sines as needed.

The FFT subroutines in Press *et al.* (1985) use double precision Repeated Multiplication for computing the FFT weights, as does the FORTRAN code in Swarztrauber (1984a). Forward Recursion is not used by anyone known to the author, for obvious reasons.

Logarithmic Recursion is not used in practice but was described primarily to illustrate that magnification errors can overwhelm roundoff errors. Repeated Multiplication consists of a stable set of computations, but has $O(j)$ steps for each c_j. Meanwhile Logarithmic Recursion has only $O(\log_2 j)$ steps per c_j, yet its absolute error is much worse than that of Repeated Multiplication because of the exponential magnification of these $O(\mathbf{u} \log_2 j)$ errors. Subvector Scaling is also an $O(\log_2 j)$ procedure but with a stable algorithm. Hence its error bound is proportion to $\log_2 j$.

Recursive Bisection is used by Buneman (1987) in his FFT codes. Buneman (1987) also describes a way of generating the c_j's and s_j's in the natural order by requiring only $O(\log_2 n)$ of table storage.

2.5 Error Analysis of the FFT

Floating Point Complex Arithmetic

In this error analysis, we compute the componentwise error of the Cooley-Tukey radix-2 FFT algorithm. Since the arithmetic done is complex, we start by deriving a model for floating-point complex arithmetic.

Lemma 2.5.1 *(Floating-Point Real Arithmetic) If a and b are floating point numbers and $\Box$ is a binary operator, then by the model of floating-point arithmetic in Golub and Van Loan (1983) we have*

$$fl(a\Box b) = (a\Box b)(1+\epsilon), \qquad |\epsilon| \le \mathbf{u}$$

where

$$\frac{|fl(a\Box b) - (a\Box b)|}{|a\Box b|} \le \mathbf{u}, \qquad a\Box b \ne 0$$

A complex number x is represented by the ordered pair

$$x = (x_R, x_I).$$

Let z be a floating point complex number and $\hat{z}$ be the computed quantity for z. Let $\mathbf{u}$ be a machine dependent constant, floating point machine precision. Then the difference between $\hat{z}$ and z can be quantified by $\delta(z)$ where

$$|\hat{z} - z| \le \delta(z)\mathbf{u} + O(\mathbf{u}^2).$$

We use this function $\delta(z)$ to keep an account of the amount of error accumulated during a series of floating point computations leading up to z.

First we establish the model for floating-point complex arithmetic as it relates to addition and multiplication.

Lemma 2.5.2 *(Floating Point Complex Addition) If $x = (x_R, x_I)$ and $y = (y_R, y_I)$ are floating-point complex numbers and if $\hat{z} = fl(x+y)$, then*

$$|\hat{z} - z| \le \mathbf{u}\delta(z)$$

where $\delta(z) = |z|$.

Proof

$$\begin{aligned}
\hat{z} &= fl(x+y) = fl(x_R + y_R) + i fl(x_I + y_I) \\
&= (x_R + y_R)(1+\epsilon_R) + i(x_I + y_I)(1+\epsilon_I) \\
&= z + \epsilon_R(x_R + y_R) + i\epsilon_I(x_I + y_I) \\
& \quad |\epsilon_R|, |\epsilon_I| \le \mathbf{u}
\end{aligned}$$

hence

$$\begin{aligned}|\hat{z}-z|^2 &\leq |\epsilon_R|^2|x_R+y_R|^2+|\epsilon_I|^2|x_I+y_I|^2\\ &\leq \mathbf{u}^2(|x+y|^2)\end{aligned}$$

———*———

Lemma 2.5.3 *(Floating-Point Complex Multiplication) If $x=(x_R,x_I)$ and $y=(y_R,y_I)$ are floating-point complex numbers and if $\hat{z}=fl(xy)$, then*

$$|\hat{z}-z|\leq \mathbf{u}\delta(z)+O(\mathbf{u}^2)$$

where $\delta(z)=2|z|$.

Proof

$$\begin{aligned}\hat{z} &= fl(xy)=[x_Ry_R(1+\delta_1)-x_Iy_I(1+\delta_2)](1+\epsilon_R)\\ &\quad +i[x_Iy_R(1+\delta_3)+x_Ry_I(1+\delta_4)](1+\epsilon_I)\\ &= x_Ry_R-x_Iy_I+\delta_1x_Ry_R-\delta_2x_Iy_I+\epsilon_R(x_Ry_R-x_Iy_I)\\ &\quad +i[x_Iy_R+x_Ry_I]+i\delta_3x_Iy_R+i\delta_4x_Ry_I+i\epsilon_I(x_Iy_R+x_Ry_I)\\ &\quad |\delta_1|,|\delta_2|,|\delta_3|,|\delta_4|,|\epsilon_R|,|\epsilon_I|\leq \mathbf{u}\end{aligned}$$

Recall that the norm of a complex number $z=(z_R,z_I)$ is equal to the two-norm of the two-vector $\begin{bmatrix} z_R \\ z_I\end{bmatrix}$, hence

$$\begin{aligned}\left\|\begin{bmatrix} Re[\hat{z}] \\ Im[\hat{z}]\end{bmatrix}-\begin{bmatrix} Re[z] \\ Im[z]\end{bmatrix}\right\|_2 &\leq \left\|\begin{bmatrix} \delta_1x_R \\ \delta_3x_I\end{bmatrix}y_R\right\|_2+\left\|\begin{bmatrix} -\delta_2x_I \\ \delta_4x_R\end{bmatrix}y_I\right\|_2\\ &\quad +\left\|\begin{bmatrix} \epsilon_R(x_Ry_R-x_Iy_I) \\ \epsilon_I(x_Iy_R+x_Ry_I)\end{bmatrix}\right\|_2\\ &\leq \mathbf{u}|x||y_R|+\mathbf{u}|x||y_I|+\mathbf{u}|z|\\ &\leq \mathbf{u}|x|\sqrt{2}|y|+\mathbf{u}|z|\\ &= (1+\sqrt{2})\mathbf{u}|z|\end{aligned}$$

or $|\hat{z}-z|\leq(1+\sqrt{2})\mathbf{u}|z|+O(\mathbf{u}^2)$ and

$$|\hat{z}-z|\leq\delta(z)\mathbf{u}+O(\mathbf{u}^2)$$

with $\delta(z)=(1+\sqrt{2})|z|$.

——*——

Having defined the bound $\delta(z)$, we now need to know how they accumulate from previous floating-point computations. For example, given computed quantities $\hat{z}_1$ and $\hat{z}_2$ with discrepancies $\delta(z_1)$ and $\delta(z_2)$, respectively; the quantity $\hat{z} = fl(\hat{z}_1 \Box \hat{z}_2)$ has a new discrepancy $\delta(z)$ derived from discrepancies of the two terms and a contribution from the operation $\Box$ performed.

Lemma 2.5.4 *(Compounding Errors—Addition) If $z = z_1 + z_2$ and $\hat{z} = fl(\hat{z}_1 + \hat{z}_2)$ where*

$$\begin{aligned}|\hat{z}_1 - z_1| &\leq \delta(z_1)\mathbf{u} + O(\mathbf{u}^2)\\ |\hat{z}_2 - z_2| &\leq \delta(z_2)\mathbf{u} + O(\mathbf{u}^2)\end{aligned}$$

then

$$|\hat{z} - z| \leq \delta(z)\mathbf{u} + O(\mathbf{u}^2)$$

where

$$\delta(z) = |z| + \delta(z_1) + \delta(z_2)$$

Proof

$$|\hat{z} - (z_1 + z_2)| \leq |\hat{z} - (\hat{z}_1 + \hat{z}_2)| + |\hat{z}_1 + \hat{z}_2 - (z_1 + z_2)|$$

we know from the lemma on complex addition that the first term on the right satisfies

$$\begin{aligned}|\hat{z} - (\hat{z}_1 + \hat{z}_2)| &\leq |\hat{z}_1 + \hat{z}_2|\mathbf{u} + O(\mathbf{u}^2)\\ &= |z|\mathbf{u} + O(\mathbf{u}^2)\end{aligned}$$

so

$$\begin{aligned}|\hat{z} - (z_1 + z_2)| &\leq |\hat{z} - (\hat{z}_1 + \hat{z}_2)| + |\hat{z}_1 - z_1| + |\hat{z}_2 - z_2|\\ &\leq (|z| + \delta(z_1) + \delta(z_2))\mathbf{u} + O(\mathbf{u}^2)\end{aligned}$$

——*——

The corresponding result for multiplication is described in the next lemma.

Lemma 2.5.5 *(Compounding Errors—Multiplication) Suppose $z = z_1 z_2$ and $\hat{z} = fl(\hat{z}_1 \hat{z}_2)$ where*

$$\begin{aligned}|\hat{z}_1 - z_1| &\leq \delta(z_1)\mathbf{u} + O(\mathbf{u}^2)\\ |\hat{z}_2 - z_2| &\leq \delta(z_2)\mathbf{u} + O(\mathbf{u}^2)\end{aligned}$$

then

$$|\hat{z} - z| \leq \delta(z)\mathbf{u} + O(\mathbf{u}^2)$$

where

$$\delta(z) = (1 + \sqrt{2})|z| + \delta(z_1)|z_2| + \delta(z_2)|z_1|$$

Proof

$$|\hat{z} - (z_1 z_2)| \leq |\hat{z} - (\hat{z}_1 \hat{z}_2)| + |\hat{z}_1 \hat{z}_2 - z_1 z_2|$$

From the lemma on complex multiplication, the first term on the right satisfies

$$\begin{aligned} |\hat{z} - \hat{z}_1 \hat{z}_2| &\leq (1 + \sqrt{2})|\hat{z}_1 \hat{z}_2|\mathbf{u} + O(\mathbf{u}^2) \\ &= (1 + \sqrt{2})|z|\mathbf{u} + O(\mathbf{u}^2) \end{aligned}$$

and

$$|\hat{z} - (z_1 z_2)| \leq |\hat{z} - (\hat{z}_1 \hat{z}_2)| + |\hat{z}_1||\hat{z}_2 - z_2| + |z_2||\hat{z}_1 - z_1|$$

or with $|\hat{z}_1| \approx |z_1|$ we get

$$|\hat{z} - (z_1 z_2)| \leq [(1 + \sqrt{2})|z| + |\delta(z_1)||z_2| + |\delta(z_2)||z_1|]\mathbf{u} + O(\mathbf{u}^2)$$

——*——

Butterfly Errors

We present two separate error analyses for the radix-2 FFT, one using the Cooley-Tukey (CT2) procedure and the second one the Gentleman-Sande (GS1) algorithm. In order to track the effect of multiplier computations we measure the error of each component of the transform with a bound given by the infinity norm of the starting vector. This type of analysis, emphasizing the error of each component, highlights the different error properties of the Cooley-Tukey algorithm versus that of the Gentleman-Sande algorithm. Although the order of magnitude between the two methods are equivalent, differences in the individual components result from the differences between the algorithms.

Recall that the CT2-FFT computes the DFT

$$\mathbf{x} \leftarrow \frac{1}{n}\mathbf{F}_n \mathbf{P}_n \mathbf{x},$$

and the CT2 algorithm can be written conveniently as follows,

```
Algorithm 2.5.1    $x \leftarrow P_n x$
    for $k = 1 : t$
      $x \leftarrow A_k x$
    end
    $x \leftarrow \frac{1}{n} x$
```

where $\mathbf{A}_k = \mathbf{I}_{n/L} \otimes \mathbf{B}_L$, $L = 2^k$ and

$$\mathbf{B}_L = \begin{bmatrix} \mathbf{I}_{L/2} & \mathbf{\Delta}_{L/2} \\ \mathbf{I}_{L/2} & -\mathbf{\Delta}_{L/2} \end{bmatrix}$$

and $\mathbf{\Delta}_{L/2} = diag(1, \omega_L, \ldots, \omega_L^{L/2-1})$. Each $\mathbf{B}_L$ has exactly two nonzero elements per row and two nonzero elements per column. Hence $\mathbf{B}_L$ is composed of independent 2-by-2 matrices superimposed on each other. We call these 2-by-2 matrices CT2 Butterfly matrices and denote the generic CT2 butterfly by

$$\mathbf{B}_\omega = \begin{bmatrix} 1 & \omega \\ 1 & -\omega \end{bmatrix}$$

where we do not worry what ω is except that it is a root of unity. We first consider the error analysis of a generic CT2 Butterfly $\mathbf{B}_\omega$.

The following lemma tells how the supremum norm of a vector grows with application of the FFT Butterfly matrices.

Lemma 2.5.6 *At each step of the FFT,*

$$||\mathbf{x}^{(k)}||_\infty \leq 2^k ||\mathbf{x}^{(0)}||_\infty$$

Proof Observe that $\mathbf{B}_\omega^T \mathbf{B}_\omega = 2\mathbf{I}$ and hence

$$||\mathbf{B}_\omega \mathbf{x}||_\infty \leq ||\mathbf{B}_\omega \mathbf{x}||_2 = \sqrt{2}||\mathbf{x}||_2 \leq 2||\mathbf{x}||_\infty$$

and $\mathbf{A}$ is composed of independent butterflies $\mathbf{B}_\omega$ so

$$||\mathbf{A}\mathbf{x}|| \leq 2||\mathbf{x}||_\infty$$

——*——

We start with the radix-2 FFT error analysis. First we look at the most simple butterfly operations for a 2-vector $\mathbf{x} = [x_T, x_B]^T$.

Lemma 2.5.7 *(CT2 Butterfly Error Analysis)* *Let*

$$\mathbf{B}_\omega = \begin{bmatrix} 1 & \omega \\ 1 & -\omega \end{bmatrix} \qquad \mathbf{E} = \begin{bmatrix} 0 & \delta(\omega)\mathbf{u} \\ 0 & -\delta(\omega)\mathbf{u} \end{bmatrix} \qquad \hat{\mathbf{x}} = [\hat{x}_T, \hat{x}_B]^T$$

with $|\mathbf{E}| \leq \begin{bmatrix} 0 & \delta(\omega) \\ 0 & \delta(\omega) \end{bmatrix} \mathbf{u}$

$$\begin{aligned} |\hat{x}_T - x_T| &\leq \delta(x_T)\mathbf{u} + O(\mathbf{u}^2) \\ |\hat{x}_B - x_B| &\leq \delta(x_B)\mathbf{u} + O(\mathbf{u}^2) \\ |\hat{\omega} - \omega| &\leq \delta(\omega)\mathbf{u} + O(\mathbf{u}^2) \end{aligned}$$

and $\hat{\mathbf{B}} = \mathbf{B} + \mathbf{E}$. *Then if*

$$\mathbf{x}^+ = \begin{bmatrix} x_T^+ \\ x_B^+ \end{bmatrix} = \mathbf{B}\mathbf{x}$$

and

$$\hat{\mathbf{x}}^+ = fl(\hat{\mathbf{B}}\hat{\mathbf{x}})$$

$$\begin{aligned} |\delta(x_T^+)| &\leq \|\mathbf{x}^+\|_\infty + (1+\sqrt{2})\|\mathbf{x}\|_\infty + \|\mathbf{x}\|_\infty|\delta(\omega)| + |\delta(x_T)| + |\delta(x_B)| \\ |\delta(x_B^+)| &\leq \|\mathbf{x}^+\|_\infty + (1+\sqrt{2})\|\mathbf{x}\|_\infty + \|\mathbf{x}\|_\infty|\delta(\omega)| + |\delta(x_T)| + |\delta(x_B)| \end{aligned}$$

Proof The CT2 butterfly is defined by

$$\begin{aligned} \hat{x}_T^+ &\leftarrow fl(\hat{x}_T + fl(\hat{\omega}\hat{x}_B)) \\ \hat{x}_B^+ &\leftarrow fl(\hat{x}_T - fl(\hat{\omega}\hat{x}_B)) \end{aligned}$$

First we have

$$|fl(\hat{\omega}\hat{x}_B) - \omega x_B| \leq [(1+\sqrt{2})|\omega x_B| + |x_B|\delta(\omega) + |\omega|\delta(x_B)]\mathbf{u} + O(\mathbf{u}^2)$$

We know that $|\omega| = 1$ so this becomes

$$\delta(\omega x_B) = (1+\sqrt{2})|x_B| + |x_B||\delta(\omega)| + |\delta(x_B)|$$

The lemma on addition shows us that

$$\delta(x_T^+) \leq |x_T^+| + |\delta(x_T)| + (1+\sqrt{2})|x_B| + |\delta(\omega)||x_B| + |\delta(x_B)|$$

which is bounded by replacing $|x_T^+|$ and $|x_B|$ by the sup norms,

$$\delta(x_T^+) \leq \|\mathbf{x}^+\|_\infty + (1+\sqrt{2})\|\mathbf{x}\|_\infty + \|\mathbf{x}\|_\infty|\delta(\omega)| + |\delta(x_T)| + |\delta(x_B)|$$

A similar analysis gives the same bound for $\delta(x_B^+)$.

———*———

This shows that a current component is dependent upon the errors incurred in the previous step added to the multiplier error and the current roundoff. Hence to do an error analysis, one must be able to "track" the errors and "know" which multipliers are used in all the previous steps in computations leading up to the current one. For the two FFT variants with multipliers applied in natural order we have explicit formulae for determining which multiplier was applied to which component at each stage. Furthermore for the decimation in time algorithm (CT2) we also know that both components involved in a particular butterfly have *identical* multiplier histories.

Theorem 2.5.1 *(Multiplier History—CT2) If $x_j^{(k)}$ stands for the jth component of* $\mathbf{x}$ *at stage k of a radix-2 FFT of length $n = 2^t$, then*

$$\begin{aligned} x_j^{(k)} &= x_j^{(k-1)} + \omega(m_j^{(k)})x_{j'}^{(k-1)} \\ x_{j'}^{(k)} &= x_j^{(k-1)} - \omega(m_j^{(k)})x_{j'}^{(k-1)} \end{aligned}$$

where $j' = j + 2^{k-1}$ and $m_j^{(k)}$ is the shorthand that points to the error coefficient of the multiplier used at step k on x_j, then

$$\{m_j^{(1)}, \ldots, m_j^{(k-1)}, m_j^{(k)}\} = \{m_{j'}^{(1)}, \ldots, m_{j'}^{(k-1)}, m_{j'}^{(k)}\}$$

i.e. both $x_j^{(k)}$ and $x_{j'}^{(k)}$ have the same sequence of multipliers.

Proof It is obvious that $m_j^{(k)} = m_{j'}^{(k)}$ so we concentrate on the $k-1$ previous steps. j and $j' = j + 2^{k-1}$ have the property that

$$j \bmod 2^q = j' \bmod 2^q, \qquad q = 1, \ldots, k-1$$

Since the CT2 FFT algorithm has at step k, $n/2^k$ ***identical*** sets of computations of size 2^k, this means that j and j' occupied the identical position in the separate independent sets of computations for all the previous steps. Hence the *exact* same set of multipliers had to have been applied to both of them.

———*———

Theorem 2.5.2 *(CT2: Error Analysis) Let* $\mathbf{x}^{(0)}$ *be a complex vector of length $n = 2^t$ with components* $\mathbf{x}_j^{(0)}$ *such that*

$$|\hat{x}_j^{(0)} - x_j^{(0)}| \le |x_j^{(0)}|\mathbf{u} + O(\mathbf{u}^2)$$

that is, $\delta(x_j^{(0)}) = |x_j^{(0)}| \leq \|\mathbf{x}^{(0)}\|_\infty$. *If* $\mathbf{x}^{(t)} = \frac{1}{n}\mathbf{F}_n\mathbf{x}^{(0)}$ *is the result of a radix-2 CT2 FFT procedure, then*

$$|\widehat{x}_j^{(t)} - x_j^{(t)}| \leq [1 + t(1 + (1+\sqrt{2})/2) + \frac{1}{2}\sum_{k=1}^{t} h_{m_j^{(k)}}]\|\mathbf{x}^{(0)}\|_\infty \mathbf{u} + O(\mathbf{u}^2)$$

Proof For the general term we have

$$\delta(x_j^+) \leq \|\mathbf{x}^+\|_\infty + 2\|\mathbf{x}\|_\infty + \|\mathbf{x}\|_\infty|\delta(\omega)| + |\delta(x_T)| + |\delta(x_B)|$$

and at a specific stage k for a specific x_j we know from the CT2 multiplier history that the error of the multiplier is $h_{m_j^{(k)}}$. Therefore we have

$$\begin{aligned} |\delta(x_j^{(k)})| \;\leq\; & \|\mathbf{x}^{(k)}\|_\infty + (1+\sqrt{2})\|\mathbf{x}^{(k-1)}\|_\infty \\ & + \|\mathbf{x}^{(k-1)}\|_\infty|h_{m_j^{(k)}}| + |\delta(x_j^{(k-1)})| + |\delta(x_{j'}^{(k-1)})| \end{aligned}$$

where $j' = j \pm 2^{k-1}$ (it does not matter). Lemma 2.5.6 tells us that $\|\mathbf{x}^{(k)}\|_\infty \leq 2^k\|\mathbf{x}^{(0)}\|_\infty$ hence

$$\begin{aligned} |\delta(x_j^{(k)})| \;\leq\; & 2^k\|\mathbf{x}^{(0)}\|_\infty + 2^{k-1}(1+\sqrt{2})\|\mathbf{x}^{(0)}\|_\infty + 2^{k-1}\|\mathbf{x}^{(0)}\|_\infty|h_{m_j^{(k)}}| \\ & + |\delta(x_j^{(k-1)})| + |\delta(x_{j'}^{(k-1)})| \\ \leq\; & 2^k\|\mathbf{x}^{(0)}\|_\infty + (1+\sqrt{2})2^{k-1}\|\mathbf{x}^{(0)}\|_\infty 2^{k-1}\|\mathbf{x}^{(0)}\|_\infty|h_{m_j^{(k)}}| \\ & + 2|\delta(x_j^{(k-1)})| \end{aligned}$$

The last term $2|\delta(x_j^{(k-1)})|$ is justified because $|\delta(x_j^{(k-1)}|$ and $|\delta(x_{j'}^{(k-1)})|$ are both bounded by the same quantity. Theorem 2.5.1 tells us that they are the result of the exact sequence of multipliers—the sequence of $h_{m_j^{(k)}}$ are identical. Furthermore, the infinity norm is used in the bound so that although $x_j^{(k-1)}$ and $x_{j'}^{(k-1)}$ are quantitatively different, their error characteristics with respect to multiplier applications are the same. Therefore it makes sense to recurse on

$$\begin{aligned} |\delta(x_j^{(k)})| \;\leq\; & 2^k\|\mathbf{x}^{(0)}\|_\infty + (1+\sqrt{2})2^{k-1}\|\mathbf{x}^{(0)}\|_\infty + 2^{k-1}\|\mathbf{x}^{(0)}\|_\infty|h_{m_j^{(k)}}| \\ & + 2|\delta(x_j^{(k-1)})| \end{aligned}$$

to get

$$\begin{aligned} |\delta(\tilde{x}_j^{(t)})| \;\leq\; & t2^t\|\mathbf{x}^{(0)}\|_\infty + (1+\sqrt{2})t2^{t-1}\|\mathbf{x}^{(0)}\|_\infty \\ & + 2^{t-1}\|\mathbf{x}^{(0)}\|_\infty \sum_{k=1}^{t}|h_{m_j^{(k)}}| + 2^t\|\mathbf{x}^{(0)}\|_\infty \end{aligned}$$

The final scaling by $\frac{1}{n} = \frac{1}{2^t}$ gives $x_j^{(t)} \leftarrow \frac{1}{n}\tilde{x}_j^{(t)}$ and

$$|\delta(x_j^{(t)})| \leq t(1 + \frac{(1+\sqrt{2})}{2})\|\mathbf{x}^{(0)}\|_\infty + \frac{1}{2}\|\mathbf{x}^{(0)}\|_\infty \sum_{k=1}^{t} h_{m_j^{(k)}} + \|\mathbf{x}^{(0)}\|_\infty$$

——*——

This shows that the errors $h_{m_j^{(k)}}$ in the multiplier computations is represented in the same proportion as the roundoff error. Our bound is consistent with the result of Ramos (1971) if we let all the $h_{m_j^{(k)}} = \gamma \geq 0$, an absolute error constant. Hence the error is of $O(\log_2 n + \sum_{k=1}^{\log_2 n} h_{m_j^{(k)}})$.

We next examine the radix-2 GS1 FFT algorithm and note a few facts. From the properties of the GS1 algorithm, we notice that each x_j has the same sequence of multipliers applied to it as in the CT2 algorithm, except in reverse order, and with one difference. If at step k, the $(k-1)$st bit of j's binary representation is equal to 0, then no multipliers were applied because x_j is the top member of the butterfly and hence created by a simple addition. Since we are interested in bounds on the *maximum* component error, we shall first look at the error analysis for x_j such that j's binary representation has nonzeros in every bit. Later we can say that if j's binary representation has a zero in position $k-1$, then the contribution of that particular multiplier is deducted.

The radix-2 GS1 algorithm has a butterfly which is the transpose of that for the radix-2 CT2 FFT. As in lemma 2.5.7, we have the following.

Lemma 2.5.8 *Let*

$$\mathbf{B}^T = \begin{bmatrix} 1 & 1 \\ \omega & -\omega \end{bmatrix} \qquad \mathbf{E}^T = \begin{bmatrix} 0 & 0 \\ \delta(\omega)\mathbf{u} & -\delta(\omega)\mathbf{u} \end{bmatrix} \qquad \hat{\mathbf{x}} = [\hat{x}_T, \hat{x}_B]^T$$

$$\begin{aligned} |\hat{x}_T - x_T| &\leq \delta(x_T)\mathbf{u} + O(\mathbf{u}^2) \\ |\hat{x}_B - x_B| &\leq \delta(x_B)\mathbf{u} + O(\mathbf{u}^2) \\ |\hat{\omega} - \omega| &\leq \delta(\omega)\mathbf{u} + O(\mathbf{u}^2) \end{aligned}$$

and $\hat{\mathbf{B}}^T = \mathbf{B}^T + \mathbf{E}^T$. *Let*

$$\mathbf{x}^+ = \begin{bmatrix} x_T^+ \\ x_B^+ \end{bmatrix} = \mathbf{B}^T\mathbf{x}$$

the maximum absolute error in $\hat{x}_T^+$ *and* $\hat{x}_B^+$ *is*

$$\begin{aligned} |\delta(x_T^+)| &\leq \|\mathbf{x}^+\|_\infty + |\delta(x_T)| + |\delta(x_B)| \\ |\delta(x_B^+)| &\leq (1+\sqrt{2})\|\mathbf{x}^+\|_\infty + (1+\sqrt{2})\|\mathbf{x}\|_\infty|\delta(\omega)| \\ &\quad +(1+\sqrt{2})\|\mathbf{x}\|_\infty + |\delta(x_T)| + |\delta(x_B)| \end{aligned}$$

Proof The GS1 butterfly is defined by

$$\begin{aligned} \hat{x}_T^+ &\leftarrow fl(\hat{x}_T + \hat{x}_B) \\ \hat{x}_B^+ &\leftarrow fl(\hat{\omega} fl(\hat{x}_T - \hat{x}_B)) \end{aligned}$$

The x_T^+ term requires only a simple addition and hence

$$\begin{aligned} \delta(x_T^+) &= |x_T^+| + |\delta(x_T)| + |\delta(x_B)| \\ &\leq \|\mathbf{x}^+\|_\infty + |\delta(x_T)| + |\delta(x_B)| \end{aligned}$$

The x_B^+ term has

$$\begin{aligned} \delta(x_B^+) &= 2|x_B^+| + |\delta(\omega)||x_T - x_B| + [|x_T - x_B| + |\delta(x_T)| + |\delta(x_B)|]|\omega| \\ &\leq (1+\sqrt{2})\|\mathbf{x}^+\|_\infty + (1+\sqrt{2})\|\mathbf{x}\|_\infty|\delta(\omega)| \\ &\quad +(1+\sqrt{2})\|\mathbf{x}\|_\infty + |\delta(x_T)| + |\delta(x_B)| \end{aligned}$$

——*——

Notice that the error bound for x_B^+ in lemma 2.5.8 is similar to the error bound for x_B^+ in lemma 2.5.7. Therefore as far as maximum absolute error is concerned the radix-2 GS1 and CT2 algorithms are almost equivalent. This was shown already in Tsao (unpublished).

The Multiplier History of the GS1 Algorithm is more complicated that that of the Cooley-Tukey Algorithm.

Theorem 2.5.3 *(Multiplier History—GS1) If $x_j^{(k)}$ stands for the jth component of x at stage k of a radix-2 FFT of length $n = 2^t$ then*

$$\begin{aligned} x_j^{(k)} &= x_j^{(k-1)} + x_{j'}^{(k-1)} \\ x_{j'}^{(k)} &= \omega(m_{j'}^{(k)})(x_j^{(k-1)} - x_{j'}^{(k-1)}) \end{aligned}$$

where $j' = j + 2^{k-1}$ and $m_j^{(k)}$ points to the multiplier used on x_j at stage k then letting $M_j^{(k)}$ be the set of all indices used in the process of creating $x_j^{(k)}$ we have

$$\begin{aligned} M_j^{(k)} &= M_j^{(k-1)} \cup M_{j'}^{(k-1)} \\ M_{j'}^{(k)} &= M_j^{(k-1)} \cup M_{j'}^{(k-1)} \cup \{m_{j'}^{(k)}\} \end{aligned}$$

Proof The GS butterfly shows that computing $x_j^{(k)}$ requires no multiplications. Therefore the multiplier history at this step is simply the union of the multiplier history $M_j^{(k-1)}$ and $M_{j'}^{(k-1)}$. The computation of $x_{j'}^{(k)}$ requires one new multiplication, hence the result.

——*——

The imbalance in the way multipliers are applied means that in the step in the error analysis where we wish to recurse on the $|\delta(x_j^{(k-1)})|$ and $|\delta(x_{j'}^{(k-1)})|$ where $j' = j + 2^{k-1}$, we cannot assume that they are basically equivalent. This is because at each butterfly step, the top element is a pure and simple addition with no multiplier applied to it. Therefore we concentrate our error bound on the element with the most complicated set of multipliers, the element x_j where $j = 11 \cdots 1_2$. To say something about the other components we use the following lemma.

Lemma 2.5.9 *In every GS butterfly between $x_j^{(k-1)}$ and $x_{j'}^{(k-1)}$ where the $(k-1)$st bit of j is 0 and the $(k-1)$st bit of j' is 1,*

$$|\delta(x_j^{(k)})| \geq |\delta(x_j^{(k)})|$$

Proof If j' has a 1 at position $(k-1)$ then $x_{j'}^{(k-1)}$ is the x_B for that butterfly step. Similarly x_j, with j having a 0 at position $(k-1)$, is the x_T. Now using Lemma 2.5.8 we see that

$$|\delta(x_B^{+})| \geq |\delta(x_T^{+})|$$

——*——

The theorem on the Gentleman-Sande Error Analysis is therefore basically a theorem about the last component.

Theorem 2.5.4 *(GS1: Error Analysis). Let $\mathbf{x}^{(0)}$ be a complex vector of length $n = 2^t$ with components $x_j^{(0)}$ such that*

$$|\hat{x}_j^{(0)} - x_j^{(0)}| \leq |x_j^{(0)}|\mathbf{u} + O(\mathbf{u}^2)$$

that is $|\delta(x_j^{(0)})| = |x_j^{(0)}| \leq \|\mathbf{x}^{(0)}\|_\infty$. If $\mathbf{x}^{(t)} = \frac{1}{n}\mathbf{F}_n\mathbf{x}^{(0)}$ is the result of a radix-2 GS1 FFT procedure, then

$$|\hat{x}_j^{(t)} - x_j^{(t)}| \leq [\frac{3}{2}t(1+\sqrt{2}) + 1 + \frac{(1+\sqrt{2})}{2}\sum_{k=1}^{t} h_{m_j^{(k)}}]\|\mathbf{x}^{(0)}\|_\infty \mathbf{u} + O(\mathbf{u}^2)$$

Proof Concentrating on the x_j term where $j = 11 \cdots 1_2 = 2^t - 1$, we have

$$\begin{aligned} |\delta(x_j^+)| &\leq (1+\sqrt{2})\|\mathbf{x}^+\|_\infty + (1+\sqrt{2})\|\mathbf{x}\|_\infty|\delta(\omega)| \\ &\quad +(1+\sqrt{2})\|\mathbf{x}\|_\infty + |\delta(x_T)| + |\delta(x_B)| \end{aligned}$$

or

$$\begin{aligned} |\delta(x_j^{(k)})| &\leq 2^k(1+\sqrt{2})\|\mathbf{x}^{(0)}\|_\infty + 2^{k-1}(1+\sqrt{2})\|\mathbf{x}^{(0)}\|_\infty|h_{m_j^{(k)}}| \\ &\quad +2^{k-1}(1+\sqrt{2})\|\mathbf{x}^{(0)}\|_\infty + |2\delta(x_j^{(k-1)})| \end{aligned}$$

recursing on this gives

$$\begin{aligned} |\delta(\tilde{x}_j^{(t)})| &\leq t(1+\sqrt{2})2^t\|\mathbf{x}^{(0)}\|_\infty + 2^{t-1}(1+\sqrt{2})\|\mathbf{x}^{(0)}\|_\infty \sum_{k=1}^{t} h_{m_j^{(k)}} \\ &\quad +t2^{t-1}(1+\sqrt{2})\|\mathbf{x}^{(0)}\|_\infty + 2^t\|\mathbf{x}^{(0)}\|_\infty \end{aligned}$$

Scaling by $\frac{1}{n} = \frac{1}{2^t}$ gives us

$$|\delta(x_j^{(t)})| \leq \frac{3}{2}t(1+\sqrt{2})\|\mathbf{x}^{(0)}\|_\infty + \frac{(1+\sqrt{2})}{2}\|\mathbf{x}^{(0)}\|_\infty \sum_{k=1}^{t} |h_{m_j^{(k)}}| + \|\mathbf{x}^{(0)}\|_\infty$$

——*——

We now incorporate the different methods of computing multipliers with their error properties into the error analysis for the radix-2 FFTs. First we summarize the error bounds in Table 2.4. We will only be concerned about the order of magnitude of the errors because individual errors of the different components vary too much. The main question we have addressed is how the errors in the multipliers transfer themselves to the errors in the FFT. It turns out that they do not become suppressed in the FFT, i.e. a multiplier computational method of $O(\log_2 n)$ combines multiplicatively to the FFT error to become $O(\log_2 n)^2$. This underscores why it is important to consider the accuracy of the multiplier computation.

Next we compute the order of the maximum error for the six methods. Let $n = 2^t$ in each of the following. Direct Call is the simplest.

Direct Call: $\|\hat{\mathbf{x}} - \mathbf{x}\|_\infty \leq O(t)\mathbf{u}$.

Recall that Forward Recursion and Repeated Multiplication start from a different initial argument at each stage. Hence the maximum index used at each stage is $j = 2^{k-1} - 1$.

Table 2.4: Error Bounds for Weight Computational Methods

Method	Bound
Direct Call	$h_j^{DC} = 1$
Forward Recursion	$h_j^{FR} = (\|c_1\| + \sqrt{\|c_1\|^2 + 1})^j$
Repeated Multiplication	$h_j^{RM} = 2j$
Logarithmic Recursion	$h_j^{LR} \leq (\|c_1\| + \sqrt{\|c_1\|^2 + 1})^{\log_2 j}$
Subvector Scaling	$h_j^{SS} \leq 2 \log_2 j$
Recursive Bisection	$h_j^{RB} \leq 2 \log_2 j$

Forward Recursion: First we notice that $|c_1| + \sqrt{|c_1|^2 + 1} \geq 2|c_1|$, however to simplify the order of magnitude, we can plug in $2|c_1|$ into $max(h_{m_j^{(k)}}) = (2|c_1|) * *(2^{k-1} - 1)$ into the error bound we get,

$$\|\hat{\mathbf{x}} - \mathbf{x}\|_\infty \leq O(t2^{n/2})\mathbf{u}.$$

Repeated Multiplication: Here $max(h_{m_j^{(k)}}) = 2(2^{k-1} - 1)$, hence

$$\|\hat{\mathbf{x}} - \mathbf{x}\|_\infty \leq O(tn)\mathbf{u}.$$

The following three methods all have the maximum index j at each step equal to $2^{t-1} - 1$, hence the maximum $\log_2 j$ is approximately $t - 1$.

Logarithmic Recursion: Here $max(h_{m_j^{(k)}}) \leq (2|c_1|) * *(t - 1)$, so

$$\|\hat{\mathbf{x}} - \mathbf{x}\|_\infty \leq O(tn)\mathbf{u}.$$

For Subvector Scaling and Recursive Bisection, $max(h_{m_j^{(k)}}) \leq 2(t - 1)$, and thus

Subvector Scaling: $\|\hat{\mathbf{x}} - \mathbf{x}\|_\infty \leq O(t^2)\mathbf{u}$.

Recursive Bisection: $\|\hat{\mathbf{x}} - \mathbf{x}\|_\infty \leq O(t^2)\mathbf{u}$.

Therefore, we see that although the ideal error bound for the FFT is $O(t)\mathbf{u}$, this is achieved only through the use of Direct Call. This is counterintuitive and shows that the order of the error bounds for the multipliers affects the total FFT error bounds multiplicatively instead of additively. This does not contradict the result of Ramos (1971) because in his case, the error bounds for the multipliers are constant and hence are of the same order as the roundoff error.

Double precision computation of the multipliers can be used for the two $O(t^2)\mathbf{u}$ methods and the $O(n)\mathbf{u}$ method, whenever $n\tilde{\mathbf{u}} \leq 1$, where $\tilde{\mathbf{u}}$ is the double precision machine precision. This is why double precision Repeated Multiplication can be used for practical implementations requiring speed and low storage requirements. [Press *et al.* (pp. 394–395, 1986)]

We summarize our novel approach to FFT error analysis with respect to multiplier computation and give some conclusions.

- We verified that errors in computing the multipliers are represented in the same proportion as the roundoff error if the error bound for the multipliers is an absolute constant.

- The order of the error incurred in the computation of the multipliers is multiplied by $t = \log_2 n$ in the final error bound of the FFT. Hence methods that bound the errors in the multipliers by $O(t)\mathbf{u}$ actually have a total error bound for the FFT of $O(t^2)\mathbf{u}$.

- Using double precision in the Repeated Multiplication method, is accurate as long as $n\tilde{\mathbf{u}} \leq 1$. Therefore the double precision, covers up for the $O(n)\tilde{\mathbf{u}}$ absolute error in the multipliers, making them effectively 0.

2.6 Practical Multiplier Generation

The six methods for the generation of FFT multipliers differ in both their numerical characteristics and their implementation properties. There are two ways in which FFT multipliers can be used. They can either be pre-computed and stored in a table as Algorithms 2.3.2– 2.3.7 show, or the particular multiplier needed at each step can be computed right before it is used. We call the latter "just-in-time" computation.

Not all of our methods are amenable to just-in-time computation. For example, Logarithmic Recursion and Subvector Scaling requires storage space of $O(n)$ for the $n/2$ multipliers needed for the FFT of length $n = 2^t$. This is because at each stage, the next set of multipliers computed is a "scaling" of the ones previously computed. On the other hand, the method of Direct Call can compute multipliers on demand without need for any table storage. After all, when a cosine of a particular angle is needed, it is obtained by a direct call to the library function COS. The other three methods, Forward Recursion, Repeated Multiplication, and Recursive Bisection vary in the amount of storage required for just-in-time computation.

The basic equations for Forward Recursion are

$$\cos(j\theta) \quad = \quad 2\cos(\theta)\cos([j-1]\theta) - \cos([j-2]\theta)$$

$$\sin(j\theta) \quad = \quad 2\cos(\theta)\sin([j-1]\theta) - \sin([j-2]\theta)$$

Hence six real numbers need to be stored for each iteration.

Algorithm 2.6.1 Forward Recursion *(just-in-time)*

```
$\theta = 2\pi/n$
$c_p \leftarrow 1;\ s_p \leftarrow 0$
$c_c \leftarrow COS(\theta);\ s_c \leftarrow SIN(\theta)$
$c_t \leftarrow 2c_c$
for $j = 2, \ldots$
  $t \leftarrow c_c$
  $c_c \leftarrow c_t \cdot c_c - c_p$
  $c_p \leftarrow t$
  $t \leftarrow s_c$
  $s_c \leftarrow c_t \cdot s_c - s_p$
end
```

Repeated multiplication requires storage for only two complex numbers since

$$\omega^j \leftarrow \omega \cdot \omega^{j-1}$$

is the basic iteration.

Algorithm 2.6.2 Repeated Multiplication *(just-in-time)*

```
$\theta = 2\pi/n$
$\omega \leftarrow 2\exp(-i\theta)$
$\omega_j \leftarrow \omega$
for $j = 2, \ldots$
  $\omega_j \leftarrow \omega \cdot \omega_j$
end
```

Recursive Bisection is a bit more tricky. The most obvious implementation requires a full table of $O(n)$. This is because the Recursive Bisection algorithm computes a full table of multipliers by first starting with $e^{i\pi/4}$, $e^{i\pi/2} = i$, and $e^{i0} = 1$, then creating $e^{i\pi/8}$, $e^{3\pi i/8}$, and then $e^{i\pi/16}$, $e^{3i\pi/16}$, $e^{5i\pi/16}$, $e^{7i\pi/16}$, and so on. Initially a short table of half-secants, $.5\sec(\pi/4)$, need to be supplied. For example, if $n = 2^t$ and $\theta = 2\pi/n$, then $(t-2)$ half-secants or $O(\log_2 n)$ storage is required. However to go from one step to the next requires all of the previously computed multipliers. So to get the odd angles of $\pi/2^k$, requires one to have available, the multipliers for the angles $\pi/2^{k-1}$. Obviously, memory space for the full table is needed. Furthermore, for the purposes of the FFT, specifically the Gentleman-Sande algorithms, the multipliers are not created in the correct order. Recall that the Gentleman-Sande algorithms start with multipliers of the smallest spacings, i.e. for step k, $k = 1, \ldots, t$, we use powers of $\omega_L = \exp(2\pi i/L)$ where $L = 2^{t-k+1}$. Therefore the spacing of multipliers is exactly the reverse of their creation by Recursive Bisection, where at step k, $k = 1, \ldots, t$, the multipliers ω_L, $L = 2^{k-1}$ are computed. This spacing, however, is consistent with the Cooley-Tukey algorithm where ω_L, $L = 2^{k-1}$ is utilized at step k.

In any case, if spacing and storage considerations are a factor, Recursive Bisection, whether pre-computed or not, requires a table of $O(n)$ to store the multipliers and another table of $O(\log_2 n)$ to store the half-secants.

Buneman's On-Line Generation of Weights

Buneman (1987) proposes a new method of on-line (just-in-time) creation of FFT multipliers with the use of two tables of order $O(\log_2 n)$. The table is dynamically updated and the multipliers are created in the consecutive order, i.e. ω_n, ω_n^2, ω_n^3, etc. An initial table is supplied and holds the cosines and/or sines of the angles 4π, 2π, π, $\pi/2$, $\pi/4$, $\pi/8, \ldots, 2\pi/2^t$ requiring $t+2$ spots. Another table of length $t+1$ contains the half-secants $.5\sec(2\pi)$, $.5\sec(\pi), \ldots, .5\sec(2\pi/2^t)$. The idea is that whenever an entry has been used, it is replaced by a new entry that is to be used at a later step. Just enough information is retained in the table so that the appropriate new entry can be completed by mid-point interpolation.

To describe Buneman's method, we first define a "degree" to be equal to the smallest desired spacing of angles, i.e.

$$\cos j \equiv \cos 2j\pi/2^t$$

so that even though the conventional degree is $\pi/180$, our "degree" is typically $2\pi/2^t$. Our initial table now looks like this

$$\cos 2^{t+1}, \cos 2^t, \cos 2^{t-1}, \ldots, \cos 4, \cos 2, \cos 1$$

or simply, (symbolically),

$$2^{t+1}, 2^t, 2^{t-1}, \ldots, 4, 2, 1$$

These initial values are all powers of two, with each slot in the table reserved for the *odd* multiplier of the power of two held originally. For example, the last slot holds progressively increasing odd degrees, $\cos 1$ replaced by $\cos 3$, replaced by $\cos 5$, etc. The second to the last slot has 2, 6, 10, etc.

A concrete example illustrates the order that multipliers appear in the table. Let $n = 16$ and $2\pi/16$ be the smallest increment, that is a "degree" is equivalent to $\pi/8$. This suffices for a 16-point FFT. Initially, the table holds cosines of

$$4\pi, 2\pi, \pi, \pi/2, \pi/4, \pi/8$$

or degrees

$$32, 16, 8, 4, 2, 1$$

Writing everything in binary, we outline the sequence of events in Table 2.5.

Keeping Table 2.5 in mind, we interpret the sequence of events by first noticing that all entries in the same column have the same number of trailing zeros. This is because they are all *odd* multiples of the same power of 2. If we index the columns in reverse order starting with zero for the last column, we see that the column index is equal to the number of trailing zeros in the binary representation of the angle degree. Each interpolation involves the entry preceeding the entry to be changed and one further back.

Let K be the index of the angle that had just been called, residing in column p. Then K is to be replaced by $K + 2 \cdot 2^p$. Letting $I \equiv 2^p$ we see that $K \leftarrow K + 2I$. The two angles needed for interpolation are $K + I$ and $K + 3I$. Since K is an odd multiple of I, $K + I$ and $K + 3I$ must be odd multiples of $2^q I$, $(q \geq p)$. Therefore $K = (2j + 1)I$ for some j, then $K + I = (j + 1)2I$ and $K + 3I = (j + 2)2I$.

Case 1: j is even. Then $K + I$ is an odd multiple of $2I$ and is therefore found in the entry immediately preceeding our current entry. $K + 3I$ is found further back in the table. Exactly where can be determined by how many trailing zero bits it has.

Case 2: j is odd. Then $K + I$ is the one further back in the table and $K + 3I$ is the one immediately preceeding our current location.

Where to find an entry can be determined by testing the number of trailing zeros it contains in its binary representation. A way to do this is to use two-s complement arithmetic.

$$I = K.\text{AND}. - K$$

for which $\log_2 I$ is the number we're looking for. We already know where one of the entries to be used in the interpolation is. It is the one immediately preceeding our current one. To locate the other, we first form $K - I$ and test to see if the bit representing $2I$ is set or not by forming $M = 2I.\text{OR}.(K - I)$. If this bit was not set, we have just set it and know that we have $M = K + I$ in our preceeding entry, i.e. j was even. If this bit was already set, we still have $M = K - I$ after the .OR. operation and know that j was odd. In any case M is a number with $p + 1$ trailing zeros and is one of the interpolants. The other is found by adding $2I$ to M; M being either $K + I$ or $K - I$ gives either $K + 3I$ or $K + I$. This number has more than $p+1$ trailing zeros and has in fact $L = \log_2(M + 2I.\mathrm{OR}. - (M + 2I))$ of them and is thus the Lth entry from the end.

This procedure is mathematically identical to the recursive bisection algorithm given in Oliver (1975) and differs only in the order with which the trigonometric functions are calculated. The minimal amount of information is kept which is needed to generate the subsequent weights. As soon as the weight is used, it is replaced by one which will be used in the future.

The following procedure is adapted from a FORTRAN program of Buneman's. The array `h( )` contains the half-secants of 2π, π, $\frac{\pi}{2}$, $\frac{\pi}{4}$, etc. The array `c( )` initially contains the cosines of 8π, 4π, 2π, π, etc. 8π and 4π are included so that we can get back the original table automatically after one complete period. The half-secant of $\frac{\pi}{2}$ is infinite and is irrelevant in the cosine routine. For a sine routine, just remember that $\sin(2j+1)\frac{\pi}{2}$ is always 0.

Algorithm 2.6.3 On-line Cosine Generation *(Buneman)*

```
    $N = 2^j$
    real h(j+2), c(j+4)
    for $K = 1 : 2 * N$
  /* find L, the number of trailing zeros in K */
      $I = K.AND. - K$
      $L = \log_2(I)$
  /* c(j+4-L) contains the Kth cosine */
  /* use it and then replace it */
      write c(j+4-L)
  /* find where the other entry for use in interpolation is located */
      $I = 2 * I + (2 * I.OR.K - I)$
      c(j+4-L) = h(j+2-L)*(c(j+3-L)+c(j+4-$\log_2$(I.AND.-I)))
    end
```

Buneman has used this algorithm for several years in FFT subroutines running on VAXes and CRAYs. In Chapter 3, we demonstrate a modified version of Buneman's method for multiprocessor FFTs.

Table 2.5: Buneman's On-Line Generation of Trigonometric Functions

32	16	8	4	2	1
100000	010000	001000	000100	000010	000001
32	16	8	4	2	3
100000	010000	001000	000100	000010	000011
32	16	8	4	6	3
100000	010000	001000	000100	000110	000011
32	16	8	4	6	5
100000	010000	001000	000100	000110	000101
32	16	8	12	6	5
100000	010000	001000	001100	000110	000101
32	16	8	12	6	7
100000	010000	001000	001100	000110	000111
32	16	8	12	10	7
100000	010000	001000	001100	001010	000111
32	16	8	12	10	9
100000	010000	001000	001100	001010	001001
32	16	24	12	10	9
100000	010000	011000	001100	001010	001001
32	16	24	12	10	11
100000	010000	011000	001100	001010	001011
32	16	24	12	14	11
100000	010000	011000	001100	001110	001011
32	16	24	12	14	13
100000	010000	011000	001100	001110	001101
32	16	24	20	14	13
100000	010000	011000	010100	001110	001101
32	16	24	20	14	15
100000	010000	011000	010100	001110	001111
32	16	24	20	18	15
100000	010000	011000	010100	010010	001111
32	16	$24 \equiv 8$	$20 \equiv 4$	$18 \equiv 2$	$17 \equiv 1$
100000	010000	011000	010100	010010	010001

Chapter 3

One Dimensional Parallel Fast Fourier Transforms

3.1 Introduction

The FFT has a lot of inherent parallelism. The signal flow graph shows clearly that at each step of the radix-p FFT, n/p independent butterfly operations are done on n/p mutually exclusive sets of p of data-points. These butterfly computations, being totally disjoint, can be done in parallel. The FFT is also a global operation. Each point in the DFT is computed as a weighted sum of every other point. This can be seen by the representation of the DFT as a matrix-vector multiplication $\mathbf{F}_n\mathbf{x}$. The FFT achieves this result efficiently by using a divide-and-conquer splitting procedure. This means that for a parallel distributed computing system, the n/p parallelism of the FFT can only be exploited profitably if the data-points that make up the disjoint pairs can be efficiently placed exactly where they are needed when they are needed. Implementation of a parallel one-dimensional FFT is primarily a data-routing problem.

One of the first things to notice when studying the radix-2 hypercube architecture and the radix-2 FFT algorithm is the similarity between the adjacency scheme of the hypercube and the signal flow graph of the FFT. The FFT butterfly data flow graph can be mapped into the hypercube graph so that adjacent nodes in the radix-2 FFT signal flow graph are also adjacent in the hypercube. In this sense the hypercube is an ideal topology for the implementation of the FFT. This explains why it is easiest to implement a radix-2 FFT on the radix-2 hypercube. Several authors have either implemented or discussed the implementation of the radix-2 complex FFT in the hypercube, including Chamberlain (1986), Chan (1986), Swarztrauber (1986) and Walton(1986).

FFTs of mixed radix can also be implemented on the hypercube by way of the "twiddle factor" approach where a one-dimensional array is written in terms of a two-dimensional array. This method is discussed in Chapter 4 along with two-dimensional FFTs because the data-flow requirements are identical. A variation to this approach allows us to implement FFT's of length $n = 2^k \cdot m$ on the radix-2 hypercube. We discuss this mixed radix-2 FFT in Chapter 7.

In this chapter we study the implementation of a single one-dimensional complex FFT. Issues such as data distribution, load balancing, communication requirements, processor topology and distributed multiplier computation are addressed. Comparisons for the four canonic in-place radix-2 algorithms of Chapter 1 are examined to determine properties of the FFT algorithm with respect to distributed processing. Deficiencies in the conventional procedures are identified and a new mapping of data into two columns (the Two-Track mapping) is proposed to remedy these problems. The new method is load balanced, executes faster and allows in-place computation of the FFT without requiring extra buffer space for message handling. We also address the issue of topology and show that a mapping of the hypercube according to a Binary Reflected Gray Code (BRGC), which is not isomorphic to the FFT butterfly signal flow graph, is a perfectly viable way of implementing the FFT. This mapping is also more versatile and better suited for use on general numerical computational problems involving FFTs and other computations that require either nearest-neighbor connections and/or the exchange permutation. The last section of this chapter examines distributed multiplier computational issues and describes how each of the methods in Chapter 2 can or cannot be implemented efficiently on the hypercube.

3.2 One-Dimensional Complex FFT on the Hypercube

Let $\mathbf{x}$ be a vector of length n where $n = 2^t$. The vector $\mathbf{x}$ can be distributed in a hypercube of dimension d consisting of P nodes ($P = 2^d$ and $d \leq t$) by partitioning $\mathbf{x}$ into subvectors $\mathbf{x}_i$ each of length $n_s = n/P$,

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_{n-1} \end{bmatrix}$$

and placing $\mathbf{x}_i$ in processor i. This is called *consecutive storage*. Here it is helpful to imagine the hypercube as a column of P nodes. We call this mapping the *natural ordered* mapping of input in natural order. See Figure 3.1.

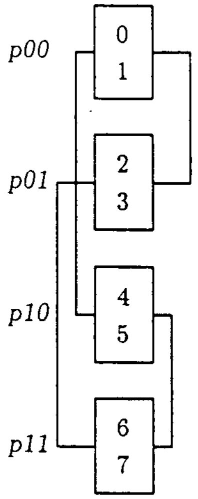

Figure 3.1: $n = 8$, $d = 2$ mapping of data in natural order

As noted before, in-place FFTs that take natural order input produce output in bit-reversed order. Specifically the procedures CT1 and GS1 overwrite $\mathbf{x} \leftarrow \mathbf{P}_n\mathbf{F}_n\mathbf{x}$. It is practical to also be able to implement the FFT algorithms that take bit-reversed input and produces natural ordered output. The CT2 and GS2 radix-2 procedures overwrite $\mathbf{x} \leftarrow \mathbf{F}_n\mathbf{P}_n\mathbf{x}$. The input is mapped so that $\mathbf{P}_n\mathbf{x}$ is stored in the hypercube in consecutive order shown in Figure 3.2

The most direct method of implementing a distributed FFT on the hypercube is to map the input vector into processors consecutively in the natural order. [Chamberlain (1986)] The hypercube graph is a compressed version of the radix-two FFT butterfly data flow graph. The signal flow graph for the original Cooley-Tukey algorithm (CT1) in Figure 3.3 shows how easy it is to map the FFT algorithm onto the hypercube architecture. For a transform of 2^t points consecutively stored in the d-dimensional hypercube, the first d butterfly steps involve across processor operations and hence require communication. We refer to these steps as the *intermingling* steps. The remaining $t - d$ butterfly steps are local and hence do not require communication. We call them the *parallel butterflies*.

The implementation of the intermingling butterflies requires communication.

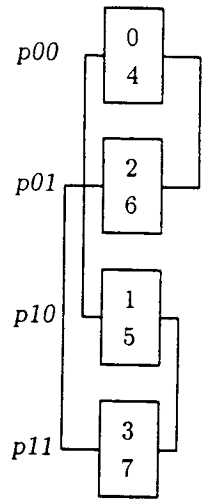

Figure 3.2: $n = 8$, $d = 2$ mapping of data in bit-reversed order

Algorithm 3.2.1 shows the direct implementation of the Cooley-Tukey butterfly

$$\begin{bmatrix} \mathbf{I} & \mathbf{\Delta} \\ \mathbf{I} & -\mathbf{\Delta} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{x} + \mathbf{\Delta y} \\ \mathbf{x} - \mathbf{\Delta y} \end{bmatrix}$$

and Algorithm 3.2.2 describes the Gentleman-Sande butterfly

$$\begin{bmatrix} \mathbf{I} & \mathbf{I} \\ \mathbf{\Delta} & -\mathbf{\Delta} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{x} + \mathbf{y} \\ \mathbf{\Delta}(\mathbf{x} - \mathbf{y}) \end{bmatrix}$$

We establish some general conventions for the description of the algorithms.

μ A generic name referring to the node that the process is running on. On the Intel iPSC, this is the value of the function `mynode()`.

ρ A generic name referring to the processor that communication, whether a send or receive, is being done with.

send($v, L_v \rho$) A primitive containing three separate arguments. v is the address of the array or vector that is to be sent to processor ρ. L_v is the length of the vector or array sent. (Note: this is not the length in bytes, but rather a higher-level abstraction that we usually think of so that 1 unit in length is

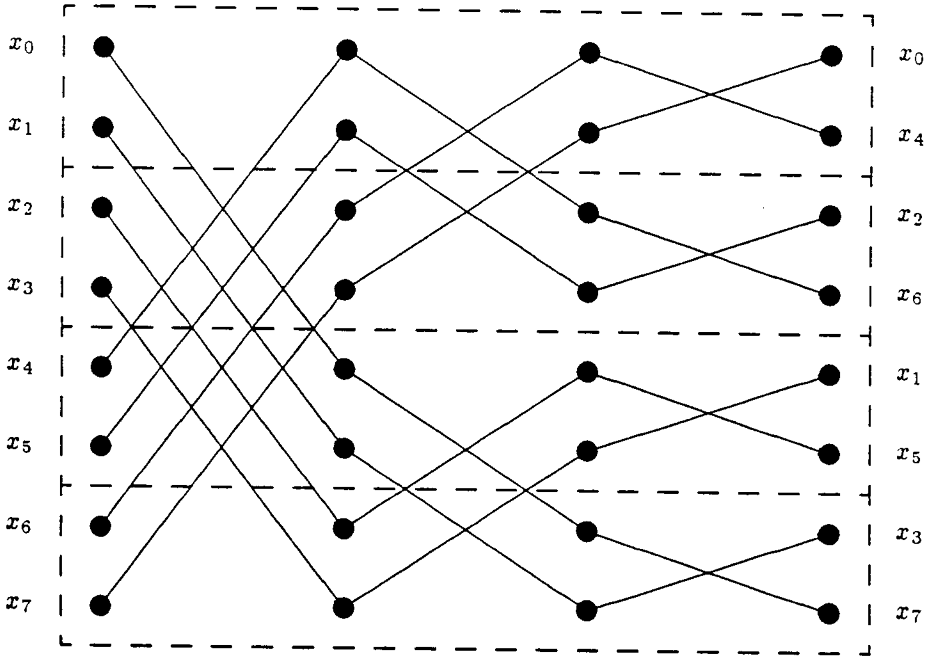

Figure 3.3: $n = 8$, $P = 4$, signal flow graph (GS1, CT1)

equivalent to 1 element in the vector. If A is an m-by-n matrix written in column-major order, then sending A is represented by send$(A, m \cdot n, \rho)$.

recv(v, L_v, ρ) A primitive designating that the calling process is to receive an array v of length L_v from the process running on processor ρ.

b A workspace buffer.

n A constant referring to the length of the transform.

t $n = 2^t$.

P The number of processors.

d The dimension of the hypercube $P = 2^d$.

$id(\mu)$ The index number that represents μ's place in the node order that the hypercube is mapped in. For example $B_2 = \{0, 1, 2, 3\}$ is the binary mapping of the nodes in a 2-cube in natural order, then $id[2] = 2$. However if $G_2 = \{0, 1, 3, 2\}$ is the Binary Reflected Gray Code mapping of the nodes in a 2-cube, $id[2] = 3$.

The basic Cooley-Tukey distributed butterfly involves two processors that send each other their data. One processor is responsible for the top portion of the butterfly $\mathbf{x} + \mathbf{\Delta y}$ and the other one the bottom portion $\mathbf{x} - \mathbf{\Delta y}$. The processor responsible for the bottom portion also computes $\mathbf{\Delta y}$ before the communication so it has a higher workload.

Algorithm 3.2.1 Cooley-Tukey Butterfly

```
/* $\mu$ = processor id, $n = 2^t$, $P = 2^d$, $m = n/P$ */
/* processor $\mu$ has $\mathbf{x}(id[\mu] \cdot m : (id[\mu] + 1) \cdot m - 1)$ */
/* $\rho$ = processor id of partner in this particular butterfly */
  if $(id[\mu] < id[\rho])$ then
/* $\mu$ houses the top input */
    top = .true.
  else
/* $\mu$ houses the bottom input */
    top = .false.
  end if
  if (.not.top) then
/* processor on the bottom has to compute the multipliers $\mathbf{\Delta}$ */
/* and scale its portion of $\mathbf{x}$ */
```

```
    $\mathbf{x} \leftarrow \Delta\mathbf{x}$
  /* communicate with processor $\rho$ */
  /* use buffer $\mathbf{b}$ for incoming messages */
    send($\mathbf{x}, m, \rho$); recv($\mathbf{b}, m, \rho$)
  /* update */
    if (top) then
      $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{b}$
    else
      $\mathbf{x} \leftarrow \mathbf{b} - \mathbf{x}$
    end if
```

The distributed Gentleman-Sande butterfly also involves two processors that send each other their entire data vector. Here the communication is done before any scaling by Δ is done. Again, the processor responsible for the bottom portion of the butterfly has a heavier workload as a result of the scaling.

Algorithm 3.2.2 Gentleman-Sande Butterfly

```
  /* $\mu$ = processor id, $n = 2^t$, $P = 2^d$, $m = n/P$ */
  /* processor $\mu$ has $\mathbf{x}(id[\mu] \cdot m : (id[\mu] + 1) \cdot m - 1)$ */
  /* $\rho$ = processor id of partner in this particular butterfly */
    if $(id[\mu] < id[\rho])$ then
  /* $\mu$ houses the top input */
      top = .true.
    else
  /* $\mu$ houses the bottom input */
      top = .false.
    end if
  /* communicate with processor $\rho$ */
  /* use buffer $\mathbf{b}$ for incoming messages */
    send($\mathbf{x}, m, \rho$); recv($\mathbf{b}, m, \rho$)
  /* update */
    if (top) then
      $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{b}$
    else
  /* bottom processor has to compute $\Delta$ and scale $\mathbf{x}$ */
      $\mathbf{x} \leftarrow \mathbf{b} - \mathbf{x}$
      $\mathbf{x} \leftarrow \Delta\mathbf{x}$
    end if
```

Figure 3.5 depicts the distributed Cooley-Tukey butterfly and Figure 3.4 shows the distributed Gentleman-Sande butterfly.

The intermingling butterflies involve two processors that pass their entire data sets to each other. Both processors then compute the updates for that step and store the results locally. A close look at the distributed butterfly operation reveals the necessity for an extra complex buffer to store the incoming data. In the case of the Gentleman-Sande distributed butterfly, processor *p0* computes $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{y}$. However in the *recv* statement, it is not possible to overwrite $\mathbf{x}$ by $\mathbf{y}$ since the sum $\mathbf{x} + \mathbf{y}$ is required. Similarly processor *p1* cannot overwrite $\mathbf{y}$ by $\mathbf{x}$. Hence the call to *recv* must provide for temporary storage of the incoming data in a buffer.

Another problem is the imbalance of workload between the two processors. As noted by Chamberlain for the Gentleman-Sande algorithm, processor *p0* computes the sum of its own data and the data it has just received. Processor *p1* has to compute the diagonal matrix of weights $\mathbf{\Delta}$ and then do a complex subtraction followed by a complex multiplication. If $\mathbf{\Delta}$ is updated by Algorithm 2.3.4, Repeated Multiplication, processor *p1* incurs two complex multiplies per data point more than its partner. This imbalance is slightly reduced in the Cooley-Tukey CT2 version reported by Chamberlain, but is nevertheless present.

Walton (1986) modifies the CT2 algorithm found in Press *et al.* (1986) and derives a straightforward implementation for use in the hypercube (Figure 3.6). He addresses the buffering problem as well as the load balancing issue. Here each processor exchanges only half of its data points with its partner for a particular intermingling step. The total butterfly computation now takes place within the processor, with each processor doing the updates for half of the points. Afterwards, a second exchange is performed and the half of the updates that "belong" to the exchange partner is sent "home". Buffering is not needed since the exchange occurs before and after the actual update step. Furthermore the butterfly computations are performed symmetrically so that processor *p0* computes a portion of both $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{\Delta y}$ for its own update as well as a portion of $\mathbf{y} \leftarrow \mathbf{x} - \mathbf{\Delta y}$ for its neighbor. Therefore Walton's method corrects the load imbalance and buffering problems, but introduces the need for twice as much communication.

Algorithm 3.2.3 Cooley-Tukey Butterfly (Walton's Implementation)

```
/* $\mu$ = processor id, $n = 2^t$, $P = 2^d$, $m = (n/P)/2$ */
/* processor $\mu$ holds: */
/* $\mathbf{z}^{(1)} = \mathbf{x}(2m \cdot id[\mu] : m \cdot (2id[\mu] + 1) - 1)$ */
/* $\mathbf{z}^{(2)} = \mathbf{x}(m \cdot (2id[\mu] + 1) : 2m \cdot (id[\mu] + 1) - 1)$ */
/* communicate with processor $\rho$ */
  send($\mathbf{z}^{(2)}, m, \rho$); recv($\mathbf{z}^{(1)}, m, \rho$)
```

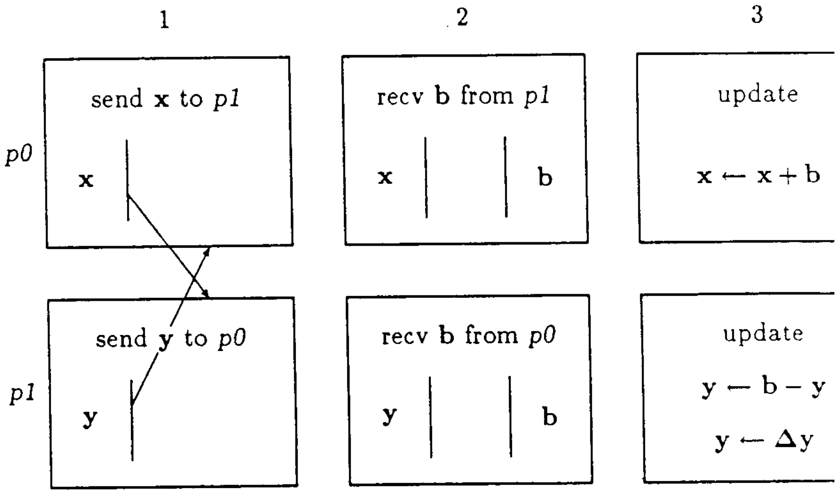

Figure 3.4: Distributed Gentleman-Sande butterfly

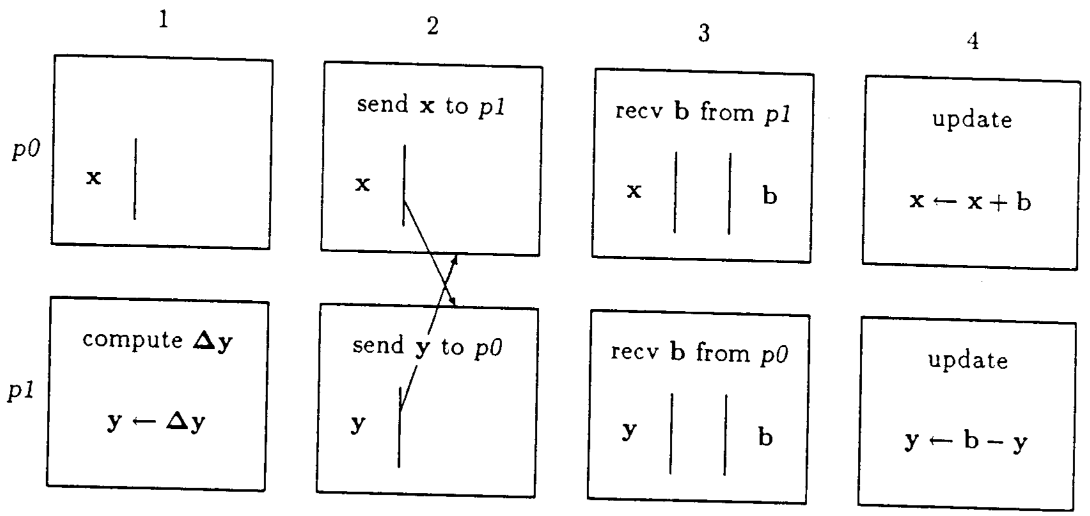

Figure 3.5: Distributed Cooley-Tukey Butterfly

Table 3.1: Four Canonic Radix-2 FFT Algorithms

Abbreviation	Formula	Intermingling	Multiplier Order
(CT1)	$\mathbf{x} \leftarrow \mathbf{P}_n\mathbf{F}_n\mathbf{x}$	first	bit-reversed
(CT2)	$\mathbf{x} \leftarrow \mathbf{F}_n\mathbf{P}_n\mathbf{x}$	last	natural
(GS1)	$\mathbf{x} \leftarrow \mathbf{P}_n\mathbf{F}_n\mathbf{x}$	first	natural
(GS2)	$\mathbf{x} \leftarrow \mathbf{F}_n\mathbf{P}_n\mathbf{x}$	last	bit-reversed

```
/* update */
  for $j = 1 : n_{p2}$
    $t \leftarrow \omega \cdot \mathbf{z}^{(2)}(j)$
    $\mathbf{z}^{(2)}(j) \leftarrow \mathbf{z}^{(1)}(j) - t;\ \mathbf{z}^{(1)}(j) \leftarrow \mathbf{z}^{(1)}(j) + t$
    $\omega \leftarrow \omega_0 \cdot \omega$
  end
/* communicate again with processor $\rho$ */
  send($\mathbf{z}^{(2)}, m, \rho$); recv($\mathbf{z}^{(1)}, m, \rho$)
```

3.3 Implementation Results for the One-Dimensional FFT

We first present the implementation of the four canonic radix-2 FFT algorithms via a method similar to Chamberlain's (1986). The properties and abbreviations for these four FFT algorithms are outlined in Table 3.1. We give a comparison of the timing results as well as a discussion on how the properties of the canonic forms affect the timing results. Ordering of the multipliers is discussed in Section 3.5.

Table 3.3 shows the result of the comparisons between the four canonic forms. The timings are given in increments of five milliseconds ranging from the fastest time to the slowest time. A '–' indicates that data is not available or unnecessary. These timings were taken from a single sample FFT of 4096 random complex points. The total time is broken down into components including computation (abbreviated, comp) and communication (abbreviated, comm) time. Part of the communication costs is the time a processor spends blocked awaiting data that it needs to arrive from another processor. Recall from Chapter 1 that a processor is considered inactive when it is blocked. Therefore the time that a processor spends

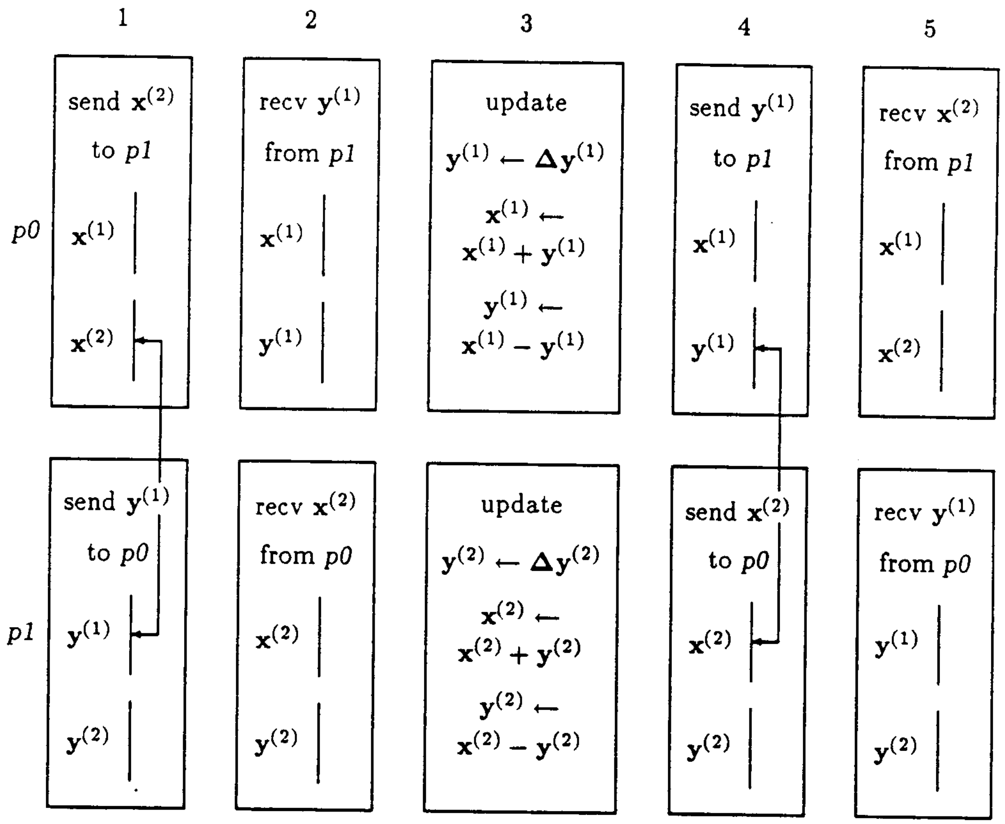
1
2
3
4
5
p0
p1
send $\mathbf{x}^{(2)}$ to *p1*
$\mathbf{x}^{(1)}$
$\mathbf{x}^{(2)}$
recv $\mathbf{y}^{(1)}$ from *p1*
$\mathbf{x}^{(1)}$
$\mathbf{y}^{(1)}$
update
$\mathbf{y}^{(1)} \leftarrow \Delta\mathbf{y}^{(1)}$
$\mathbf{x}^{(1)} \leftarrow \mathbf{x}^{(1)} + \mathbf{y}^{(1)}$
$\mathbf{y}^{(1)} \leftarrow \mathbf{x}^{(1)} - \mathbf{y}^{(1)}$
send $\mathbf{y}^{(1)}$ to *p1*
$\mathbf{x}^{(1)}$
$\mathbf{y}^{(1)}$
recv $\mathbf{x}^{(2)}$ from *p1*
$\mathbf{x}^{(1)}$
$\mathbf{x}^{(2)}$
send $\mathbf{y}^{(1)}$ to *p0*
$\mathbf{y}^{(1)}$
$\mathbf{y}^{(2)}$
recv $\mathbf{x}^{(2)}$ from *p0*
$\mathbf{x}^{(2)}$
$\mathbf{y}^{(2)}$
update
$\mathbf{y}^{(2)} \leftarrow \Delta\mathbf{y}^{(2)}$
$\mathbf{x}^{(2)} \leftarrow \mathbf{x}^{(2)} + \mathbf{y}^{(2)}$
$\mathbf{y}^{(2)} \leftarrow \mathbf{x}^{(2)} - \mathbf{y}^{(2)}$
send $\mathbf{x}^{(2)}$ to *p0*
$\mathbf{x}^{(2)}$
$\mathbf{y}^{(2)}$
recv $\mathbf{y}^{(1)}$ from *p0*
$\mathbf{y}^{(1)}$
$\mathbf{y}^{(2)}$

Figure 3.6: Walton's Implementation of the Distributed CT2 Butterfly

Table 3.2: One-Track FFT FORTRAN Code

Algorithm	Code numbers
CT2	(3.1), (3.6)
GS1	(3.2), (3.7)
CT1	(3.8), (3.10)
GS2	(3.9), (3.11)
all	(3.3), (3.4), (3.5)

blocked lowers its overall utilization and is thus detrimental to the total speedup of the parallel computation.

The FORTRAN code is listed in the appendix. The same code is used for the host and node driver routines for all of the different implementations. Table 3.2 outlines the code that is used for the different FFT algorithms.

The results show markedly different characteristics between the Gentleman-Sande (GS) algorithms and the Cooley-Tukey (CT) algorithms in terms of computational load balance and communication blocking costs. The load imbalance between the processors shows up in both cases. However in the Gentleman-Sande implementations, it does not cause an inordinate amount of processor blocking, so that nodes with a higher computational time also had a higher total time. In the Cooley-Tukey implementations, however, the nodes with the shortest computational times showed the longest blocking time since they were waiting for nodes with had longer computational times. This results in a more balanced total time, since the spread between faster and slower processors is smaller than in the Gentleman-Sande implementations. However, more total time is wasted in blocking because slower processors cause faster ones to block. This is primarily due to the difference between the butterfly structures of the two methods.

A look at the diagram in Figure 3.4 shows that in the distributed Gentleman-Sande butterfly, processor *p0* does not have to wait for processor *p1* to compute its multipliers $\mathbf{\Delta}$ before it can go ahead and do its update. Therefore slower processors did not cause faster ones to block. Furthermore, the data communication is done *before* the butterfly operation is commenced. In contrast, the distributed Cooley-Tukey butterfly (Figure 3.5) requires that processor *p0* wait for processor *p1* to compute $\mathbf{\Delta}$ and scale its contents *before* communication takes place. In this case, communication occurs in the middle of the butterfly operation causing the processor that has no work during the first half of the butterfly to block and wait for its partner.

A property that shows up in the computational timings is related to the order

Table 3.3: Forward FFT (Natural Implementation)

$n = 4096$

		CT1	
dim	total	comp	comm/blocked
0	12290	–	–
1	6305–6430	6080–6255	50-345/10-315
2	3240–3350	2995–3195	50–350/5–320
3	1670–1745	1475–1635	40–270/0–240
4	865–915	730–820	35–180/15–160
		GS1	
dim	total	comp	comm/blocked
0	9255	–	–
1	4605–5280	4550–5225	55/5–10
2	2290–2985	2235–2925	55–60/5–10
3	1145–1670	1100–1620	45–50/5–10
4	640–995	555–900	80–95/5–70
		CT2	
dim	total	comp	comm/blocked
0	9915	–	–
1	5655-5660	4830–5610	50–825/5–790
2	3305–3320	2365–3270	50–940/10–900
3	1870–1880	1165–1840	40–700/10–675
4	1045–1055	565–1025	30–470/5–440
		GS2	
dim	total	comp	comm/blocked
0	11250	–	–
1	5955–6205	4775–5525	430/10
2	2950–3210	2710–2970	240–250/10
3	1470–1665	1325–1525	135–145/0–15
4	740–915	650–785	85–130/0–60

of the multiplier computation. Algorithm GS1 and CT2 require the multipliers to be applied in natural order. In Chapter 2 we discussed various methods to compute naturally ordered multipliers. Double precision Repeated Multiplication (Algorithm 2.3.4) was used for GS1 and CT2 whereas the Direct Call method (Algorithm 2.3.2) was used for CT1 and GS2. The computational times show that the use of Repeated Multiplication to compute multipliers paid off in faster timings. Therefore methods which employ naturally ordered multipliers are favored for two reasons, they are faster and simpler.

Finally we notice some differences in communication time that depend on whether the intermingling steps come first or last. The communication times of the two algorithms (CT1 and GS1) that intermingle first are faster than those of their counterparts where intermingling is last (CT2 and GS2). This is because the weight of all the past computational .imbalances causes some processors to arrive at the intermingling steps ahead of the other processors. Of course these processors with a lighter workload block to wait for those with a heavier workload to reach the intermingling steps. This is especially so in the heavy imbalance in blocking exhibited by the CT2 algorithm.

Hypercube Mapped in BRGC Order

The topology of the hypercube mapped in Natural order is well-suited for computation of the radix-2 FFT because all messages are between neighboring nodes. However other computations may be best mapped into an entirely different topology. For example if the FFT is involved in the solution of partial differential equations, it is natural to use a finite difference discretization where a Nearest-Neighbor mesh type topology would be best suited. It would be inefficient to arrange the data in a nearest neighbor configuration for the finite difference steps and later rearranging it for the FFT steps. Therefore Chamberlain (1986) and Chan (1986) have proposed a compromise whereby the data is mapped into nodes ordered in a BRGC. See Figure 3.7.

Let a sequence G_d represent a BRGC of dimension k. For example, $G_3 = \{0, 1, 3, 2, 6, 7, 5, 4\}$. Recall from Chapter 1, the properties of this special gray code that make it useful.

Nearest Neighbor: Consecutive elements in the BRGC differ at exactly one bit. Therefore hypercube processors mapped in this order are adjacent to their nearest neighbors.

Exchange: Corresponding positions in G and and $\mathbf{E}G$ differ at exactly one bit. This means that processors are adjacent to their "mirror images".

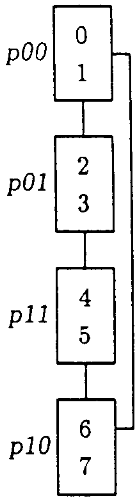

Figure 3.7: $n = 8$, $d = 2$ mapping of data in natural order, cube in BRGC

Global Connectivity: Elements of the BRGC $G(j)$ and $G(j \pm 2^k)$ differ at most in two bits for any $k > 0$. The subscripts are to be taken modulo 2^d. This means that processors are at most a Hamming distance two from any processor which is ordered a power-of-two apart.

Table 3.4 summarizes the implementation of the CT2 and GS1 algorithm on a hypercube mapped in BRGC order. We choose to discuss the two variants where the multipliers are computed in natural order because they are simpler and more convenient to implement. The communication time ranges are roughly the same as those for the hypercube mapped in Natural order, see Table 3.3.

3.4 The Two-Track FFT Implementation

We now describe a new method that solves the problems of load imbalance and extra buffering needs with just minimal communication. We have already seen in Section 3.2 how an attempt to correct the load imbalance and buffering problems requires a processor to communicate two times during an intermingling step. By considering the second exchange in Walton's implementation we see that it is unnecessary. Recall that in efficient distributed in-place FFT implementations, no attempt is made to perform the bit-reversal permutation. Hence either the input

Table 3.4: Forward FFT (BRGC Implementation)

$n = 4096$

		CT2	
dim	total	comp	comm/blocked
0	9450	–	–
1	5420–5435	4595–5385	50–825/10–790
2	3065–3150	2235–3020	125–830/10–790
3	1720–1790	1090–1680	105–630/10–595
4	955–1030	725–925	40–390/15–440
		GS1	
dim	total	comp	comm/blocked
0	9205	–	–
1	4545–5290	4490–5230	55/5–10
2	2555–3305	2190–2925	365–375/70–295
3	1415–1935	1075–1630	300–395/50–270
4	795–1095	530–800	190–300/90–270

or the output is in bit-reversed scrambled order. Our new implementation method forgoes the second exchange and in doing so, introduces an additional scrambling of the data. This scrambling is shown to be manageable. The important point is that it is reversible. Hence our implementation achieves load balancing, in-place computations, no need for additional buffering space, as well as non-redundant communication requirements.

The FFT of a vector $\mathbf{x}$ of length n ($n = 2^t$) is broken into two pieces, the first half denoted by $\mathbf{x}^{(1)}$ and the second half $\mathbf{x}^{(2)}$ and distributed into P processors. We partition $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ into subvectors of length $n/2P$, $x_i^{(1)}$ and $x_i^{(2)}$, respectively. These are then placed into the processors such that the $x_i^{(k)}$ are placed in processor i, with $k = 1, 2$. We define the left track to consist of the subvectors of $\mathbf{x}^{(1)}$ and the right track the subvectors of $\mathbf{x}^{(2)}$.

At each intermingling step, each processor exchanges its portion of $\mathbf{x}^{(2)}$ for the portion of $\mathbf{x}^{(1)}$ that its exchange partner for that step has. The butterfly computation is done and the results are *not* sent back. Instead, we just accept that an additional permutation is put into the data. The inverse transform can be done by starting out with this permuted data and reversing the exchange steps until the final result is again in the correct order. This scheme is already load balanced and does not require extra buffers as it is similar to Walton's method. However in addition, by accepting an extra permutation, we have eliminated the

	Step 1		Step2		Step 3		Step 4	
p00	0	**8**	0	**4**	0	2	0	2
	1	**9**	1	**5**	1	3	1	3
p01	2	**10**	**2**	6	4	6	4	6
	3	**11**	**3**	7	5	7	5	7
p11	**4**	12	8	**12**	8	10	8	10
	5	13	9	**13**	9	11	9	11
p10	**6**	14	**10**	14	12	14	12	14
	7	15	**11**	15	13	15	13	15

Figure 3.8: Two-Track Forward Transform Data Swapping Pattern

need for extra communication.

The permutation is illustrated with a simple example: $(n = 16), (P = 4)$ in Figure 3.8, with elements for the next exchange at each step highlighted in bold. Notice that the last stage involves no intermingling and does not swap data between the columns, because parallel butterflies take place independently without need for communication.

An inverse transform with data originating in the scrambled order is done with sub-vectors swapped in the reverse order as depicted in Figure 3.9.

Figures 3.10 and 3.11 illustrate Two-Track Implementation of the Gentleman-Sande and Cooley-Tukey distributed butterflies and Algorithms 3.4.1 and 3.4.2 show how a distributed butterfly is done in the Two-Track method. Recall the two primitives *send* and *recv* used for the description of the algorithms. We also use a function

$$\texttt{flip}(m, \nu)$$

that takes the binary expansion of an integer m and complements the νth bit of that expansion. For example, $\texttt{flip}(19, 4) = 27$, since $19 = 10011_2$ and $27 = 11011_2$.

The Gentleman-Sande GS1 FFT algorithm utilizing the Two-Track data mapping performs the distributed butterflies first. Each update step precedes the communication step during the distributed butterfly stages. And the last $t - d$ steps are done independently and locally in the processors.

	Step 1		Step2		Step 3		Step 4	
p00	0	2	0	**2**	0	**4**	0	8
	1	3	8	**3**	1	**5**	1	9
p01	4	6	**4**	6	2	**6**	2	10
	5	7	**5**	7	3	**7**	3	11
p11	8	10	8	**10**	**8**	12	4	12
	9	11	9	**11**	**9**	13	5	13
p10	12	14	**12**	14	**10**	14	6	14
	13	15	**13**	15	**11**	15	7	15

Figure 3.9: Two-Track Inverse Transform Data Swapping Pattern

Algorithm 3.4.1 GS1 Two-Track FFT

```
/* $\mu$ = processor id; $n = 2^t$, $P = 2^d$, $m = (n/P)/2$ */
/* processor $\mu$ holds: */
/* $\mathbf{z}^{(1)} = \mathbf{x}(m \cdot id[\mu] : m \cdot (id[\mu] + 1) - 1)$ */
/* $\mathbf{z}^{(2)} = \mathbf{x}(\frac{n}{2} + m \cdot id[\mu] : \frac{n}{2} + m \cdot (id[\mu] + 1) - 1)$ */
/* s tells which multiplier to initialize */
  $s \leftarrow n$;
  for $L = 1 : d$
    $\omega_0 \leftarrow \omega_s$
    $p = \mu/m \pmod{s/2}$
    $\omega \leftarrow \omega_0^p$
    for $j = 1 : m$
      $t \leftarrow \mathbf{z}^{(1)}(j) - \mathbf{z}^{(2)}(j)$; $\mathbf{z}^{(1)}(j) \leftarrow \mathbf{z}^{(1)}(j) + \mathbf{z}^{(2)}(j)$
      $\mathbf{z}^{(2)}(j) \leftarrow \omega \cdot t$; $\omega \leftarrow \omega_0 \cdot \omega$
    end
    $s \leftarrow s/2$
/* communicate */
    $\nu = d - L$; $id[\rho] = flip(\mu, \nu)$; $\rho = id^{-1}(id[\rho])$
    send($\mathbf{z}^{(2)}, m, \rho$); recv($\mathbf{z}^{(1)}, m, \rho$)
  end
/* do local FFT (GS1) algorithm */
  call localGS1($\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, m$)
```

The Cooley-Tukey CT2 FFT using the Two-Track data mapping commences

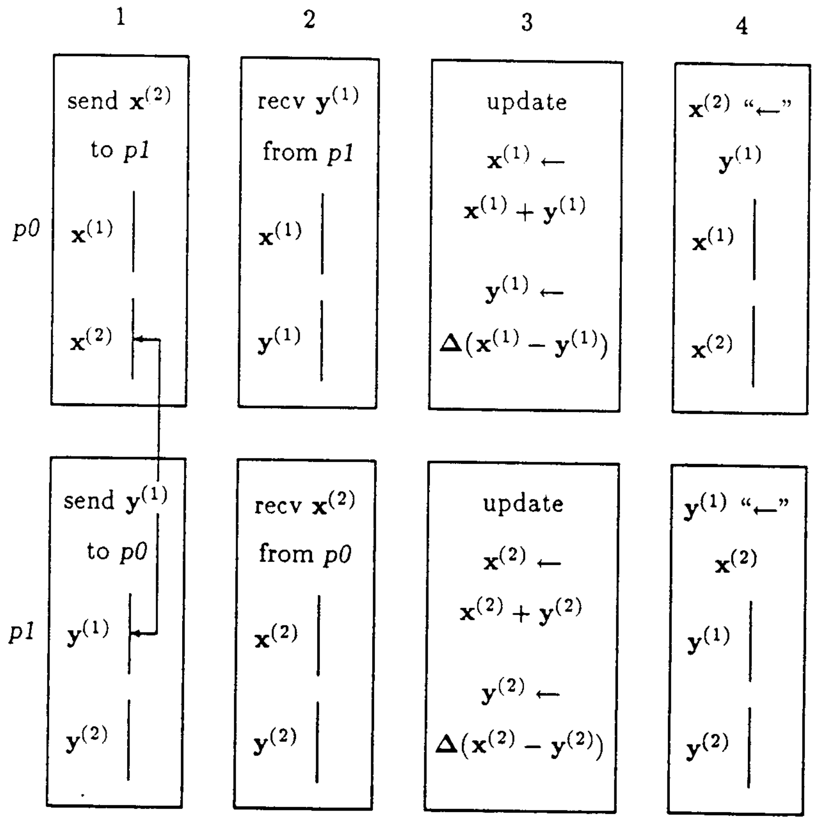
1
2
3
4
send $\mathbf{x}^{(2)}$ to *p1*
recv $\mathbf{y}^{(1)}$ from *p1*
update
$\mathbf{x}^{(1)} \leftarrow \mathbf{x}^{(1)} + \mathbf{y}^{(1)}$
$\mathbf{y}^{(1)} \leftarrow \Delta(\mathbf{x}^{(1)} - \mathbf{y}^{(1)})$
$\mathbf{x}^{(2)}$ "$\leftarrow$" $\mathbf{y}^{(1)}$
p0
$\mathbf{x}^{(1)}$
$\mathbf{x}^{(2)}$
$\mathbf{x}^{(1)}$
$\mathbf{y}^{(1)}$
$\mathbf{x}^{(1)}$
$\mathbf{x}^{(2)}$
send $\mathbf{y}^{(1)}$ to *p0*
recv $\mathbf{x}^{(2)}$ from *p0*
update
$\mathbf{x}^{(2)} \leftarrow \mathbf{x}^{(2)} + \mathbf{y}^{(2)}$
$\mathbf{y}^{(2)} \leftarrow \Delta(\mathbf{x}^{(2)} - \mathbf{y}^{(2)})$
$\mathbf{y}^{(1)}$ "$\leftarrow$" $\mathbf{x}^{(2)}$
p1
$\mathbf{y}^{(1)}$
$\mathbf{y}^{(2)}$
$\mathbf{x}^{(2)}$
$\mathbf{y}^{(2)}$
$\mathbf{y}^{(1)}$
$\mathbf{y}^{(2)}$

Figure 3.10: Two-Track Implementation of the Distributed GS Butterfly

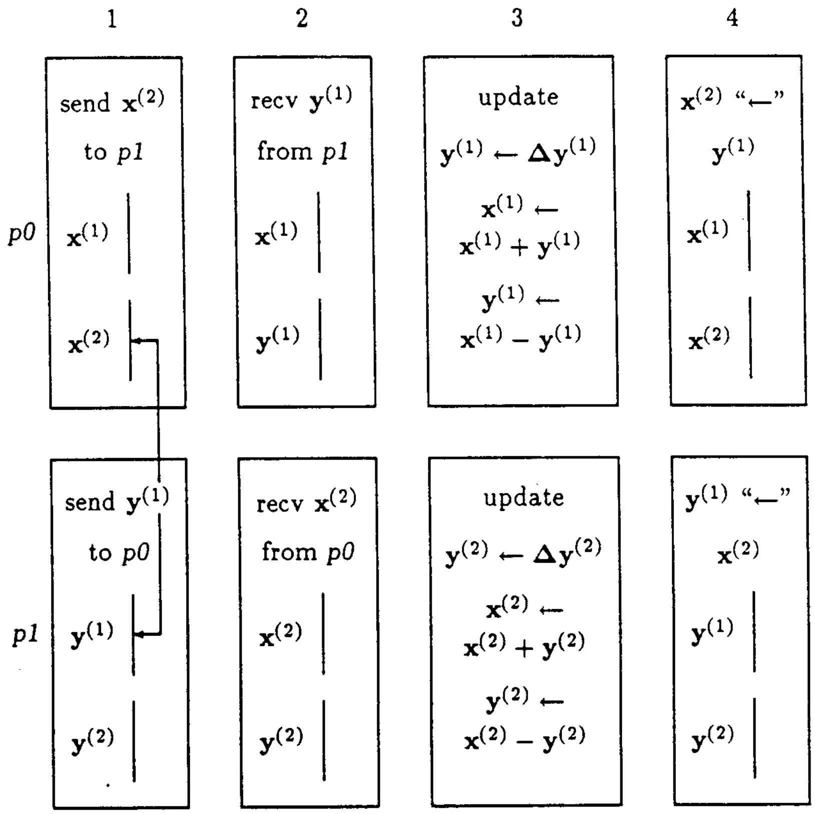
1
2
3
4
send $\mathbf{x}^{(2)}$ to *p1*
recv $\mathbf{y}^{(1)}$ from *p1*
update $\mathbf{y}^{(1)} \leftarrow \Delta\mathbf{y}^{(1)}$ $\mathbf{x}^{(1)} \leftarrow \mathbf{x}^{(1)} + \mathbf{y}^{(1)}$ $\mathbf{y}^{(1)} \leftarrow \mathbf{x}^{(1)} - \mathbf{y}^{(1)}$
$\mathbf{x}^{(2)}$ "$\leftarrow$" $\mathbf{y}^{(1)}$
p0
$\mathbf{x}^{(1)}$
$\mathbf{x}^{(2)}$
$\mathbf{x}^{(1)}$
$\mathbf{y}^{(1)}$
$\mathbf{x}^{(1)}$
$\mathbf{x}^{(2)}$
send $\mathbf{y}^{(1)}$ to *p0*
recv $\mathbf{x}^{(2)}$ from *p0*
update $\mathbf{y}^{(2)} \leftarrow \Delta\mathbf{y}^{(2)}$ $\mathbf{x}^{(2)} \leftarrow \mathbf{x}^{(2)} + \mathbf{y}^{(2)}$ $\mathbf{y}^{(2)} \leftarrow \mathbf{x}^{(2)} - \mathbf{y}^{(2)}$
$\mathbf{y}^{(1)}$ "$\leftarrow$" $\mathbf{x}^{(2)}$
p1
$\mathbf{y}^{(1)}$
$\mathbf{y}^{(2)}$
$\mathbf{x}^{(2)}$
$\mathbf{y}^{(2)}$
$\mathbf{y}^{(1)}$
$\mathbf{y}^{(2)}$

Figure 3.11: Two-Track Implementation of the Distributed CT Butterfly

with local independent FFTs and ends with distributed butterflies. Here the communication is done before the updating.

Algorithm 3.4.2 CT2 Two-Track FFT

```
/* $\mu$ = processor id; $n = 2^t$, $P = 2^d$, $m = (n/P)/2$ */
/* processor $\mu$ holds: */
/* $\mathbf{z}^{(1)} = \mathbf{x}(m \cdot id[\mu] : m \cdot (id[\mu] + 1) - 1)$ */
/* $\mathbf{z}^{(2)} = \mathbf{x}(\frac{n}{2} + m \cdot id[\mu] : \frac{n}{2} + m \cdot (id[\mu] + 1) - 1)$ */
/* do local FFT */
   call localCT2($\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, m$)
   $s \leftarrow 2 \cdot (n/P)$
   for $L = 1 : d$
/* communicate */
      $\nu = L$; $id[\rho] = flip(\mu, \nu)$; $\rho = id^{-1}(id[\rho])$
      send($\mathbf{z}^{(2)}, m, \rho$); recv($\mathbf{z}^{(1)}, m, \rho$)
/* update */
      $\omega_0 \leftarrow \omega_s$
      $p = \mu/m \pmod{s/2}$
      $\omega \leftarrow \omega_0^p$
      for $j = 1 : m$
         $t \leftarrow \omega \cdot \mathbf{z}^{(2)}(j)$; $\mathbf{z}^{(2)}(j) \leftarrow \mathbf{z}^{(1)}(j) - t$
         $\mathbf{z}^{(1)}(j) \leftarrow \mathbf{z}^{(1)}(j) + t$; $\omega \leftarrow \omega_0 \cdot \omega$
      end
      $s \leftarrow 2 \cdot s$
   end
```

Data Permutation of the Two-Track Method

The Two-Track method achieves its objective by utilizing an unconventional mapping of data into the processors. The sequence of data transfers results in an output array that is permuted in another ordering. We describe this data permutation pattern in the language of permutation matrices and Kronecker products.

Let $\mathbf{x}$ be a vector of length $n = 2^t$

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}^{(1)} \\ \mathbf{x}^{(2)} \end{bmatrix}$$

inside of $P = 2^k$ processors. Every processor has a piece of $\mathbf{x}^{(1)}$ as well as a piece of $\mathbf{x}^{(2)}$. During the communication of the Two-Track method, portions of $\mathbf{x}^{(1)}$ are swapped with portions of $\mathbf{x}^{(2)}$, resulting in a new vector $\tilde{\mathbf{x}}$ which is also split into $\tilde{\mathbf{x}}^{(1)}$ and $\tilde{\mathbf{x}}^{(2)}$ mapped into two columns.

Algorithm 3.4.3 gives the data permutation pattern of the Forward Two-Track method. Recall from Chapter 1 that $\mathbf{B}_n$ is the matrix representing the Butterfly Permutation and $\mathbf{M}_n$ is the inverse perfect shuffle matrix of a sequence of length n.

```
Algorithm 3.4.3 for $j = 2 : k+1$
      $r = 2^j$
      $\mathbf{x} \leftarrow (\mathbf{B}_r \otimes \mathbf{I}_{n/r})\mathbf{x}$
   end
```

From Theorem 1.6.6 we have

$$\tilde{\mathbf{x}} \leftarrow (\mathbf{M}_{2P}^{(2)} \otimes \mathbf{I}_{n/2P})\mathbf{x}$$

The Inverse Two-Track data pattern starts from

$$\tilde{\mathbf{x}} = (\mathbf{M}_{2P}^{(2)} \otimes \mathbf{I}_{n/2P})\mathbf{x}$$

and works backwards as in Algorithm 3.4.4

```
Algorithm 3.4.4 for $j = k+1 : -1 : 2$
      $r = 2^j$
      $\tilde{\mathbf{x}} \leftarrow (\mathbf{B}_r \otimes \mathbf{I}_{n/r})\tilde{\mathbf{x}}$
   end
```

The transpose of Theorem 1.6.6 gives,

$$\mathbf{x} \leftarrow (\mathbf{\Pi}_{2P}^{(2)} \otimes \mathbf{I}_{n/2P})\tilde{\mathbf{x}}$$

the original $\mathbf{x}$. If $n = 2P$, then the result of the forward Two-Track data permutation is that $\tilde{\mathbf{x}} = \mathbf{M}_n^{(2)}\mathbf{x}$ and similarly the Inverse Two-Track pattern gives back $\mathbf{x}$.

Timing Results

The timing results in Table 3.6 were done with the processors mapped in natural order and those of Table 3.7 with processors mapped in BRGC order. The relevant lines of Table 3.3 are reproduced in Table 3.6 in italicized characters for easy comparison. Similarly, lines in Table 3.4 appear in italics below their corresponding entry in Table 3.7.

Table 3.5: Two-Track FFT FORTRAN Code

Algorithm	Code Number
CT2	(3.12), (3.17)
GS1	(3.13), (3.18)
both	(3.14), (3.15), (3.16)

Table 3.6: Forward FFT (Two-Track) Natural

$n = 4096$

		CT2	
dim	total	comp	comm/blocked
0	8685	8685	–
0	*9915*	–	–
1	4655	4625–4630	30/0–10
1	*5655-5660*	*4830-5610*	*50-825/5-790*
2	2485–2490	2445–2460	30/0–10
2	*3305-3320*	*2365-3270*	*50-940/10-900*
3	1325–1335	1300–1305	25–30/5–10
3	*1870-1880*	*1165-1840*	*40-700/10-675*
4	745–765	690–695	50–70/0–55
4	*1045-1055*	*565-1025*	*30-470/5-440*
		GS1	
0	8365	8365	–
0	*9255*	–	–
1	4490–4530	4465–4500	25–30/5
1	*4605-5280*	*4550-5225*	*55/5-10*
2	2415–2445	2385–2415	30/5
2	*2290-2985*	*2235-2925*	*55-60/5-10*
3	1295–1320	1265–1285	25–30/0–10
3	*1145-1670*	*1100-1620*	*45-50/5-10*
4	700–760	675–690	20–85/0–65
4	*640-995*	*555-900*	*80-95/5-70*

* Italicized timings are for the One-Track Method.

The implementation code (in FORTRAN) appears in the Appendix. Table 3.5 tells the subroutine number associated with each Two-Track FFT algorithm.

As one can see, in each case these times for the Two-Track method are balanced and moreover faster than the slowest processor in Section 3.2. This is because a lot of the blocking inefficiencies have been eliminated.

To summarize:

- Load balancing is achieved by placing sub-vectors so that complete butterflies are done inside each processor during the intermingling steps.
- Only $n/2P$ elements are exchanged at each step rather than n/P elements.
- No extra buffering is needed for the distributed butterflies.
- Data is not sent "home", hence only one exchange per intermingling step is utilized.
- Timing results are faster than for one column method, because each processor only has to calculate $n/2P$ multipliers rather than n/P multipliers (for the two algorithms with natural ordering of multipliers). The speedup of the Two-Track method over the one column method increases from 10% to 31% with increasing dimension for the GS1 algorithm, and from 14% to 38% with increasing dimension for the CT2 algorithm.

3.5 Distributed Multiplier Computation

In distributed FFT's each individual processor requires a set of multipliers that is different from those required by other processors. This is because each processor is responsible for only a subset of the FFT butterflies.

Let $\omega_n = e^{-2\pi i/n}$ where $n = 2^t$ and p be a integer ranging from $0, \ldots, n/2 - 1$. Then the powers of ω_n, p required by processor x, in natural order, are exactly those p whose binary representation satisfy the pattern,

$$x\hat{\ }j, \quad j = 0, 1, \ldots, \frac{n}{2P} - 1$$

($\hat{\ }$denotes concatenation of binary numbers), and x is assumed to be in binary form. ($0 \leq x \leq P - 1$), with $P = 2^d$, the numbers of nodes in a d-dimensional hypercube. The mapping the integers $0, \ldots, n/2$ in consecutive order through a hypercube whose nodes are in natural order shows this. The highest order d bits denote the processor number.

Table 3.7: Forward FFT (Two-Track) BRGC

$n = 4096$

		CT2	
dim	total	comp	comm/blocked
0	8685	8685	–
0	*9450*	–	–
1	4655	4630	25–30/5
1	*5420–5435*	*4595–5385*	*50–825/10–790*
2	2565–2650	2450–2465	100–195/0–105
2	*3065–3150*	*2235–3020*	*125–830/10–790*
3	1405–1420	1295–1305	100–115/10–95
3	*1720–1790*	*1090–1680*	*105–630/10–595*
4	760–870	695–700	65–175/10–155
4	*955–1030*	*725–925*	*40–390/15–440*
		GS1	
dim	total	comp	comm/blocked
0	8365	8365	–
0	*9205*	–	–
1	4490–4530	4460–4495	30/5
1	*4545–5290*	*4490–5230*	*55/5–10*
2	2525–2560	2380–2410	125–175/5–110
2	*2555–3305*	*2190–2925*	*365–375/70–295*
3	1395–1460	1265–1290	115–175/50–140
3	*1415–1935*	*1075–1630*	*300–395/50–270*
4	790–865	670–695	110–185/25–165
4	*795–1095*	*530–800*	*190–300/90–270*

* Italicized timings are for the One-Track Method.

Example $n = 32$, $d = 2$

$$\begin{aligned} p00 &: \{00\hat{\ }00, 00\hat{\ }01, 00\hat{\ }10, 00\hat{\ }11\} \\ p01 &: \{01\hat{\ }00, 01\hat{\ }01, 01\hat{\ }10, 01\hat{\ }11\} \\ p10 &: \{10\hat{\ }00, 10\hat{\ }01, 10\hat{\ }10, 10\hat{\ }11\} \\ p11 &: \{11\hat{\ }00, 11\hat{\ }01, 11\hat{\ }10, 11\hat{\ }11\} \end{aligned}$$

Since the left-most d bits designate the processor number, we refer to these d bits as the processor field. Meanwhile, the right-most $t - d - 1$ bits run from $0, \ldots, \frac{n}{2P} - 1$ and are designated the working field.

We now show how each of the five algorithms from Chapter 2 can be distributed in the multiprocessor setting. We do not use the symmetry described on page 34 since each processor only has a selected subset of the trigonometric functions.

Algorithm 3.5.1 Direct Call

```
/* $\mu$ = processor id, $n = 2^t$, $P = 2^d$, $m = (n/P)/2$ */
  $\theta = 2\pi/n$
  $x = 2^{t-d-1} \cdot id[\mu]$
  for $j = 0 : m$
    c(j) ← $COS(x + j)\theta$; s(j) ← $SIN(x + j)\theta$
  end
```

Algorithm 3.5.2 Forward Recursion

```
/* $\mu$ = processor id, $n = 2^t$, $P = 2^d$, $m = (n/P)/2$ */
  $\theta = 2\pi/n$
  $x = 2^{t-d-1} \cdot id[\mu]$
  c ← $COS(\theta)$
  tc1 ← 2 · c
  c(0) ← $COS(x\theta)$; s(0) ← $SIN(x\theta)$
  c(1) ← $COS(x + 1)\theta$; s(1) ← $SIN(x + 1)\theta$
  for $j = 2 : m$
    c(j) ← tc1 · c(j − 1) − c(j − 2)
    s(j) ← tc1 · s(j − 1) − s(j − 2)
  end
```

Algorithm 3.5.3 Repeated Multiplication

```
/* $\mu$ = processor id, $n = 2^t$, $P = 2^d$, $m = (n/P)/2$ */
  $\theta = 2\pi/n$
  $x = 2^{t-d-1} \cdot id[\mu]$
  c $\leftarrow COS(\theta)$; s $\leftarrow SIN(\theta)$
  c(0) $\leftarrow COS(x\theta)$; s(0) $\leftarrow SIN(x\theta)$
  for $j = 1 : m$
    c(j) $\leftarrow$ c $\cdot$ c(j $-$ 1) + s $\cdot$ s(j $-$ 1)
    s(j) $\leftarrow$ $-$s $\cdot$ c(j $-$ 1) + c $\cdot$ s(j $-$ 1)
  end
```

Logarithmic Recursion is not practical in the multiprocessor setting. A simple example illustrates this. Let $n = 16$, $n/2 = 8$, and $d = 1$. Let c_j denote $\cos(j\theta)$. Recall that $c_j = c_{-j}$ if $j\theta$ is in the first quadrant.

p0	$c_0 = 1$ $c_1 = \cos(\theta)$ $c_2 = \cos(2\theta)$ $c_3 = 2c_2 \cdot c_1 - c_{-1}$
p1	$c_4 = \cos(4\theta)$ $c_5 = 2c_4 \cdot c_1 - c_{-3}$ $c_6 = 2c_4 \cdot c_2 - c_{-2}$ $c_7 = 2c_4 \cdot c_3 - c_{-1}$

One glance at the data that processor *p1* needs to compute c_5, c_6, and c_7 shows that they are not local and hence require communication.

Algorithm 3.5.4 Subvector Scaling

```
/* $\mu$ = processor id, $n = 2^t$, $P = 2^d$, $m = (n/P)/2$ */
  $\theta = 2\pi/n$; $m \leftarrow n/2$
  $x = 2^{t-d-1} \cdot id[\mu]$
  c $\leftarrow COS(\theta)$; s $\leftarrow SIN(\theta)$
  c(0) $\leftarrow COS(x\theta)$; s(0) $\leftarrow SIN(x\theta)$
  c(1) $\leftarrow COS(x+1)\theta$; s(1) $\leftarrow SIN(x+1)\theta$
  for $q = 1 : t-d-1$
    $L \leftarrow 2^q$; $L_o \leftarrow L/2$
    for $j = 0 : L_o - 1$
      c($L_o$+j) $\leftarrow$ c $\cdot$ c(j) + s $\cdot$ s(j)
      s($L_o$+j) $\leftarrow$ $-$s $\cdot$ c(j) + c $\cdot$ s(j)
```

```
        end
        $m \leftarrow m/2$
        $\theta = \pi/m$
        c $\leftarrow COS(\theta)$; s $\leftarrow SIN(\theta)$
        c(L) $\leftarrow$ c; s(L) $\leftarrow$ s
    end
```

Algorithm 3.5.5 Recursive Bisection

```
  /* $\mu$ = processor id, $n = 2^t$, $P = 2^d$, $m = (n/P)/2$ */
    $\theta = 2\pi/n$
    $x \leftarrow 2^{t-d-1} \cdot id[\mu]$; $y \leftarrow 2^{t-d-1} \cdot (id[\mu] + 1)$
    c(0) $\leftarrow COS(x\theta)$; s(0) $\leftarrow SIN(x\theta)$
    c($m$) $\leftarrow COS(y\theta)$; s($m$) $\leftarrow SIN(y\theta)$
    for $L = 1 : t - d - 1$
      $i \leftarrow m/2^L$; $j \leftarrow m/2^L$
      c(j) $\leftarrow COS(x + i)\theta$; s(j) $\leftarrow SIN(x + i)\theta$
      $h \leftarrow$ 1/2c(j)
      for $k = 1 : 2^{L-1} - 1$
        $j \leftarrow j + 2i$
        c(j) $\leftarrow h \cdot$ (c(j − i) + c(j + i)); s(j) $\leftarrow h \cdot$ s(j − i) + c(j + i))
      end
    end
```

Storage Considerations

The methods listed above generate a table of length $(n/P)/2$ complex numbers as the multipliers needed by each processor. This may or may not be desirable depending on whether there is sufficient storage and/or multiple transforms are done. If only one FFT is done and array storage space is precious, then methods which do not require table storage should be utilized. Such is the case with our implementations, where the multipliers are computed as needed.

Recursive Bisection

Recall that Buneman (1987) describes a procedure that implements the Recursive Bisection algorithm with tables of $O(\log_2 n)$ rather than $O(n)$.

Now suppose we are in a multiprocessor setting where each processor requires only a subset of the weights. This situation is applicable to distributing FFT butterflies among many processors.

Let $n = 2^t$ so that the multipliers needed have arguments

$$2\pi j/n \quad j = 0, \dots, \frac{n}{2} - 1$$

Mapping the weights in the natural order among $P = 2^d$ processors, $d \leq t$, we see that processor k needs to have the weights

$$\cos k\hat{\ }j, \quad j = 0, \dots, 2^{t-d-1}$$

($\hat{\ }$denotes concatenation of two binary numbers).

Example $(t = 5, d = 2)$

p(00): 00^000, 00^001, 00^010, 00^011, 00^100, 00^101, 00^110, 00^111.

p(01): 01^000, 01^001, 01^010, 01^011, 01^100, 01^101, 01^110, 01^111.

p(10): 10^000, 10^001, 10^010, 10^011, 10^100, 10^101, 10^110, 10^111.

p(11): 11^000, 11^001, 11^010, 11^011, 11^100, 11^101, 11^110, 11^111.

Notice that the left-most two bits designate the processor number and the right-most ones run from 0 to 7. In general, we refer to the left-most d bits as the processor field, and the rightmost $t-d-1$ bits, the working field. If we ignore the processor field, we notice that each processor needs to be able to generate 2^{t-d-1} sines and/or cosines. This means that it must have a table of length $t-d+1$ to follow Buneman's scheme. In fact the procedure is exactly the same for each processor as if it were working on a 2^{t-d-1} problem *except* for the initial values. Instead of initially having the cosines of 2π, π, $\frac{\pi}{2}$, $\frac{\pi}{4}$, etc., processor k would have the cosines of $2^{t-d-1} \cdot \frac{k\pi}{n} + 2\pi$, $2^{t-d-1} \cdot \frac{k\pi}{n} + \pi$, $2^{t-d-1} \cdot \frac{k\pi}{n} + \frac{\pi}{2}$, $2^{t-d-1} \cdot \frac{k\pi}{n} + \frac{\pi}{4}$, etc. In other words, the initial values are the cosines of the angles we would have had **plus** the quantity $2^{t-d-1} \cdot \frac{k\pi}{N}$. The characteristics of the working bits are all that are used, even though the initial and subsequent values are for angles which have degree $(+2^{t-d-1} \cdot k)$.

To illustrate, this is what processor 1 initially contains in our example:

$$\cos 40 \quad \cos 24 \quad \cos 16 \quad \cos 12 \quad \cos 10 \quad \cos 9$$

cos 8 must also be computed and used, but does not enter into our discussion here. Pretend for a minute that we try to work with the binary representation of 40, 24, 16, 12, 10, and 9.

101000 011000 010000 001100 001010 001001

We quickly see that we run into trouble in that the first three columns do not have the proper number of trailing zeros for Buneman's procedure. Instead, looking back at table 1, we see that if we add 8 to every number, the angles we need are generated in precisely the correct order. Therefore using the binary representations of the working field only, we see that despite what the initial values are, one can still get the correct order with which to generate the weights. Since the position and order is determined by the working bits and the values are determined by the initial values, we can generalize this to any other processor, let's say processor 3 whose initial values are

$$\cos 56 \quad \cos 40 \quad \cos 32 \quad \cos 28 \quad \cos 26 \quad \cos 25$$

and use the exact same algorithm. In fact Buneman's algorithm need only be changed in a few spots, the initialization of the cosine table is different and the loop runs through fewer values.

Algorithm 3.5.6 Multiprocessor On-line Cosine Generation

```
 /* $\mu$ = processor id, $n = 2^t$, $P = 2^d$ */
   $n = 2^t$
   real h(t+1), c(t+3)
 /* generate the portion of the cosine table */
 /* depending on processor number id[$\mu$] */
   arg = $8\pi$
   pin = $\frac{2\pi}{n}$
   inc = $2^{t-d-1} \cdot id[\mu]$
   for $i = 1 : t+1$
     arg = arg/2.0
     ang = arg + inc*pin
     c(i) = COS(ang)
   end
   c(t+2) = COS(ang-pin)
   write c(t+2)
   for $K = 1 : 2^{t-d-1} - 1$
     I = K.AND.-K
     L = $\log_2$(I)
/* c(t+2-L) contains the Kth cosine; use it and then replace it */
     write c(t+2-L)
     I = 2*I + (2*I.OR.K-I)
     c(t+2-L) = h(t+2-L)*(c(t+1-L)+c(t+2-$\log_2$(I.AND.-I)))
   end
```

Chapter 4

Comparison of Two-Dimensional FFT Methods on the Hypercube

4.1 Introduction

Multidimensional Fourier transforms, as in the single dimensional case, can also be broken into pieces that can be done in parallel. Here the possibilities are even richer than in the one-dimensional case. This is because multidimensional transforms can be done either as a sequence of separable one-dimensional transforms, or by directly splitting them into blocks of smaller multidimensional transforms, as in the vector-radix methods [Rivard (1977), Harris *et al.* (1977)]. We study only the case of the two-dimensional Fourier transform because the discussion and algorithmic methods can be extended directly to computing higher dimensional Fourier transforms.

Lower dimensional Fourier transforms can be computed by way of multidimensional transforms through the Twiddle Factor Algorithm (see Chapter 1). Therefore large one-dimensional Fourier transform problems can be approached from the multi-dimensional point of view. This multidimensional mapping of one-dimensional DFTs underlies the derivation of prime-factor FFTs, and the very efficient Winograd Fourier transform algorithm [Nussbaumer (1982)].

Work on two-dimensional FFTs on distributed processors has so far been restricted to the row-column approach. The strips method partitions the two-dimensional array, or matrix, into rows, mapping block rows into processors. The transform of each row is then found, and the matrix is transposed before a second row transform pass is done on rows that previously had been columns. We follow McBryan and Van de Velde (1985) in terming this approach the *Transpose-Split* (TS) 2D-FFT.

Another row-column method partitions the matrix into blocks of submatrices.

assigning one block per node. The hypercube is then viewed as a two-dimensional cross-product of smaller dimensional hypercubes with distributed FFTs performed along both the rows and the columns. No transposing of data is needed here. We term this method the ***Block*** (B) 2D-FFT method.

We present two new methods of implementing 2D-FFTs. The first one is a row-column approach that partitions data into strips much like the Transpose-Split method. The difference is that no transpose is done between the horizontal and vertical steps. Instead, the horizontal FFTs are done locally inside each processor and the vertical FFTs are distributed. We call this the ***Local-Distributed*** (LD) 2D-FFT method.

Finally we implement a partial Vector-Radix (VR) 2D-FFT on the hypercube. What this means is that the individual 2D-FFTs that are done locally inside the processors are row-column 2D-FFTs, however, the intermingling steps use the vector radix update scheme.

The Transpose-Split 2D-FFT is favored by several, including John Gustafson (1987), because all FFT computations are performed locally. The only communication that takes place occurs within the transpose step. Gustafson has implemented the Transpose-Split FFT on a 1024-node NCUBE machine which has the pleasant property that each node can perform up to 9 simultaneous communications, thereby allowing the use of almost all the links of the hypercube during the transpose stage. He can reduce communication time by a factor of d, the hypercube dimension. Therefore all that is needed to effectively implement this method is a fast efficient matrix transpose procedure. See Ho and Johnsson (1986) for an in-depth analysis of the hypercube matrix transpose problem.

The Block method was implemented by Miles *et al.* (1987) on the Floating Point Systems T-Series hypercube. By considering the signal flow graph of the radix-2 FFT algorithm, we see that this implementation requires communication during both the vertical and horizontal passes. At each step where the butterfly computation is split between two processors, each node exchanges with its neighbor exactly half of its data points. Each processor computes the butterfly updates for the points it possesses after which it contains updates for half of its own points and half of the points belonging to its communicating partner. Another exchange is then necessary to repatriate these updates. Therefore two exchanges are required for one butterfly step. This may seem inefficient unless the communications and computations are overlapped during the distributed butterfly calculations. This is indeed possible on the T-Series since each node possesses a transputer that allows a processor to send data to its neighbor in the next butterfly step even before it has totally completed the present step. Hence, two communication stages can be overlapped in one computational step. This method of implementation is referred to as the Block method since the matrix is mapped by sub-blocks into the hypercube such that the (i,j)th block is mapped into the node whose label is the binary representation of i concatenated with the binary representation of j. [See Miles *et al.* (1987)]. The powerful vector boards on the T-Series allows this method to be used advantageously.

The Local-Distributed method and the Vector-Radix-2 method are implemented on the Intel iPSC. The Intel iPSC, unlike the NCUBE and T-Series machines, does not allow simultaneous communication. This is detrimental for the Vector-Radix-2 method as the intermingling stages involve total exchanges between four processors instead of two. A total exchange within a subcube of processors means that each processor in the subcube exchanges data with every other processor in the subcube. This particular property of Intel communication also means that the full cross-bar interconnection scheme cannot be simulated efficiently, and with Intel iPSC/System 286 capabilities, a transpose takes $2d = 2\log_2 P$ steps to perform, as each node can only do one send followed by one receive in one direction at a time. Another drawback is that computation and communication cannot be overlapped and thus distributed FFTs will exhibit blocking during the intermingling butterfly steps caused by one processor waiting for data from another. The Local-Distributed method does not require a transpose and does distributed FFTs along only one direction instead of two (the Block method). Meanwhile the Vector-Radix FFT performs local 2D-FFTs followed by intermingling stages requiring the summation and multiplication of the local FFTs by "twiddle" factors. The Vector-Radix FFT has a lot of potential that is not reflected in our implementation on the Intel iPSC precisely because of communication inefficiencies. But we think it is useful to offer it as an alternative to the row-column approach because of its rich parallelism.

4.2 Two Dimensional FFT Algorithms

Row-Column

The Discrete Fourier Transform (DFT) of a vector x of length n is defined as

$$\mathbf{y} \leftarrow \mathbf{F}_n \mathbf{x}$$

where $\mathbf{F}_n$ is the matrix consisting of powers of the nth root of unity $\omega_n = e^{-2\pi i/n}$.

$$[\mathbf{F}_n]_{jk} = \omega_n^{jk}$$

The two-dimensional (2-D) discrete Fourier transform (DFT) of a two-dimensional array $\mathbf{X} \in \mathcal{C}^{n_1 \times n_2}$ is defined as

$$\mathbf{Y} \leftarrow \mathbf{F}_{n_1} \mathbf{X} \mathbf{F}_{n_2}^T$$

This matrix notation clearly demonstrates the row-column or column-row method of computing the 2-D transform, since matrix-multiplication is associative. If the fast Fourier transform (FFT) is used to evaluate the 1-D FFTs along both the rows and the columns, the number of complex multiplications required is $n^2 \log_2 n$ for $n = n_1 = n_2$. In addition, a matrix transposition algorithm is required.

A one-dimensional FFT of a long vector of length $n = n_1 \cdot n_2$ can be computed in a 2-D "fashion" by viewing it as a DFT of an array of size $n_1 \times n_2$ [Nussbaumer (1982)], that is, by writing $\mathbf{x}$ as an array $\mathbf{x}_{n_1 \times n_2}$. This notation is defined in Chapter 7. Using the Twiddle Factor Algorithm (Chapter 7), we can compute the n-point DFT of $\mathbf{x}$ by an n_1-point FFT of the rows, a point-wise multiplication of $\mathbf{x}$ by the *twiddle factors*, followed by another n_2-point FFT on the columns. The matrix of twiddle factors $\mathbf{T}$ is defined

$$[\mathbf{T}]_{jk} = \omega_n^{jk}, \quad j = 0, \ldots, n_1 - 1; \quad k = 0, \ldots, n_2 - 1.$$

and $*$ denotes the point-wise multiplication of two matrices. Hence the DFT $\mathbf{y}$ of $\mathbf{x}$ is another two-dimensional array given by

$$\mathbf{y}_{n_2 \times n_1}^T = \mathbf{F}_{n_1}[(\mathbf{T}_{n_1 \times n_2}) * [\mathbf{x}_{n_1 \times n_2}\mathbf{F}_{n_2}]]$$

The row-column or column-row method can be used to compute the horizontal and vertical DFTs.

Vector-Radix

The Vector-Radix FFT is a direct decomposition of the two-dimensional DFT into sums of smaller two-dimensional DFTs multiplied by "twiddle factors", (the matrix $\mathbf{\Delta}$). Here a 2-D DFT is recursively broken down into four 2-D DFTs until only trivial 2D-DFTs need to be evaluated. The number of complex multiplications is now $\frac{3}{4}n^2 \log_2 n$, 25% lower than the row-column method. [Rivard (1977), Harris (1977)] Moreover, no matrix transpose routine is required.

The recursive block structure of the DFT matrix $\mathbf{F}_n$ is used in two-dimensions to derive the method. The matrix $\mathbf{X} \in \mathcal{C}^{n \times n}$ is segregated into four sets; one over those samples $\mathbf{X}(j,k)$ for which j and k are both even, one for which j is even and k is odd, one for which j is odd and k is even and one for which both j and k are odd. The diagram in Figure 4.1 shows schematically the decimations of data in both the one and two dimensional decompositions.

Theorem 4.2.1 *Let* $\mathbf{X} \in \mathcal{C}^{n \times n}$ *with* $n = 2^t$, *then the 2D vector-radix splitting of the 2D-DFT of* $\mathbf{X}$ *is*

$$\mathbf{F}_n \mathbf{X} \mathbf{F}_n^T \;=\; (\mathbf{F}_n \mathbf{\Pi}_n)(\mathbf{M}_n \mathbf{X} \mathbf{M}_n^T)(\mathbf{\Pi}_n^T \mathbf{F}_n^T)$$

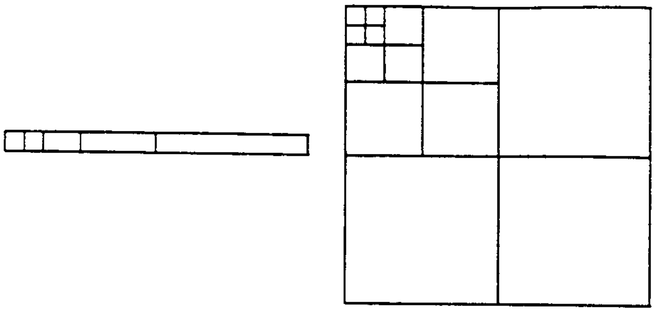

Figure 4.1: One and Two Dimensional Decimation Schemes

$$= \begin{bmatrix} \mathbf{I}_{n/2} & \mathbf{\Delta}_{n/2} \\ \mathbf{I}_{n/2} & -\mathbf{\Delta}_{n/2} \end{bmatrix} \begin{bmatrix} \widehat{\mathbf{X}}_{11} & \widehat{\mathbf{X}}_{12} \\ \widehat{\mathbf{X}}_{21} & \widehat{\mathbf{X}}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{I}_{n/2} & \mathbf{I}_{n/2} \\ \mathbf{\Delta}_{n/2} & -\mathbf{\Delta}_{n/2} \end{bmatrix}$$

where $\mathbf{\Delta}_{n/2} = diag(1, \omega_n, \ldots, \omega_n^{n/2-1})$ *and*

$$\begin{aligned} \widehat{\mathbf{X}}_{11} &= \mathbf{F}_{n/2}\mathbf{X}(0:2:n-2, 0:2:n-2)\mathbf{F}_{n/2}^T \\ \widehat{\mathbf{X}}_{12} &= \mathbf{F}_{n/2}\mathbf{X}(0:2:n-2, 1:2:n-1)\mathbf{F}_{n/2}^T \\ \widehat{\mathbf{X}}_{21} &= \mathbf{F}_{n/2}\mathbf{X}(1:2:n-1, 0:2:n-2)\mathbf{F}_{n/2}^T \\ \widehat{\mathbf{X}}_{22} &= \mathbf{F}_{n/2}\mathbf{X}(1:2:n-1, 1:2:n-1)\mathbf{F}_{n/2}^T \end{aligned}$$

Proof From Theorem 1.8.1 we have

$$\mathbf{F}_n\mathbf{\Pi}_n = \begin{bmatrix} \mathbf{I} & \mathbf{\Delta} \\ \mathbf{I} & -\mathbf{\Delta} \end{bmatrix} \begin{bmatrix} \mathbf{F}_{n/2} & 0 \\ 0 & \mathbf{F}_{n/2} \end{bmatrix}$$

Applying this to both sides of $\mathbf{M}_n\mathbf{X}\mathbf{M}_n^T$ gives the required decomposition.

_____*_____

Theorem 4.2.1 is the basic two-dimensional Cooley-Tukey (CT2) splitting of the Vector-Radix method for computing the 2-D FFT. The complete algorithm is obtained by recursive application of this basic decomposition.

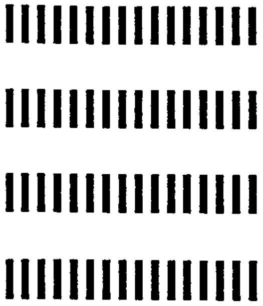

Figure 4.2: 16 columns mapped into 4 processors.

Algorithm 4.2.1 Vector-Radix 2-D FFT

```
    $n = 2^t$
    $\mathbf{X} \leftarrow \mathbf{P}_n \mathbf{X} \mathbf{P}_n^T$
    for $q = 1 : t$
      $L \leftarrow 2^q$
      $\mathbf{\Delta}_{L/2} \leftarrow diag(1, \omega_L, \ldots, \omega_L^{L/2-1})$
      $\mathbf{X} \leftarrow \left[\mathbf{I}_{n/L} \otimes \begin{bmatrix} \mathbf{I}_{L/2} & \mathbf{\Delta}_{L/2} \\ \mathbf{I}_{L/2} & -\mathbf{\Delta}_{L/2} \end{bmatrix}\right] \mathbf{X} \left[\mathbf{I}_{n/L} \otimes \begin{bmatrix} \mathbf{I}_{L/2} & \mathbf{I}_{L/2} \\ \mathbf{\Delta}_{L/2} & -\mathbf{\Delta}_{L/2} \end{bmatrix}\right]$
    end
```

4.3 Data Structures and Mapping

The Local-Distributed method as well as the Transpose-Split method are based on the natural mapping of block rows into the hypercube, where block row i is stored in processor i. The particular way we map these block rows into the hypercube is equivalent to McBryan and Van de Velde's (1987) *distributed column* mapping. Figure 4.2 shows the mapping of a matrix into 4 processors. Local FFTs are done along the rows, with the vectors oriented perpendicular to the direction of the FFT computations.

The Vector-Radix-2 method and the Block method both partition the matrix into submatrices, placing one submatrix in each processor. The Block method involves distributed FFTs along both rows and columns. The Vector-Radix-2

000^00	000^01	000^10	000^11
001^00	001^01	001^10	001^11
010^00	010^01	010^10	010^11
011^00	011^01	011^10	011^11
100^00	100^01	100^10	100^11
101^00	101^01	101^10	101^11
110^00	110^01	110^10	110^11
111^00	111^01	111^10	111^11

Figure 4.3: 5-cube = 3-cube $\otimes$ 2-cube.

intermingling steps involve total exchanges among two-dimensional subcubes. The adjacency structure of the communication in both methods requires viewing a d-dimensional hypercube as a "cross-product" of a k-dimensional by an l-dimensional hypercube where $d = k + l$.

To illustrate what we mean, Figure 4.3 gives a low dimension example. Suppose we have a 5-cube viewed as a "cross-product" of a 3-cube and a 2-cube. Each node has a binary label A with 5 bits that can be construed as the concatenation of a binary label $b_1b_2b_3$ of 3 bits with a binary label c_1c_2 of 2 bits, i.e.

$$A = b_1b_2b_3c_1c_2$$

Now we place each node so that in the x-direction c_1c_2 go from 00 to 11; and in the y-direction $b_1b_2b_3$ go from 000 to 111.

This produces a two-dimensional array of nodes possessing "powers of two" adjacency for both rows *and* columns. Notice that we have exactly the adjacency structure necessary for a two-dimensional radix-2 FFT of a matrix mapped into the processors in a natural fashion.

This scheme extends to arbitrary dimensions. Suppose we have a d-dimensional hypercube where $d = k + l + m$. We can write the binary label of each node

$$A = b_1b_2 \dots b_kc_1c_2 \dots c_lf_1f_2 \dots f_m$$

and map our nodes in three dimensions. Denoting concatenation by exponentiation, we order the $f_1f_2 \dots f_m$ from 0^m to 1^m along the x-direction, the $c_1c_2 \dots c_l$ from 0^l to 1^l along the y-direction and the $b_1b_2 \dots b_k$ from 0^k to 1^k along the z-direction. Therefore if $d = d_1 + d_2 + \dots + d_j$ ($j=$ dimension of data structure), a j-dimensional array can be mapped into the d-dimensional hypercube so that in each direction $k = 1, 2, \dots, j$, the interconnections are exactly those of an d_k-dimensional hypercube.

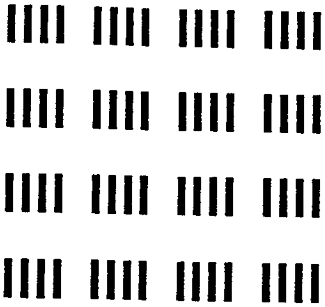

Figure 4.4: 16 columns mapped into 16 processors.

A block matrix mapping maps submatrices into the hypercube so that block A_{ij} goes into processor $i\hat{}j$ (where ^denotes concatenation). Figure 4.4 shows the mapping of a matrix into a hypercube of 16 nodes.

In the Block method, the FFT computations must include distributed butterfly computations in both directions. This method can obviously be extended to higher dimensions. For the Vector-Radix method, higher dimensional FFTs require total exchanges among higher dimensional subcubes.

4.4 Efficient Data Transposition on the Intel iPSC

An extensive study on matrix transposition on hypercube architectures was done by Ching-Tien Ho and S. Lennart Johnsson (1986). The algorithm we use in the implementations comes from a recursive block transpose algorithm described in Johnsson (1985a, 1985b) and McBryan and Van de Velde (1985) and coded by Ho. Basically a matrix of dimension n mapped into P processors can be fully transposed in a time of order $(n^2/P)\log_2 P$. The following recursion fully describes the transpose algorithm.

Algorithm 4.4.1 Transpose(A):

```
    $A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}$
    $A_{01} \leftrightarrow A_{10}$
    $Transpose(A_{00})$   $Transpose(A_{01})$
    $Transpose(A_{10})$   $Transpose(A_{11})$
```

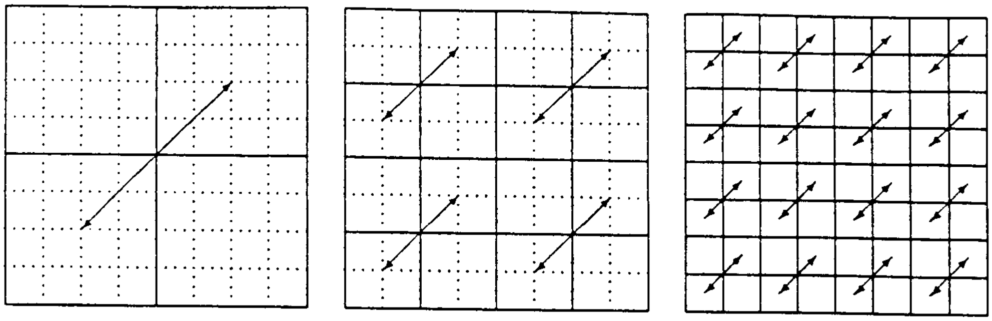

Figure 4.5: Block Recursive Transpose.

Figure 4.5 describes the recursive block transpose for an 8-by-8 block matrix mapped into a hypercube of 8 nodes. Each node receives one block row of the matrix. And the arrows denote the scope of the blocks that are transposed at each step. Figure 4.6 concentrates on processor 2 and highlights in black the portions of the matrix that are to be exchanged at each step. Since the matrix is mapped in the distributed columns formation, each dark block consists of contiguous elements and can be sent as a single message. The initial step involves an exchange between processors whose node numbers differ at the highest order bit, or bit number $d-1$. The messages are of length $L/2$, where L is the length of the portion of our matrix resident in each processor. In our case, L is the number of data-points in a block row multiplied by the number of bytes per data-point. In the next step, each processor now has 2 messages of length $L/4$ and these are exchanged between processors whose node numbers differ at the 2nd-highest order bit, (bit number $d-2$). Counting the steps from $k=0$ to $k=d-1$, we have the property that at the kth step, there are 2^k messages of length $L/2^{k+1}$ exchanged between processors whose node numbers differ at the $(d-k-1)$th bit.

A simple procedure describes where each chunk goes at each step. First we define a function

$$ithbit(\mu,k)=\begin{cases}0 & \text{if the } k\text{th bit of } \mu \text{ is } 0\\ 1 & \text{if the } k\text{th bit of } \mu \text{ is } 1\end{cases}$$

For example: $ithbit(20,2)=1$ since $20=10100_2$.

Algorithm 4.4.2 Recursive Block Transpose

```
/* $\mu$ = processor id; $n = 2^t$; $m = n/P$ */
```

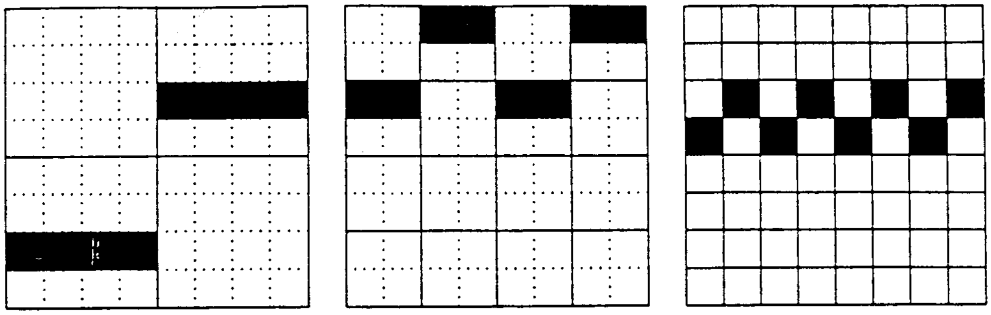

Figure 4.6: Blocks Exchanged by Processor # 2.

```
/* processor $\mu$ has: */
/* $\mathbf{A}_{id[\mu]} \equiv \mathbf{A}(id[\mu] \cdot m : (id[\mu]+1) \cdot m - 1, 0 : n-1)$*/
  for $k = 0 : d-1$
/* partition block row $\mathbf{A}_{id[\mu]}$ into $2^{k+1}$ submatrices of width $n/2^{k+1}$ */
/* with the submatrices numbered from $j = 0 \ldots 2^{k+1} - 1$ */
  $w \leftarrow n/2^{k+1}$
  for $j = 0 : 2^{k+1} - 1$
    $\mathbf{A}_j \leftarrow \mathbf{A}_{id[\mu]}(:, j \cdot w : (j+1) \cdot w - 1)$
  end
/* send $2^k$ submatrices to neighbor in $(d-k-1)$st direction */
/* receive $2^k$ submatrices from neighbor in $(d-k-1)$st direction */
/* determine processor $\rho$ to exchange with */
  $id[\rho] = flip(id[\mu], d-k-1)$; $\rho = id^{-1}[id[\rho]]$
  if $ithbit(\mu, d-k-1) = 0$ then
    send($\{\mathbf{A}_j : j$  is odd$\}, m \cdot w, \rho$)
    recv($\{\mathbf{A}_j : j$  is odd$\}, m \cdot w, \rho$)
  end if
  if $ithbit(\mu, d-k-1) = 1$ then
    send($\{\mathbf{A}_j : j$  is even$\}, m \cdot w, \rho$)
    recv($\{\mathbf{A}_j : j$  is even$\}, m \cdot w, \rho$)
  end if
  end
```

Notice that as the algorithm progresses, there are more and more messages of smaller and smaller sizes to exchange. This can cause a tremendous amount of overhead in terms of message start-ups. Since the start-up time is, in general, several orders of magnitude slower than the data transfer rate, we would prefer to buffer the smaller chunks into the largest possible array and transmit a fewer messages. Of course there are constraints to this approach also, taking into consideration the largest message size B_m and also the "copy" rate of the processors.

Ho and Johnsson (1986) present an optimized version of this algorithm for the Intel iPSC taking into account the communication startup time, the copying time for buffering, and other peculiarities of the Intel cube such as its allowing communication over only one port at time. They have discovered that the time it takes to copy $4K$ bytes is about 37 milliseconds and that the time for copying 256 bytes is approximated the same as one communication start-up. Hence their algorithm sends blocks of length at least 64 floating-point numbers without buffering. We have chosen to implement their algorithm for complex numbers. One should also note that if $P = 2^d$, then after d steps in the block recursive transpose algorithm, the remaining transpose operations are local to each processor.

4.5 Comments on Data Ordering, Vectorization, Mixed Radix Implementations, and Storage

One-dimensional transforms of long vectors can also be implemented in procedures similar to the methods described here by utilizing the *twiddle factor* algorithm where the sequence is initially mapped in column order by distributed columns, i.e. block rows of columns. DFTs are taken along the rows, followed by a point-wise multiplication of twiddle factors. Then the whole array is transposed and the "column" DFTs are again done on the rows.

Algorithm 4.5.1 Twiddle Factor FFT of $\mathbf{X} = \mathbf{x}_{n_1 \times n_2}$

```
    $\mathbf{X} \leftarrow \mathbf{X}\mathbf{F}_{n_2}$
    $\mathbf{X} \leftarrow \mathbf{T} * \mathbf{X}$
    $\mathbf{X} \leftarrow \mathbf{X}^T$
    $\mathbf{X} \leftarrow \mathbf{X}\mathbf{F}_{n_1}$
```

Recall that $*$ denotes the point-wise multiplication of two matrices. Since the results $\mathbf{x}(k_1, k_2)$ are in transposed order, the resulting array $\mathbf{x}$, which actually contains $\mathbf{x}^T$ is in the correct order. Therefore no extra transpose is needed to bring the data back to its original order. A two-dimensional DFT proceeds in exactly the same manner except for the omission of the twiddle factor multiplication. However, the resulting array is in transposed order.

The usual radix-two in-place FFT algorithms require a data permutation either at the start of the procedure or at the end. This permutation is the well-known bit-reversing permutation. In the Transpose-Split method, the FFTs are performed locally so that this permutation is also done independently and locally. We can also avoid bit-reversing by using Stockham FFTs locally, albeit this necessitates an extra array of workspace. In any case, no distributed bit-reversing operation is necessary. The Local-Distributed methods, require up to an extra d communications [Swarztrauber (1986a)], on a cube which can communicate simultaneously in all directions, to perform distributed bit-reversal. This is also discussed by Van Loan (1987). The Block and the Vector-Radix FFT results in a need for distributed bit-reversal along both the horizontal and vertical directions, or an extra $2\sqrt{d}$ communications if Swarztrauber's method is utilized. On the Intel cube, autosort FFTs incur too much communication overhead to be efficient. However if the data is to be transformed and then inverse transformed, there would be no need to unscramble in the transform domain since there are algorithms which take bit-reversed data on input and return the inverse transform in natural order.

If the data array is mapped within each processor so that its vector orientation is perpendicular to the direction of the FFT, one can effectively vectorize the butterfly computations. The vector length is equal to the number of multiple transforms and hence one would prefer to do the FFT of m transforms in parallel rather than do one transform after the other. See [Hockney and Jesshope (1981) and Swarztrauber (1986a)]. The basic operation here is a vector SAXPY where

$$\mathbf{v} \leftarrow \alpha \mathbf{x} + \mathbf{y}$$

$\mathbf{v}$, $\mathbf{x}$, and $\mathbf{y}$ are vectors with α a scalar. Since the direction of the second FFT pass is perpendicular to that of the first pass, a transpose is needed between the two stages. For the Transpose-Split method this means that *after* the block transpose, one must complete the transpose by performing transposes on all the submatrices locally. For the two distributed methods, a local transpose of the array resident in each node is required to retain the correct orientation of the vectors. The Floating Point Systems implementation of the Block method consists of mapping the array X in sub-blocks into the nodes of the hypercube. Since the T-Series consists of vector boards, local transposes are done on each submatrix to keep the correct orientation for vectorizing.

Vector length is also an important issue when doing multiple transforms on vector processors. Suppose we have an N-by-N array and p processors. The length of the vectors in the direction perpendicular to the FFT computations should be as long as possible (up to the length of the vector register) so as to take full advantage of vector operations. The Transpose-Split method requires each processor to do N/p FFTs of length N simultaneously and thus has an effective

Table 4.1: Multiple Transforms and Vector Length.

	horizontal				vertical			
	TS	LD	B	VR	TS	LD	B	VR
m[†]	$\frac{N}{p}$	$\frac{N}{p}$	$\frac{N}{\sqrt{p}}$	$\frac{N}{\sqrt{p}}$	$\frac{N}{p}$	N	$\frac{N}{\sqrt{p}}$	$\frac{N}{\sqrt{p}}$
n[‡]	N	N	$\frac{N}{\sqrt{p}}$	$\frac{N}{\sqrt{p}}$	N	$\frac{N}{p}$	$\frac{N}{\sqrt{p}}$	$\frac{N}{\sqrt{p}}$

[†] # of FFTs = vector length

[‡] length of portion in each processor

vector length of N/p. The Block method has each processor responsible for $N/\sqrt{p}$ FFTs of which only $N/\sqrt{p}$ elements of each FFT are processor-local. The vector length here is $N/\sqrt{p}$ and is $\sqrt{p}$ times longer than the Transpose-Split method. The Local-Distributed method has the same characteristics of the Transpose-Split method during its local phase, vectors of length N/p; however in its distributed phase, the effective vector lengths are N. These observations are summarized in Table 4.1

On the Floating Point Systems T-Series hypercube, the vector registers are of length 128, and thus the Block method is favored here since its average vector length is the longest. For example, if $p = 64$, and $N = 1024$, the Block method would give vectors of length 128, whereas the Transpose-Split produces vectors of length 16. Since the vector boards are so fast (50–60 Mflops for the 2D-FFT) [Miles *et al.* (1987)], one would like to "fill up" the registers to minimize the loading and storage of data. In fact, this was a compelling reason for chosing the Block method for the FPS T-Series implementation.

A further consideration is whether the problem at hand requires vectors whose lengths are not a power of two. As the powers of two get large, they are spaced further and further apart. A partial differential equation problem may take place on a grid which is 100-by-100. Most of the distributed methods in the literature presume the use of a radix-two FFT on vector length $= 2^k$ (some k an integer). Therefore a point brought out by McBryan is that the use of the Transpose-Split method permits one to use any FFT subroutine that accepts general n. Given that there is a good transpose algorithm, such a method is more portable and simple for the end-user to implement. However, the Local-Distributed Method which we have proposed here can be used to handle transforms of length $n = 2^k m$. This is because the twiddle factor algorithm can be adapted so that the local in-house FFTs are of length m and the distributed FFTs running the other direction are of length 2^k, where a radix-two algorithm is used. This method which we call the Mixed-Radix-2 will be discussed in Chapter 7. The Vector-Radix method is generalized by Harris *et al.* (1977) to include arbitrary radices and non-square arrays.

Finally, storage requirements differ for these methods. In general, the distributed methods require an extra N^2/p points of storage per processor if one wants to do only one exchange per distributed butterfly step. This is because all information must be sent to the other processor before the updating is done. If one is allowed to incur two exchanges per distributed butterfly step, as in the FPS implementation, there is no extra buffer space necessary for storage. The Vector-Radix-2 method can also be done with two exchanges per intermingling step. Here each processor sends a quarter of its data to each of three members belonging to a specific two-dimensional subcube as itself. It also receives three quarter pieces from each member. Updates are done and another total exchange is necessary to repatriate the data. This means that no extra buffer is needed. The Transpose-Split method can be done totally in place, if unbuffered transposing is used. However, as Ho and Johnsson (1986) have noticed, this can cause the transpose time to grow exponentially in the number of dimensions. Therefore, one extra buffer is needed to collect the noncontigous blocks and ship them all out at once. For the Intel iPSC, this buffer is not too large (only 64 floating point numbers).

4.6 Implementation and Timings

The Transpose-Split, Local-Distributed, Block and Vector-Radix methods for the two dimensional FFT are implemented on the Intel iPSC/D4MX hypercube running XENIX R3.4, iPSC Release 3.1 with Exelan R3.3 networking software. The code was written using Ryan-McFarland FORTRAN. Vector boards are *not* available so that all computations within a node are done serially. The Transpose-Split method (Algorithm 4.6.1) uses local FFTs in both the vertical and horizontal stages and a recursive block transpose routine. The transpose code used was a modified version of Ching-Tien Ho's with the only change being the removal of the aforementioned buffering.

The following algorithms give a general overview of what the whole system of processors are doing. For example, the construct

$$\textbf{do}(\textit{in parallel}, \quad i = 0, \ldots, P-1)$$

means that all P processors are asynchronously performing the same operation. The construct

$$\textbf{do}(\textit{for processor } p(i)) \quad (\textit{in parallel})$$

means that processor *p(i)* is performing that portion of the computations, and that all processors are asynchronously performing their respective portions of work. with the necessary communications and updating.

Algorithm 4.6.1 Transpose-Split 2-D FFT

```
/* Find the 2D-FFT of $\mathbf{X}$ by using $P$ processors */
  partition $\mathbf{X}$ into block rows $\mathbf{X}_i$, $i = 0, \ldots, P-1$
  map $\mathbf{X}_i$ into processor p(i)
  do (in parallel, $i = 0, \ldots, P-1$)
    $\mathbf{X}_i \leftarrow \mathbf{X}_i\mathbf{F}_n^T$ /* GS1 FFT */
  end do
  call Recursive Block Transpose ($\mathbf{X}$)
  do (in parallel, $i = 0, \ldots, P-1$)
    $\mathbf{X}_i \leftarrow \mathbf{X}_i\mathbf{F}_n^T$ /* GS1 FFT */
  end do
```

The method of implementation for distributed FFTs is that of Chapter 3 where the sequences are mapped in natural order into the hypercube. The isomorphism of the radix-2 FFT signal flow graph to the hypercube interconnection scheme is used to determine the inter-node communication pattern. The FFT subroutine implemented is the Gentleman-Sande (GS1) algorithm. The Local-Distributed method is illustrated by Algorithm 4.6.2 and the Block method by Algorithm 4.6.3. Figure 4.7 shows the communications necessary during the horizontal and vertical FFT passes for the Block method.

Algorithm 4.6.2 Local-Distributed 2D-FFT

```
/* Find the 2D-FFT of $\mathbf{X}$ by using $P$ processors */
  partition $\mathbf{X}$ into block rows $\mathbf{X}_i$, $i = 0, \ldots, P-1$
  map $\mathbf{X}_i$ into processor p(i)
/* Local FFTs */
  do (in parallel, $i = 0, \ldots, P-1$)
    $\mathbf{X}_i \leftarrow \mathbf{X}_i\mathbf{F}_n^T$ /* GS1 FFT */
  end do
/* Distributed FFTs */
  do (for processor p(i)) (in parallel)
    $\mathbf{X}_i \leftarrow [\mathbf{F}_n\mathbf{X}]_i$ /* distributed GS1 FFT */
  end
```

Algorithm 4.6.3 Block 2D-FFT

```
/* Find the 2D-FFT of $\mathbf{X}$ by using */
/* an array of $\sqrt{P} \times \sqrt{P}$ processors */
  partition $\mathbf{X}$ into blocks of sub-matrices $\mathbf{X}_{ij}$
    $i = 0, \ldots, \sqrt{P} - 1,\ j = 0, \ldots, \sqrt{P} - 1$
  map $\mathbf{X}_{ij}$ into processor p(i^j)
/* Horizontal Distributed FFTs */
  do (for processor p(i^j)) (in parallel)
    $\mathbf{X}_{ij} \leftarrow [\mathbf{X}\mathbf{F}_n^T]_{ij}$
  end
/* Vertical Distributed FFTs */
  do (for processor p(i^j)) (in parallel)
    $\mathbf{X}_{ij} \leftarrow [\mathbf{F}_n\mathbf{X}]_{ij}$
  end
```

The Vector-Radix method (Algorithm 4.6.4) is implemented only partially. In other words, the local independent 2D-DFTs are done by a conventional row-column 2D-FFT subroutine found in Press *et al.* (1986), and only the intermingling steps involve updating by the Vector-Radix method. Since our hypercube has only 16 nodes, only two such intermingling steps are done. In order to perform in-place computation and do away with the need for extra buffers, the Vector-Radix updating was done so that each processor in the corner of its two-dimensional subcube was responsible for all the updating of that particular corner of the data for its three partners as well as for itself. For example, the processor in the south-west corner of the two-dimensional subcube for that iteration will send its north-west corner to the processor above it in exchange for the south-west corner of that processor. It will also send its north-east corner to the processor diagonally across from it in exchange for the south-west corner of that processor. And it will send its south-east corner to the processor to its right in exchange for the south-east corner of that processor. This processor will update all the submatrices it receives and then do a reverse exchange with all its partners to get its own updated corners back. The intermingling steps for processor *p0100* is illustrated schematically in in Figure 4.8.

Algorithm 4.6.4 Vector-Radix 2D-FFT

```
/* Find the 2D-FFT of $\mathbf{X}$ */
/* an array of $\sqrt{P} \times \sqrt{P}$ processors */
  partition $\mathbf{X}$ into blocks of sub-matrices $\mathbf{X}_{ij}$
    $i = 0, \ldots, \sqrt{P} - 1,\ j = 0, \ldots, \sqrt{P} - 1$
  map $\mathbf{X}_{ij}$ into processor p(i^j)
```

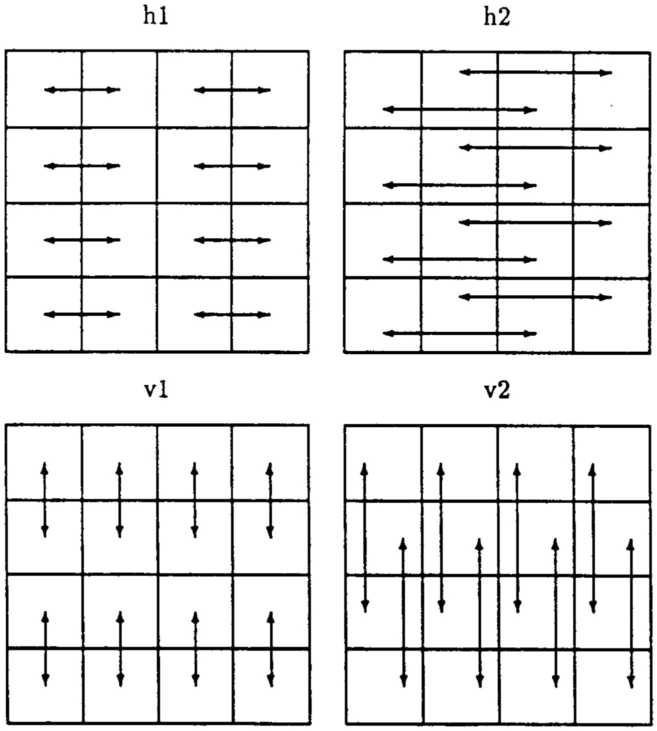

Figure 4.7: Block Method 2D-FFT Communications for a 16-node hypercube

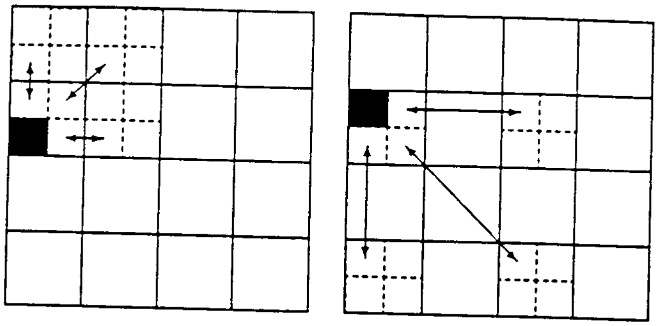

Figure 4.8: Vector-Radix Exchanges for Processor *p0100*

```
/* Local 2D-FFTs of $\mathbf{X}_{ij}$
  do (in parallel, $i = 0, \ldots, \sqrt{P}-1,\ j = 0, \ldots, \sqrt{P}-1$)
    $\mathbf{X}_{ij} \leftarrow \mathbf{F}_{n/\sqrt{P}}\mathbf{X}_{ij}\mathbf{F}^T_{n/\sqrt{P}}$ /* 2D-FFT */
  end do
/* intermingling steps for Vector-Radix method */
  for $q = 0, \ldots, d$
    $\mathbf{X}_{ij} \leftarrow [\mathbf{A}_q\mathbf{X}\mathbf{A}_q^T]_{ij}$
  end
```

Here $\mathbf{A}_q$ is defined as in Chapter 1, ($L = 2^q$)

$$\mathbf{A}_q = \mathbf{I}_{n/L} \otimes \begin{bmatrix} \mathbf{I}_{L/2} & \mathbf{\Delta}_{L/2} \\ \mathbf{I}_{L/2} & -\mathbf{\Delta}_{L/2} \end{bmatrix}$$

where $\mathbf{\Delta}_{L/2} = diag(1, \omega_L, \ldots, \omega_L^{L/2-1})$.

Computation time as well as communication time is displayed for all four methods. A portion of the communication time is reflected in a processor blocking while awaiting data that it is to receive. The total execution time is shown in Table 4.2. Table 4.3 shows the computation time required by each method while Table 4.4 displays the communication time whereas Table 4.5 displays the amount of time a processor spends blocked. Times are given in milliseconds and range from the fastest processor to the slowest processor.

Table 4.2: Total Execution Time.

size	Total Execution Time			
dim	TS	LD	B	VR
16 × 16				
1	155	145–160	–	–
2	90	80–95	90–110	160
3	75–125	55–70	65–85	–
4	140–145	105–150	90–110	125–190
32 × 32				
1	710	665–715	–	–
2	385–390	340–395	365–430	465–470
3	230–275	180–230	235–290	–
4	170–220	105–160	110–205	215–330
64 × 64				
1	3280	3110–3305	–	–
2	1700	1550–1750	1675–1895	1850–1895
3	910–945	785–935	1045–1230	–
4	525–570	415–585	425–555	685–840
128 × 128				
1	15130	15025–15230	–	–
2	7825–7840	7165–7930	7820–8630	7735–7820
3	3965	3575–4145	4835–5480	–
4	2155	1800–2225	1895–2350	2345–2490
256 × 256				
3	17650–17665	16870–18605	22220–24660	–
4	9040–9215	8135–9615	8695–10370	9040–9215

We also mention that our implementation of the Block method differs from Floating Point System's in that only one exchange is incurred during each distributed butterfly step. Since the Intel cannot overlap communication and computation, the added cost of performing two exchanges would degrade the performance of the distributed methods without giving any basis for comparison.

The results show timings that are roughly within 10% of each other for the four different methods. However if we look at the break-down of communication versus computational time, some interesting differences surface.

The computational times of the Transpose-Split method and the Vector-Radix method are the most load balanced. The Transpose-Split computations exhibit the most parallelism as all of the actual FFT steps are independent and done in parallel. For the Vector-Radix method the individual 2D-FFTs done locally within each processor are also done entirely independently and in parallel. As expected, the computational times in the distributed methods show a small amount of imbalance, with greater gaps between the faster and slowest processor than shown by the Transpose-Split method and the Vector-Radix method. While the Local-Distributed method stays about even with the Transpose-Split method as far as computation is concerned, the Block method on the average takes a bit longer. Since the same FFT algorithm was used for the three row-column methods, we conjecture that this might be due to the fact that processors in the Block method get interrupted during computation at two stages, during both the vertical and horizontal FFTs, whereas the Local-Distributed method only gets interrupted during its vertical FFT. The Transpose-Split method is only interrupted once, between the horizontal and vertical stages. Of course the effects of these interruptions can be minimized if the processors are able to simultaneously communicate and compute. The computational time for the Vector-Radix method is on the average faster than any of the row-column methods, especially on large problems. Keep in mind that we have not taken advantage of the 25% reduction of multiplications since our Vector-Radix implementation does not recurse all the way down to the 2 × 2 trivial 2-D transform. Instead, only the intermingling steps are done via the Vector-Radix splitting and the processor local 2D-FFTs are done by the regular row-column approach. Hence even at this limited level, we see that the Vector-Radix shows potential in speeding up computation.

The analysis of the communication times show a different story. Here the three row-column methods exhibit roughly the same range of times for communication. As already mentioned, our implementation of the Vector-Radix 2-cube total exchange is very primitive since the Intel iPSC cannot communicate across more than one link at the same time. Hence a huge amount of blocking is seen to be responsible for the slowness of the Vector-Radix communication. The blocking time for the three row-column methods is about the same. This demonstrates that

Table 4.3: Computational Time of 2D-FFT.

size	Computation Time			
dim	TS[†]	LD	B	VR
16 × 16				
1	150/10	140–155	–	–
2	90/10	70–85	80–100	90–100
3	55/5	40–65	50–70	–
4	40/5	15–40	25–40	30–45
32 × 32				
1	705/25	645–695	–	–
2	375/15	320–380	345–410	380–385
3	210/10	155–215	210–270	–
4	125/10	85–120	80–135	115–130
64 × 64				
1	3250/80	3055–3250	–	–
2	1670/50	1490–1690	1615–1835	1625–1630
3	880/30	735–885	995–1180	–
4	480/20	380–470	385–515	445–460
128 × 128				
1	15015/285	14805–15010	–	–
2	7575/155	6935–7705	7565–8405	7235–7250
3	3870/90	3400–3975	4660–5315	–
4	2015/60	1680–2075	1775–2225	1895–1915
256 × 256				
3	17325/315	16150–17875	21515–23975	–
4	8815/185	7650–9145	8205–9910	8300–8320

[†] total computation time and time for internal transpose

the load imbalance of the distributed methods is not really much of a problem as far as blocking between **send** and **recv** are concerned. One interesting point is that the Transpose-Split communication results actually show an *increase* in time for the $N = 16 \times 16$ problem with increasing number of processors. This is due primarily to the added complexity of having to send smaller and smaller messages or extra buffering costs. Even though we have implemented the unbuffered transpose, the results from the use of buffered transposing still show this increase. See Section 4.7.

In the next section we shall see that while the distributed methods require $O(\log_2 P)$ start-ups for communication, the transpose method could possibly require up to $O(P)$ startups if not done carefully. We also consider the effective vector length of the different methods and hypothesize what would happen if the node processors have vector boards.

4.7 Discussion and Model

Models of computational and communicational complexity are often useful in giving general guidelines to the benefits of various methods of implementation. Since FFT implementations are usually communication bound, we first consider the analysis of simply transferring data among the processors as specified by the recursive block transform procedure and the distributed methods. The vectorization of multiple transforms are dealt with next. Finally we give an estimate of the total time required.

Suppose we have an n-by-n array and $P = 2^d$ processors. Throughout this discussion we shall assume that P is an even power of two. Assuming that P divides n, each processor would have n^2/P points. The recursive block transpose algorithm requires d steps where $n^2/2P$ points are exchanged per step. Meanwhile both distributed FFT algorithms have d steps involving trans-processor butterflies. Each step requires the exchange of n^2/P points. One can see immediately that twice as much data points are exchanged at each step by the distributed methods. Notice however, that these points are all contiguous, so that there is no overhead of sending multiple messages or need to copy into a buffer. However, due to the algebraic structure of the butterflies, an extra buffer array of n^2/P points is needed for each processor since it cannot overwrite its array until after the butterfly computation. The extra buffer is not required in the Floating Point System implementation, however an extra exchange per trans-processor butterfly step is needed. Three extra buffers would be needed for the Vector-Radix method *if* we did not use this trick of exchanging twice per intermingling step. Therefore in our implementation, we incur the cost of the extra exchange and hence no extra

Table 4.4: Communication Time Including Blocking Overhead.

size	Communication Time/Blocked Time			
dim	TS	LD	B	VR
16 × 16				
1	5	5	–	–
2	5–10	5–10	5–10	60–65
3	15–70	60–65	5–10	–
4	95–110	65–120	55-60	80–155
32 × 32				
1	5	15	–	–
2	10–15	15–20	15–20	85–90
3	25–65	15–20	15	–
4	35–95	5–20	15–65	90–205
64 × 64				
1	30	55–60	–	–
2	30–35	60	50–55	220–265
3	25–70	45–85	40–45	–
4	40–90	35–40	30–40	230–385
128 × 128				
1	110	220	–	–
2	245–265	220	220–255	495–570
3	80–95	165–170	160–165	–
4	115–145	105–120	105–120	445–585
256 × 256				
3	310–335	650–710	660–695	–
4	230–400	445–510	435–495	860–1355

Table 4.5: Time Spent Blocked during Communication

size	Blocked Time			
dim	TS	LD	B	VR
16 × 16				
1	0	0	–	–
2	0–5	0–5	0–5	10–45
3	5–60	0–60	0–5	–
4	15–85	0–110	5–50	15–90
32 × 32				
1	0	0	–	–
2	0–5	0–5	5	30–65
3	0–50	0–10	0–5	–
4	5–65	0–15	0–60	30–175
64 × 64				
1	5	10	–	–
2	5	0	10	10–175
3	5–20	5–55	0–15	–
4	5–60	0–15	5–10	115–355
128 × 128				
1	15–40	35	–	–
2	20–155	30–35	35–60	60–340
3	10–25	20–30	20–25	–
4	20–85	10–20	15–20	125–405
256 × 256				
3	40–60	90–150	95–130	–
4	45–220	65–170	60–135	95–715

buffers are necessary.

Let τ be the data startup time or communication latency time, B_m the maximum packet size that can be transferred at a time, and t_c the per element transfer time. Denote time by

$$T^{method}_{operation},$$

the time required for a certain operation by a certain method.

The total communication overhead is measured by

$$t_c\text{ (number of elements to be sent)} + \tau\text{ (number of start-ups)}$$

First let's look at unbuffered transpose communication. Ho and Johnsson [1986] show that the complexity for unbuffered communication is

$$T^{TS}_{comm} = d\frac{n^2}{P}t_c + \left(P + \left\lceil\frac{n^2}{2B_mP}\right\rceil \min(d, \log_2\left\lceil\frac{n^2}{B_mP}\right\rceil) - \frac{n^2}{B_mP}\right)2\tau$$

The complexity for startups is $O(P)$ and grows exponentially with the dimension $d = \log_2 P$ of the cube. This can be seen easily where, ignoring B_m, the complexity becomes

$$T^{TS}_{comm} = d\frac{n^2}{P}t_c + (\sum_{i=0}^{d-1} 2^i)2\tau.$$

When $\frac{n^2}{P} > B_m$, we must take into account these extra start-ups and the number of start-ups is $O(P + \log_2 P\left\lceil n^2/2PB_m\right\rceil)$.

Buffered communication makes sense only when n^2/P remains small and the complexity is approximately $O(\log_2 P)$ start-ups growing linearly with cube dimension. Here one must also take into account the extra time required for buffering as well as the fact that the effective buffer is small, so that on large problems the transpose is essentially unbuffered.

The Local-Distributed method has communication complexity

$$T^{LD}_{comm} = 2d\frac{n^2}{P}t_c + 2d\tau\max(1, \left\lceil\frac{n^2}{B_mP}\right\rceil).$$

as does the Block method.

$$T^{B}_{comm} = T^{LD}_{comm}$$

Here the complexity only grows linearly with the number of cube dimensions. however, there is twice as much data to transfer and when the problem gets large. we get a measurement proportional to $\left\lceil n^2/B_mP\right\rceil$ times the start-up costs. Thus the complexity of the distributed methods is $O(\log_2 P)$ start-ups when $n^2/P < B_m$ and $O(\log_2 P \lceil n^2/B_m P \rceil)$ when $n^2/P > B_m$.

The communication complexity of our in-place Vector-Radix FFT necessitates two exchanges per intermingling step. Recall that this scheme is similar to that of the Floating Point Systems implementation and also Walton's one-dimensional FFT implementation where portions of the matrix or vector are exchanged and then sent "home". Here each portion is of size $\frac{3}{4}n^2/P$ and each processor communicates with three processors. Hence the communication complexity is

$$T^{VR}_{comm} = 4\log_2\sqrt{P}(\frac{3}{4}\frac{n^2}{P}t_c + 3\tau)$$

If the three exchanges per intermingling step can be done simultaneously, the startup term becomes just

$$4\log_2\sqrt{P}\tau$$

In summary,

$$\begin{aligned}
T^{TS}_{comm} &\asymp 2(P-1)\tau + (\log_2 P)\frac{n^2}{P}t_c \\
T^{LD}_{comm} &\asymp (2\log_2 P)\left(\frac{n^2}{P}t_c + \tau\right) \\
T^{B}_{comm} &\asymp (2\log_2 P)\left(\frac{n^2}{P}t_c + \tau\right) \\
T^{VR}_{comm} &\asymp (2\log_2 P)\left(\frac{3}{4}\frac{n^2}{P}t_c + 3\tau\right)
\end{aligned}$$

Table 4.6 illustrates data transfer time (in milliseconds) without any computation. Two different transposes are compared for the Transpose-Split method, the unbuffered and buffered methods described in Ho and Johnsson (1986). One can see that for small problems, the buffered recursive transpose and the distributed methods take about the same time communicating, even though the distributed methods send twice as much data. This is because the data lie in contiguous locations and thus require only one message. By comparing buffered and unbuffered transposing we can find the cut-off point beyond which it makes no difference whether to buffer or not to buffer. Here we see that buffering just does not matter when the size is larger than 64 × 64. The communication times of the distributed methods are comparable to buffered transpose times until the problem size gets larger enough so that the maximum packet size B_m is reached. Here we see that for the 128 × 128 problem, communication for the distributed methods is roughly twice the time for transposing. Since the number of data points moved is also twice that of transposing, this is consistent. For $d = 4$, the timings are about equal since the message packets are small enough so that they can be all transferred in one step. Finally as expected the Vector-Radix communication was the slowest because of the inefficient implementation on the Intel iPSC. It is expected, however that given an efficient "total exchange" capability where processors can communicate simultaneously, that the communication times should speed up making the Vector-Radix method viable and feasible.

Next we model computational time for the three row-column methods on vector nodes. Hockney and Jesshope (1981) give a model for performing M independent transforms of length N by using the best serial algorithm and vectorizing the arithmetic. Let α^{-1} be the per flop computation time, and $N_{1/2}$ be the vector length required to achieve half of the asymptotic performance. Then

$$T_{multft} = 5\alpha N(N_{1/2} + M)\log_2 N$$

This model cannot be used for the Vector-Radix method since the "multiple transforms" are done simultaneously in two-dimensions. However we know that the Vector-Radix method, when recursed down to the trivial 2 x 2 2D-FFT has 25% less multiplications than the row-column approach.

Using this model we see that for the Transpose-Split method each processor does n/P transforms of length n twice, hence

$$T^{TS}_{comp} = 2\alpha(5n(N_{1/2} + \frac{n}{P})\log_2 n)$$

For the Block method, each processor is responsible for $1/\sqrt{P}$th of the work for $n/\sqrt{P}$ transforms of length n and this occurs twice, so

$$T^{B}_{comp} = 2\alpha(5\frac{n}{\sqrt{P}}(N_{1/2} + \frac{n}{\sqrt{P}})\log_2 n)$$

Finally for the Local-Distributed, each processor does n/P transforms of length n during the horizontal phase, and $(1/P)$th of the work for n transforms of length n during the vertical stage.

$$T^{LD}_{comp} = 5\alpha n(N_{1/2} + \frac{n}{P})\log_2 n + 5\alpha\frac{n}{P}(N_{1/2} + n)\log_2 n$$

The difference in computation time among the three methods comes from the $N_{1/2}$ term, with $T^B_v = (n/\sqrt{P})N_{1/2}$, $T^{TS}_v = nN_{1/2}$ and $T^{LD}_v = 1/2(n + n/P)N_{1/2}$. Comparing coefficients, we see that

$$T^{B}_{comp} < T^{LD}_{comp} < T^{TS}_{comp} \quad \text{when } P > 1$$

Table 4.6: Communication Times without Intervening Computation.

Analysis of Communication w/o Computation					
size	TS		LD	B	VR
dim	unbuff.	buff.			
16 × 16					
1	5	5	5	–	–
2	10	10	10	10–15	50–90
3	20–60	65–70	10–45	15–20	–
4	105–110	75–140	70–110	65–70	10–115
32 × 32					
1	10	10	20	–	–
2	15	10	20	20–25	45–80
3	20–65	20	15–20	25–60	–
4	45–140	20–25	20–25	25–75	15–190
64 × 64					
1	30	25–30	55	–	–
2	35–70	30–65	60	60–65	55–175
3	35–75	30–75	50–85	50–90	–
4	105–110	35–90	35–85	45–105	85–285
128 × 128					
1	115	110	220–225	–	–
2	110–115	110–115	220–225	225–230	110–235
3	140	90–95	170	175–180	–
4	80–125	80–125	120–150	175–180	95–385
256 × 256					
3	335–340	330–335	660–690	670–760	–
4	235–240	235–345	445–450	450–455	295–740

Hence in terms of vectorization, the Block method is at an advantage which increases with increasing parallelism as measured by $N_{1/2}$.

All of the vectorized FFT implementations require an internal transform of the data in order to set up the vectors in the correct orientation. Since each processor has n^2/P points the overhead here is $n^2/P \cdot t_{copy}$.

Our final model combines the communication and computational portions of the row-column methods in a straight-forward manner.

$$T^{TS} = 2\alpha(5n(N_{1/2} + \frac{n}{P})\log_2 n) + 2(P-1)\tau + (\log_2 P)\frac{n^2}{P}t_c$$

$$T^{B} = 2\alpha(5\frac{n}{\sqrt{P}}(N_{1/2} + \frac{n}{\sqrt{P}})\log_2 \frac{n}{\sqrt{P}}) + (2\log_2 P)(\frac{n^2}{P}t_c + \tau)$$

$$T^{LD} = 5\alpha n(N_{1/2} + \frac{n}{P})\log_2 n + 5\alpha\frac{n}{P}(N_{1/2} + n)\log_2 \frac{n}{P} + (2\log_2 P)(\frac{n^2}{P}t_c + \tau)$$

An analysis of the communication time shows that the coefficient of the t_c term is of the same order, but that of the τ term is clearly against the Transpose-Split method. Since the τ term represents the latency time or startup time for each communication and is several orders of magnitude larger than t_c, it is obvious that as P increases, the overhead for transpose communication will become significant. Of course buffering can reduce this overhead. However the minimum number of communications needed for recursive block transpose is still of order $\log_2 P$. Therefore, the communication needs of the distributed methods present a lower bound for the transpose communication.

4.8 Conclusion

After consideration of the model and timing results we can draw the following conclusions.

- The distributed methods are hurt by the interruption of computation during the trans-processor butterfly stages, hence their performance should be enhanced on systems which interleave computation and communication.

- The complexity of communication for distributed methods is $\log_2 P$, therefore they could show promise on systems where P is large.

- In the presence of vector processor nodes, the Block method exhibits better vectorization than any of the other methods since its working vector length is approximate $\sqrt{P}$ times longer.

- The Transpose-Split method is superior on systems which support an efficient transpose algorithm since its major deficiency is the asymptotic complexity of recursive block transpose communication growing exponentially to the number of dimensions. Presently on the Intel iPSC/System 286 where each node is connected by an Ethernet communication channel, the penalties for traversing several nodes to get a message across increases linearly with the distance. Therefore on the current system, Transpose-Split is likely to be superior when the number of processors is small.

- The Vector-Radix method is promising both in the fact that it is highly parallel and also that it requires 25% fewer complex multiplications than the row-column approaches. However a d-dimensional Vector-Radix FFT requires total exchanges between processors in a d-dimensional subcube during each intermingling step. Therefore it is possible to obtain good results for this method only if the hypercube in question is able to perform efficient "total exchange" communications.

- Future generations of hypercubes will likely support more efficient transpose algorithms and routing hardware, making the Transpose-Split methodology more efficient. We have already seen the difference between transpose methods in Table 4.6.

- The Local-Distributed method is a compromise between the first two methods. In our implementation it is competitive with the Transpose-Split method and can be used when one wishes to avoid the transpose.

Our implementation results show that on the Intel iPSC/System 286 without vector boards on the nodes, there is essentially no difference between the three row-column methods. Efficient implementation of the Vector-Radix method depends on efficient total exchange communication and is therefore this method is promising given its faster computational potential. The final analysis is that this problem is highly system dependent, and one should be aware of the advantages and disadvantages of these different methods in order to best utilize the parameters of a particular system.

Our discussion of higher-dimensional transform methods naturally leads to insights on further areas of research. A higher-dimensional DFT can be approached from any combination of the following: row-column, transposing along any dimension, or distributed block manipulations. For example, one way of looking at a 3-dimensional transform is as a 2-dimensional transform and a 1-dimensional transform mapped in planes within nodes of a linear array of processors. Another way of viewing it is to map the data into solid chunks within a three-dimensional array of processors. Finally, the use of the vector-radix approach is promising in the sense that no transposing is involved and maximal parallelism is exhibited during the independent stages. In the distributed stages, there is a total exchange between subcubes of the same dimension as the transform. While our implementation of this total exchange on the Intel is extremely primitive, we believe that this method can be viable in systems containing optimal total exchange algorithms invisible to the user. Transposing distributed data in three dimensions is extremely awkward, hence we are especially looking at vector-radix approaches to 3-dimensional FFTs as an avenue for further work.

Chapter 5

Parallel Sine Transform for the Hypercube

5.1 Introduction

In some situations, the DFT of a sequence with some sort of symmetry is required. This sequence might be real, or complex even or odd. The DFT of such a sequence usually has special structure, e.g., the DFT of a real sequence is conjugate even. Hence both storage space and computational time can be saved by utilizing the redundancy of information presented by symmetry.

The implementation of symmetric FFTs on distributed systems is nowhere as straightforward as that of the complex FFT. While most symmetric FFT algorithms utilize the complex FFT as the core computation, they are implemented with a series of pre- and post- processing steps designed to take advantage of associated symmetries to minimize the operation count. In the conventional single processor setting the savings of both storage and computational effort is a reasonable goal to pursue. We present a new parallel sine transform algorithm that minimizes the amount of communication required to perform its limited pre- and post- processing steps.

An important detail in multiprocessor FFTs is the fact that the output of the Cooley-Tukey complex FFT algorithm is generated in bit-reversed order. In the single processor case this array can be unscrambled easily with index permutations. However in the case of the hypercube, the bit-reverse unscrambling takes d parallel transmission steps, where d is the dimension of the particular hypercube. [Swarztrauber (1986a)] If the data is not unscrambled, post- processing operations for the sine transform require a significant amount of communication between processors. Since the complex FFT itself is extremely efficient and also efficiently parallelizable, it would be a shame if the cost of the pre- and post- processing operations and communications were to outweigh the benefits of parallel computation.

Our new sine transform algorithm attempts to keep pre- and post- processing communications to a minimum at the expense of a relatively small number of redundant calculations. We believe that with the high overhead associated with communication on the Intel hypercube, the total return of this algorithm will be superior to the savings of some computational effort at the expense of a severe tie-up in communications.

To summarize this chapter, we first describe the symmetric transforms and their properties. The reflection and exchange permutations are introduced to facilitate the matrix description of the transform procedure. We also show how to work these two permutations into bit-reverse permuted data vectors. A new permutation, the *recursive exchange* is introduced to manipulate the reflection permutation of sequences in bit-reversed form.

The implementation of permutations on a distributed processor system is done by communication between processors as well as local permutation of the data vector. We describe each of the relevant permutations and how to efficiently implement them on multiprocessor systems.

The parallel sine transform of this chapter algorithm pares the necessary permutation/communication needs down to a bare minimum without doubling the length of the core complex FFT computation. This is done by partially exploiting symmetry and at the same time performing some redundant computations in parallel. Our procedure is practical and can be utilized on data that is both input in natural order as well as in bit-reversed order. This provides a complete forward and inverse sine transform package.

We finish by discussing possible implementations of the existing symmetric transform algorithms on the hypercube and show how the communication complexity of these methods can potentially hinder performance. Symmetric transforms are a subset of real transforms and as such, we discuss procedures for finding the FFT of real sequences and their possible implementation on hypercubes via the complex FFTs of Chapter 3. We then consider the conventional sine and cosine transform methods of Cooley *et al.* (1970) and Swarztrauber (1982). These methods pre- process the input sequence into either conjugate even sequences or real sequences and then utilize efficient algorithms for finding real FFTs. The symmetric transform is then extracted from the real FFT by post- processing computations. We show that the associated permutations involved with these data manipulations, when compounded to that of the real FFTs and bit-reversal permutation result in unwieldy communication requirements. This high degree of communication traffic means that the savings in communication time consistent with exploiting symmetry is negated by the extra communication overhead incurred.

5.2 Symmetries and Properties of the DFT

We start with some definitions of the different kinds of symmetries that a given sequence may have. These symmetries can be described with the permutation matrices **T**, the reflection matrix for even and odd sequences, and **E**, the exchange matrix for quarter-wave even and odd sequences. The exchange permutation $\mathbf{E}_n$ is described by

$$\mathbf{E}_n = [\mathbf{e}_{n-1}, \mathbf{e}_{n-2}, \ldots, \mathbf{e}_1, \mathbf{e}_0]$$

If $\mathbf{y} = \mathbf{E}_n\mathbf{x}$, then $\mathbf{y}$ consists of the elements of $\mathbf{x}$ flipped upside-down. The reflection permutation matrix $\mathbf{T}_n$ is defined as follows

$$\mathbf{T}_n = \begin{bmatrix} 1 & 0 \\ 0 & \mathbf{E}_{n-1} \end{bmatrix}.$$

If $\mathbf{y} = \mathbf{T}_n\mathbf{x}$, then $\mathbf{y}$ is best described by fixing the top element, and flipping the rest of the $n-1$ elements upside down. The reflection permutation can also be described via a circular upshift permutation followed by the exchange permutation.

$$\mathbf{T}_n = \mathbf{E}_n\mathbf{R}_n^T$$

where $\mathbf{R}_n^T$ is the circular upshift matrix defined by

$$\mathbf{R}_n^T \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-2} \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-2} \\ x_{n-1} \\ x_0 \end{bmatrix}$$

The symmetric properties can now be described in matrix-vector notation in a compact format. Table 5.1 shows the different symmetries that a sequence might have along with an eight-point example.

The following lemmas provide a link between the symmetries **T** and **E** and the DFT matrix $\mathbf{F}_n$. These relationships allow us to describe some of the symmetry properties that $\mathbf{F}_n$ possesses in a simple matrix-oriented language.

Lemma 5.2.1

$$\mathbf{F}_n\mathbf{T}_n = \mathbf{T}_n\mathbf{F}_n = \overline{\mathbf{F}}_n$$

Table 5.1: Symmetric Sequences

Property	Definition	Example
even (E)	$\mathbf{x} = \mathbf{T}\mathbf{x}$	$[x_0, x_1, x_2, x_3, x_4, x_3, x_2, x_1]$
odd (O)	$\mathbf{x} = -\mathbf{T}\mathbf{x}$	$[0, x_1, x_2, x_3, 0, -x_3, -x_2, -x_1]$
quarter-wave even (QE)	$\mathbf{x} = \mathbf{E}\mathbf{x}$	$[x_0, x_1, x_2, x_3, x_3, x_2, x_1, x_0]$
quarter-wave odd (QO)	$\mathbf{x} = -\mathbf{E}\mathbf{x}$	$[x_0, x_1, x_2, x_3, -x_3, -x_2, -x_1, -x_0]$
conjugate even (CE)	$x = \mathbf{T}\overline{\mathbf{x}}$	$[x_0, x_1, x_2, x_3, x_4, \overline{x}_3, \overline{x}_2, \overline{x}_1]$
conjugate odd (CO)	$\mathbf{x} = -\mathbf{T}\overline{\mathbf{x}}$	$[0, x_1, x_2, x_3, 0, -\overline{x}_3, -\overline{x}_2, -\overline{x}_1]$

Lemma 5.2.2

$$\overline{\mathbf{F}}_n\mathbf{E} = \mathbf{\Delta}_n\mathbf{F}_n$$

where $\mathbf{\Delta} = diag(1, \omega_n, \ldots, \omega_n^{n-1})$, $\omega_n = \exp(-2\pi i/n)$.

The next two lemmas are due to Swarztrauber (1986b) and provide a method of representing the DFT of quarter-even (QE-symmetric) and quarter-odd (QO-symmetric) sequences by purely real or purely imaginary sequences. This allows the programmer to save on array storage space as a complex number occupies twice as much space as a real number. These lemmas also simplify the splitting equations for the cosine and sine transforms by eliminating the need for complex arithmetic.

Lemma 5.2.3 *The DFT* $\mathbf{y}$ *of a QE-symmetric sequence* $\mathbf{x}$ *satisfies*

$$\mathbf{y} = \mathbf{\Delta}_n^{-1}\overline{\mathbf{y}}$$

where $\mathbf{\Delta}_n = diag(1, \omega_n, \ldots, \omega_n^{n-1})$ *(*$\omega_n = \exp(-2\pi i/n)$*) and can be represented in terms of a purely real sequence* $\tilde{\mathbf{y}}$.

Proof Swarztrauber (1986b).

——*——

Lemma 5.2.4 *The DFT* $\mathbf{y}$ *of a QO-symmetric sequence* $\mathbf{x}$ *satisfies*

$$\mathbf{y} = -\mathbf{\Delta}^{-1}\overline{\mathbf{y}}$$

where $\mathbf{\Delta}_n = diag(1, \omega_n, \ldots, \omega_n^{n-1})$ *and can be represented in terms of a purely imaginary sequence* $\tilde{\mathbf{y}}$.

Proof Swarztrauber (1986b).

———*———

Swarztrauber (1986b) devises splitting equations for various symmetric sequences. These equations provide the basic recurrence for symmetric transforms. We state them here in our matrix notation.

Splitting Equations for Real Sequences

Let $\mathbf{z}$ be a real sequence of length $2n$ and let

$$\begin{aligned} \mathbf{z}_1 &\leftarrow \mathbf{z}(0:2:n-1) \\ \mathbf{z}_2 &\leftarrow \mathbf{z}(1:2:n-1) \end{aligned}$$

Since $\mathbf{z}$, $\mathbf{z}_1$ and $\mathbf{z}_2$ are real sequences their transforms are conjugate even symmetric. Let $\mathbf{y} = \mathbf{F}_{2n}\mathbf{z}$, $\mathbf{y}_1 = \mathbf{F}_n\mathbf{z}_1$, and $\mathbf{y}_2 = \mathbf{F}_n\mathbf{z}_2$. This means that only $n+1$ points of $\mathbf{y}$ need to be computed. Suppose $\mathbf{y}_1(0:n/2)$ and $\mathbf{y}_2(0:n/2)$ are available. The radix-2 splitting algorithm gives

$$\mathbf{y}(0:n/2) = \mathbf{y}_1(0:n/2) + \mathbf{\Delta}_{2n}\mathbf{y}_2(0:n/2)$$

and

$$\mathbf{y}(n:-1:n/2+1) = \mathbf{T}_{n/2}\mathbf{y}_1(0:n/2-1) + \mathbf{T}_{n/2}\mathbf{\Delta}_{2n}\mathbf{y}_2(0:n/2-1)$$

where $\mathbf{\Delta}_{2n} = (1, \omega_{2n}, \ldots, \omega_{2n}^{n/2-1})$ $(\omega_{2n} = \exp^{2\pi i/2n})$. $\mathbf{T}_{n/2}\mathbf{\Delta}_{2n} = -\mathbf{\Delta}_{2n}$ since $e^{-i(n-k)2\pi/2n} = -e^{ik2\pi/2n}$ so that the second equation becomes

$$\overline{\mathbf{y}}(n:-1:n/2+1) = \mathbf{y}_1(0:n/2-1) - \mathbf{\Delta}_{2n}\mathbf{y}_2(0:n/2-1)$$

Splitting Equations for Even Sequences

Let $\mathbf{z}$ be a real even sequence of length $2n$ with $\mathbf{z}_1$ comprised of the even points of $\mathbf{z}$ and $\mathbf{z}_2$ the odd points. If $\mathbf{y}$, $\mathbf{y}_1$, and $\mathbf{y}_2$ are their transforms, respectively, we have the following symmetries. $\mathbf{y}_1$ is also E-symmetric, $\mathbf{y}_2$ is QE-symmetric, and $\mathbf{y}$ is E-symmetric. Both $\mathbf{y}$ and $\mathbf{y}_1$ are real and $\mathbf{y}_2 = \mathbf{\Delta}_{2n}^{-1}\tilde{\mathbf{y}}_2$ where $\tilde{\mathbf{y}}_2$ is also real. Substituting this into the splitting equations for real sequences gives

$$\begin{aligned} \mathbf{y}(0:n/2) &= \mathbf{y}_1(0:n/2) + \tilde{\mathbf{y}}_2(0:n/2) \\ \mathbf{y}(n:-1:n/2+1) &= \mathbf{y}_1(0:n/2-1) - \tilde{\mathbf{y}}_2(0:n/2-1) \end{aligned}$$

Since all quantities are real, only real arithmetic is needed. $\tilde{\mathbf{y}}_2$ is computed instead of $\mathbf{y}_2$ so as to avoid complex arithmetic.

Splitting Equations for Odd Sequences

Now let $\mathbf{z}$ be a real odd sequence of length $2n$ and $\mathbf{z}_1$ and $\mathbf{z}_2$ defined as before. $\mathbf{y}_1$ is also O-symmetric and $\mathbf{y}_2$ is QO-symmetric, where $\mathbf{y}$ and $\mathbf{y}_1$ are strictly imaginary. $\mathbf{y}_2 = \mathbf{\Delta}_{2n}^{-1}\tilde{\mathbf{y}}_2$ and $\tilde{\mathbf{y}}_2$ is strictly imaginary. Substituting the whole lot into the splitting equations for real sequences gives,

$$\begin{aligned} \mathbf{y}(1:n/2) &= \mathbf{y}_1(1:n/2) + \tilde{\mathbf{y}}_2(1:n/2) \\ -\mathbf{y}(n-1:-1:n/2+1) &= \mathbf{y}_1(1:n/2-1) - \tilde{\mathbf{y}}_2(1:n/2-1) \end{aligned}$$

All quantities are purely imaginary so that we can multiply through by i and avoid complex arithmetic.

Block Structure of the DFT Matrix

The DFT matrix $\mathbf{F}_n$ has an interesting block structure when we consider real and imaginary parts separately. First we define some special vectors and matrices to simplify the following discussion.

$$\begin{aligned} \mathbf{w}_m &= (1,1,\ldots,1)^T \in \mathcal{R}^m \\ \mathbf{v}_m &= (-1,1,-1,\ldots,(-1)^m)^T \in \mathcal{R}^m \end{aligned}$$

Let $\mathbf{E}_n$ be the exchange matrix of order n defined as $\mathbf{E}_n = \mathbf{I}_n(:,n:-1:1)$.

The n-by-n real cosine and sine matrices are defined as

$$\mathbf{C}_n = \begin{bmatrix} \cos(\pi/n+1) & \cdots & \cos(n\pi/n+1) \\ \vdots & \ddots & \vdots \\ \cos(n\pi/n+1) & \cdots & \cos(n^2\pi/n+1) \end{bmatrix}$$

and

$$\mathbf{S}_n = \begin{bmatrix} \sin(\pi/n+1) & \cdots & \sin(n\pi/n+1) \\ \vdots & \ddots & \vdots \\ \sin(n\pi/n+1) & \cdots & \sin(n^2\pi/n+1) \end{bmatrix}$$

Theorem 5.2.1 *(Matrix Structure of* $\mathbf{F}_n$*): Let* $n = 2m$. *If* $\mathbf{w} = \mathbf{w}_{m-1}$, $\mathbf{v} = \mathbf{v}_{m-1}$, $\mathbf{E} = \mathbf{E}_{m-1}$, $\mathbf{C} = \mathbf{C}_{m-1}$, *and* $\mathbf{S} = \mathbf{S}_{m-1}$, *then*

$$\mathbf{F}_{2m} = \mathrm{Re}[\mathbf{F}_{2m}] - i\mathrm{Im}[\mathbf{F}_{2m}]$$

where

$$\mathrm{Re}[\mathbf{F}_{2m}] = \begin{array}{c} \\ 1 \\ m-1 \\ 1 \\ m-1 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & m-1 & 1 & m-1 \end{array} \\ \left[\begin{array}{cccc} 1 & \mathbf{w}^T & 1 & \mathbf{w}^T \\ \mathbf{w} & \mathbf{C} & \mathbf{v} & \mathbf{CE} \\ 1 & \mathbf{v}^T & (-1)^m & \mathbf{v}^T\mathbf{E} \\ \mathbf{w} & \mathbf{EC} & \mathbf{Ev} & \mathbf{ECE} \end{array}\right] \end{array}$$

$$\mathrm{Im}[\mathbf{F}_{2m}] = \begin{array}{c} \\ 1 \\ m-1 \\ 1 \\ m-1 \end{array} \begin{array}{cccc} 1 & m-1 & 1 & m-1 \\ \left[\begin{array}{cccc} 0 & 0 & 0 & 0 \\ 0 & \mathbf{S} & 0 & -\mathbf{SE} \\ 0 & 0 & 0 & 0 \\ 0 & -\mathbf{ES} & 0 & \mathbf{ESE} \end{array}\right] \end{array}$$

Proof See Van Loan (1987).

——*——

This matrix structure proves handy in the development of symmetric FFT algorithms and the description of cosine and sine transforms.

5.3 Permutations and Communication on Distributed Processors

A permutation of data mapped in a distributed fashion in separate nodes of a parallel computer requires communication between the processors. Sometimes permutations are unavoidable and on loosely coupled systems they can present quite a problem.

Several permutations come into play when implementing real and symmetric FFT procedures. We have already seen the exchange permutation **E**, the bit-reversal permutation **P**, and the reflection permutation **T**. Because distributed FFTs are more efficiently implemented without distributed bit-reversal, the data in the transform domain is usually permuted by **P**. Hence a facility must be developed for working with the combination of the exchange and reflection permutations with the bit-reversal permutation.

To motivate, we first present an easy example concerning the exchange permutation. Suppose that during the course of our computations we are required to find

$$\mathbf{x} + \mathbf{Ex}$$

where **x** is distributed among processors in the consecutive order. This can be done by inter-processor communication that "flips" **x** upside down. We will suppress, for the moment, how this is done. Now suppose we really had **Px** available and wanted to find

$$\mathbf{Px} + \mathbf{P(Ex)}$$

What would be an easy way to implement the communications necessary? It would be extremely inefficient to first unscramble $\mathbf{x} \leftarrow \mathbf{Px}$, then find $\mathbf{x} \leftarrow \mathbf{Ex}$ and finally scramble by **P** again. Since we have **Px** available, we would like to work directly with $\mathbf{Px}$. Luckily in the case of the exchange matrix $\mathbf{E}$, $\mathbf{PE} = \mathbf{EP}$ and therefore we find that

$$\mathbf{Px} + \mathbf{E}(\mathbf{Px})$$

can be implemented by a simple exchange permutation on the vector $\mathbf{Px}$.

Lemma 5.3.1 *If $n = 2^t$, then*

$$\mathbf{P}_n\mathbf{E}_n = \mathbf{E}_n\mathbf{P}_n$$

Proof We give an intuitive proof for the case $n = 2^t$. Recall from Chapter 1 that an exchange permutation is defined on the binary representation of the index $j = (b_{t-1}b_{t-2}\dots b_1b_0)_2$ by complementing all of j's bits. Hence

$$\epsilon(j) = (\bar{b}_{t-1}\bar{b}_{t-2}\dots \bar{b}_1\bar{b}_0)_2$$

Meanwhile the bit-reversal permutation is defined on the binary representation of the index $j = (b_{t-1}b_{t-2}\dots b_1b_0)_2$ by reversing all of j's bits, i.e.

$$\rho(j) = (b_0b_1\dots b_{t-2}b_{t-1})_2$$

The commutativity comes from observing that if matters not a bit whether the bits of j are first complemented and then reversed in order, or vice-versa. The same observation works for the general radix $n = p^t$ by defining complementation as the additive inverse, i.e. $\bar{b} + b = 0 \bmod p$.

——*——

We now turn to the reflection permutation $\mathbf{T}$ and show that things are not so easy here primarily because $\mathbf{PT} \neq \mathbf{TP}$. Now supposing we want to find

$$\mathbf{Px} + \mathbf{P}(\mathbf{Tx})$$

and we only had $\mathbf{Px}$ to work with. What we now need is a permutation $\mathbf{G}$ such that $\mathbf{PT} = \mathbf{GP}$. It turns out that a special permutation $\mathbf{G}$, that we call *Recursive Exchange* satisfies this criterion. A look at the structure of $\mathbf{G}_n$, $n = 2^t$ shows that it is the direct sum of a sequence of smaller exchange matrices.

Definition 5.3.1 *Let $\mathbf{G}_n$ be a permutation matrix which is a direct sum of smaller exchange matrices, $n = 2^t$.*

$$\mathbf{G}_n = \begin{bmatrix} \mathbf{E}_1 & & & & & \\ & \mathbf{E}_1 & & & & \\ & & \mathbf{E}_2 & & & \\ & & & \mathbf{E}_4 & & \\ & & & & \ddots & \\ & & & & & \mathbf{E}_{n/2} \end{bmatrix}$$

$\mathbf{G}_n$ can be defined recursively as

$$\mathbf{G}_n = \begin{bmatrix} \mathbf{G}_{n/2} & 0 \\ 0 & \mathbf{E}_{n/2} \end{bmatrix}$$

Example

$$\mathbf{G}_8 \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 3 \\ 2 \\ 7 \\ 6 \\ 5 \\ 4 \end{bmatrix}$$

The following lemma provides a decomposition of the reflection matrix $\mathbf{T}_n$ into a direct sum of two permutation matrices of half the size coupled with a perfect shuffle of both the rows and columns.

Lemma 5.3.2

$$\mathbf{T}_n = \mathbf{\Pi}_n \begin{bmatrix} \mathbf{T}_{n/2} & 0 \\ 0 & \mathbf{E}_{n/2} \end{bmatrix} \mathbf{M}_n$$

Proof We apply the permutations on the right hand side to the vector $(0 : n-1)^T$.

1. $\mathbf{M}_n(0 : n-1)^T = \begin{bmatrix} (0:2:n-2)^T \\ (1:2:n-1)^T \end{bmatrix}$

2. $\begin{bmatrix} \mathbf{T}_{n/2} & 0 \\ 0 & \mathbf{E}_{n/2} \end{bmatrix} \begin{bmatrix} (0:2:n-2)^T \\ (1:2:n-1)^T \end{bmatrix} = \begin{bmatrix} (0, n-2:-2:2)^T \\ (n-1:-2:1)^T \end{bmatrix}$

3. $\mathbf{\Pi}_n \begin{bmatrix} (0, n-2:-2:2)^T \\ (n-1:-2:1)^T \end{bmatrix} = \begin{bmatrix} 0 \\ n-1 \\ n-2 \\ n-3 \\ \vdots \\ 2 \\ 1 \end{bmatrix}$

The result of applying the permutations on the right hand side is exactly $\mathbf{T}_n(0 : n-1)^T$.

——*——

Now we are ready to prove a result tieing together the reflection matrix $\mathbf{T}_n$ and the recursive exchange matrix $\mathbf{G}_n$ through the bit-reversal permutation $\mathbf{P}_n$. Recall that $\mathbf{P}_n$ is always involved somewhere in distributed FFT procedures where unscrambling is not done. This next lemma is of great use in the analysis of bit-reverse permutation data that also exhibit symmetry.

Lemma 5.3.3

$$\mathbf{P}_n\mathbf{T}_n = \mathbf{G}_n\mathbf{P}_n$$

Proof We use induction. This is obviously true for $n = 2$. Now

$$\begin{aligned}
\mathbf{T}_n &= \mathbf{P}_n\mathbf{G}_n\mathbf{P}_n \\
&= \mathbf{\Pi}_n \begin{bmatrix} \mathbf{P}_{n/2} & 0 \\ 0 & \mathbf{P}_{n/2} \end{bmatrix} \begin{bmatrix} \mathbf{G}_{n/2} & 0 \\ 0 & \mathbf{E}_{n/2} \end{bmatrix} \begin{bmatrix} \mathbf{P}_{n/2} & 0 \\ 0 & \mathbf{P}_{n/2} \end{bmatrix} \mathbf{M}_n \\
&= \mathbf{\Pi}_n \begin{bmatrix} \mathbf{T}_{n/2} & 0 \\ 0 & \mathbf{E}_{n/2} \end{bmatrix} \mathbf{M}_n
\end{aligned}$$

using the induction assumption $\mathbf{P}_{n/2}\mathbf{T}_{n/2} = \mathbf{G}_{n/2}\mathbf{P}_{n/2}$ and the fact that $\mathbf{EP} = \mathbf{PE}$.

——*——

Armed with this fact, we can find

$$\mathbf{Px} + \mathbf{PTx}$$

by the equivalent sum

$$\mathbf{Px} + \mathbf{G}(\mathbf{Px})$$

Permutations on distributed vectors are equivalent to data movement between and within the processors of a loosely coupled system. In fact after having established the equivalency of the two concepts, we can use the permutation matrix as a shorthand for a sequence of sometimes complicated communication specifications.

In this section we assume that the nodes of a hypercube are mapped in the Binary Reflected Grey Code (BRGC) order. The input vector is mapped in the two-track fashion of Chapter 3. The following theorems tell how data mapped in the Two-Track method of Chapter 3 is rearranged inside the processors when permuted by the various permutations. Keeping in mind that our main goal is to provide a setting where the corresponding components of $\mathbf{x}$ and let's say $\mathbf{Tx}$ are in the same processor so that they can be combined. For example, the Two-Track

$$\begin{array}{cc} 0 & \mathbf{8} \\ 1 & \mathbf{9} \\ 2 & \mathbf{10} \\ 3 & \mathbf{11} \\ \mathbf{4} & 12 \\ \mathbf{5} & 13 \\ \mathbf{6} & 14 \\ \mathbf{7} & 15 \end{array} \rightarrow \begin{array}{cc} 0 & \mathbf{4} \\ 1 & \mathbf{5} \\ \mathbf{2} & 6 \\ \mathbf{3} & 7 \\ 8 & \mathbf{12} \\ 9 & \mathbf{13} \\ \mathbf{10} & 14 \\ \mathbf{11} & 15 \end{array} \rightarrow \begin{array}{cc} 0 & \mathbf{2} \\ \mathbf{1} & 3 \\ 4 & \mathbf{6} \\ \mathbf{5} & 7 \\ 8 & \mathbf{10} \\ \mathbf{9} & 11 \\ 12 & \mathbf{14} \\ \mathbf{13} & 15 \end{array} \rightarrow \begin{array}{cc} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \\ 8 & 9 \\ 10 & 11 \\ 12 & 13 \\ 14 & 15 \end{array}$$

Figure 5.1: Permutation Pattern of Two-Track FFT

mapping matches each element with its corresponding element that is half of the vector length away. Hence the computation

$$\mathbf{x} + \begin{bmatrix} \mathbf{x}(n/2 : n-1) \\ \mathbf{x}(0 : n/2 - 1) \end{bmatrix}$$

can be done locally without communication.

Theorem 5.3.1 *The permutation resulting from the implementation of the FFT of the Two-Track method is equivalent to* $\mathbf{M}_n\mathbf{P}_n$. *(See Figure 5.1).*

Proof The in-place FFT results in a sequence permuted by $\mathbf{P}_n$ while Theorem 1.6.6 tells us that repeatedly performing super-butterfly permutations giving us $\mathbf{M}_n$.

———*———

Algorithm 5.3.1 Permutation $\mathbf{M}_n$

```
/* processors mapped in BRGC order */
/* $\mu$ = processor id; $n = 2^t$; $P = 2^d$; $m = (n/P)/2$ */
/* processor $\mu$ holds: */
/* $\mathbf{z}^{(1)} = \mathbf{x}(m \cdot id[\mu] : m \cdot (id[\mu]+1) - 1)$ */
/* $\mathbf{z}^{(2)} = \mathbf{x}(\frac{n}{2} + m \cdot id[\mu] : \frac{n}{2} + m \cdot (id[\mu]+1) - 1)$ */
  for $L = 1 : d$
    $\nu = d - L$; $id[\rho] = flip(id[\mu], \nu)$
    $\rho = id^{-1}(id[\rho])$
    send($\mathbf{z}^{(2)}, m, \rho$); recv($\mathbf{z}^{(1)}, m, \rho$)
  end
  $\begin{bmatrix} \mathbf{z}^{(1)} \\ \mathbf{z}^{(2)} \end{bmatrix} \leftarrow \mathbf{M}_{2m} \begin{bmatrix} \mathbf{z}^{(1)} \\ \mathbf{z}^{(2)} \end{bmatrix}$
```

$$\begin{array}{ll c ll}
0 & \mathbf{8} & & 0 & 15 \\
1 & \mathbf{9} & & 1 & 14 \\
2 & \mathbf{10} & & 2 & 13 \\
3 & \mathbf{11} & & 3 & 12 \\
4 & \mathbf{12} & \rightarrow & 4 & 11 \\
5 & \mathbf{13} & & 5 & 10 \\
6 & \mathbf{14} & & 6 & 9 \\
7 & \mathbf{15} & & 7 & 8
\end{array}$$

Figure 5.2: Permutation Pattern of Two-Track $\mathbf{E}_n$

Theorem 5.3.2 *Exchange. Suppose* $\mathbf{x}$ *is mapped into the hypercube by the Two-Track mapping. Then getting matching components of* $\mathbf{x}$ *and* $\mathbf{Ex}$ *into the same processor is equivalent to performing the exchange permutation on the second half of* $\mathbf{x}$ *only, i.e. applying* $\begin{bmatrix} \mathbf{I}_{n/2} & 0 \\ 0 & \mathbf{E}_{n/2} \end{bmatrix}\mathbf{x}$. *(See Figure 5.2).*

Proof The two-track mapping means that $\mathbf{x}(j)$ and $\mathbf{x}(j+\frac{n}{2})$ are in the same processor. To get $\mathbf{x}$ and $\mathbf{Ex}$ together we want $\mathbf{x}(j)$ and $\mathbf{x}(n-j-1)$ together. Therefore we need to replace

$$\mathbf{x}(j+\frac{n}{2}) \leftarrow \mathbf{x}(n-j-1), \quad j = 0,\ldots,\frac{n}{2}-1$$

But doing this is precisely the same as applying $\begin{bmatrix} \mathbf{I}_{n/2} & 0 \\ 0 & \mathbf{E}_{n/2} \end{bmatrix}\mathbf{x}$.

———*———

The permutation pattern for an example $n = 16$ is given in Figure 5.2.

Algorithm 5.3.2 Permutation $\mathbf{E}_n$

```
/* processors mapped in BRGC order */
/* $\mu$ = processor id; $n = 2^t$; $P = 2^d$; $m = (n/P)/2$ */
/* processor $\mu$ holds: */
/* $\mathbf{z}^{(1)} = \mathbf{x}(m \cdot id[\mu] : m \cdot (id[\mu]+1)-1)$ */
/* $\mathbf{z}^{(2)} = \mathbf{x}(\frac{n}{2} + m \cdot id[\mu] : \frac{n}{2} + m \cdot (id[\mu]+1)-1)$ */
  $\nu = d-1$; $\rho = flip(\mu,\nu)$
  send($\mathbf{z}^{(2)}, m, \rho$); recv($\mathbf{z}^{(2)}, m, \rho$)
  $\mathbf{z}^{(2)} \leftarrow \mathbf{E}_m\mathbf{z}^{(2)}$
```

$$\begin{array}{cc c cc} 0 & \mathbf{8} & & 0 & 8 \\ 1 & \mathbf{9} & & 1 & 15 \\ 2 & \mathbf{10} & & 2 & 14 \\ 3 & \mathbf{11} & & 3 & 13 \\ 4 & \mathbf{12} & \rightarrow & 4 & 12 \\ 5 & \mathbf{13} & & 5 & 11 \\ 6 & \mathbf{14} & & 6 & 10 \\ 7 & \mathbf{15} & & 7 & 9 \end{array}$$

Figure 5.3: Permutation Pattern of Two-Track $\mathbf{T}_n$

Theorem 5.3.3 *Reflection. Suppose* $\mathbf{x}$ *is mapped into the hypercube by the two-track mapping. Then getting matching components of* $\mathbf{x}$ *and* $\mathbf{Tx}$ *into the same processor is equivalent to performing the reflection permutation on the second half of* $\mathbf{x}$ *only, i.e. applying* $\begin{bmatrix} \mathbf{I}_{n/2} & 0 \\ 0 & \mathbf{T}_{n/2} \end{bmatrix} \mathbf{x}$. *(See Figure 5.3).*

Proof In the two-track setting $\mathbf{x}(j)$ and $\mathbf{x}(j+\frac{n}{2})$ are in the same processor. To get $\mathbf{x}$ and $\mathbf{Tx}$ together we want $\mathbf{x}(j)$ and $\mathbf{x}(n-j)$ together. Therefore we need to replace

$$\mathbf{x}(j+\frac{n}{2}) \leftarrow \mathbf{x}(n-j), \quad j=0,\ldots,\frac{n}{2}-1$$

Doing this is precisely equivalent to applying

$$\begin{bmatrix} \mathbf{I}_{n/2} & 0 \\ 0 & \mathbf{T}_{n/2} \end{bmatrix} \mathbf{x}$$

——*——

Algorithm 5.3.3 Permutation $\mathbf{T}_n$

```
  /* processors mapped in BRGC order */
  /* $\mu$ = processor id; $n = 2^t$; $P = 2^d$; $m = (n/P)/2$ */
  /* processor $\mu$ holds: */
  /* $\mathbf{z}^{(1)} = \mathbf{x}(m \cdot id[\mu] : m \cdot (id[\mu]+1) - 1)$ */
  /* $\mathbf{z}^{(2)} = \mathbf{x}(\frac{n}{2} + m \cdot id[\mu] : \frac{n}{2} + m \cdot (id[\mu]+1) - 1)$ */
  /* send top element to previous processor */
  /* upshift second column */
    $\rho = id^{-1}[(id[\mu] - 1) \bmod P]$
    $\nu = id^{-1}[(id[\mu] + 1) \bmod P]$
```

```
send($\mathbf{z}^{(2)}(0), 1, \rho$); recv($t, 1, \nu$)
$\mathbf{z}^{(2)} \leftarrow \mathbf{R}_m^T \mathbf{z}^{(2)}$
$\mathbf{z}^{(2)}(m-1) \leftarrow t$
call exchange($\mathbf{z}^{(2)}$)
```

Theorem 5.3.4 *Recursive Exchange. Suppose $\mathbf{x}$ is mapped into the hypercube by the two-track mapping. Then getting matching components of $\mathbf{x}$ and $\mathbf{Gx}$ in the same processor is equivalent to first performing $\mathbf{M}_n$ on the sequence and then applying $\mathbf{G}_{n/2}$ to the second half of $\mathbf{x}$ only. In other words, mapping*

$$\begin{bmatrix} \mathbf{I}_{n/2} & 0 \\ 0 & \mathbf{G}_{n/2} \end{bmatrix} \mathbf{M}_n \mathbf{x}$$

into the processors in the two-track manner. (See Figure 5.4).

Proof In the two-track mapping, $\mathbf{x}(j)$ and $\mathbf{x}(j+\frac{n}{2})$ are in the same processor. Having matching components of $\mathbf{x}$ and $\mathbf{Gx}$ in the same processor means that $\mathbf{x}(j)$ and $\mathbf{x}(2^{\lceil \log_2 j \rceil} - j \bmod 2^{\lfloor \log_2 j \rfloor} - 1)$, $j = 0, \ldots, \frac{n}{2} - 1$ are directly matched. The effect of $\mathbf{M}_n$ is

$$\begin{aligned} \mathbf{x}(j) &\leftarrow \mathbf{x}(2j) \\ \mathbf{x}(j+\frac{n}{2}) &\leftarrow \mathbf{x}(2j+1) \\ & \qquad j = 0, \ldots, \frac{n}{2} - 1 \end{aligned}$$

We want $\mathbf{x}(2j)$ together with

$$\mathbf{x}(2^{\lceil \log_2 2j \rceil} - 2j \bmod 2^{\lfloor \log_2 2j \rfloor} - 1)$$

or

$$\mathbf{x}(2j+1) \leftarrow \mathbf{x}(2^{\lceil \log_2 2j \rceil} - 2j \bmod 2^{\lfloor \log_2 2j \rfloor} - 1)$$

This is equivalent to applying $\mathbf{G}_{n/2}$ to $\mathbf{x}(j')$ where $j' \leftarrow 2j+1$ is the change of variables.

——*——

$$\begin{array}{cc} \mathbf{0} & \mathbf{8} \\ \mathbf{1} & \mathbf{9} \\ \mathbf{2} & \mathbf{10} \\ \mathbf{3} & \mathbf{11} \\ \mathbf{4} & \mathbf{12} \\ \mathbf{5} & \mathbf{13} \\ \mathbf{6} & \mathbf{14} \\ \mathbf{7} & \mathbf{15} \end{array} \longrightarrow \begin{array}{cc} 0 & 1 \\ 2 & 3 \\ 4 & \mathbf{5} \\ 6 & \mathbf{7} \\ 8 & \mathbf{9} \\ 10 & \mathbf{11} \\ 12 & \mathbf{13} \\ 14 & \mathbf{15} \end{array} \longrightarrow \begin{array}{cc} 0 & 1 \\ 2 & 3 \\ 4 & 7 \\ 6 & 5 \\ 8 & 15 \\ 10 & 13 \\ 12 & 11 \\ 14 & 9 \end{array}$$

Figure 5.4: Permutation Pattern of Two-Track $\mathbf{G}_n$

Algorithm 5.3.4 Permutation $\mathbf{G}_n$

```
/* processors mapped in BRGC order */
/* $\mu$ = processor id; $n = 2^t$; $P = 2^d$; $m = (n/P)/2$ */
/* processor $\mu$ holds: */
/* $\mathbf{z}^{(1)} = \mathbf{x}(m \cdot id[\mu] : m \cdot (id[\mu] + 1) - 1)$ */
/* $\mathbf{z}^{(2)} = \mathbf{x}(\frac{n}{2} + m \cdot id[\mu] : \frac{n}{2} + m \cdot (id[\mu] + 1) - 1)$ */
/* apply $\mathbf{M}_n$ only if Two-Track FFT was not done */
  if(Two-Track FFT not done) then
    call Inverse Shuffle($\mathbf{z}^{(1)}, \mathbf{z}^{(2)}$)
  end if
  if ($id[\mu] \neq 0.and.id[\mu] \neq 1$) then
    $id[\rho] \leftarrow 2^{\lceil \log_2 id[\mu] \rceil} - id[\mu] \bmod 2^{\lfloor \log_2 id[\mu] \rfloor} - 1)$
    $\rho = id^{-1}(id[\rho])$
    send($\mathbf{z}^{(2)}, m, \rho$); recv($\mathbf{z}^{(2)}, m, \rho$)
    $\mathbf{z}^{(2)} \leftarrow \mathbf{E}_m \mathbf{z}^{(2)}$
  end if
  if ($id[\mu] = 0$) then
    $\mathbf{z}^{(2)} \leftarrow \mathbf{G}_m \mathbf{z}^{(2)}$
  end if
  if ($id[\mu] = 1$) then
    $\mathbf{z}^{(2)} \leftarrow \mathbf{E}_m \mathbf{z}^{(2)}$
  end if
```

Armed with this fact we can find

$$\mathbf{Px} + \mathbf{PTx}$$

by the equivalent sum

$$\mathbf{Px} + \mathbf{G}(\mathbf{Px}).$$

Now that we have related permutations to hypercube communication we can simplify our ensueing discussion of algorithms. Using the "language" of a permutation matrix, let us say $\mathbf{T}_n$, we can talk about data movements in distributed systems by referring to Algorithm 5.3.3. This means that in the course of our discussion of symmetric transforms, we can simply say

call Permutation($\mathbf{T}_n$)

and know that the communications detail in Algorithm 5.3.3 are performed.

5.4 Development of the Parallel Sine Transform

In this section we present our new sine transform algorithm designed to be implementable on the hypercube with a simplified pre- and post- processing scheme. In fact, all of the post-processing is local to a processor and we require only one upshift ($\mathbf{R}_n$) communication (or ***shift-vector*** operation of McBryan and Van de Velde (1987)) for the pre-processing, i.e. every processor sends one real number to its preceeding node and receives one real number from its succeeding node. In the case that the original sequence is generated by a formula instead of entered in, we can circumvent this communication by having each processor generate one extra real number. This method results in the sine transform being laid out so that just two exchanges are required for pre-processing the inverse sine transform. This simplified procedure is accomplished by doing some redundant computations and computing a complex FFT of length n, instead of a real FFT of length n as was the case for the conventional methods.

Since the sine transform of an $(n-1)$-point sequence is derived from the imaginary part of the complex transform of the extended odd $(2n)$-point sequence of real numbers, we first consider the use of an n-point complex FFT to find the DFT of this extended sequence. Note that the analogous cosine transform method can be derived by using the real part of the complex transform of the extended even $(2n)$-point sequence of real numbers.

Let $\mathbf{z}$ be the real odd extension for our original $(n-1)$-point sequence $\mathbf{x}$,

$$\mathbf{z} = \begin{bmatrix} 0 \\ \mathbf{x}(1:n-1) \\ 0 \\ -\mathbf{E}\mathbf{x}(1:n-1) \end{bmatrix}$$

We can split this sequence into two sequences, one comprised of the even points and the other the odd points. Let

$$\mathbf{z}_1 \quad \leftarrow \quad \mathbf{z}(0:2:2n-1)$$

$$\mathbf{z}_2 \leftarrow \mathbf{z}(1:2:2n-1)$$

Swarztrauber (1986) has shown that these sequences exhibit symmetry also, that is

$$\mathbf{T}_n\mathbf{z}_1 = \mathbf{z}(2n-2:-2:0) = -\mathbf{z}(0:2:2n-2) = -\mathbf{z}_1$$

so that $\mathbf{z}$ is an odd sequence, and

$$\mathbf{E}_n\mathbf{z}_2 = \mathbf{z}(2n-1:-1:1) = -\mathbf{z}(1:2:2n-1) = -\mathbf{z}_2$$

and thus $\mathbf{z}_2$ is quarter-wave odd.

The result is that we have two real sequences each with special symmetry. We proceed in a similar vein as Procedure 2 except that we have two additional symmetries to exploit.

Form the complex sequence

$$\mathbf{s} = \mathbf{z}_1 + i\mathbf{z}_2$$

and let

(5.4-1) $$\mathbf{w} = \mathbf{F}_n\mathbf{s} = \mathbf{y}_1 + i\mathbf{y}_2$$

$\mathbf{y}_1$ and $\mathbf{y}_2$ possess properties in addition to the fact that they are conjugate even

$$\begin{aligned} \mathbf{y}_1 &= \mathbf{T}\overline{\mathbf{y}}_1 \\ \mathbf{y}_1 &= -\overline{\mathbf{y}}_1 \\ \mathbf{y}_2 &= \mathbf{T}\overline{\mathbf{y}}_2 \\ \mathbf{y}_2 &= -\mathbf{\Delta}^{-1}\overline{\mathbf{y}}_2 \\ \mathbf{y}_2 &= \mathbf{\Delta}_{2n}^{-1}\tilde{\mathbf{y}}_2 \\ \tilde{\mathbf{y}}_2 &= -\overline{\tilde{\mathbf{y}}}_2 \end{aligned}$$

Recall that $\tilde{\mathbf{y}}_2$ is pure imaginary. Equation 5.4-1 gives

(5.4-2) $$\mathbf{w} = \mathbf{y}_1 + i\mathbf{\Delta}_{2n}^{-1}\tilde{\mathbf{y}}_2$$

and

(5.4-3) $$\begin{aligned} \overline{\mathbf{w}} &= \overline{\mathbf{y}}_1 - i\mathbf{\Delta}_{2n}\overline{\tilde{\mathbf{y}}}_2 \\ &= -\mathbf{y}_1 + i\mathbf{\Delta}_{2n}\tilde{\mathbf{y}}_2 \end{aligned}$$

Solving equations 5.4-2 and 5.4-3 gives

$$\tilde{\mathbf{y}}_2 = \frac{1}{i} diag\left(\frac{1}{\cos k\pi/n}\right) \mathrm{Re}[\mathbf{w}]$$

and

$$\mathbf{y}_1 = i\mathrm{Im}[\mathbf{w}] - i diag\,(\tan k\pi/n)\,\mathrm{Re}[\mathbf{w}].$$

Because of the $\mathbf{y}_1$ and $\mathbf{y}_2$ are conjugate even, we only have to solve for the first terms $0, \ldots, n/2$. Finally

$$\begin{aligned} \mathbf{w}(n/2) &= \mathbf{y}_1(n/2) + i\exp(i\pi/2)\tilde{\mathbf{y}}_2(n/2) \\ &= -\tilde{\mathbf{y}}_2(n/2) \end{aligned}$$

and

$$\begin{aligned} \mathbf{y}_1(n/2) &= 0 \\ \tilde{\mathbf{y}}_2(n/2) &= -\mathbf{w}(n/2) = -i\mathrm{Im}[\mathbf{w}(n/2)] \end{aligned}$$

The splitting equation for odd sequences gives

$$\begin{aligned} \mathbf{y}(1:n/2) &= \frac{1}{2}[\mathbf{y}_1(1:n/2) + \tilde{\mathbf{y}}_2(1:n/2)] \\ \mathbf{y}(n-1:-1:n/2+1) &= \frac{1}{2}[\tilde{\mathbf{y}}_2(1:n/2) - \mathbf{y}_1(1:n/2)] \end{aligned}$$

From the definition of a sine transform, we know that $\mathbf{b} = 2i\mathbf{y}$ so that we can multiply through by i and obtain the following equations

$$\begin{aligned} \mathbf{b}(1:n/2) &= \mathbf{y}_1(1:n/2) + \tilde{\mathbf{y}}_2(1:n/2) \\ \mathbf{b}(n-1:-1:n/2+1) &= \tilde{\mathbf{y}}_2(1:n/2-1) - \mathbf{y}_2(1:n/2-1) \end{aligned}$$

And we never had to compute $\mathbf{y}$ in the first place. None of our arrays need to be larger than n and the extended sequence $\mathbf{z}$ was used for illustration and derivation purposes only. In fact, $\mathbf{y}_1$ and $\mathbf{y}_2$ can be real arrays of length $n/2+1$.

> **Procedure 1.** *Given:* A real sequence $\mathbf{x}$ of length n. *Find:* The sine transform $\mathbf{P}_n\mathbf{b}$ by the parallelizable sine transform algorithm.

- Form the complex sequence $\mathbf{s}$ as follows.

$$\mathbf{s} \leftarrow \begin{bmatrix} \mathbf{x}(0:2:n-1) + i\mathbf{x}(1:2:m-1) \\ -\mathbf{T}_{n/2}\mathbf{x}(0:2:n-1) - i\mathbf{E}_{n/2}\mathbf{x}(1:2:n-1) \end{bmatrix}$$

  This requires one linear-shift communication $\mathbf{R}_{n/2}^T$ and one exchange permutation $\mathbf{E}_{n/2}$

- Find $\mathbf{P}_n\mathbf{w} = \mathbf{P}_n\mathbf{F}_n\mathbf{s}$. This requires the communication pattern for distributed FFTs using the Two-Track method.

- Set

$$\mathbf{P}_n\mathbf{y}_1 = -\mathrm{Im}[\mathbf{P}_n\mathbf{w}] + \mathbf{P}_n diag(\tan\frac{k\pi}{n})\mathrm{Re}[\mathbf{P}_n\mathbf{w}]$$
$$\mathbf{P}_n\tilde{\mathbf{y}}_2 = \mathrm{Re}[\mathbf{P}_n\mathbf{w}]diag(1/\cos\frac{k\pi}{n})$$

  with $\mathbf{P}_n\mathbf{y}_1(n/2) = 0$ and $\mathbf{P}_n\tilde{\mathbf{y}}_2(n/2) = \mathrm{Im}[\mathbf{P}_n\mathbf{w}(n/2)]$. This step is local to a processor.

- Let

$$\mathbf{P}_n\mathbf{b} = \mathbf{P}_n\mathbf{y}_1 + \mathbf{P}_n\tilde{\mathbf{y}}_2$$

  Local to processor.

Algorithm 5.4.1 Forward Sine Transform

```
/* $\mu$ = processor id; $n = 2^t$; $P = 2^d$; $m = (n/P)/2$ */
/* Initially, processor $\mu$ holds: */
/* $\mathbf{x}(2m \cdot id[\mu] : 2m \cdot (id[\mu]+1) - 1)$ */
/* Form the two-tracks, $\mathbf{s}^{(1)}$ and $\mathbf{s}^{(2)}$ */
  $\mathbf{s}^{(1)} \leftarrow \mathbf{x}(0:2:m-1) + i\mathbf{x}(1:2:m-1)$
/* form $\mathbf{E}_{n/2}\mathbf{s}^{(2)} \leftarrow -\mathbf{R}^T_{n/2}\mathbf{x}(0:2:n-1) - \mathbf{x}(1:2:n-1)$ */
  $\rho = id^{-1}[(id[\mu]-1) \bmod P]$
  $\nu = id^{-1}[(id[\mu]+1) \bmod P]$
  send($Re[\mathbf{s}^{(1)}(0)], 1, \rho$); recv($temp, 1, \nu$)
  $\mathbf{s}^{(2)}(m-1) \leftarrow -temp - i(Im[\mathbf{s}^{(1)}(m-1)])$
  $\mathbf{s}^{(2)}(0:m-2) \leftarrow -Re[\mathbf{s}^{(1)}(1:m-1)] - i(Im[\mathbf{s}^{(2)}(0:m-2)])$
  call Exchange ($\mathbf{s}^{(2)}$) /* code (5.3) */
/* $\mathbf{w} \leftarrow \mathbf{P}_n\mathbf{F}_n\mathbf{s}$ */
  call Two-Track FFT($\mathbf{s}^{(1)}, \mathbf{s}^{(2)}$) /* code (5.2) */
/* post-processing code (5.5) */
/* k runs through the indices */
/* $\mathbf{v}(2m \cdot id[\mu] : 2m \cdot (id[\mu]+1) - 1)$ */
/* where $\mathbf{v} = \mathbf{P}_n(0:n-1)$ */
  $\mathbf{y}_1 \leftarrow -Im[\mathbf{w}] + diag(\tan\frac{k\pi}{n})Re[\mathbf{w}]$
  $\mathbf{y}_2 \leftarrow Re[\mathbf{w}]diag(1/\cos\frac{k\pi}{n})$
/* $\mathbf{b}$ is the sine transform of $\mathbf{x}$
  $\mathbf{P}_n\mathbf{b} \leftarrow \mathbf{y}_1 + \mathbf{y}_2$
```

Notice that we have chosen to compute all n points of $\mathbf{y}_1$ and $\tilde{\mathbf{y}}_2$ when only $n/2+1$ are needed. This is where the redundant computations come in. Therefore everything after the complex FFT is local to a processor with the introduction of n extra multiplications and $n/2$ extra additions. However these extra operations are not noticeable in the parallel setting since the processors are working independently. If the redundant computations were not introduced, some of the processors would be sitting idle anyhow. Therefore we have achieved load balancing as well as saved $O(d)$ communication steps for a total of $2d-1$ exchanges and one linear-shift. Notice the $2d-1$ exchanges are solely used for the computation of the complex FFT on a hypercube mapped on a BRGC, therefore the pre-processing resulted in only one (optional) communication.

5.5 The Sine Transform of a Sequence in Bit-Reversed Order

Even though the sine transform is its own inverse (scaled by a constant) our output from the forward sine transform is in bit-reversed order. Therefore we require an inverse sine transform which takes data originally in bit-reversed order and finishes with it in the natural order.

We have $\mathbf{P}_n\mathbf{b}$. Recall that

$$\mathbf{P}_n\mathbf{b} = \begin{bmatrix} \mathbf{P}_{n/2}\mathbf{b}(0:2:n-1) \\ \mathbf{P}_{n/2}\mathbf{b}(1:2:n-1) \end{bmatrix}$$

We would like to construct $\mathbf{M}_n\mathbf{P}_n\mathbf{s}$ so that we can use the inverse Two-Track method to obtain $\mathbf{w} \leftarrow \mathbf{F}_n\mathbf{s}$. Recall that

$$\mathbf{s} \leftarrow \begin{bmatrix} \mathbf{b}(0:2:n-1) + i\mathbf{b}(1:2:n-1) \\ -\mathbf{T}_{n/2}\mathbf{b}(0:2:n-1) - i\mathbf{E}_{n/2}\mathbf{b}(1:2:n-1) \end{bmatrix}$$

From Theorem 1.6.7 we have $\mathbf{P}_n = \mathbf{\Pi}_n(\mathbf{I}_2 \otimes \mathbf{P}_{n/2})$ hence

$$\mathbf{P}_n\mathbf{s} \leftarrow \mathbf{\Pi}_n \begin{bmatrix} \mathbf{P}_{n/2}\mathbf{b}(0:2:n-1) + i\mathbf{P}_{n/2}\mathbf{b}(1:2:n-1) \\ -\mathbf{P}_{n/2}\mathbf{T}_{n/2}\mathbf{b}(0:2:n-1) - i\mathbf{P}_{n/2}\mathbf{E}_{n/2}\mathbf{b}(1:2:n-1) \end{bmatrix}$$

Since the permutations

$$\mathbf{PT} = \mathbf{GP} \quad \text{and} \quad \mathbf{PE} = \mathbf{EP}$$

we can form

$$\mathbf{M}_n\mathbf{P}_n\mathbf{s} \leftarrow \begin{bmatrix} \mathbf{P}_{n/2}\mathbf{b}(0:2:n-1) + i\mathbf{P}_{n/2}\mathbf{b}(1:2:n-1) \\ -\mathbf{G}_{n/2}\mathbf{P}_{n/2}\mathbf{b}(0:2:n-1) - i\mathbf{E}_{n/2}\mathbf{P}_{n/2}\mathbf{b}(1:2:n-1) \end{bmatrix}$$

Working on $\mathbf{M}_n\mathbf{P}_n\mathbf{s}$ the the Inverse Two-Track algorithm, we get

$$\mathbf{w} \leftarrow \mathbf{F}_n\mathbf{s}$$

The post-processing continues locally.

Therefore the rest of Procedure 1a is identical to that of Procedure 1, except that we have sequences in the natural order. Since the sine transforms is its own inverse we have just outlined two procedures which can take both the sine transform and its inverse on data mapped either in the natural order or bit-reversed order.

Procedure 1a. *Given:* A real sequence $\mathbf{P}_n\mathbf{b}$ of length n. *Find:* The sine transform $\mathbf{x}$ by the parallelizable sine transform algorithm.

- Form the complex sequence

$$\mathbf{M}_n\mathbf{P}_n\mathbf{s} \leftarrow \begin{bmatrix} \mathbf{P}_{n/2}\mathbf{b}(0:2:n-1) + i\mathbf{P}_{n/2}\mathbf{b}(1:2:n-1) \\ -\mathbf{G}_{n/2}\mathbf{P}_{n/2}\mathbf{b}(0:2:n-1) - i\mathbf{E}_{n/2}\mathbf{P}_{n/2}\mathbf{b}(1:2:n-1) \end{bmatrix}$$

  The communication patterns for $\mathbf{G}_{n/2}$ and $\mathbf{E}_{n/2}$ are needed.

- Find $\mathbf{w} = \mathbf{F}_n\mathbf{P}_n(\mathbf{P}_n\mathbf{s})$. Need communication pattern for distributed FFTs via the Inverse Two-Track method.

- Set

$$\begin{aligned} \mathbf{x}_1 &= -\mathrm{Im}[\mathbf{w}] + diag(\tan\frac{k\pi}{n})\mathrm{Re}[\mathbf{w}] \\ \tilde{\mathbf{x}}_2 &= \mathrm{Re}[\mathbf{w}]diag(1/\cos\frac{k\pi}{n}) \end{aligned}$$

  No communication is needed here.

-

$$\mathbf{x} = \mathbf{x}_1 + \tilde{\mathbf{x}}_2$$

  No communication is needed here.

Algorithm 5.5.1 Inverse Sine Transform

```
/* $\mu$ = processor id; $n = 2^t$; $P = 2^d$; $m = (n/P)/2$ */
/* Initially, processor $\mu$ holds: */
/* $\mathbf{g}^{(1)} = \mathbf{g}(m \cdot id[\mu] : m \cdot (id[\mu]+1) - 1)$ */
/* $\mathbf{g}^{(2)} = \mathbf{g}(\frac{n}{2} + m \cdot id[\mu] : \frac{n}{2} + m \cdot (id[\mu] + 1) - 1)$ */
/* where $\mathbf{g} = \mathbf{P}_n\mathbf{b}$ */
/* $\mathbf{b}$ is the sine transform of $\mathbf{x}$ */
/* Form the two-tracks, $\mathbf{s}^{(1)}$ and $\mathbf{s}^{(2)}$ */
    $\mathbf{s}^{(1)} \leftarrow \mathbf{g}^{(1)} + i\mathbf{g}^{(2)}$
    call Exchange($\mathbf{g}^{(2)}$)
    call Recursive Exchange($\mathbf{g}^{(2)}$)
    $\mathbf{g}^{(2)} \leftarrow -\mathbf{g}^{(1)} - i\mathbf{g}^{(2)}$
/* $\mathbf{w} \leftarrow \mathbf{F}_n\mathbf{s}$ */
    call Two-Track FFT($\mathbf{s}^{(1)}, \mathbf{s}^{(2)}$)
    $\mathbf{x}_1 \leftarrow -Im[\mathbf{w}] + diag(\tan\frac{k\pi}{n})Re[\mathbf{w}]$
    $\mathbf{x}_2 \leftarrow Re[\mathbf{w}]diag(1/cos\frac{k\pi}{n})$
    $\mathbf{x} \leftarrow \mathbf{x}_1 + \mathbf{x}_2$
```

Procedures 1 and 1a are efficiently parallelizable because in both cases the preprocessing requires minimal communication and the post- processing is entirely local.

5.6 Implementation Results on the Intel Hypercube

The complete procedure was implemented on a four-dimensional Intel iPSC hypercube with extended memory in the nodes. Since our goal is to compute a forward sine transform, scale it and then do an inverse sine transform, our timings are for both the forward and inverse transform done back to back. We have also implemented the Cooley *et al.* (1970) algorithm on a single node and obtained a time of 121685 milliseconds for a sine and inverse transform of length 16384.

The results show almost linear speedup and reflect the efficiency with which the complex FFT can be parallelized. The speedup* against the sequential algorithm is also very good. Recall that the Cooley *et al.* sequential algorithm does a complex FFT of length $N/2$ or (8192 for our example). Therefore it should be roughly twice as fast as our algorithm in the sequential case. But our algorithm using only one processor is only about 4% slower than the conventional algorithm. So we see that the layers of pre- and post- processing are quite wasteful even in the totally sequential case.

One may wonder how this method would compare to doing a $2N$-point complex transform of the extended odd sequence (the only other feasible method on the hypercube). Results for the $N = 32768$ sequence are given below. Comparing

Table 5.2: Timing Results for the Parallel Sine Transform

Time (milliseconds) for Sine and Inverse Transform
$N = 16384$

# proc	time	speedup	speedup*
1	127515	1.00	0.95
2	67515–67535	1.89	1.80
4	36870–36915	3.45	3.30
8	19315–19375	6.58	6.28
16	10430–10535	12.10	11.55

(* speedup against conventional algorithm)

Table 5.3: Timing Results for Conventional Complex Transform

Time (milliseconds) for Complex Forward and Inverse Transform
$N = 32768$

# proc	time	speedup
1	197840	1.00
2	97785–97825	2.02
4	52450–52605	3.76
8	28800–28865	6.85
16	15055–15585	12.69

results shows that our method is faster than the direct computation of the $2N$-point sequence by 45%, 43%, 49% and 48% for hypercubes of dimension 1, 2, 3, and 4 respectively. Sequentially, it is also 55% faster.

5.7 The DFT and IDFT of Real Sequences

Real sequences exhibit special symmetry in the transform domain, that is their transforms are conjugate even. Thus an efficient algorithm computes the FFT of a real sequence of a fixed length n with the same amount of effort in terms of storage space and operations count as that of the complex FFT of length $n/2$. The economies come from the symmetry. This is indeed the case in the procedures described in Cooley *et al.* (1970). We repeat some of the procedures in matrix notation.

In each case we first derive the procedure and then give a high-level description, mentioning communication patterns with our permutation matrix language. We also give the equivalent procedure in bit-reversed order and output in consecutive order for completeness.

Procedure 2. The DFT of Two Real Sequences done as one

Suppose $\mathbf{x}_1$ and $\mathbf{x}_2$ are two real sequences. We wish to find $\mathbf{y}_1 = \mathbf{F}_n\mathbf{x}_1$ and $\mathbf{y}_2 = \mathbf{F}_n\mathbf{x}_2$. Since the transforms of real sequences possess symmetry, and $\mathbf{x}_1$ and $\mathbf{x}_2$ have zero imaginary parts, we can exploit this by forming the complex sequence $\mathbf{z} = \mathbf{x}_1 + i\mathbf{x}_2$ and taking the complex FFT of $\mathbf{z}$. Let $\mathbf{w} = \mathbf{F}_n\mathbf{z} = \mathbf{y}_1 + i\mathbf{y}_2$. $\mathbf{y}_1$ and $\mathbf{y}_2$ are conjugate even, so that we have

$$\begin{aligned} \mathbf{w} &= \mathbf{y}_1 + i\mathbf{y}_2 \\ \mathbf{T}\overline{\mathbf{w}} &= \mathbf{T}\overline{\mathbf{y}}_1 - i\mathbf{T}\overline{\mathbf{y}}_2 \\ &= \mathbf{y}_1 - i\mathbf{y}_2 \end{aligned}$$

hence

$$\begin{aligned} \mathbf{y}_1 &= \frac{1}{2}[\mathbf{T}\overline{\mathbf{w}} + \mathbf{w}] \\ \mathbf{y}_2 &= \frac{i}{2}[\mathbf{T}\overline{\mathbf{w}} - \mathbf{w}] \end{aligned}$$

Procedure 2. *Given:* Two real sequences $\mathbf{x}_1$ and $\mathbf{x}_2$. *Find:* the DFT $\mathbf{y}_1$ and $\mathbf{y}_2$ respectively of the two real sequences by using a radix-two in-place complex FFT which computes $\mathbf{P}_n\mathbf{F}_n\mathbf{z}$ (either CT1 or GS1).

- Form the complex vector $\mathbf{z} \leftarrow \mathbf{x}_1 + i\mathbf{x}_2$. No communication is needed if both $\mathbf{x}_1$ and $\mathbf{x}_2$ are distributed among the processors in the same manner.
- Compute $\mathbf{P}_n\mathbf{w} = \mathbf{P}_n\mathbf{F}_n\mathbf{z}$. This requires the communication pattern of distributed FFTs discussed in Chapter 3.
- Find

$$\begin{aligned} \mathbf{P}_n\mathbf{y}_1 &= \frac{1}{2}[\mathbf{G}_n(\mathbf{P}_n\overline{\mathbf{w}}) + \mathbf{P}_n\mathbf{w}] \\ \mathbf{P}_n\mathbf{y}_2 &= \frac{i}{2}[\mathbf{G}_n(\mathbf{P}_n\overline{\mathbf{w}}) - \mathbf{P}_n\mathbf{w}] \end{aligned}$$

The communication pattern required for $\mathbf{G}_n$ is needed.

Procedure 2a. *Given:* Two real sequences $\mathbf{P}_n\mathbf{x}_1$ and $\mathbf{P}_n\mathbf{x}_2$. *Find:* The DFT $\mathbf{y}_1$ and $\mathbf{y}_2$ respectively of the two real sequences using an in-place complex FFT which computes $\mathbf{F}_n\mathbf{P}_n\mathbf{z}$ (either CT2 or GS2).

- Form the complex vector $\mathbf{P}_n\mathbf{z} \leftarrow \mathbf{P}_n\mathbf{x}_1 + i\mathbf{P}_n\mathbf{x}_2$. No communication is needed.
- Compute $\mathbf{w} = \mathbf{F}_n\mathbf{P}_n(\mathbf{P}_n\mathbf{z})$. Requires the communication pattern for distributed FFTs.
- Find

$$\begin{aligned} \mathbf{y}_1 &= \frac{1}{2}[\mathbf{T}\overline{\mathbf{w}} + \mathbf{w}] \\ \mathbf{y}_2 &= \frac{i}{2}[\mathbf{T}\overline{\mathbf{w}} - \mathbf{w}] \end{aligned}$$

The communication pattern for permutation $\mathbf{T}_n$ is required.

Procedure 3. The DFT of a real sequence done as a complex FFT of half length

Let $\mathbf{x}$ be a real vector of length n, with $\tilde{\mathbf{x}} = \mathbf{M}_n^{(2)}\mathbf{x}$. Denote $\mathbf{x}^e = \tilde{\mathbf{x}}(0 : \frac{n}{2} - 1)$ and $\mathbf{x}^o = \tilde{\mathbf{x}}(\frac{n}{2} : n - 1)$. Form the complex sequence $\mathbf{z} = \mathbf{x}^e + i\mathbf{x}^o$ of length $\frac{n}{2}$. Notice that $\mathbf{x}^e$ and $\mathbf{x}^o$ are also real sequences so we can use the above procedure to find $\mathbf{y}^e = \mathbf{F}_{n/2}\mathbf{x}^e$ and $\mathbf{y}^o = \mathbf{F}_{n/2}\mathbf{x}^o$. Let $\mathbf{w} = \mathbf{F}_{n/2}\mathbf{z}$, then

$$\begin{aligned} \mathbf{y}^e &= \frac{1}{2}[\mathbf{T}\overline{\mathbf{w}} + \mathbf{w}] \\ \mathbf{y}^o &= \frac{i}{2}[\mathbf{T}\overline{\mathbf{w}} - \mathbf{w}] \end{aligned}$$

The Radix-2 Splitting algorithm gives us

$$\begin{aligned} \mathbf{F}_n\mathbf{x} &= \begin{bmatrix} \mathbf{F}_{n/2} & \mathbf{\Delta F}_{n/2} \\ \mathbf{F}_{n/2} & -\mathbf{\Delta F}_{n/2} \end{bmatrix} \begin{bmatrix} \mathbf{x}^e \\ \mathbf{x}^o \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{I}_{n/2} & \mathbf{\Delta} \\ \mathbf{I}_{n/2} & -\mathbf{\Delta} \end{bmatrix} \begin{bmatrix} \mathbf{y}^e \\ \mathbf{y}^o \end{bmatrix} \end{aligned}$$

where $\mathbf{\Delta} = diag(1, \omega_n, \ldots, \omega_n^{n/2-1})$ and $\omega_n = \exp(-2\pi i/n)$. Since $\mathbf{y} = \mathbf{T}\bar{\mathbf{y}}$, we only need to find $\mathbf{y}(0 : n/2)$. Hence

$$\begin{aligned} \mathbf{y}(0 : n/2 - 1) &= \mathbf{y}^e + \mathbf{\Delta y}^o \\ \mathbf{y}(n/2) &= \mathbf{y}^e(0) - \mathbf{y}^o(0) \end{aligned}$$

This procedure is entirely reversible and can be used to find the IDFT or DFT of a conjugate even complex sequence.

> **Procedure 3.** *Given:* A real sequence $\mathbf{x}(0 : n-1)$. *Find:* The DFT of $\mathbf{y}(0 : n-1)$ by using a radix-two complex FFT of half the length and post-sorting of data in bit-reversed order.

- Form two real sequences $\mathbf{x}^e$ of the even points of $\mathbf{x}$ and $\mathbf{x}^o$ of the odd points of $\mathbf{x}$. No communication is needed if $\mathbf{x}$ is mapped consecutively and $n = 2^t > P = 2^k$. Let $\mathbf{z} = \mathbf{x}^e + i\mathbf{x}^o$.

- Implement Procedure 2.

- Form $\mathbf{P}_n\mathbf{y} = \mathbf{P}_n\mathbf{y}^e + (\mathbf{P}_n\mathbf{\Delta})\mathbf{P}_n\mathbf{y}^o$. No communication is needed here.

Note that Procedure 3 requires the same data pattern as Procedure 2.

> **Procedure 3a.** *Given:* A real vector $\mathbf{P}_n\mathbf{x}$. *Find:* The DFT of $\mathbf{x}$ by using a complex FFT of half the length.

- Let $\hat{\mathbf{x}} = \mathbf{P}_n\mathbf{x}$. Form $\mathbf{P}_{n/2}\mathbf{z} = \hat{\mathbf{x}}(0 : n/2 - 1) + i\hat{\mathbf{x}}(n/2 : n-1)$. Processor local.

- Find $\mathbf{w} = \mathbf{F}_{n/2}\mathbf{P}_{n/2}(\mathbf{P}_{n/2}\mathbf{z})$ This requires distributed FFT communication pattern.

- Form $\mathbf{y}^e$ and $\mathbf{y}^o$. The communication pattern for $\mathbf{T}_n$ is needed.

- Form $\mathbf{y} = \mathbf{y}^e + \mathbf{\Delta y}^o$. No communication is needed.

Procedure 4. IDFT or DFT of a Conjugate Even Complex Sequence

Suppose we have a conjugate even complex sequence $\mathbf{y} = \mathbf{T}\overline{\mathbf{y}}$. This means that

$$\mathbf{y} = \begin{bmatrix} \mathbf{y}(0) \\ \mathbf{y}(1:n/2-1) \\ \mathbf{y}(n/2) \\ \mathbf{E}\overline{\mathbf{y}}(1:n/2-1) \end{bmatrix}$$

Going backwards, we can find

$$\begin{aligned} \mathbf{y}^e(0) &= \frac{1}{2}[\mathbf{y}(0) + \mathbf{y}(n/2)] \\ \mathbf{y}^e(1:n/2-1) &= \frac{1}{2}[\mathbf{y}(1:n/2-1) + \mathbf{E}\overline{\mathbf{y}}(1:n/2-1)] \\ \mathbf{y}^o(0) &= \frac{1}{2}[\mathbf{y}(0) - \mathbf{y}(n/2)] \\ \mathbf{y}^o(1:n/2-1) &= \frac{1}{2}\mathbf{\Delta}^{-1}(1:n/2-1)[\mathbf{y}(1:n/2-1) - \mathbf{E}\overline{\mathbf{y}}(1:n/2-1)] \end{aligned}$$

Form $\mathbf{w} = \mathbf{y}^e + i\mathbf{y}^o$ and find $\mathbf{z} = \mathbf{F}_n^H\mathbf{w}$. Now the real sequence $\mathbf{x}$ is found

$$\begin{aligned} \mathbf{x}^e &= \mathrm{Re}[\mathbf{z}] \\ \mathbf{x}^o &= \mathrm{Im}[\mathbf{z}] \end{aligned}$$

Procedure 4. *Given:* A complex conjugate even sequence $\mathbf{y}(0:n/2)$ (we only need to store the first $n/2+1$ elements because of the symmetry). *Find:* The IDFT or DFT of $\mathbf{y}$.

- Form $\mathbf{y}^e$ and $\mathbf{y}^o$. Communication pattern for permutation $\mathbf{T}_{n/2}$ is required. Let $\mathbf{w} = \mathbf{y}^e + i\mathbf{y}^o$.

- Find $\mathbf{P}_{n/2}\mathbf{z} = \mathbf{P}_{n/2}\mathbf{F}_{n/2}\mathbf{w}$. The communication pattern for distributed FFTs is required here.

- Set

$$\begin{aligned} \mathbf{P}_{n/2}\mathbf{x}^e &= \mathrm{Re}[\mathbf{P}_{n/2}\mathbf{z}] \\ \mathbf{P}_{n/2}\mathbf{x}^o &= \mathrm{Im}[\mathbf{P}_{n/2}\mathbf{z}] \end{aligned}$$

No communication is needed.

Procedure 4a. *Given:* A complex conjugate even sequence $\mathbf{P}_{n/2}\mathbf{y}(0:n/2-1)$. *Find:* The DFT $\mathbf{x}$ by using a complex FFT of half the length.

- Form $\mathbf{P}_{n/2}\mathbf{y}^e$ and $\mathbf{P}_{n/2}\mathbf{y}^o$. This requires communication for $\mathbf{T}_n$ permutation. Let $\mathbf{P}_{n/2}\mathbf{w} = \mathbf{P}_{n/2}\mathbf{y}^e + i\mathbf{P}_{n/2}\mathbf{y}^o$.
- Find $\mathbf{z} = \mathbf{F}_{n/2}\mathbf{P}_{n/2}(\mathbf{P}_{n/2}\mathbf{w})$. This requires communication for the distributed FFT.
- Let $\mathbf{x}^e = \mathrm{Re}[\mathbf{z}]$ and $\mathbf{x}^o = \mathrm{Im}[\mathbf{z}]$. Processor local.

5.8 The Cosine and Sine Transform

The DFT of a real even sequence corresponds to a special transform called the cosine transform. Similarly the DFT of a real odd sequence corresponds to the sine transform. These transforms come up in special applications ranging from image processing to the solution of elliptic partial differential equations with various boundary conditions.

In this section we introduce these symmetric transforms in matrix notation. We then focus on the sine transform and discuss the methods of Dollimore (1973) and Cooley *et al.* (1970) to compute the sine transform efficiently by exploiting all possible symmetries. Finally we talk our way through a hypothetical implementation of these methods on a parallel distributed system such as the hypercube and show how the plethora of symmetry exploitation causes a very complicated sequence of permutations (communication).

Matrix Description

Definition 5.8.1 *The cosine series of a sequence* $\mathbf{x}(j)$, $j = 0, \ldots, n$ *is defined to be*

$$\mathbf{a} = \frac{2}{n}\begin{bmatrix} \frac{1}{2} & \mathbf{w}^T & \frac{1}{2} \\ \frac{1}{2}\mathbf{w} & \mathbf{C}_{n-1} & \frac{1}{2}\mathbf{v} \\ \frac{1}{2} & \mathbf{v}^T & \frac{1}{2}(-1)^n \end{bmatrix}\mathbf{x}$$

Notice that the cosine series is only defined on $n+1$ points $\mathbf{x}(0:n)$.

Theorem 5.8.1 *Cosine Series: Let* $\mathbf{z}$ *be the even extension of* $\mathbf{x}$,

$$\mathbf{z} = \begin{bmatrix} \mathbf{x}(0) \\ \mathbf{x}(1:n-1) \\ \mathbf{x}(n) \\ \mathbf{E}\mathbf{x}(1:n-1) \end{bmatrix}$$

then if $\mathbf{y} = \mathbf{F}_{2n}\mathbf{z}$, *then*

$$\mathbf{a}(0:n) = 2\mathrm{Re}[\mathbf{y}(0:n)]$$

Proof $\mathbf{z}$ is real even, implying that $\mathbf{y}$ is purely real. Then

$$\mathbf{F}_{2n}\mathbf{z} = \frac{1}{2n}\begin{bmatrix} 1 & \mathbf{w}^T & 1 & \mathbf{w}^T \\ \mathbf{w} & \mathbf{C} & \mathbf{v} & \mathbf{CE} \\ 1 & \mathbf{v}^T & (-1)^n & \mathbf{v}^T\mathbf{E} \\ \mathbf{w} & \mathbf{EC} & \mathbf{Ev} & \mathbf{ECE} \end{bmatrix}\begin{bmatrix} \mathbf{x}(0) \\ \mathbf{x}(1:n-1) \\ \mathbf{x}(n) \\ \mathbf{Ex}(1:n-1) \end{bmatrix}$$

$$\begin{aligned}
\mathbf{y}(0) &= \frac{1}{2n}[\mathbf{x}(0) + 2\mathbf{w}^T\mathbf{x}(1:n-1) + \mathbf{x}(n)] \\
\mathbf{y}(1:n) &= \frac{1}{2n}[\mathbf{x}(0)\mathbf{w} + 2\mathbf{Cx}(1:n-1) + \mathbf{x}(n)\mathbf{v}] \\
\mathbf{y}(n) &= \frac{1}{2n}[\mathbf{x}(0) + 2\mathbf{v}^T\mathbf{x}(1:n-1) + (-1)^n\mathbf{x}(n)] \\
\mathbf{y}(n+1:2n-1) &= \frac{1}{2n}[\mathbf{x}(0)\mathbf{w} + 2\mathbf{ECx}(1:n-1) + \mathbf{Evx}(n)]
\end{aligned}$$

Notice that $\mathbf{y}(n+1:2n-1) = \mathbf{Ey}(1:n)$, which is what we expect.

——*——

Definition 5.8.2 *The Inverse Cosine Transform is defined as*

$$\mathbf{x} = \begin{bmatrix} \frac{1}{2} & \mathbf{w}^T & \frac{1}{2} \\ \frac{1}{2}\mathbf{w} & \mathbf{C}_{n-1} & \frac{1}{2}\mathbf{v} \\ \frac{1}{2} & \mathbf{v}^T & \frac{1}{2}(-1)^n \end{bmatrix}\mathbf{a}$$

Hence the cosine transform is its own inverse up to a multiplicative constant.

Definition 5.8.3 *The DFT of a real odd sequence corresponds to the sine series defined on a sequence* $\mathbf{x}(j)$, $j = 1, \ldots, n$.

$$\mathbf{b}(1:n) = \frac{2}{n}\mathbf{S}_{n-1}\mathbf{x}(1:n)$$

Notice that the sine series is defined on only $n-1$ points $\mathbf{x}(1:n)$.

Theorem 5.8.2 *Let* $\mathbf{z}$ *be the odd extension of* $\mathbf{x}$.

$$\mathbf{z} = \begin{bmatrix} 0 \\ \mathbf{x}(1:n-1) \\ 0 \\ -\mathbf{Ex}(1:n-1) \end{bmatrix}$$

then if $\mathbf{y} = \mathbf{F}_{2n}\mathbf{z}$,

$$\mathbf{b}(1:n) = 2i\mathbf{y}(1:n)$$

Proof Since **z** is real and odd, the $\mathbf{y} = \mathbf{F}_{2n}\mathbf{z}$ is purely imaginary, therefore

$$\mathbf{F}_{2n}\mathbf{z} = \frac{-i}{2n}\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \mathbf{S} & 0 & -\mathbf{SE} \\ 0 & 0 & 0 & 0 \\ 0 & -\mathbf{ES} & 0 & \mathbf{ESE} \end{bmatrix}\begin{bmatrix} 0 \\ \mathbf{x}(1:n-1) \\ 0 \\ -\mathbf{Ex}(1:n-1) \end{bmatrix}$$

$$\begin{aligned} \mathbf{y}(1:n) &= \frac{-i}{2n}[2\mathbf{Sx}(1:n-1)] \\ \mathbf{y}(n+1:2n-1) &= \frac{-i}{2n}[-2\mathbf{ESx}(1:n-1)] \end{aligned}$$

Here $\mathbf{y}(n+1:2n-1) = -\mathbf{Ey}(1:n)$ as expected.

——*——

Definition 5.8.4 *The Inverse Sine Transform is defined as*

$$\mathbf{x} = \mathbf{S}_{n-1}\mathbf{b}$$

so that the sine transform is also its own inverse up to a multiplicative constant.

The Sine Transform Algorithm

The most straightforward sine transform algorithm would be to follow the derivation in Theorem 5.8.2 and extend the given sequence of length n into its odd extension. The FFT of this $2n$-point sequence is then computed and the sine transform is found in the imaginary part of the transform of the extended sequence. This method has a number of drawbacks in the single processor case. For one, it requires four times the array storage than necessary. This is because the extended sequence is both odd and real. But if a complex FFT were to be done, we would need $2n$ points of complex storage. It is also obvious that we are doing computationally more work than is necessary. However given the difficulties with which the conventional sine transform procedures of Dollimore (1973) and Cooley *et al.* (1970) are implemented on the hypercube, this naive method is actually recommended by some researchers for parallel implementation. This is because given an efficient complex FFT and parallel computing, one may not care too much about whether a $2n$-point transform is done instead of an n-point or $n/2$-point transform. It should be noted that it requires a non-trivial set of communication to convert an n-point vector mapped in consecutive order into a $2n$-point odd extension of itself. Try it with paper and pencil!

The sine transform can be calculated efficiently via the complex FFT by taking advantage of the symmetry in the extended sequence. These algorithms usually involve the use of a real FFT of length n coupled with some pre- and post- processing of the sequence to exploit the additional symmetry. There are two different ways of doing this. Each is the inverse of the other. The first method is usually attributed to Dollimore (1973) and is presented by Swarztrauber (1982) and Press *et al.* (1986).

Procedure 5. Dollimore's Sine Transform

Suppose we have a sequence $\mathbf{x}(0:n)$ where $\mathbf{x}(0) = 0$. An auxilliary array is constructed so that the first term is even and the second term is odd. Let $\mathbf{\Delta}_s = diag(\sin\frac{j\pi}{n})$, $j = 0, 1, \ldots, n-1$. The constructed sequence is

$$\mathbf{d} = \mathbf{\Delta}_s(\mathbf{x} + \mathbf{T}\mathbf{x}) + \frac{1}{2}(\mathbf{x} - \mathbf{T}\mathbf{x}) \tag{5.8-1}$$

Taking the FFT of this sequence yields

$$\mathbf{y} = \mathbf{F}_n\mathbf{d} = \mathbf{F}_n\mathbf{\Delta}_s(\mathbf{x} + \mathbf{T}\mathbf{x}) + \frac{1}{2}\mathbf{F}_n(\mathbf{x} - \mathbf{T}\mathbf{x})$$

with

$$\begin{aligned} \mathrm{Re}[\mathbf{y}] &= \mathrm{Re}[\mathbf{F}_n]\mathbf{\Delta}_s(\mathbf{x} + \mathbf{T}\mathbf{x}) \\ \mathrm{Im}[\mathbf{y}] &= \frac{1}{2}\mathrm{Im}[\mathbf{F}_n](\mathbf{x} - \mathbf{T}\mathbf{x}) \end{aligned}$$

First we look at Re[$\mathbf{y}$]. We see that $\mathbf{T}\mathbf{\Delta}_s = \mathbf{\Delta}_s\mathbf{T} = \mathbf{\Delta}_s$ so that

$$\mathrm{Re}[\mathbf{y}] = 2\mathrm{Re}[\mathbf{F}_n]\mathbf{\Delta}_s\mathbf{x}$$

Now

$$\begin{aligned} [\mathrm{Re}[\mathbf{F}_n]\mathbf{\Delta}_s]_{pq} &= \cos\frac{2\pi pq}{n}\sin\frac{\pi p}{n} \\ &= \frac{1}{2}\left\{\sin\left(\frac{2q+1}{n}p\pi\right) - \sin\left(\frac{2q-1}{n}p\pi\right)\right\} \end{aligned}$$

Hence

$$\mathrm{Re}[\mathbf{y}] = \mathbf{R}_n\mathrm{Re}[\mathbf{F}_n]\mathbf{x} - \mathbf{R}_n^T\mathrm{Re}[\mathbf{F}_n]\mathbf{x}$$

where $\mathbf{R}_n^T$ is the down-shift matrix. For Im[$\mathbf{y}$], we have $\mathrm{Im}[\mathbf{F}_n]\mathbf{T} = -\mathrm{Im}[\mathbf{F}_n]$ and thus

$$\begin{aligned} \mathrm{Im}[\mathbf{y}] &= \frac{1}{2}[\mathrm{Im}[\mathbf{F}_n]](\mathbf{x} - \mathbf{T}\mathbf{x}) \\ &= \mathrm{Im}[\mathbf{F}_n]\mathbf{x} \end{aligned}$$

Im[**y**] gives us the even terms of the sine transform, and a recursion on Re[**y**] gives us the odd terms. So

$$\begin{aligned}
\mathbf{b}_1 &\leftarrow \frac{1}{2}\mathbf{y}(0) \\
\mathbf{b}_{2k} &\leftarrow \mathrm{Im}[\mathbf{y}_k] \\
\mathbf{b}_{2k+1} &\leftarrow b_{2k-1} + \mathrm{Re}[\mathbf{y}_k] \\
& \quad k = 0, \ldots, \frac{n}{2} - 1.
\end{aligned}$$

This algorithm is not efficiently vectorized because of this last recurrence. Temperton (1980) has shown however, that the numerical properties of Dollimore's method is superior to the vectorizable method we present next.

The Sine Transform of Cooley *et al.* (1970)

The inverse of Dollimore's algorithm is the one presented by Cooley *et al.* (1970). A complex conjugate even sequence is formed out of the original real sequence. Let $\mathbf{b}(0:n)$ be a sequence of real numbers with $\mathbf{b}(0) = 0$. Form the complex conjugate even sequence **z** as follows

$$\begin{aligned}
\mathbf{z}(0) &\leftarrow (2\mathbf{b}(1), 0) \\
\mathrm{Re}[\mathbf{z}(j)] &\leftarrow \mathbf{b}(2j+1) - \mathbf{b}(2j-1) \\
\mathrm{Im}[\mathbf{z}(j)] &\leftarrow \mathbf{b}(2j) \\
& \quad j = 1, \ldots, n/2 - 1 \\
\mathbf{z}(n/2) &\leftarrow (-2\mathbf{b}(n-1), 0)
\end{aligned}$$

Procedure 4 is used to find $\mathbf{d} = \mathbf{F}_n\mathbf{z}$. **d** is a real sequence and the inverse of equation 5.8-1 gives

$$\mathbf{x} = \frac{1}{4}\mathbf{\Delta}_s^{-1}(\mathbf{Td} + \mathbf{d}) + \frac{1}{2}(\mathbf{Td} - \mathbf{d})$$

Implementation of the Conventional Sine Transform Methods on Distributed Systems

We outline how the sine transform can be implemented via Dollimore's algorithm and the algorithm of Cooley *et al.* and describe the permutations necessary. Here we see that the implementation of either of these two sine transform algorithms would be inefficient due to the complicated maneuvering of the permutations.

> **Procedure 5.** *Given:* A real sequence **x**. Find the sine transform **b** of **x** via Dollimore's algorithm.

- Form the auxilliary sequence

$$\mathbf{d} = \mathbf{\Delta}_s(\mathbf{x} + \mathbf{T}\mathbf{x}) + \frac{1}{2}(\mathbf{x} - \mathbf{T}\mathbf{x})$$

  The communication pattern for $\mathbf{T}_n$ is needed.

- Find $\mathbf{P}_n\mathbf{y} = \mathbf{P}_n\mathbf{F}_n\mathbf{d}$ via Procedure 3.

- Let the even points of $\mathbf{P}_n\mathbf{b} = \mathrm{Im}[\mathbf{P}_n\mathbf{y}]$. Solve the recurrence for the odd points. This is difficult because the entries of $\mathbf{y}$ are in bit-reversed order and we require adjacent entries to be in adjacent processors. Hence a distributed bit-reversal must be done first before the linear shift nearest-neighbor permutations $\mathbf{R}_n^T$ and $\mathbf{R}_n$ are performed.

Procedure 6. *Given:* A real sequence $\mathbf{x}$. Find the sine transform $\mathbf{b}$ of $\mathbf{x}$ via the algorithm of Cooley *et al.*

- Set up the conjugate even sequence $\mathbf{z}$. Nearest neighbor connections or $\mathbf{R}_n^T$ and $\mathbf{R}_n$ permutations are required.

- Use Procedure 4 to find the real sequence $\mathbf{d} = \mathbf{P}_n\mathbf{F}_n\mathbf{z}$.

- Find

$$\mathbf{P}_n\mathbf{x} = \frac{1}{4}\mathbf{\Delta}_s^{-1}(\mathbf{G}_n\mathbf{P}_n\mathbf{d} + \mathbf{P}_n\mathbf{d}) + \frac{1}{2}(\mathbf{G}_n\mathbf{P}_n\mathbf{d} - \mathbf{P}_n\mathbf{d})$$

  The communication pattern for recursive exchange $\mathbf{G}_n$ is required.

Since the sine transform is its own inverse, any sine transform algorithm can be used to invert a transform. Hence we outline Procedure 5 and 6 with bit-reversed input.

Procedure 5a. *Given:* A real sequence $\mathbf{P}_n\mathbf{x}$. Find the sine transform $\mathbf{b}$ via Dollimore's algorithm.

- Form the auxilliary sequence

$$\mathbf{P}_n\mathbf{d} = \mathbf{P}_n\mathbf{\Delta}_s(\mathbf{P}_n\mathbf{x} + \mathbf{G}_n(\mathbf{P}_n\mathbf{x})) + \frac{1}{2}(\mathbf{P}_n\mathbf{x} - \mathbf{G}_n(\mathbf{P}_n\mathbf{x}))$$

  Here we need the communication pattern for $\mathbf{G}_n$.

- Find $\mathbf{y} = \mathbf{F}_n\mathbf{P}_n(\mathbf{P}_n\mathbf{d})$ via Procedure 3a.

- Let the even points of $\mathbf{b} = \mathrm{Im}[\mathbf{y}]$, and the odd points are solved by recurrence. This is inefficiently parallelized.

Procedure 6a. *Given:* A real sequence $\mathbf{P}_n\mathbf{b}$. Fnd the sine transform $\mathbf{x}$ via the algorithm of Cooley *et al.*

- Set up the complex sequence $\mathbf{P}_n\mathbf{z}$. This is difficult because the real part of $\mathbf{P}_n\mathbf{z}$ requires nearest neighbor connections, or shift permutations, but everything we have is scrambled up. Now we have $\mathbf{P}_n\mathbf{b}$, but there is no easy way to commute $\mathbf{R}_n^T$ and $\mathbf{P}_n$.
- Use Procedure 4a to find the real sequence $\mathbf{d} = \mathbf{F}_n\mathbf{P}_n(\mathbf{P}_n\mathbf{z})$.
- Find
$$\mathbf{x} = \frac{1}{4}\mathbf{\Delta}_s^{-1}(\mathbf{Td} + \mathbf{d}) + \frac{1}{2}(\mathbf{Td} - \mathbf{d})$$
The communication pattern for $\mathbf{T}_n$ is required.

A quick analysis of the communication needed to implement these two sine transforms show that the recurrence for Dollimore's algorithm is inefficiently parallelized. Setting up $\mathbf{P}_n\mathbf{z}$ for the algorithm of Cooley *et al.* is also difficult and requires unnecessary communications to unscramble data. So basically, to implement these conventional sine transform algorithms on distributed systems, we need to be able to do distributed bit-reversal permutations efficiently. Since many algorithms require a transform followed by an inverse transform of the data, without need for natural ordered data in the transform domain, unscrambling of the data is unnecessary. Therefore the cost of pre- and post- processing operations and the communication needed to carry out these algorithms outweighs the benefits of their design to minimize operations counts. This is a typical problem in parallel computation where problems which are optimized for a single processor can actually be far more difficult to implement and carry much more overhead to parallelize than an algorithm which is less efficient in a single processor environment. This is because with parallel computing, we desire work to be as independent of each other to maximize parallelism. Sometimes this means that it is cheaper to do some redundant computations to defer the need for communication until absolutely necessary. This is how we approach our new sine transform, an efficiently parallelizable version.

5.9 Conclusion and Discussion

Although we have mainly concerned ourselves with the sine transform in this chapter, the exact same type of reasoning behind our parallel sine transform can be used for a parallel cosine transform algorithm.

The sine transform is used extensively in the Fourier method for the solution of boundary value problems in partial differential equations with Dirichlet-Dirichlet boundary conditions. [Swarztrauber (1977)] In fact, the tensor product method for finding the solution to the Poisson equation with Dirichlet boundary conditions involves a two-dimensional sine transform [see Lynch, Rice and Thomas (1964)]. While in the single processor case, one would choose to use the matrix decomposition method [Buzbee (1973)], involving a fast transform in one direction and tridiagonal solves in the other, with the hypercube, it might be interesting to substitute the tridiagonal solves with fast transforms that are more efficiently parallelizable than tridiagonal solvers usually are. Sine and cosine transforms are also used extensively in image processing which may require transforms in two or more dimensions. Therefore fast efficiently implemented algorithms for vector computers and parallel systems are important and should be considered a basic building block in many other more complicated algorithms.

Chapter 6

Fast Symmetric Transforms

6.1 Introduction

Sine and cosine transforms arise naturally from the Fourier transform of symmetric sequences. They are used in a variety of problems in applied mathematics ranging from the digital coding of images [Clarke (1985)] to the fast solution of Poisson's partial differential equation with various boundary conditions [Swarztrauber (1984b)].

Since they exhibit certain special symmetries, the computation of symmetric transforms should be up to four times more efficient, in terms of storage and operations count, than the complex FFT. This is because the transform of a real sequence is conjugate even, so there is a two-fold redundancy of information. Furthermore, the sine transform comes from the DFT of an extended odd real sequence and is itself odd. Therefore another two-fold redundancy of information is present. Hence a smart efficient Fast Symmetric Transform algorithm should require a quarter of the array storage and roughly a quarter of the computational effort of a complex FFT algorithm. There are several sequential algorithms that achieve this objective (discussed in Chapter 5 for the sine transform). [Cooley *et al.* (1970), Swarztrauber (1982), Press *et al.* (1986)] These algorithms require pre- and post- processing and are very difficult to implement efficiently on a parallel memory-passing system. Therefore in Chapter 5 we present a parallel algorithm specifically for the sine transform. The technique of Chapter 5 can be used on other symmetric transforms but each case must be worked out individually.

In this chapter we present a new algorithm that can simultaneously find the sine, cosine, quarter-wave sine and cosine transform of a single real sequence by computing only one complex FFT of the same length. Furthermore, all multiplication in our method is by roots of unity and hence the method is numerically stable. Portability is no problem because after an initial permutation of the data, any standard FFT subroutine can be used to find the results.

Other methods used to find symmetric transforms include the method of transforming extended sequences. A sequence is extended so that its symmetry becomes obvious. For example, given a data vector x_i, $i = 1, \ldots, n-1$, we can extend it into an odd function

$$(0, x_1, \ldots, x_{n-1}, 0, -x_{n-1}, \ldots, -x_1).$$

The DFT of this extended function is both odd and conjugate even, and thus purely imaginary. Furthermore, the sine transform of x_i (up to a factor of $2i$). is found in the imaginary portion of the first n points of the DFT of the extended function. [See Press *et al.* (1986)]. This method is as numerically stable as the Fast Fourier Transform, a sequence of unitary transformations, but requires the use of an extended sequence of twice the length. Hence, instead of reducing the work by a factor of four, we have actually increased the work by a factor of two. For higher-dimensional transforms, the factor increases as 2^d where d is the dimension of the transform. This also means that 2^d times the storage space is needed to transform these points. Therefore this method is neither efficient in the computational sense nor in the use of memory. The advantage however is that this method is portable to any architecture or machine where efficient FFT subroutines have already been worked out, implemented and tested.

Direct methods are discussed by Chen, Smith and Fralick (1977), Yip and Rao (1980), Wang (1981a–c), (1984), the gist of which is best explained by Swarztrauber (1986b). The basic idea is to emulate the factorization of the DFT matrix. By factoring the discrete sine transform (DST) and discrete cosine transform (DCT) matrices into sparse matrices, a fast algorithm for these symmetric transforms can be derived with reduced operations counts since redundant operations are not done. Furthermore since the operations done are a subset of the ones done by the FFT algorithm, the direct methods possess the numerical stability of the complex FFT algorithm. The major drawback of these direct methods is the complexity of the signal flow diagrams. This makes it difficult to implement in parallel, as significant time could be taken just to find out where something has to go. See Wang (1984) and references therein for a complete description of the algorithms as well as the theoretical background leading up to these factorizations.

This chapter is summarized as follows. The four types of cosine and sine transforms are first defined in terms of their representation as matrix-vector multiplication. The new method is presented and a demonstration is given on how the symmetries are exploitable. This method is then implemented on the hypercube and timing results are analyzed. A few applications are discussed and in the appendix details of the proof for the type III and IV transforms are presented.

Finally the FORTRAN code for this set of procedures is given to show how easy it is to implement.

6.2 Definitions and Matrix Notation

The four types of symmetric transforms are described by Wang (1981a–c) as a real approach to harmonic analysis. If $f(x)$ is a bounded real function it can be expressed in terms of the orthonormal set of the trigonometric functions

$$\sqrt{\frac{1}{\pi}}\sin\left[\frac{\pi}{4}+(n+\alpha)x\right], \quad n=\ldots,-2,-1,0,1,2,\ldots$$

n has to be an integer, but α can be any real number. In practice, however, only integer and half integer ($\alpha = k$ and $\alpha = k + \frac{1}{2}$, for k integer) harmonics are used. This is because in these two cases, symmetry properties of a sequence can be used to decompose the real transform into discrete cosine and sine transforms. Another reason is that it is easier to find fast algorithms for these two types of harmonics.

The matrices for the four versions of the discrete cosine transforms (DCTs) and the discrete sine transforms (DSTs) corresponding to the integer and half-integer harmonics are described by defining their p-qth element. The transform of a particular sequence is then a simple matrix-vector multiplication.

DCT-I: $\sqrt{\frac{2}{n}}[k_p k_q \cos(pq\pi/n)], \quad p,q=0,1,\ldots,n.$

DCT-II: $\sqrt{\frac{2}{n}}[k_p \cos(p(q+\frac{1}{2})\pi/n)], \quad p,q=0,1,\ldots,n-1.$

DCT-III: $\sqrt{\frac{2}{n}}[k_q \cos((p+\frac{1}{2})q\pi/n)], \quad p,q=0,1,\ldots,n-1.$

DCT-IV: $\sqrt{\frac{2}{n}}[\cos(p+\frac{1}{2})(q+\frac{1}{2})\pi/n)], \quad p,q=0,1,\ldots,n-1.$

DST-I: $\sqrt{\frac{2}{n}}[\sin(pq\pi/n)], \quad p,q=1,\ldots,n-1.$

DST-II: $\sqrt{\frac{2}{n}}[k_p \sin(p(q-\frac{1}{2})\pi/n)], \quad p,q=1,\ldots,n.$

DST-III: $\sqrt{\frac{2}{n}}[k_q \sin((p-\frac{1}{2})q\pi/n)], \quad p,q=1,\ldots,n.$

DST-IV: $\sqrt{\frac{2}{n}}[\sin((p+\frac{1}{2})(q+\frac{1}{2})\pi/n)], \quad p,q=0,1,\ldots,n-1.$

where

$$k_l = \begin{cases} 1 & \text{if } l \neq 0 \text{ or } n \\ \sqrt{\frac{1}{2}} & \text{if } l = 0 \text{ or } n. \end{cases}$$

It can be shown by simple verification that DCT-I is its own inverse, DCT-II and DCT-III are inverses of each other, DCT-IV is its own inverse, DST-I is its own inverse, DST-II and DST-III are inverses of each other, and DST-IV is its own inverse [Wang and Hunt (1985)].

These symmetric transforms are not artificial creations, but appear naturally as the eigenvectors of the finite difference operator of Poisson's equation on a square with various boundary conditions. [Swarztrauber (1977)] DCT-I and DST-I are the usual cosine and sine transforms, respectively. And DCT-II and DST-II are the quarter-wave cosine and sine transforms with DCT-III and DST-III their inverses, respectively. These transforms are also used in digital signal processing applications and in image processing. [Ahmed *et al.* (1974), Kekre and Solanki (1978), Jain (1976, 1979), Wang and Hunt (1985)]

Matrix Notation

Each of these transform matrices can be simplified into an n-by-n matrix consisting purely of cosines or sines. We get rid of the k_l scaling terms. The matrix-vector multiplication of this is almost the corresponding discrete transform with a few adjustments that appear later.

$$\mathbf{C}_n^I = [\cos(pq\pi/n)], \quad p,q = 0,1,\ldots,n-1.$$

$$\mathbf{C}_n^{II} = [\cos(p(q+\frac{1}{2})\pi/n)], \quad p,q = 0,1,\ldots,n-1.$$

$$\mathbf{C}_n^{III} = [\cos((p+\frac{1}{2})q\pi/n)], \quad p,q = 0,1,\ldots,n-1.$$

$$\mathbf{C}_n^{IV} = [\cos((p+\frac{1}{2})(q+\frac{1}{2})\pi/n)], \quad p,q = 0,1,\ldots,n-1.$$

$$\mathbf{S}_n^I = [\sin(pq\pi/n)], \quad p,q = 0,1,\ldots,n-1.$$

$$\mathbf{S}_n^{II} = [\sin(p(q+\frac{1}{2})\pi/n)], \quad p,q = 0,1,\ldots,n-1.$$

$$\mathbf{S}_n^{III} = [\sin((p+\frac{1}{2})q\pi/n)], \quad p,q = 0,1,\ldots,n-1.$$

$$\mathbf{S}_n^{IV} = [\sin((p+\frac{1}{2})(q+\frac{1}{2})\pi/n)], \quad p,q = 0,1,\ldots,n-1.$$

Notice that these definitions are not exactly the same as the corresponding definitions of the discrete transforms, however, once we have figured out let's say $\mathbf{C}_{n+1}^I x$, it is easy to find the DCT-I of x by subtracting $\sqrt{\frac{1}{2}}x_0$ from each component, adding $\sqrt{\frac{1}{2}}x_n$ to the even components, and subtracting $\sqrt{\frac{1}{2}}x_n$ from the odd components. Finally the whole thing is scaled by $\sqrt{\frac{2}{n}}$. A summary of these "retrieval" procedures appears later.

6.3 Description and Development

In this section, we present our new method of computing the symmetric transforms so that those of type I and II and computed together and those of type III and IV are computed together. First we introduce a permutation matrix that is used in this analysis. We then give an overview of the relationships between the symmetric transforms, after which we derive these relationships in detail.

Quarter-wave Splitting Permutations

The permutation that is used to simplify the description of sequences with quarter-wave symmetries is defined as follows.

$$\mathbf{K}_n = \begin{bmatrix} \mathbf{I}_{\lceil n/2\rceil} & \\ & \mathbf{E}_{\lfloor n/2\rfloor} \end{bmatrix} \mathbf{M}_n$$

A highly related "manipulation" of a data vector is

$$\mathbf{L}_n = \begin{bmatrix} \mathbf{I}_{\lceil n/2\rceil} & \\ & -\mathbf{E}_{\lfloor n/2\rfloor} \end{bmatrix} \mathbf{M}_n$$

In English, $\mathbf{K}_n$ takes all the even points of a sequence first, and then the odd points going backwards. $\mathbf{L}_n$ does the same thing except it takes the negative of the odd points going backwards.

Example

$$\mathbf{K}_n \begin{bmatrix} 0\\1\\2\\3\\4\\5\\6\\7 \end{bmatrix} = \begin{bmatrix} 0\\2\\4\\6\\7\\5\\3\\1 \end{bmatrix} \qquad \mathbf{L}_n \begin{bmatrix} 0\\1\\2\\3\\4\\5\\6\\7 \end{bmatrix} = \begin{bmatrix} 0\\2\\4\\6\\-7\\-5\\-3\\-1 \end{bmatrix}$$

The Four Types of Symmetric Transforms

All four types of cosine and sine transforms are related to a certain complex matrix-vector multiplication. Recall that the discrete Fourier transform matrix of dimension n is a matrix consisting of powers of $\omega_n = e^{-2\pi i/n}$. Let $\mathbf{F}_n$ be the DFT matrix of order n and $\mathbf{F}_{2n}$ be the DFT matrix of order $2n$. We consider the upper-left n-by-n corner of the matrix $\mathbf{F}_{2n}$ denoted by

$$\hat{\mathbf{F}}_{2n} = \mathbf{F}_{2n}(0 : n-1, 0 : n-1)$$

and show that all of these transforms involve either the term $\hat{\mathbf{F}}_{2n}\mathbf{x}$ or $\hat{\mathbf{F}}_{2n}(\mathbf{\Delta x})$ where $\mathbf{\Delta}$ is a particular scaling of $\mathbf{x}$ by a diagonal matrix of twiddle factors, i.e. $4n$-th roots of unity.

The matrix $\hat{\mathbf{F}}_{2n}$ provides the link between the various transforms. The diagonal matrices are n-by-n matrices consisting of

$$\begin{aligned} \mathbf{\Delta} &= diag(e^{-\pi ik/2n}) \\ \mathbf{\Delta}^2 &= diag(e^{-\pi ik/n}) \end{aligned}$$

Table 6.1 displays the formulas involved. One can see that the type I transform can be obtained from the type II transforms and the type II transforms from the type IV transforms. From Table 6.1 we see that the DFT of $\mathbf{K}_n\mathbf{x}$ or $\mathbf{L}_n\mathbf{x}$ (permuted versions of $\mathbf{x}$ are all that are required to be computed via the FFT algorithm. Further the FFT is of length n rather than $2n$. Basically two DFTs are needed for each set of four symmetric transforms: $\mathbf{F}_n(\mathbf{K}_n\mathbf{x})$ and $\mathbf{F}_n(\mathbf{L}_n\mathbf{x})$ for the type I and type II transforms, and $\mathbf{F}_n(\mathbf{\Delta}^2\mathbf{L}_n\mathbf{x})$ and $\mathbf{F}_n(\mathbf{\Delta}^2\mathbf{K}_n\mathbf{x})$ for the type III and type IV transforms. The first two are real transforms and the second two are complex transforms.

Makhoul (1980) describes a way of finding $\mathbf{C}_n^{II}\mathbf{x}$ by find the real part of the transform $\mathbf{\Delta F}_n(\mathbf{K}_n\mathbf{x})$. He establishes the equivalence between $Re[\mathbf{\Delta F}_n(\mathbf{K}_n\mathbf{x})]$ and $Re[\mathbf{\Delta}\hat{\mathbf{F}}_{2n}\mathbf{x}]$. What we do is establish the same sort of procedure for the type II sine transform $\mathbf{S}_n^{II}\mathbf{x}$. Once we have $\mathbf{C}_n^{II}\mathbf{x}$ and $\mathbf{S}_n^{II}\mathbf{x}$ available, we show how it is an easy step to finding the type I transforms. We form a sequence by putting $\mathbf{C}_n^{II}\mathbf{x}$ in the real part and $\mathbf{S}_n^{II}\mathbf{x}$ in the imaginary part. We then scale the sequence by $\mathbf{\Delta}^{-1}$ and then find $\mathbf{C}_n^{I}\mathbf{x}$ in the real part of the resulting sequence and $\mathbf{S}_n^{I}\mathbf{x}$ in the imaginary part. A similar procedure is used for the types III and IV transforms. The algorithms that follow describe the techniques used.

Summary of Procedures

Algorithm 6.3.1 describes how to find the cosine and sine transforms of type I and type II simultaneously.

Algorithm 6.3.1 Procedure Type12:

```
$\mathbf{v} \leftarrow \mathbf{K}_n\mathbf{x}$; $\mathbf{z} \leftarrow \mathbf{L}_n\mathbf{x}$
$\mathbf{v} \leftarrow \mathbf{F}_n\mathbf{v}$; $\mathbf{z} \leftarrow \mathbf{F}_n\mathbf{z}$
$\mathbf{v} \leftarrow \mathbf{\Delta}_{4n}\mathbf{v}$; $\mathbf{z} \leftarrow \mathbf{\Delta}_{4n}\mathbf{z}$
$\mathbf{C}_n^{II}\mathbf{x} = Re(\mathbf{v})$; $\mathbf{S}_n^{II}\mathbf{x} = -Im(\mathbf{z})$
$Re(\mathbf{a}) \leftarrow Re(\mathbf{v})$; $Im(\mathbf{a}) = Im(\mathbf{z})$
$\mathbf{a} \leftarrow \mathbf{\Delta}^{-1}\mathbf{a}$
$\mathbf{C}_n^{I}\mathbf{x} = Re(\mathbf{a})$; $\mathbf{S}_n^{I}\mathbf{x} = -Im(\mathbf{a})$
```

Table 6.1: Symmetric Transform Formulae

transform	formulae
DCT-I	$\mathbf{C}_n^I\mathbf{x} = Re[\hat{\mathbf{F}}_{2n}\mathbf{x}]$ $= Re[\mathbf{\Delta}^{-1}(\mathbf{C}_n^{II}\mathbf{x} + i\mathbf{S}_n^{II}\mathbf{x})]$
DST-I	$\mathbf{S}_n^I\mathbf{x} = -Im[\hat{\mathbf{F}}_{2n}\mathbf{x}]$ $= -Im[\mathbf{\Delta}^{-1}(\mathbf{C}_n^{II}\mathbf{x} + i\mathbf{S}_n^{II}\mathbf{x})]$
DCT-II	$\mathbf{C}_n^{II}\mathbf{x} = Re[\mathbf{\Delta}\hat{\mathbf{F}}_{2n}\mathbf{x}]$ $= Re[\mathbf{\Delta}\mathbf{F}_n\mathbf{K}_n\mathbf{x}]$
DST-II	$\mathbf{S}_n^{II}\mathbf{x} = -Im[\mathbf{\Delta}\hat{\mathbf{F}}_{2n}\mathbf{x}]$ $= -Im[\mathbf{\Delta}\mathbf{F}_n\mathbf{L}_n\mathbf{x}]$
DCT-III	$\mathbf{C}_n^{III}\mathbf{x} = Re[\hat{\mathbf{F}}_{2n}\mathbf{\Delta}\mathbf{x}]$ $= Re[e^{\pi i/4n}\mathbf{\Delta}^{-1}(\mathbf{C}_n^{IV}\mathbf{x} + i\mathbf{S}_n^{IV}\mathbf{x})]$
DST-III	$\mathbf{S}_n^{III}\mathbf{x} = -Im[\hat{\mathbf{F}}_{2n}\mathbf{\Delta}\mathbf{x}]$ $= -Im[e^{\pi i/4n}\mathbf{\Delta}^{-1}(\mathbf{C}_n^{IV}\mathbf{x} + i\mathbf{S}_n^{IV}\mathbf{x})]$
DCT-IV	$\mathbf{C}_n^{IV}\mathbf{x} = Re[e^{-\pi i/4n}\mathbf{\Delta}\hat{\mathbf{F}}_{2n}\mathbf{\Delta}\mathbf{x}]$ $= Re[e^{-\pi i/4n}\mathbf{\Delta}\mathbf{F}_n\mathbf{\Delta}^2\mathbf{L}_n\mathbf{x}]$
DST-IV	$\mathbf{S}_n^{IV}\mathbf{x} = -Im[e^{-\pi i/4n}\mathbf{\Delta}\hat{\mathbf{F}}_{2n}\mathbf{\Delta}\mathbf{x}]$ $= -Im[e^{-\pi i/4n}\mathbf{\Delta}\mathbf{F}_n\mathbf{\Delta}^2\mathbf{K}_n\mathbf{x}]$

Algorithm 6.3.2 describes the procedure for obtaining the type III and type IV sine and cosine transforms simultaneously.

Algorithm 6.3.2 Procedure Type34:

```
    $\mathbf{v} \leftarrow \mathbf{K}_n\mathbf{x}$; $\mathbf{z} \leftarrow \mathbf{L}_n\mathbf{x}$
    $\mathbf{v} \leftarrow \mathbf{\Delta}^2\mathbf{v}$; $\mathbf{z} \leftarrow \mathbf{\Delta}^2\mathbf{z}$
    $\mathbf{v} \leftarrow \mathbf{F}_n\mathbf{v}$; $\mathbf{z} \leftarrow \mathbf{F}_n\mathbf{z}$
    $\mathbf{v} \leftarrow e^{-\pi i/4n}\mathbf{\Delta v}$; $\mathbf{z} \leftarrow e^{-\pi i/4n}\mathbf{\Delta z}$
    $\mathbf{C}_n^{IV}\mathbf{x} = Re(\mathbf{z})$; $\mathbf{S}_n^{IV}\mathbf{x} = -Im(\mathbf{v})$
    $Re(\mathbf{a}) = Re(\mathbf{z})$; $Im(\mathbf{a}) = Im(\mathbf{v})$
    $\mathbf{a} \leftarrow e^{\pi i/4n}\mathbf{\Delta}^{-1}\mathbf{a}$
    $\mathbf{C}_n^{III}\mathbf{x} = Re(\mathbf{a})$; $\mathbf{S}_n^{III}\mathbf{x} = -Im(\mathbf{a})$
```

We next describe how Algorithm 6.3.1 is derived. The derivation of Algorithm 6.3.2 is similar, but more complicated and is hence relegated to the Appendix.

Procedure 1: $\mathbf{C}_n^I$, $\mathbf{S}_n^I$, $\mathbf{C}_n^{II}$, and $\mathbf{S}_n^{II}$ from two real FFTs of length n

The method of computing the type II cosine transform $\mathbf{C}_n^{II}\mathbf{x}$ by equating

$$\mathbf{C}_n^{II}\mathbf{x} = Re[\mathbf{\Delta F}_n\mathbf{K}_n\mathbf{x}]$$

is derived and demonstrated by Makhoul (1980).

Theorem 6.3.1 (Makhoul (1980)) *Let* $\mathbf{x}$ *be a real vector of length* n*, then*

$$\mathbf{C}_n^{II}\mathbf{x} = Re[\mathbf{\Delta}\hat{\mathbf{F}}_{2n}\mathbf{x}] = Re[\mathbf{\Delta F}_n\mathbf{K}_n\mathbf{x}]$$

Proof Makhoul (1980).

——*——

The type II DST can be computed in a similar manner to that of the type II DCT. We first present a lemma equating the imaginary part of the $2n$-point DFT matrix multiplied by diagonal matrix of $4n$ roots of unity with the type II sine transform matrix.

Lemma 6.3.1

$$\mathbf{\Delta}\hat{\mathbf{F}}_{2n} - \mathbf{\Delta}^{-1}\overline{\hat{\mathbf{F}}}_{2n} = -i\mathbf{S}_n^{II}$$

Proof

$$\begin{aligned}[\boldsymbol{\Delta}\hat{\mathbf{F}}_{2n} - \boldsymbol{\Delta}^{-1}\overline{\hat{\mathbf{F}}}_{2n}]_{pq} &= e^{-\pi i p(2q+1)/2n} - e^{\pi i p(2q+1)/2n} \\ &= -2i\sin(\pi p(2q+1)/2n)\end{aligned}$$

——*——

The following theorem shows that the type II sine transform has two expressions as imaginary parts the complex sequence $-\boldsymbol{\Delta}\hat{\mathbf{F}}_{2n}\mathbf{x}$ and $-\boldsymbol{\Delta}\mathbf{F}_n\mathbf{L}_n\mathbf{x}$, respectively.

Theorem 6.3.2 *Let* $\mathbf{x}$ *be a real vector of length* n *then*

(6.3-1) $$\mathbf{S}_n^{II}\mathbf{x} = -\mathrm{Im}[\boldsymbol{\Delta}\hat{\mathbf{F}}_{2n}\mathbf{x}] = -Im[\boldsymbol{\Delta}\mathbf{F}_n\mathbf{L}_n\mathbf{x}]$$

Proof Let $\mathbf{y}$ be the quarter-odd extension of $\mathbf{x}$

$$\mathbf{y} = \begin{bmatrix} \mathbf{x} \\ -\mathbf{E}\mathbf{x} \end{bmatrix}$$

Partition the $2n$-point DFT matrix $\mathbf{F}_{2n}$ as follows:

$$\mathbf{F}_{2n} = [\mathbf{F}^L, \mathbf{F}^R]$$

Then letting $\mathbf{Y}$ be the DFT of $\mathbf{y}$ we have

$$\mathbf{Y} = \mathbf{F}_{2n}\mathbf{y} = \mathbf{F}^L\mathbf{x} - \mathbf{F}^R\mathbf{E}\mathbf{x}$$

However it can be easily shown that $\mathbf{F}^R\mathbf{E} = \boldsymbol{\Delta}^{-2}\overline{\mathbf{F}}^L$ so

$$\mathbf{Y} = \mathbf{F}^L\mathbf{x} - \boldsymbol{\Delta}^{-2}\overline{\mathbf{F}}^L\mathbf{x}$$

Defining $\hat{\mathbf{F}}_{2n} = \mathbf{F}_{2n}(0:n-1, 0:n-1)$ we have for the first n terms of $\mathbf{Y}$,

$$\begin{aligned}\mathbf{Y}(0:n-1) &= \hat{\mathbf{F}}_{2n}\mathbf{x} - \boldsymbol{\Delta}^{-2}\hat{\mathbf{F}}_{2n}\mathbf{x} \\ &= \boldsymbol{\Delta}^{-1}[\boldsymbol{\Delta}\hat{\mathbf{F}}_{2n} - \boldsymbol{\Delta}^{-1}\overline{\hat{\mathbf{F}}}_{2n}]\mathbf{x} \\ &= \boldsymbol{\Delta}^{-1}2iIm[\boldsymbol{\Delta}\hat{\mathbf{F}}_{2n}\mathbf{x}]\end{aligned}$$

At the same time we have

$$\boldsymbol{\Delta}\hat{\mathbf{F}}_{2n} - \boldsymbol{\Delta}^{-1}\overline{\hat{\mathbf{F}}}_{2n} = -i\mathbf{S}_n^{II}$$

hence

$$\mathbf{Y}(0:n-1) = -\boldsymbol{\Delta}^{-1}2i\mathbf{S}_n^{II}\mathbf{x}.$$

Thus

$$\mathbf{S}_n^{II}\mathbf{x} = -Im[\mathbf{\Delta}\widehat{\mathbf{F}}_{2n}\mathbf{x}]$$

and the first part of the proof is complete. The second part of the proof is to establish the relationship

$$\mathbf{S}_n^{II}\mathbf{x} = -Im[\mathbf{\Delta}\mathbf{F}_n(\mathbf{L}_n\mathbf{x})]$$

As above, we let $\mathbf{y}$ be the quarter-even extension of $\mathbf{x}$, and let

$$\begin{aligned} \mathbf{z}(j) &= \mathbf{y}(2j) \\ \mathbf{w}(j) &= \mathbf{y}(2j+1) \\ & \quad j = 0,\ldots,n-1 \end{aligned}$$

and note that $\mathbf{z} = \mathbf{L}_n\mathbf{x}$ and $\mathbf{w} = -\mathbf{E}\mathbf{z}$. Again let $\mathbf{Y} = \mathbf{F}_{2n}\mathbf{y}$. Denoting

$$\begin{aligned} \mathbf{F}_{2n}^e &= \mathbf{F}_{2n}(:,0:2:2n-1) \\ \mathbf{F}_{2n}^o &= \mathbf{F}_{2n}(:,1:2:2n-1) \end{aligned}$$

be the $2n$-by-n matrices containing the even and odd columns of $\mathbf{F}_{2n}$, respectively, we have

$$\begin{aligned} \mathbf{Y} = \mathbf{F}_{2n}\mathbf{y} &= \mathbf{F}_{2n}^e\mathbf{z} + \mathbf{F}_{2n}^o\mathbf{w} \\ &= \mathbf{F}_{2n}^e\mathbf{z} - \mathbf{F}_{2n}^o\mathbf{E}\mathbf{z} \end{aligned}$$

Denoting the top half or $\mathbf{F}_{2n}^e$ and $\mathbf{F}_{2n}^o$, respectively:

$$\begin{aligned} \widehat{\mathbf{F}}_{2n}^e &= \mathbf{F}_{2n}^e(0:n-1,:) \\ \widehat{\mathbf{F}}_{2n}^o &= \mathbf{F}_{2n}^o(0:n-1,:) \end{aligned}$$

then

$$\begin{aligned} \mathbf{Y}(0:n-1) &= \widehat{\mathbf{F}}_{2n}^e\mathbf{z} - \widehat{\mathbf{F}}_{2n}^o\mathbf{E}\mathbf{z} \\ &= \mathbf{F}_n\mathbf{z} - \mathbf{\Delta}^{-2}\overline{\mathbf{F}}_n\mathbf{z} \\ &= \mathbf{\Delta}^{-1}[\mathbf{\Delta}\mathbf{F}_n - \mathbf{\Delta}^{-1}\overline{\mathbf{F}}_n]\mathbf{z} \\ &= \mathbf{\Delta}^{-1}2iIm[\mathbf{\Delta}\mathbf{F}_n\mathbf{z}] \end{aligned}$$

we already know that

$$\mathbf{Y}(0:n-1) = -\mathbf{\Delta}^{-1}2i\mathbf{S}_n^{II}\mathbf{x}$$

hence this establishes the relationship

$$\mathbf{S}_n^{II}\mathbf{x} = -Im[\mathbf{\Delta}\mathbf{F}_n\mathbf{L}_n\mathbf{x}]$$

——*——

This shows that the type II sine transform can also be found by an FFT of a permuted real sequence $\mathbf{z}$ of length n.

Expressions involving $\widehat{\mathbf{F}}_{2n}\mathbf{x}$ can also be found for the type I cosine and sine transforms.

Theorem 6.3.3 *If* $\mathbf{x}$ *is a sequence of length* n*, then*

$$\mathbf{C}_n^I\mathbf{x} = Re[\widehat{\mathbf{F}}_{2n}\mathbf{x}]$$

Proof The even extension of $\mathbf{x}$ is

$$\mathbf{y} = \begin{bmatrix} \mathbf{x} \\ \mathbf{Tx} \end{bmatrix}$$

Let $\mathbf{Y} = \mathbf{F}_{2n}\mathbf{y}$, and $\mathbf{F}_{2n} = [\mathbf{F}^L, \mathbf{F}^R]$, then

$$\begin{aligned} \mathbf{F}_{2n}\mathbf{y} &= \mathbf{F}^L\mathbf{x} + \mathbf{F}^R\mathbf{Tx} \\ &= (\mathbf{F}^L + \overline{\mathbf{F}}^L)\mathbf{x} \\ &= 2Re[\mathbf{F}^L\mathbf{x}] \end{aligned}$$

But $\mathbf{C}_n^I\mathbf{x} = \frac{1}{2}[(\mathbf{F}_{2n}\mathbf{y})(0 : n-1)]$, hence the result

$$\mathbf{C}_n^I\mathbf{x} = Re[\widehat{\mathbf{F}}_{2n}\mathbf{x}]$$

——*——

Theorem 6.3.4 *If* $\mathbf{x}$ *is a sequence of length* n*, then*

$$\mathbf{S}_n^I\mathbf{x} = -\mathrm{Im}[\widehat{\mathbf{F}}_{2n}\mathbf{x}]$$

Proof The odd extension of $\mathbf{x}$ is

$$\mathbf{y} = \begin{bmatrix} \mathbf{x} \\ -\mathbf{Tx} \end{bmatrix}$$

Let $\mathbf{Y} = \mathbf{F}_{2n}\mathbf{y}$, and $\mathbf{F}_{2n} = [\mathbf{F}^L, \mathbf{F}^R]$, then

$$\begin{aligned} \mathbf{F}_{2n}\mathbf{y} &= \mathbf{F}^L\mathbf{x} - \mathbf{F}^R\mathbf{Tx} \\ &= (\mathbf{F}^L - \overline{\mathbf{F}}^L)\mathbf{x} \\ &= 2i\mathrm{Im}[\mathbf{F}^L\mathbf{x}] \end{aligned}$$

But $\mathbf{S}_n^I\mathbf{x} = \frac{i}{2}(\mathbf{F}_{2n}\mathbf{y})(0 : n-1)$, hence the result

$$\mathbf{S}_n^I\mathbf{x} = -\mathrm{Im}[\widehat{\mathbf{F}}_{2n}\mathbf{x}]$$

——*——

Now that all the relevant relationships are present:

$$\mathbf{C}_n^{II}\mathbf{x} = Re[\mathbf{\Delta}\widehat{\mathbf{F}}_{2n}\mathbf{x}] = Re[\mathbf{\Delta}\mathbf{F}_n(\mathbf{K}_n\mathbf{x})]$$

$$\mathbf{S}_n^{II}\mathbf{x} = -Im[\mathbf{\Delta}\widehat{\mathbf{F}}_{2n}\mathbf{x}] = -Im[\mathbf{\Delta}\mathbf{F}_n(\mathbf{L}_n\mathbf{x})]$$

$$\mathbf{C}_n^{I}\mathbf{x} = Re[\widehat{\mathbf{F}}_{2n}\mathbf{x}]$$

$$\mathbf{S}_n^{I}\mathbf{x} = -Im[\widehat{\mathbf{F}}_{2n}\mathbf{x}]$$

we can produce the procedure that computes the four type I and II symmetric transforms.

Let

$$\mathbf{A} = \mathbf{\Delta}\widehat{\mathbf{F}}_{2n}\mathbf{x}$$

Then

$$\begin{aligned} \mathrm{Re}\mathbf{A} &= \mathrm{Re}\mathbf{\Delta}\mathbf{F}_n\mathbf{v} \\ \mathrm{Im}\mathbf{A} &= \mathrm{Im}\mathbf{\Delta}\mathbf{F}_n\mathbf{z} \end{aligned}$$

and

$$\mathbf{B} = \mathbf{\Delta}^{-1}\mathbf{A}$$

gives

$$\begin{aligned} \mathbf{C}_n^{I}\mathbf{x} &= \mathrm{Re}\mathbf{B} \\ \mathbf{S}_n^{I}\mathbf{x} &= -\mathrm{Im}\mathbf{B} \end{aligned}$$

Thus we need to find two real DFTs of length n. It is a well-known fact that one can find these two DFTs in one pass through the complex FFT subroutine. Thus by computing only one complex FFT of length n, we have managed to find four sine and cosine transforms of length n.

Procedure 2: $\mathbf{C}_n^{III}$, $\mathbf{S}_n^{III}$, $\mathbf{C}_n^{IV}$, and $\mathbf{S}_n^{IV}$ from two complex FFTs of length n.

The type III and type IV symmetric transforms have a very similar relationship whereby the type III transforms can be computed in linear time once the type IV transforms are present. The type IV transforms can be found efficiently by taking the complex FFT of vectors of length n, i.e. a scaled version of either $\mathbf{K}_n\mathbf{x}$ or $\mathbf{L}_n\mathbf{x}$.

The derivation and proof is long-winded and is thus relegated to the appendix. The important results are theorems 6.8.1 and 6.8.2 giving:

$$\begin{aligned} \mathbf{C}_n^{IV}\mathbf{x} &= Re[e^{-\pi i/4n}\mathbf{\Delta}\widehat{\mathbf{F}}_{2n}\mathbf{\Delta}\mathbf{x}] \\ &= Re[e^{-\pi i/4n}\mathbf{\Delta}\mathbf{F}_n\mathbf{\Delta}^2\mathbf{L}_n\mathbf{x}] \end{aligned}$$

and theorems 6.8.3 and 6.8.4 giving:

$$\begin{aligned} \mathbf{S}_n^{IV}\mathbf{x} &= -Im[e^{-\pi i/4n}\mathbf{\Delta}\widehat{\mathbf{F}}_{2n}\mathbf{\Delta}\mathbf{x}] \\ &= -Im[e^{-\pi i/4n}\mathbf{\Delta}\mathbf{F}_n\mathbf{\Delta}^2\mathbf{K}_n\mathbf{x}] \end{aligned}$$

Also neeed are the relationships

$$\mathbf{C}_n^{III}\mathbf{x} = Re[\widehat{\mathbf{F}}_{2n}\mathbf{\Delta}\mathbf{x}]$$

and

$$\mathbf{S}_n^{III}\mathbf{x} = -Im[\widehat{\mathbf{F}}_{2n}\mathbf{\Delta}\mathbf{x}]$$

Now Algorithm 6.3.2 is derived as follows. Let

$$\mathbf{A} = e^{-\pi i/4n}\mathbf{\Delta}_{4n}\widehat{\mathbf{F}}_{2n}\mathbf{\Delta}_{4n}\mathbf{x}$$

Then

$$\begin{aligned} \mathrm{Re}[\mathbf{A}] &= \mathrm{Re}[e^{-\pi i/4n}\mathbf{\Delta}\mathbf{F}_n\mathbf{\Delta}^2\mathbf{z}] \\ \mathrm{Im}[\mathbf{A}] &= \mathrm{Im}[e^{-\pi i/4n}\mathbf{\Delta}\mathbf{F}_n\mathbf{\Delta}^2\mathbf{v}] \end{aligned}$$

so letting

$$\mathbf{B} = e^{\pi i/4n}\mathbf{\Delta}^{-1}\mathbf{A}$$

gives

$$\begin{aligned} \mathbf{C}_n^{III}\mathbf{x} &= \mathrm{Re}[\mathbf{B}] \\ \mathbf{S}_n^{III}\mathbf{x} &= \mathrm{Im}[\mathbf{B}] \end{aligned}$$

Therefore we need to find two complex DFT's of length n

$$\mathbf{F}_n(\mathbf{\Delta}^2\mathbf{L}_n\mathbf{x}) \quad \mathbf{F}_n(\mathbf{\Delta}^2\mathbf{K}_n\mathbf{x})$$

to get the four sine and cosine transforms of length n.

How to get the Transforms

In the **procedures** above we have always assumed that $\mathbf{x}$ was a vector of length n, with subscripts ranging from 0 to $n-1$. However, the exact definitions of some of these transforms are slightly different. For example, DST-III assumes that the subscripts run from 1 to n. Furthermore there is that constant k_l to deal with. We now show how easy it is to get the exact definitions from the output of the **procedures**.

DCT-I of $\mathbf{x}(0:n)$

Given: $\mathbf{Y} = \mathbf{C}_n^I \mathbf{x}(0:n-1)$. Overwrite $\mathbf{Y}$ with the DCT-I of $\mathbf{x}$.

$$\begin{aligned}
\mathbf{Y}(0) &\leftarrow \sqrt{\frac{2}{n}}\left(\sqrt{\frac{1}{2}}(\mathbf{Y}(0)-\mathbf{x}_0)+\frac{1}{2}\mathbf{x}_0+\frac{1}{2}\mathbf{x}_n\right)\\
\mathbf{Y}(k) &\leftarrow \sqrt{\frac{2}{n}}\left(\mathbf{Y}(k)-(1-\sqrt{\frac{1}{2}})\mathbf{x}_0+(-1)^k\sqrt{\frac{1}{2}}\mathbf{x}_n\right)\\
&\quad 1 \le k \le n-1\\
\mathbf{Y}(n) &\leftarrow \sqrt{\frac{2}{n}}\left(\frac{1}{2}\mathbf{x}_0+\sum_{j=1}^{n-1}(-1)^j\sqrt{\frac{1}{2}}\mathbf{x}_j+\frac{1}{2}(-1)^n\mathbf{x}_n\right)
\end{aligned}$$

DST-I of $\mathbf{x}(1:n-1)$

Given: $\mathbf{Y} = \mathbf{S}_n^I \mathbf{x}$. Overwrite $\mathbf{Y}$ with the DST-I of $\mathbf{x}$.

$$\begin{aligned}
\mathbf{Y}(k) &\leftarrow \sqrt{\frac{2}{n}}(\mathbf{Y}(k))\\
&\quad 1 \le k \le n-1
\end{aligned}$$

DCT-II of $\mathbf{x}$

Given: $\mathbf{Y} = \mathbf{C}_n^{II} \mathbf{x}$. Overwrite $\mathbf{Y}$ with the DCT-II of $\mathbf{x}$.

$$\mathbf{Y}(0) \leftarrow \sqrt{\frac{2}{n}}\left(\sqrt{\frac{1}{n}}\mathbf{Y}(0)\right)$$

$$\begin{aligned} \mathbf{Y}(k) &\leftarrow \sqrt{\frac{2}{n}}\mathbf{Y}(k) \\ & \quad 1 \le k \le n-1 \end{aligned}$$

DST-II of $\mathbf{x}(1:n)$

Given: $\mathbf{Y} = \mathbf{S}_n^{II}\mathbf{x}$. Overwrite $\mathbf{Y}$ with the DST-II of $\mathbf{x}$.

$$\begin{aligned} \mathbf{Y}(1) &\leftarrow \sqrt{\frac{2}{n}}\left(\sqrt{\frac{1}{2}}\mathbf{Y}(1) - \mathbf{x}_0\right) \\ \mathbf{Y}(k) &\leftarrow \sqrt{\frac{2}{n}}(\mathbf{Y}(k) - \mathbf{x}_0) \\ & \quad 1 \le k \le n-1 \\ \mathbf{Y}(n) &\leftarrow \sqrt{\frac{2}{n}}\left(\sqrt{\frac{1}{2}}\sum_{j=1}^{n}(-1)^{j-1}\mathbf{x}_j\right) \end{aligned}$$

DCT-III of x

Given: $\mathbf{Y} = \mathbf{C}_n^{III}\mathbf{x}$. Overwrite $\mathbf{Y}$ with the DCT-III of $\mathbf{x}$.

$$\begin{aligned} \mathbf{Y}(k) &\leftarrow \sqrt{\frac{2}{n}}\left(\mathbf{Y}(k) - \left(1 - \sqrt{\frac{1}{2}}\right)\mathbf{x}_0\right) \\ & \quad 0 \le k \le n-1 \end{aligned}$$

DST-III of $\mathbf{x}(1:n)$

Given: $\mathbf{Y} = \mathbf{S}_n^{III}\mathbf{x}$. Overwrite $\mathbf{Y}$ with the DST-III of $\mathbf{x}$.

$$\begin{aligned} \mathbf{Y}(k) &\leftarrow \sqrt{\frac{2}{n}}\left(\mathbf{Y}(k) - \mathbf{x}_0 + \sqrt{\frac{1}{2}}(-1)^{k-1}\mathbf{x}_n\right) \\ & \quad 1 \le k \le n-1 \\ \mathbf{Y}(n) &\leftarrow \sqrt{\frac{2}{n}}\left(\sum_{j=1}^{n-1}(-1)^{j-1}\mathbf{x}_j + \sqrt{\frac{1}{2}}(-1)^{n-1}\mathbf{x}_n\right) \end{aligned}$$

DCT-IV of x

Given: $\mathbf{Y} = \mathbf{C}_n^{IV}\mathbf{x}$. Overwrite $\mathbf{Y}$ with the DCT-IV of $\mathbf{x}$.

$$\mathbf{Y}(k) \leftarrow \sqrt{\frac{2}{n}}\mathbf{Y}(k)$$
$$0 \le k \le n-1$$

DST-IV of x

Given: $\mathbf{Y} = \mathbf{S}_n^{IV}\mathbf{x}$. Overwrite $\mathbf{Y}$ with the DST-IV of $\mathbf{x}$.

$$\mathbf{Y}(k) \leftarrow \sqrt{\frac{2}{n}}\mathbf{Y}(k)$$
$$0 \le k \le n-1$$

6.4 Error Analysis

Our method is much more stable than that of the traditional methods, including the Cooley *et al.* (1970) algorithm. A quick error analysis of the Cooley *et al.* algorithm follows.

Cooley et al. Sine Transform

(a) Given values $\mathbf{Y}(j)$, $j = 1, \ldots, n-1$ of a real odd sequence $\mathbf{Y}(j)$, $j = 0, 1, \ldots, 2n-1$, form the complex array $\mathbf{X}(j)$:

$$\mathbf{X}(j) = -[\mathbf{Y}(2j+1) - \mathbf{Y}(2j-1)] + \mathbf{Y}(2j)i$$

(b) Find the DFT $\mathbf{A}(k) = \frac{1}{n}F_n\mathbf{X}(j)$. Since $\mathbf{X}(j)$ was constructed to be conjugate even, $\mathbf{A}(k)$ is real.

(c) Compute

$$\mathbf{b}(j) = \frac{1}{2}\left\{[A(k) - A(-k)] - \frac{1}{2\sin(\pi k/n)}[A(k) + A(-k)]\right\}$$
$$k = 1, 2, \ldots, n-1$$

There are two sources of trouble (assuming that $\mathbf{A}(k)$ was computed accurately):

- the formation of $\mathbf{X}(j)$, whose real part can have cancellation errors if the $\mathbf{Y}(j)$'s are close together.
- the divide by $\sin(\pi k/n)$ when n is large.

When n is large, $\sin(\pi k/n) \approx \frac{n\pi}{k}$. To isolate the error from this step only, we assume that all the other quantities were computed correctly. Let $\hat{\mathbf{b}}$ be the computed solution and $\mathbf{b}$ be the true solution. Then

$$\|\hat{\mathbf{b}} - \mathbf{b}\|_\infty \leq \frac{n}{\pi}\mathbf{u} \rightarrow \infty \quad n \rightarrow \infty$$

where $\mathbf{u}$ is machine precision.

Type I Sine Transform (Chu)

The error analysis for the method of this chapter is fairly straight-forward once we have the error analysis of the complex n-point FFT (See Chapter 2).

$$\mathbf{S}_n^I\mathbf{x} = -Im[\mathbf{\Delta}^{-1}(\mathbf{C}_n^{II}\mathbf{x} + i\mathbf{S}_n^{II}\mathbf{x})]$$

Therefore we first present the error analyses of $\mathbf{C}_n^{II}\mathbf{x}$ and $\mathbf{S}_n^{II}\mathbf{x}$. These two error analyses are equivalent as the transforms are just the real and imaginary parts of the complex sequences

$$\mathbf{\Delta F}_n\mathbf{v} \text{ and } \mathbf{\Delta F}_n\mathbf{z}$$

respectively. ($\mathbf{v} = \mathbf{K}_n\mathbf{x}$ and $\mathbf{z} = \mathbf{L}_n\mathbf{x}$)

Let $\mathbf{y} = \mathbf{F}_n\mathbf{v}$ such that the computed result $\hat{\mathbf{y}}$ is obtained from a complex FFT subroutine. Then from Chapter 2, we have

$$\|\hat{\mathbf{y}} - \mathbf{y}\|_2 \leq c\log_2 n\mathbf{u}\|\mathbf{y}\|_2 + O(\mathbf{u}^2)$$

Lemma 6.4.1 *If $y = \mathbf{F}_n\mathbf{v}$ is computed by the complex FFT such that*

$$\|\hat{\mathbf{y}} - \mathbf{y}\|_2 \leq c\log_2 n\mathbf{u}\|\mathbf{y}\|_2 + O(\mathbf{u}^2)$$

and $\mathbf{\Delta} = diag(e^{-2j\pi i/4n})$, $j = 0, 1, \ldots, n-1$, such that

$$\|\widehat{\mathbf{\Delta}} - \mathbf{\Delta}\|_2 \leq \mathbf{u} + O(\mathbf{u}^2)$$

then

$$\|fl(\widehat{\mathbf{\Delta}}\hat{\mathbf{y}}) - \mathbf{\Delta y}\|_2 \leq [(2+\sqrt{2}) + c\log_2 n]\mathbf{u}\|\mathbf{y}\|_2 + O(\mathbf{u}^2)$$

where c is a constant of order unity and $\mathbf{u}$ is the machine precision.

Proof

$$\begin{aligned} \|fl(\widehat{\mathbf{\Delta}}\hat{\mathbf{y}}) - \mathbf{\Delta}\mathbf{y}\|_2 &\le \|fl(\widehat{\mathbf{\Delta}}\hat{\mathbf{y}}) - \widehat{\mathbf{\Delta}}\hat{\mathbf{y}}\|_2 + \|\widehat{\mathbf{\Delta}}\hat{\mathbf{y}} - \mathbf{\Delta}\mathbf{y}\|_2 \\ &\le (1+\sqrt{2})\|\mathbf{\Delta}\mathbf{y}\|_2\mathbf{u} + \|\widehat{\mathbf{\Delta}} - \mathbf{\Delta}\|_2\|\mathbf{y}\|_2 + \|\widehat{\mathbf{\Delta}}\|_2\|\hat{\mathbf{y}} - \mathbf{y}\|_2 \\ &\le (2+\sqrt{2})\|\mathbf{y}\|_2 + c\log_2 n\mathbf{u}\|\mathbf{y}\|_2 \end{aligned}$$

Using lemma 2.5.3 and the result from the radix-2 complex FFT error analysis.

—*—

Corollary 6.4.1 *Let $\hat{\mathbf{z}}$ be the computed version of a complex vector $\mathbf{z}$ such that*

$$\|\hat{\mathbf{z}} - \mathbf{z}\|_2 \le \delta(\mathbf{z})\mathbf{u} + O(\mathbf{u}^2)$$

and $\widehat{\mathbf{\Delta}}$ be a matrix of "twiddle" factors $e^{-2\pi i/m}$, m is any integer such that

$$\|\widehat{\mathbf{\Delta}} - \mathbf{\Delta}\|_2 \le \mathbf{u} + O(\mathbf{u}^2)$$

then the computed product of $\mathbf{\Delta}$ and $\mathbf{z}$ is such that

$$\|fl(\widehat{\mathbf{\Delta}}\hat{\mathbf{z}}) - \mathbf{\Delta}\mathbf{z}\|_2 \le [(2+\sqrt{2}) + \delta\mathbf{z}]\mathbf{u}\|\mathbf{z}\|_2$$

Using this corollary we see that for $\mathbf{b} = \mathbf{S}_n^I\mathbf{x}$, the computed sine transform $\hat{\mathbf{b}}$ is such that

$$\|\hat{\mathbf{b}} - \mathbf{b}\| \le [2(2+\sqrt{2}) + c\log_2 n]\mathbf{u}\|\mathbf{b}\|_2 + O(\mathbf{u}^2)$$

Comparisons between the sine transform presented in this chapter and the Cooley *et al.* one implemented by Rabiner (1979a, 1979b) show that the error grows as n increases.

Three functions were used, $y = x$, $y = random(0,1)$, and $y = sin(x)$ for x in $[0,1]$. The base solution method entails taking the odd extension of y and transforming this $2n$-point sequence by a complex FFT algorithm. Thus any errors that show up would only be caused by the pre- and post- processing. The maximum component-wise error for the Cooley *et al.* implementation is shown in Table 6.2 and the error for the method of this chapter is shown in Table 6.3.

6.5 Implementation on the Hypercube

The hypercube implementation of this method is fairly obvious. Aside from the initial permutation of the data, everything else proceeds exactly as an n-point complex FFT. We use the Two-Track method of Chapter 3 to do this.

Table 6.2: Cooley *et al.* Sine Transform Errors

Cooley et al. Sine Transform			
$n = 2^d$	$y = x$	random	$y = \sin(x)$
$4 = 2^2$	2.98023e-08	5.96046e-08	2.98023e-08
$8 = 2^3$	5.96046e-08	5.96046e-08	2.98023e-08
$16 = 2^4$	1.19209e-07	1.19209e-07	1.19209e-07
$32 = 2^5$	1.19209e-07	1.19209e-07	1.19209e-07
$64 = 2^6$	4.17233e-07	2.38419e-07	2.98023e-07
$128 = 2^7$	4.17233e-07	2.38r19e-07	2.98023e-07
$256 = 2^8$	1.37091e-06	9.83477e-07	1.01328e-06
$512 = 2^9$	1.37091e-06	8.94070e-07	3.34465e-07
$1024 = 2^{10}$	5.42402e-06	3.24845e-06	3.66569e-06
$2048 = 2^{11}$	5.06639e-06	3.45707e-06	3.21865e-06
$4096 = 2^{12}$	2.13981e-05	1.54972e-05	1.50502e-05
$8192 = 2^{13}$	1.99080e-05	9.20892e-06	1.34706e-05
$16384 = 2^{14}$	8.55923e-05	6.00219e-05	6.03199e-05
$32768 = 2^{15}$	7.90358e-05	3.43919e-05	5.42402e-05

- Form $\mathbf{v} \leftarrow \mathbf{K}_n\mathbf{x}$ and $\mathbf{z} \leftarrow \mathbf{L}_n\mathbf{x}$. Data is entered into the hypercube as $\mathbf{M}_n\mathbf{x}$. This is okay because consecutive elements of $\mathbf{x}$ are still in the same processor. Apply the permutation

$$\begin{bmatrix} \mathbf{I} & \\ & \mathbf{E} \end{bmatrix}$$

meaning that the second track $\mathbf{x}^{(2)}$ is exchange permuted.

- Implement Two-Track FFT with BRGC mapping on $\mathbf{v}$ and $\mathbf{z}$ to get

$$\tilde{\mathbf{v}} \leftarrow (\mathbf{M}_{2P}^{(2)} \otimes \mathbf{I}_{n/2P})\mathbf{P}_n\mathbf{F}_n\mathbf{v}$$
$$\tilde{\mathbf{z}} \leftarrow (\mathbf{M}_{2P}^{(2)} \otimes \mathbf{I}_{n/2P})\mathbf{P}_n\mathbf{F}_n\mathbf{z}$$

Communication of the Two-Track FFT described in Chapter 3.

- Post-processing is multiplication by $\mathbf{\Delta}$ a diagonal matrix and other local operations.

Algorithm 6.5.1 Type 12 Transform

Table 6.3: Sine Transform (Type I) Errors

Type I Sine Transform			
$n = 2^d$	$y = x$	random	$y = \sin(x)$
$4 = 2^2$	1.49012e-08	1.49012e-08	5.96046e-08
$8 = 2^3$	5.96046e-08	5.96046e-08	2.98023e-08
$16 = 2^4$	5.96046e-08	5.96046e-08	2.00234e-08
$32 = 2^5$	2.98023e-08	5.96046e-08	5.96046e-08
$64 = 2^6$	2.98023e-08	4.47035e-08	5.96046e-08
$128 = 2^7$	1.19209e-07	5.02914e-08	2.98023e-08
$256 = 2^8$	1.78814e-07	1.86265e-08	1.19209e-07
$512 = 2^9$	5.96046e-08	2.98023e-08	5.96046e-08
$1024 = 2^{10}$	2.38419e-07	5.96046e-08	2.98023e-08
$2048 = 2^{11}$	1.78814e-07	5.96046e-08	5.96046e-08
$4096 = 2^{12}$	5.96046e-08	2.98023e-08	5.96046e-08
$8192 = 2^{13}$	5.96046e-08	2.98023e-08	5.96046e-08
$16384 = 2^{14}$	1.19209e-07	2.49129e-08	5.96046e-08
$32768 = 2^{15}$	5.96046e-08	2.21189e-08	5.96046e-08
$65536 = 2^{16}$	5.96046e-08	2.21189e-08	2.98023e-08

```
/* $\mu$ = processor id; $n = 2^t$; $P = 2^d$; $m = (n/P)/2$ */
/* Data is entered into the hypercube as $\mathbf{M}_n\mathbf{v}$ */
/* Initially, processor $\mu$ holds: */
/* $\mathbf{v}^{(1)} = \mathbf{v}(2m \cdot id[\mu] : 2 : 2m \cdot (id[\mu]+1) - 1)$ */
/* $\mathbf{v}^{(2)} = \mathbf{v}(2m \cdot id[\mu] + 1 : 2 : 2m \cdot (id[\mu]+1) - 1)$ */
  call Exchange($\mathbf{v}^{(2)}$) /* code (6.6) */
  $\mathbf{z}^{(1)} \leftarrow \mathbf{v}^{(1)}$
  $\mathbf{z}^{(2)} \leftarrow -\mathbf{v}^{(2)}$
  call Two-Track FFT($\mathbf{v}^{(1)}, \mathbf{v}^{(2)}, \mathbf{z}^{(1)}, \mathbf{z}^{(2)}$) /* code (6.2) */
  $\mathbf{v} \leftarrow \mathbf{\Delta v}$
  $\mathbf{z} \leftarrow \mathbf{\Delta z}$
```

Of course, for any hypercube procedure the facility to start with bit-reverse permuted data is necessary in order to inverse transform a sequence. Here we outline the reverse procedure. Starting with

$$(\mathbf{M}_{2P}^{(2)} \otimes \mathbf{I}_{n/2P})\mathbf{P}_n\mathbf{x}$$

- Apply the permutation $(\mathbf{\Pi}_{2P}^{(2)} \otimes \mathbf{I}_{n/2P})$ to get back just $\mathbf{P}_n\mathbf{x}$.

- Now we need to have $\mathbf{P}_n\mathbf{K}_n\mathbf{x}$ so that we can apply the inverse FFT to $\mathbf{P}_n\mathbf{v}$. But we have $\mathbf{P}_n\mathbf{x}$. So we apply

  $$\begin{bmatrix} \mathbf{I} & \\ & \mathbf{E} \end{bmatrix} \mathbf{P}_n\mathbf{x} = \mathbf{P}_n\mathbf{K}_n\mathbf{x}$$

  Need to do exchange permutation of second track.

- Do Two-Track FFT and find $\mathbf{F}_n\mathbf{P}_n(\mathbf{P}_n\mathbf{x})$.

- Multiply by the diagonal matrix $\mathbf{\Delta}$, and other local post-processing.

Timing results are given for Algorithm 6.3.1 in Table 6.4 for the forward transform with data in natural order and in Table 6.5 for the inverse transform with data in bit-reversed order. The hypercube is mapped by the BRGC mapping. The speedup here is excellent.

6.6 Applications: Updating Shifted Sequences

The stability of this method is good for updating shifted sequences as most of the error of the traditional methods is concentrated in the first few terms and last few terms.

Table 6.4: Algorithm Type12 $n = 16384$ Timings (Forward)

dim	total	comp	comm	blocked	copy	diag	per	speedup
0	95735	94780	0	0	950	24310	23935	–
1	49430	48620	335	50	480	12155	12360	1.94
2	25985	24935	810	175	235	6080	6495	3.69
3	13515	12790	600	265	120	3040	3380	7.08
4	7220	6560	600	290	60	1520	1805	13.26

Table 6.5: Algorithm Type12 $n = 16384$ Timings (Inverse)

dim	total	comp	comm	blocked	copy	diag	per	speedup
0	84445	75765	0	0	8665	11395	21110	–
1	43730	39125	440	65	4165	5705	10930	1.93
2	24250	20185	2060	495	2000	2855	6060	3.48
3	12600	10410	1220	530	1205	1425	3150	6.70
4	6720	5375	880	410	460	715	1680	12.57

Suppose we have sequence $\mathbf{x} = [x_0, \ldots, x_{n-1}]^T$ and we wish to find the sine and cosine transforms of its shift $\mathbf{x}_+ = [x_1, \ldots, x_n]^T$. First let us define the upshift matrix $\mathbf{R}_n^{-1} = [\mathbf{e}_{n-1}, \mathbf{e}_0, \mathbf{e}_1, \ldots, \mathbf{e}_{n-1}]$. Then

$$\mathbf{x}_+ = \mathbf{R}_n^{-1}\mathbf{x} - [0, \ldots, x_0]^T + [0, \ldots, x_n]^T$$

If we had $\mathbf{C}_n^I\mathbf{x}$ and $\mathbf{S}_n^I\mathbf{x}$, we can find $\mathbf{C}_n^I\mathbf{x}_+$ and $\mathbf{S}_n^I\mathbf{x}_+$ easily without having to do any more transforms. This is because

$$\begin{aligned} \mathbf{C}_n^I\mathbf{x}_+ &= \mathbf{C}_n^I(\mathbf{R}_n^{-1}\mathbf{x} - [0, \ldots, x_0]^T + [0, \ldots, x_n]^T) \\ \mathbf{S}_n^I\mathbf{x}_+ &= \mathbf{S}_n^I(\mathbf{R}_n^{-1}\mathbf{x} - [0, \ldots, x_0]^T + [0, \ldots, x_n]^T) \end{aligned}$$

Lemma 6.6.1

$$\begin{aligned} \mathbf{C}_n^I\mathbf{R}_n^{-1} &= \mathbf{\Delta}_c\mathbf{C}_n^I + \mathbf{\Delta}_s\mathbf{S}_n^I \\ \mathbf{S}_n^I\mathbf{R}_n^{-1} &= \mathbf{\Delta}_c\mathbf{S}_n^I - \mathbf{\Delta}_s\mathbf{C}_n^I \end{aligned}$$

letting $\mathbf{\Delta}_c = \mathit{diag}[\cos(\pi p/n)]$ *and* $\mathbf{\Delta}_s = \mathit{diag}[\sin(\pi p/n)]$.

Proof

$$[\mathbf{C}_n^I\mathbf{R}_n^{-1}]_{pq} =$$

$$\cos(\pi p(q-1)/n) = \cos(\pi pq/n)\cos(\pi p/n) + \sin(\pi pq/n)\sin(\pi p/n)$$

Table 6.6: Updating of Shifted Sequences

transform	update
DCT-I	$\mathbf{C}_n^I\mathbf{x}_+ = \mathbf{\Delta}_c\mathbf{C}_n^I\mathbf{x} + \mathbf{\Delta}_s\mathbf{S}_n^I\mathbf{x} + (x_n - x_0)\mathbf{C}_n^I(:, n-1)$
DST-I	$\mathbf{S}_n^I\mathbf{x}_+ = \mathbf{\Delta}_c\mathbf{S}_n^I\mathbf{x} - \mathbf{\Delta}_s\mathbf{C}_n^I\mathbf{x} + (x_n - x_0)\mathbf{S}_n^I(:, n-1)$
DCT-II	$\mathbf{C}_n^{II}\mathbf{x}_+ = \mathbf{\Delta}_c\mathbf{C}_n^{II}\mathbf{x} + \mathbf{\Delta}_s\mathbf{S}_n^{II}\mathbf{x} + (x_n - x_0)\mathbf{C}_n^{II}(:, n-1)$
DST-II	$\mathbf{S}_n^{II}\mathbf{x}_+ = \mathbf{\Delta}_c\mathbf{S}_n^{II}\mathbf{x} - \mathbf{\Delta}_s\mathbf{C}_n^{II}\mathbf{x} + (x_n - x_0)\mathbf{S}_n^{II}(:, n-1)$
DCT-III	$\mathbf{C}_n^{III}\mathbf{x}_+ = \mathbf{\Delta}_c\mathbf{C}_n^{III}\mathbf{x} + \mathbf{\Delta}_s\mathbf{S}_n^{III}\mathbf{x} + (x_n - x_0)\mathbf{C}_n^{III}(:, n-1)$
DST-III	$\mathbf{S}_n^{III}\mathbf{x}_+ = \mathbf{\Delta}_c\mathbf{S}_n^{III}\mathbf{x} - \mathbf{\Delta}_s\mathbf{C}_n^{III}\mathbf{x} + (x_n - x_0)\mathbf{S}_n^{III}(:, n-1)$
DCT-IV	$\mathbf{C}_n^{IV}\mathbf{x}_+ = \mathbf{\Delta}_c\mathbf{C}_n^{IV}\mathbf{x} + \mathbf{\Delta}_s\mathbf{S}_n^{IV}\mathbf{x} + (x_n - x_0)\mathbf{C}_n^{IV}(:, n-1)$
DST-IV	$\mathbf{S}_n^{IV}\mathbf{x}_+ = \mathbf{\Delta}_c\mathbf{S}_n^{IV}\mathbf{x} - \mathbf{\Delta}_s\mathbf{C}_n^{IV}\mathbf{x} + (x_n - x_0)\mathbf{S}_n^{IV}(:, n-1)$

and

$$[\mathbf{S}_n^I\mathbf{R}_n^{-1}]_{pq} =$$
$$\sin(\pi p(q-1)/n) = \sin(\pi pq/n)\cos(\pi p/n) - \cos(\pi pq/n)\sin(\pi p/n)$$

——*——

Next notice that

$$\mathbf{A}[0, \ldots, b]^T = b\mathbf{A}(:, n-1)$$

that is, equal to b times the last column of $\mathbf{A}$.

Therefore updating shifted sequences involves some scaling only. Table 6.6 summarizes the updating procedure. See also Yip and Rao (1987).

6.7 Applications: Fast Poisson Solvers.

Following Van Loan (1987) and others, the eigenvector matrix of the discrete finite difference operator of Poisson's equation with various boundary conditions can be written in terms of DCT or DST matrices. Here, types I–III are involved.

Dirichlet-Dirichlet

$$\mathbf{Q}_{DD}(n-1) = \frac{2}{n}[\sin(pq\pi/n)], \quad p, q = 1, \ldots, n-1.$$

$$\mathbf{Q}_{DD}^{-1} = [\sin(pq\pi/n)], \quad p, q = 1, \ldots, n-1.$$

Neumann-Neumann

$$\mathbf{Q}_{NN}(n+1) = \frac{2}{n}[k_q \cos(pq\pi/n)] \quad p,q = 0,1,\ldots,n.$$

$$\mathbf{Q}_{NN}^{-1} = [k_q \cos(pq\pi/n)] \quad p,q = 0,1,\ldots,n.$$

Dirichlet-Neumann

$$\mathbf{Q}_{DN}(n) = \frac{2}{n}[k_q \sin((p-\frac{1}{2})q\pi/n)] \quad p,q = 1,\ldots,n$$

and

$$\mathbf{Q}_{DN}^{-1} = [\sin(p(q-\frac{1}{2})\pi/n)] \quad p,q = 1,\ldots,n.$$

Neumann-Dirichlet

$$\mathbf{Q}_{ND}(n) = \frac{2}{n}[k_q \cos((p+\frac{1}{2})q\pi/n)] \quad p,q = 0,1,\ldots,n-1.$$

$$\mathbf{Q}_{ND}^{-1} = [\cos(p(q+\frac{1}{2})\pi/n)] \quad p,q = 0,1,\ldots,n-1.$$

where

$$k_l = \begin{cases} 1 & \text{if } l \neq 0 \text{ or } n \\ \frac{1}{2} & \text{if } l = 0 \text{ or } n. \end{cases}$$

The stability of this method is not an issue until $h^3 < \epsilon$ since the discretization error of finite differencing is already h^2, and the error growth rate of the Cooley *et al.* algorithm is ϵ/h.

6.8 Appendix: Proof for Type III and Type IV Symmetric Transforms

We derive a procedure that allows us to find the type III and type IV sine and cosine transforms all at the same time. Since the symmetry is a bit more complicated, this requires two complex FFT's of length n instead of one.

More notation is needed:

$$\begin{aligned}
\mathbf{F}^1_{4n} &= \mathbf{F}_{4n}(:, 0 : n-1) \\
\mathbf{F}^2_{4n} &= \mathbf{F}_{4n}(:, n : 2n-1) \\
\mathbf{F}^3_{4n} &= \mathbf{F}_{4n}(:, 2n : 3n-1) \\
\mathbf{F}^4_{4n} &= \mathbf{F}_{4n}(:, 3n : 4n-1) \\
\mathbf{F}_{4n} &= [\mathbf{F}^1, \mathbf{F}^2, \mathbf{F}^3, \mathbf{F}^4] \\
\mathbf{F}^{o1}_{4n} &= \mathbf{F}_{4n}(1 : 2 : 2n-1, 0 : n-1) \\
\mathbf{F}^{o1}_{2n} &= \mathbf{F}_{2n}(1 : 2 : 2n-1, 0 : n-1) \\
\mathbf{D} &= diag(1, -1, \ldots) \\
\mathbf{\Delta} &= diag(e^{-2\pi ik/4n})
\end{aligned}$$

The following lemmas are used in the main result.

Lemma 6.8.1 *Let* $\mathbf{D} = diag(1, -1, \ldots)$ *and* $\mathbf{\Delta} = diag[\exp(-p2\pi i/4n)]$, *then*

$$\mathbf{F}^2\mathbf{E} = \mathbf{D}\mathbf{\Delta}^{-1}\overline{\mathbf{F}}^1_{4n}$$

$$\mathbf{F}^3_{4n} = \mathbf{D}\mathbf{F}^1_{4n}$$

$$\mathbf{F}^4_{4n}\mathbf{E} = \mathbf{\Delta}^{-1}\overline{\mathbf{F}}^1_{4n}$$

Proof

$$[\mathbf{F}^2_{4n}\mathbf{E}]_{pq} = [\exp(-2\pi ip(2n - q - 1)/4n)] = [\mathbf{D}\mathbf{\Delta}^{-1}\overline{\mathbf{F}}^1_{4n}]_{pq}$$

$$[\mathbf{F}^3_{4n}]_{pq} = [\exp(-2\pi ip(q + 2n)/4n)] = [\mathbf{D}\mathbf{F}^1_{4n}]_{pq}$$

$$[\mathbf{F}^4_{4n}\mathbf{E}]_{pq} = [\exp(-2\pi ip(4n - q - 1)/4n)] = [\mathbf{\Delta}^{-1}\overline{\mathbf{F}}^1_{4n}]_{pq}$$

——*——

Lemma 6.8.2

$$\mathbf{F}^{o1}_{4n} = \mathbf{F}_{4n}(1 : 2 : 2n-1, 0 : n-1)$$

Proof We look at the pqth element of each matrix.

$$\begin{aligned}
[\mathbf{F}^{o1}_{4n}]_{pq} &= \exp(-2\pi i(2p+1)q/4n) \\
&= \exp\left(-\frac{2\pi ipq}{2n} - \frac{2\pi iq}{4n}\right) \\
&= [\widehat{\mathbf{F}}_{2n}\mathbf{\Delta}]_{pq}
\end{aligned}$$

——*——

Lemma 6.8.3

$$e^{-\pi i/4n}\boldsymbol{\Delta}\widehat{\mathbf{F}}_{2n}\boldsymbol{\Delta} + e^{\pi i/4n}\boldsymbol{\Delta}^{-1}\overline{\widehat{\mathbf{F}}}_{2n}\boldsymbol{\Delta}^{-1} = 2\mathbf{C}_n^{IV}$$

Proof Look at the pqth element and see that

$$\begin{aligned} e^{-\pi i/4n}e^{-2\pi ip/4n}e^{-4\pi ipq/4n}e^{-2\pi iq/4n} &+ e^{\pi i/4n}e^{2\pi ip/4n}e^{4\pi ipq/4n}e^{2\pi iq/4n} \\ &= 2\cos\left(\frac{(2p+1)(2q+1)\pi}{4n}\right) \end{aligned}$$

is the pqth element of $2\mathbf{C}_n^{IV}$.

——*——

Now we relate the type IV cosine transform $\mathbf{C}_n^{IV}\mathbf{x}$ with the transform of an extended sequence.

Theorem 6.8.1 *Let* $\mathbf{x}$ *be a real vector of length* n *Then*

$$\mathbf{C}_n^{IV}\mathbf{x} = \text{Re}\left[e^{-\pi i/4n}\boldsymbol{\Delta}\widehat{\mathbf{F}}_{2n}\boldsymbol{\Delta}\mathbf{x}\right]$$

Proof Let $\mathbf{y}$ be

$$\mathbf{y} = \begin{bmatrix} \mathbf{x} \\ -\mathbf{E}\mathbf{x} \\ -\mathbf{x} \\ \mathbf{E}\mathbf{x} \end{bmatrix}$$

$$\mathbf{Y} = \mathbf{F}_{4n}\mathbf{y} = \mathbf{F}_{4n}^1\mathbf{x} - \mathbf{F}_{4n}^2\mathbf{E}\mathbf{x} - \mathbf{F}_{4n}^3\mathbf{x} + \mathbf{F}_{4n}^4\mathbf{E}\mathbf{x}$$

Notice that because of the $\mathbf{D}$ matrices, the even components of $\mathbf{Y}$ are zero, thus we are only interested in $\mathbf{Y}^o = \mathbf{Y}(1:2:2n-1)$.

$$\begin{aligned} \mathbf{Y}^o &= 2\mathbf{F}_{4n}^{o1}\mathbf{x} + 2[diag(\exp(\pi i(2p+1)/2n)]\overline{\mathbf{F}}_{4n}^{o1}\mathbf{x} \\ &= 2\mathbf{F}_{4n}^{o1}\mathbf{x} + 2e^{\pi i/2n}\boldsymbol{\Delta}^{-2}\overline{\mathbf{F}}_{4n}^{o1}\mathbf{x} \\ &= 2\widehat{\mathbf{F}}_{2n}\boldsymbol{\Delta}\mathbf{x} + 2e^{\pi i/2n}\boldsymbol{\Delta}^{-2}\overline{\widehat{\mathbf{F}}}_{2n}\boldsymbol{\Delta}^{-1}\mathbf{x} \\ &= 2e^{\pi i/4n}\boldsymbol{\Delta}^{-1}\left(2\text{Re}[e^{-\pi i/4n}\boldsymbol{\Delta}\widehat{\mathbf{F}}_{2n}\boldsymbol{\Delta}\mathbf{x}]\right) \\ &= 2e^{\pi i/4n}\boldsymbol{\Delta}^{-1}[2\mathbf{C}_n^{IV}\mathbf{x}] \end{aligned}$$

or

$$\mathbf{C}_n^{IV}\mathbf{x} = \text{Re}\left[e^{-\pi i/4n}\boldsymbol{\Delta}\widehat{\mathbf{F}}_{2n}\boldsymbol{\Delta}\mathbf{x}\right]$$

——*——

Fast Cosine^{IV} Transform (FCT^{IV})

More relationships concerning the $4n$-point DFT matrix follow in the next lemma.

Lemma 6.8.4

$$\mathbf{F}_{4n}(:,0:2:2n-1) = \mathbf{F}_{2n}(:,0:n-1)$$
$$\mathbf{F}_{4n}(:,2n:2:4n-1) = \mathbf{D}\mathbf{F}_{2n}(:,0:n-1)$$
$$\mathbf{F}_{4n}(:,1:2:2n-1)\mathbf{E} = \mathbf{D}\mathbf{\Delta}^{-1}\overline{\mathbf{F}}_{2n}(:,0:n-1)$$
$$\mathbf{F}_{4n}(:,2n+1:2:4n-1)\mathbf{E} = \mathbf{\Delta}^{-1}\overline{\mathbf{F}}_{2n}(:,0:n-1)$$

Proof

$$[\mathbf{F}_{4n}(:,0:2:2n-1)]_{pq} = [\exp(-2\pi i(p2q)/4n)] = [\mathbf{F}_{2n}(:,0:n-1)]_{pq}$$
$$[\mathbf{F}_{4n}(:,2n:2:4n-1)]_{pq} = [\exp(-2\pi ip(2q+2n)/4n)] = [\mathbf{D}\mathbf{F}_{2n}(:,0:n-1)]_{pq}$$

$$\begin{aligned}[\mathbf{F}_{4n}(:,1:2:2n-1)\mathbf{E}]_{pq} &= [\exp(-2\pi ip(2n-2q-1)/4n)] \\ &= [\mathbf{D}\mathbf{\Delta}^{-1}\overline{\mathbf{F}}_{2n}(:,0:n-1)]_{pq}\end{aligned}$$

$$\begin{aligned}[\mathbf{F}_{4n}(:,2n+1:2:4n-1)\mathbf{E}]_{pq} &= [\exp(-2\pi ip(4n-2q-1)/4n)] \\ &= [\mathbf{\Delta}^{-1}\overline{\mathbf{F}}_{2n}(:,0:n-1)]_{pq}\end{aligned}$$

——*——

Lemma 6.8.5

$$\mathbf{F}_{2n}^{o1} = \mathbf{F}_n\mathbf{\Delta}^2$$

Proof

$$\begin{aligned}[\mathbf{F}_{2n}^{o1}]_{pq} &= \mathbf{F}_{2n}(1:2:2n-1,0:n-1) \\ &= \exp\left(-\frac{2\pi i(2p+1)q}{2n}\right) \\ &= \exp\left(-\frac{2\pi ipq}{n} - \frac{2\pi iq}{2n}\right) \\ &= [\mathbf{F}_n\mathbf{\Delta}^2]_{pq}\end{aligned}$$

——*——

The following theorem relates $\mathbf{C}_n^{IV}\mathbf{x}$ to another transform, but this time of a sequence of length n.

Theorem 6.8.2 *Let* $\mathbf{x}$ *be a real vector of length* n*, then*

$$\mathbf{C}_n^{IV}\mathbf{x} = Re[e^{-\pi i/4n}\mathbf{\Delta}\mathbf{F}_n\mathbf{\Delta}^2\mathbf{z}]$$

Proof Let $\mathbf{y} = [\mathbf{x}, -\mathbf{E}\mathbf{x}, -\mathbf{x}, \mathbf{E}\mathbf{x}]^T$ and define

$$\begin{array}{rcl} \mathbf{z}'(j) & = & \mathbf{y}(2j) \\ \mathbf{w}'(j) & = & \mathbf{y}(2j+1) \\ & & j = 0, \ldots, 2n-1 \end{array}$$

Notice that $\mathbf{w}' = \mathbf{E}\mathbf{z}'$ and $\mathbf{z}' = [\mathbf{z}, -\mathbf{z}]$ where

$$\mathbf{z} = \begin{cases} \mathbf{x}(2j), & \text{if } 0 \le j \le n/2-1 \\ -\mathbf{x}(2n-2j-1) & \text{if } n/2 \le j \le n-1 \end{cases}$$

Here we have $\mathbf{z} = \mathbf{L}_n\mathbf{x}$. Then

$$\begin{array}{rcl} \mathbf{Y} & = & \mathbf{F}_{4n}(:, 0:2:4n-1)\mathbf{z}' + \mathbf{F}_{4n}(:, 1:2:4n-1)\mathbf{w}' \\ & = & \mathbf{F}_{4n}(:, 0:2:2n-1)\mathbf{z} - \mathbf{F}_{4n}(:, 2n:2:4n-1)\mathbf{z} \\ & - & \mathbf{F}_{4n}(:, 1:2:2n-1)\mathbf{E}\mathbf{z} + \mathbf{F}_{4n}(:, 2n+1:2:4n-1)\mathbf{E}\mathbf{z} \end{array}$$

The lemma above gives,

$$\mathbf{Y} = \mathbf{F}_{2n}^1\mathbf{z} - \mathbf{D}\mathbf{F}_{2n}^1\mathbf{z} - \mathbf{D}\mathbf{\Delta}^{-1}\overline{\widehat{\mathbf{F}}}_{2n}\mathbf{z} + \mathbf{\Delta}^{-1}\overline{\widehat{\mathbf{F}}}_{2n}\mathbf{z}$$

The even components of $\mathbf{Y}$ are again zero due to the effect of $\mathbf{D}$. So

$$\begin{array}{rcl} \mathbf{Y}(1:2:2n-1) & = & 2\mathbf{F}_{2n}^{o1}\mathbf{z} + 2diag[\exp(\pi i(2p+1)/2n)]\overline{\mathbf{F}}_{2n}^{o1}\mathbf{z} \\ & = & (2\mathbf{F}_n\mathbf{\Delta}^2 + 2e^{\pi i/2n}\mathbf{\Delta}^{-2}\overline{\mathbf{F}}_n\mathbf{\Delta}^{-2})\mathbf{z} \\ & = & 2e^{\pi i/4n}\mathbf{\Delta}^{-1}[2\mathrm{Re}[e^{-\pi i/4n}\mathbf{\Delta}\mathbf{F}_n\mathbf{\Delta}^2\mathbf{z}]] \end{array}$$

Hence

$$\mathbf{C}_n^{IV}\mathbf{x} = \mathrm{Re}\left[e^{-\pi i/4n}\mathbf{\Delta}\mathbf{F}_n\mathbf{\Delta}^2\mathbf{z}\right]$$

____*____

Together Theorems 6.8.1 and 6.8.2 establish the relations we need for the type IV cosine transform

$$\begin{array}{rcl} \mathbf{C}_n^{IV}\mathbf{x} & = & Re[e^{-\pi i/4n}\mathbf{\Delta}\widehat{\mathbf{F}}_{2n}\mathbf{\Delta}\mathbf{x}] \\ & = & Re[e^{-\pi i/4n}\mathbf{\Delta}\mathbf{F}_n\mathbf{\Delta}^2\mathbf{L}_n\mathbf{x}] \end{array}$$

The Fast Sine^{IV} Transform (FST^{IV})

The derivation for the type IV sine transform is very similar.

Lemma 6.8.6

$$e^{-\pi i/4n}\mathbf{\Delta}\hat{\mathbf{F}}_{2n}\mathbf{\Delta} - e^{\pi i/4n}\mathbf{\Delta}^{-1}\bar{\hat{\mathbf{F}}}_{2n}\mathbf{\Delta}^{-1} = -2i\mathbf{S}_n^{IV}$$

Proof The pqth element of the expression on the left is

$$\begin{aligned} e^{-\pi i/4n}e^{-2\pi ip/4n}e^{-4\pi ipq/4n}e^{-2\pi iq/4n} \quad &- \quad e^{\pi i/4n}e^{2\pi ip/4n}e^{4\pi ipq/4n}e^{2\pi iq/4n} \\ &= \quad -2i\sin\left(\frac{(2p+1)(2q+1)\pi}{4n}\right) \end{aligned}$$

———*———

Theorem 6.8.3 *Let* $\mathbf{x}$ *be a real vector of length* n *then*

$$\mathbf{S}_n^{IV}\mathbf{x} = -Im\left[e^{-\pi i/4n}[\mathbf{\Delta}\hat{\mathbf{F}}_{2n}\mathbf{\Delta}\mathbf{x}]\right]$$

Proof Take $\mathbf{y}$ to be

$$\mathbf{y} = \begin{bmatrix} \mathbf{x} \\ \mathbf{Ex} \\ -\mathbf{x} \\ -\mathbf{Ex} \end{bmatrix}$$

and $\mathbf{Y} = \mathbf{F}_{4n}\mathbf{y}$. Then

$$\begin{aligned} \mathbf{Y} &= \mathbf{F}_{4n}^1\mathbf{x} + \mathbf{F}_{4n}^2\mathbf{Ex} - \mathbf{F}_{4n}^3\mathbf{x} - \mathbf{F}_{4n}^4\mathbf{Ex} \\ &= \mathbf{F}_{4n}^1\mathbf{x} + \mathbf{D}\mathbf{\Delta}^{-1}\bar{\mathbf{F}}_{4n}^1\mathbf{x} - \mathbf{D}\mathbf{F}_{4n}^1\mathbf{x} - \mathbf{\Delta}^{-1}\bar{\mathbf{F}}_{4n}^1\mathbf{x} \end{aligned}$$

As usual, the $\mathbf{D}$ matrix causes the even terms to be zero, hence

$$\begin{aligned} \mathbf{Y}^o &= \mathbf{Y}(1:2:2n-1) \\ &= 2\mathbf{F}_{4n}^{o1}\mathbf{x} - 2[diag(\exp(\pi i(2p+1)/2n))]\bar{\mathbf{F}}_{4n}^{o1}\mathbf{x} \\ &= 2\mathbf{F}_{4n}^{o1}\mathbf{x} - 2e^{\pi i/2n}\mathbf{\Delta}^{-2}\bar{\mathbf{F}}_{4n}^{o1}\mathbf{x} \\ &= 2\hat{\mathbf{F}}_{2n}\mathbf{\Delta}\mathbf{x} - 2e^{\pi i/2n}\mathbf{\Delta}^{-2}\bar{\hat{\mathbf{F}}}_{2n}\mathbf{\Delta}^{-1}\mathbf{x} \\ &= 2e^{\pi i/4n}\mathbf{\Delta}^{-1}[2i\mathrm{Im}[e^{-\pi i/4n}\mathbf{\Delta}\hat{\mathbf{F}}_{2n}\mathbf{\Delta}\mathbf{x}]] \\ &= 2e^{\pi i/4n}\mathbf{\Delta}^{-1}[-2i\mathbf{S}_n^{IV}\mathbf{x}] \end{aligned}$$

Another designation for the type IV sine transform is in terms of the transform of a permuted sequence of length n.

Theorem 6.8.4 *Let* $\mathbf{x}$ *be a real vector of length* n, *then then*

$$\mathbf{S}_n^{IV}\mathbf{x} = -Im\left[e^{-\pi i/4n}\mathbf{\Delta}\mathbf{F}_n\mathbf{\Delta}^2\mathbf{v}\right]$$

Proof Let $\mathbf{y} = [\mathbf{x}, \mathbf{E}\mathbf{x}, -\mathbf{x}, -\mathbf{E}\mathbf{x}]^T$ and define

$$\mathbf{v} = \begin{cases} \mathbf{x}(2j), & \text{if } 0 \le j \le n/2-1 \\ \mathbf{x}(2n-2j-1) & \text{if } n/2 \le j \le n-1 \end{cases}$$

If $\mathbf{Y} = \mathbf{F}_{4n}\mathbf{y}$, then

$$\begin{aligned} \mathbf{Y} &= \mathbf{F}_{4n}(:, 0:2:4n-1)\mathbf{v} - \mathbf{F}_{4n}(:, 2n:2:4n-1)\mathbf{v} \\ &+ \mathbf{F}_{4n}(:, 1:2:2n-1)\mathbf{E}\mathbf{v} - \mathbf{F}_{4n}(:, 2n+1:2:4n-1)\mathbf{E}\mathbf{v} \\ &= \mathbf{F}_{2n}^{1}\mathbf{v} + \mathbf{D}\mathbf{F}_{2n}^{1}\mathbf{v} - \mathbf{D}\mathbf{\Delta}^{-1}\overline{\widehat{\mathbf{F}}}_{2n}\mathbf{v} - \mathbf{\Delta}^{-1}\overline{\widehat{\mathbf{F}}}_{2n}\mathbf{v} \end{aligned}$$

The even components of $\mathbf{Y}$ are zero due to $\mathbf{D}$ so that

$$\begin{aligned} \mathbf{Y}(1:2:2n-1) &= \mathbf{F}_{2n}^{o1}\mathbf{v} - 2diag[\exp(\pi i(2p+1)/2n)]\overline{\mathbf{F}}_{2n}^{o1}\mathbf{v} \\ &= (2\mathbf{F}_n\mathbf{\Delta}^2 - 2e^{\pi i/2n}\mathbf{\Delta}^{-2}\overline{\mathbf{F}}_n\mathbf{\Delta}^{-2})\mathbf{v} \\ &= 2e^{\pi i/4n}\mathbf{\Delta}^{-1}[2i\mathrm{Im}[e^{-\pi i/4n}\mathbf{\Delta}\mathbf{F}_n\mathbf{\Delta}^2\mathbf{v}]] \\ &= 2e^{\pi i/4n}\mathbf{\Delta}^{-1}[-2i\mathbf{S}_n^{IV}\mathbf{x}] \end{aligned}$$

——*——

Together Theorems 6.8.3 and 6.8.4 give the relation needed for the type IV sine transform.

$$\begin{aligned} \mathbf{S}_n^{IV}\mathbf{x} &= -Im[e^{-\pi i/4n}\mathbf{\Delta}\widehat{\mathbf{F}}_{2n}\mathbf{\Delta}\mathbf{x}] \\ &= -Im[e^{-\pi i/4n}\mathbf{\Delta}\mathbf{F}_n\mathbf{\Delta}^2\mathbf{K}_n\mathbf{x}] \end{aligned}$$

Fast Cosine^{III} and Sine^{III} Transforms (FCT^{III} and FST^{III})

The equations for the type III cosine and sine transforms,

$$\mathbf{C}_n^{III}\mathbf{x} = \mathrm{Re}[\widehat{\mathbf{F}}_{2n}\mathbf{\Delta}\mathbf{x}]$$

and

$$\mathbf{S}_n^{III}\mathbf{x} = -\mathrm{Im}[\widehat{\mathbf{F}}_{2n}\mathbf{\Delta}\mathbf{x}]$$

can be derived, in a similar manner, from the Fourier transform of

$$\mathbf{y} = [\mathbf{x}, -\mathbf{T}\mathbf{x}, -\mathbf{x}, \mathbf{T}\mathbf{x}]^T$$

and

$$\mathbf{y} = [\mathbf{x}, \mathbf{T}\mathbf{x}, -\mathbf{x}, -\mathbf{T}\mathbf{x}]^T,$$

respectively.

6.9 Fortran Code

The methods presented in this chapter are extremely easy to implement. All that is needed is an efficient complex FFT subroutine. Here we use the package by Paul Swarztrauber called FFTPACK. The subroutines we use from the package are:

- cffti(n,wsave) initialize by factoring n and computing the multipliers
- cfftf(n,a,wsave) transform a complex vector stored in **a**

Programs using FFTPACK for the complex transform are presented here.

```
      program type12
c
c Calculates the un-normalized type I and II
c sine and cosine transforms of a real vector y
c
c Using FFTPACK complex FFT subroutine
c
      real x(n), c1x(n), c2x(n), s1x(n), s2x(n)
      complex a(n), v(n), z(n), twid
      real wsave(4*n+15)
      double precision pi, pi2n
c
      write(6,*) 'enter n'
      read(5,*) n
c
      pi = 4.0d0*datan(1.0d0)
      pi2n = pi/dfloat(2*n)
c
      call cffti(n,wsave)
c
      do 10 i = 1, n
        x(i) = func(i)
      10 continue
c
c put v in the real part
c put z in the imaginary part
c
      do 20 i = 1, n/2
        i1 = i-1
```

```
    a(i) = cmplx(x(2*i1+1),x(2*i1+1))
    a(n-i+1) = cmplx(x(2*i1+2),-x(2*i1+2))
  20 continue
  if (n/2.lt.float(n)/2.0) then
    a(n/2+1) = cmplx(x(n),x(n))
  end if
c
  call cfftf(n,a,wsave)
c
  do 30 i = 1, n
    i1 = i - 1
    v(i) = (.5,0.0)*(conjg(a(mod(n-i1,n)+1))+a(i))
    z(i) = ((0.0,1.0)/(2.0,0.0))
  $      *(conjg(a(mod(n-i1,n)+1))-a(i))
  30 continue
c
  do 40 i = 1, n
    i1 = i - 1
    twid = cmplx(dcos(pi2n*i1),-dsin(pi2n*i1))
    v(i) = twid*v(i)
    z(i) = twid*z(i)
    c2x(i) = real(v(i))
    s2x(i) = -aimag(z(i))
    a(i) = cmplx(c2x(i),-s2x(i))
    twid = conjg(twid)
    a(i) = twid*a(i)
    c1x(i) = real(a(i))
    s1x(i) = -aimag(a(i))
  40 continue
c
  write(6,*) 'cosine I transform'
  do 50 i = 1, n
    write(6,*) c1x(i)
  50 continue
  write(6,*) 'sine I transform'
  do 60 i = 1, n
    write(6,*) s1x(i)
  60 continue
  write(6,*) 'cosine II transform'
  do 70 i = 1, n
```

```
          write(6,*) c2x(i)
       70 continue
       write(6,*) 'sine II transform'
       do 80 i = 1, n
         write(6,*) s2x(i)
       80 continue
c
       end
       program type34
c
c Calculates the un-normalized type III and IV
c sine and cosine transforms of a real vector y
c
c Using FFTPACK complex FFT subroutine
c
       real x(n), c3x(n), c4x(n), s3x(n), s4x(n)
       complex a(n), v(n), z(n), twid
       real wsave(4*n+15)
       double precision pi, pin, pi2n, pi4n, arg
c
       write(6,*) 'enter n'
       read(5,*) n
c
       pi = 4.0d0*datan(1.0d0)
       pin = pi/dfloat(n)
       pi2n = pi/dfloat(2*n)
       pi4n = pi/dfloat(4*n)
c
       call cffti(n,wsave)
c
       do 10 i = 1, n
         x(i) = func(i)
       10 continue
c
       do 20 i = 1, n/2
         i1 = i-1
         v(i) = cmplx(x(2*i1+1),0.0)
         z(i) = v(i)
         v(n-i+1) = cmplx(x(2*i1+2),0.0)
         z(n-i+1) = -v(n-i+1)
```

```
 20 continue
c
 if (n/2.lt.float(n)/2.0) then
  v(n/2+1) = cmplx(x(n),0.0)
  z(n/2+1) = v(n/2+1)
 end if
c
 do 30 i = 1, n
  i1 = i - 1
  twid = cmplx(dcos(pin*i1),-dsin(pin*i1))
  v(i) = twid*v(i)
  z(i) = twid*z(i)
 30 continue
c
 call cfftf(n,v,wsave)
 call cfftf(n,z,wsave)
c
 do 40 i = 1, n
  i1 = i - 1
  arg = pi4n + pi2n*dfloat(i1)
  twid = cmplx(dcos(arg),-dsin(arg))
  v(i) = twid*v(i)
  z(i) = twid*z(i)
  c4x(i) = real(z(i))
  s4x(i) = -aimag(v(i))
  a(i) = cmplx(c4x(i),-s4x(i))
  twid = conjg(twid)
  a(i) = twid*a(i)
  c3x(i) = real(a(i))
  s3x(i) = -aimag(a(i))
 40 continue
c
 write(6,*) 'cosine III transform'
 do 50 i = 1, n
  write(6,*) c3x(i)
 50 continue
 write(6,*) 'sine III transform'
 do 60 i = 1, n
  write(6,*) s3x(i)
 60 continue
```

```
  write(6,*) 'cosine IV transform'
  do 70 i = 1, n
    write(6,*) c4x(i)
  70 continue
  write(6,*) 'sine IV transform'
  do 80 i = 1, n
    write(6,*) s4x(i)
  80 continue
c
  end
```

Chapter 7

Distributed Mixed-Radix FFTs on the Hypercube

7.1 Introduction

In the previous chapters we have discussed the implementation of radix-two FFT algorithms that work on vectors of length $n = 2^k$. This is because radix-two FFT algorithms map very well into the binary hypercube. A practical question is whether vectors of length other than a power of two can be done on the hypercube architecture efficiently without excessive data permutations and its associated communication costs.

One method of computing the single dimensional transform of length $n = mq$ is to arrange the FFT of a one-dimensional array as a two-dimensional array. This is commonly known as the "twiddle" factor approach [Gentleman and Sande (1966), Brigham (1974), Nussbaumer (1982)]. In this fashion, if $n = mq$, and x, the input vector, is mapped as an m-by-q array in column-major order, m transforms of length q are done vertically and q transforms of length m are done horizontally with an intervening point-wise multiplication by powers of the nth root of unity. Obviously, m and q can be any length and computed by any of the mixed-radix FFT routines.

The approach which we present in this chapter computes mixed-radix FFTs in a distributed fashion, where some of the butterfly computations require the co-operation of two nodes. The distributed approach intermingles computation with communication and in systems such as the Floating Point Systems T-Series where these two tasks can be overlapped and done simultaneously, this methodology can be pursued efficiently. Here we also make use of the "twiddle" factor approach by factoring n into a power of two times any other integer, i.e. $n = m2^k$. We perform the FFTs of length $q = 2^k$ by mapping the signal flow graph into the hypercube as before, and the FFTs of length m locally inside a processor and therefore obviate the need for a transpose. The method of implementation is presented in Chapter 4 and is called the Local-Distributed Method.

Since our method mixes a Mixed-Radix FFT algorithm with a Radix-2 algorithm, we shall term it the Mixed-Radix-2 FFT procedure. While it is true that not all integers can be factored by a power of two (excluding $2^0 = 1$), one can always find such a number in the vicinity of the number one wishes. For example, all even numbers automatically satisfy this criteria, although choosing which power of two to be q may be dependent on how many processor nodes are available and the cost of communication.

The Transpose-Split heuristic [McBryan and Van de Velde (1985)] (Chapter 4) presupposes a two-dimensional array of data which is acted upon both vertically and horizontally, with the vertical computations totally independent of the horizontal computations. That is, during one stage of the procedure, the algorithm works on the columns of the array and in a separate stage the rows. Obviously in between the two stages, the array must be transposed. In this method all the communication takes place in the transpose operation and the computations are all processor local. The implementation of a single long FFT of length $n = mq$ can be done by using this approach. Since the FFT computations are entirely local, any of the available FFT subroutines can be used. The implementation characteristics are almost exactly the same as that for the Transpose-Split 2-dimensional FFT of Chapter 4 except that there is a point-wise multiplication by twiddle factors in between the two FFT stages.

7.2 Twiddle Factor Algorithm Splitting of the DFT

Another approach to defining the FFT is to notice that one can split the DFT of a sequence of length n, if $n = mq$, n, m, and q integers, into two DFTs, one of length m and one of length q.

Suppose $n = mq$, and $\mathbf{x} \in C^n$, we can write $\mathbf{x}$ as an m-by-q array by the following definition:

$$\mathbf{x}_{m\times q} = [x(0:m-1), x(m:2m-1), \ldots, x(n-m:n-1)] \in C^{m\times q}$$

Definition 7.2.1 *Twiddle Factor Matrix:* $\mathbf{T}_{m\times q}$ *is an m-by-q matrix composed of the first m rows of* $\mathbf{F}_n$ *and the first q columns of* $\mathbf{F}_n$*,*

$$\mathbf{T}_{m\times q} = \mathbf{F}_n(0:m-1, 0:q-1)$$

Definition 7.2.2 *Point-wise Multiplication: Let the operator $*$ denote point-wise multiplication of two matrices.*

$$\mathbf{A} * \mathbf{B} = [a_{jk} b_{jk}]$$

$\mathbf{A}$ *and* $\mathbf{B}$ *are of the same dimension.*

If $\mathbf{F}_m$ and $\mathbf{F}_q$ are the DFT matrices of dimensions m and q, respectively, and $\mathbf{y} = \mathbf{F}_n \mathbf{x}$, then

Theorem 7.2.1 *(Twiddle Factor)*

$$\mathbf{y}^T_{q \times m} = \mathbf{F}_m \left[(\mathbf{T}_{m \times q}) * [\mathbf{x}_{m \times q} \mathbf{F}_q] \right]$$

Theorem 7.2.2 *(Inverse)*

$$\mathbf{x}_{m \times q} = [[\mathbf{F}^H_m \mathbf{y}_{q \times m}] * \mathbf{T}^H_{m \times q}] \mathbf{F}^H_q$$

where $\mathbf{F}^H_n$ *is the conjugate transpose of* $\mathbf{F}_n$ *and* $\mathbf{T}^H_{m \times q} = \mathbf{F}^H_n(0 : m-1, 0 : q-1)$.

If the DFT is computed by matrix-vector multiplication $\mathbf{F}_n \mathbf{x}$, there would be $O(n^2)$ operations. Computed by the twiddle factor method, we would only have $mq^2 + qm^2 + mq$ steps which is less than $n^2 = m^2 q^2$. If n is a highly composite number, m and q can be broken down in a similar manner with resulting savings. In fact if $n = 2^k$, one can recursively split n into k factors, resulting in $O(\log_2 n)$ operations. This is in fact the radix-2 FFT Algorithm.

7.3 Procedure for Hypercube Implementation

Let $q = 2^k$, for some k an integer and $P = 2^d$, $(d < k)$ be the number of processors. Distribute x column-wise in processors which are lined up row-wise, i.e. processor i gets block column i of $x_{m \times q}$. Using the CT1 Cooley-Tukey algorithm. the distributed FFTs are computed across the rows of $\mathbf{x}_{m \times q}$. This permutes the columns of x by the bit-reversal permutation P_q. Next point-wise multiplications by the twiddle factor matrix is done. Since we want to be sure to keep the columns consistent, we also permute the columns of $\mathbf{T}_{m \times q}$ by $\mathbf{P}_q$. (The twiddle factors are computed locally so there is no need to actually do distributed permutation.)

Next m-point FFTs are done along the columns. These are all processor local and can be unscrambled if desired. The final answer is read across the rows in bit-reversed order, but down the columns naturally. Notice that while q is a power of two, m can be any number, including a prime number. This is where the mixed-radix idea comes in. In practice, however, one should try to make m as composite as possible.

The inverse consists of first performing processor local transforms of the column, then multiplication by $\overline{\mathbf{T}}_{m\times q}\mathbf{P}_q$ and finally, the Gentleman-Sande (GS2) algorithm where data is in bit-reversed order on input, is used to obtain the answers in natural order. The inverse gives back our original sequence in column-wise natural order.

Procedure (Forward and Inverse Mixed-Radix-2 FFT)

Step 1: Map $\mathbf{x}_{m\times q}$ by the *contiguous column* mapping into the hypercube.

Step 2: Apply CT1 to the rows of $\mathbf{x}_{m\times q}$.

Step 3: Compute and store the twiddle factors $\mathbf{T}_{m\times q}\mathbf{P}_q$.

Step 4: Apply the twiddle factors.

Step 5: Call FFTPACK to forward transform the columns of $\mathbf{x}_{m\times q}$ of length m.

- a: Factor m and store in array **wsave**.
- b: Calculate the forward transform of each column of $\mathbf{x}_{m\times q}$.

Step 6: Call FFTPACK to back transform the columns of $x_{m\times q}$.

- a: Use the saved array **wsave**.
- b: Calculate the inverse transform of the columns of $\mathbf{x}_{m\times q}$.

Step 7: Apply the conjugate of the twiddle factors stored, $\overline{\mathbf{T}}_{m\times q}P_q$.

Step 8: Apply GS2 to the rows of $\mathbf{x}_{m\times q}$.

End: Get back what you started with.

Algorithm 7.3.1 Mixed-Radix-2 Forward FFT

```
/* $\mu$ = processor id; $n = mq$; $P = 2^d$; $r = n/P$; $s = q/P$ */
/* Initially, processor $\mu$ holds: */
/* $\mathbf{x}(r \cdot id[\mu] : r \cdot (id[\mu]+1)-1)$ */
  call Distributed CT1($\mathbf{x}$)
/* compute $\mathbf{x}_{m\times q} \leftarrow \mathbf{T}_{m\times q}\mathbf{P}_q\mathbf{x}_{m\times q}$ */
  for $j = 0 : q-1$
    $j' = bitrev(s \cdot id[\mu] + j)$
    for $i = 0 : m-1$
      $h = id[\mu] \cdot r + j' \cdot m + i$
      $\mathbf{x}(i + j \cdot m) \leftarrow \omega_n^h \mathbf{x}(i + j \cdot m)$
    end
  end
  call FFTPACK($\mathbf{x}$)
```

Algorithm 7.3.2 Mixed-Radix-2 Inverse FFT

```
  /* $\mu$ = processor id; $n = mq$; $P = 2^d$; $r = n/P$; $s = q/P$ */
  /* Initially, processor $\mu$ holds: */
  /* $\mathbf{y}(r \cdot id[\mu] : r \cdot (id[\mu] + 1) - 1)$ */
  /* where $\mathbf{y} \equiv (\mathbf{y}_{q \times m}^T \mathbf{P}_q)$ */
    call FFTPACK($\mathbf{y}$)
  /* compute $\mathbf{y}_{m \times q} \overline{\mathbf{T}}_{m \times q} \mathbf{P}_q$ */
    for $j = 0 : q - 1$
      $j' = bitrev(s \cdot id[\mu] + j)$
      for $i = 0 : m - 1$
        $h = id[\mu] \cdot r + j' \cdot m + i$
        $\mathbf{x}(i + j \cdot m) \leftarrow \overline{\omega}_n^h \mathbf{x}(i + j \cdot m)$
      end
    end
    call distributed GS2($\mathbf{y}$)
```

This procedure has been coded up on the Intel iPSC using FFTPACK for the processor-local FFT and distributed Cooley-Tukey and Gentleman-Sande algorithms documented in Chapter 3 and Appendix A.

Times in milliseconds for a 1280-point forward and inverse transform pair is given in Table 7.1.

Our examples show that the minimum total time occurs roughly where the local transform and the distributed transform times are close. The overhead of twiddle factors is unavoidable, so to maximally exploit the parallelism in there, one should use the largest cube available. Finally the factorization time for m increases as m increases. However if many transforms are done with the same m, the factorization can be calculated beforehand and stored, likewise for the twiddle factors. In our implementation, the factorization is calculated once and stored, as are the twiddle factors.

A model for determining how large P should be takes into account both computational complexity and communication costs. The computational workload is directly divided between the numerous processors, hence favoring the use of more processors. Yet as we add more processors, we get into more complex communication because the distributed FFTs require communication complexity of

Table 7.1: Mixed Radix-2 Timings

$m \times q = 1280$ point MR2-FFT (Forward and Inverse)							
$\log_2 p$	q	m	total	dist.	local	twiddle	factor
1	256	5	3540–3640	2015–2115	670	355	5
1	128	10	3265–3360	1670–1755	720	865–875	10
1	64	20	3005–3100	1410–1490	720	860–875	15
1	32	40	2945–3040	1190–1270	875–880	850–865	30
1	16	80	2810	995	940	820	55
1	8	160	2735	810	1040	780	110
1	4	320	2715	680	1125	695	215
1	2	640	2930	630	1335	540	430
2	256	5	1815–1915	1050–1150	335	425	5
2	128	10	1680–1775	880–965	365	440	10
2	64	20	1550–1655	745–840	365	425–440	15
2	32	40	1555–1655	670–755	440	415–435	30
2	16	80	1465–1515	545–570	470	395–425	55
2	8	160	1465	480	530	855	110
2	4	320	1520	450	580	280	215
3	256	5	965–1040	580–650	170	215	5
3	128	10	890–945	490–535	185	210–220	10
3	64	20	815–890	405–475	185	220	15
3	32	40	800–875	350–410	220–260	200–225	30
3	16	80	790–840	310–335	240	180–215	55
3	8	160	820–875	265–300	255–270	145–220	105
4	256	5	500–555	300–360	85	110	5
4	128	10	460–510	255–300	90	105–115	10
4	64	20	430–480	230–260	95	115	15
4	32	40	435–490	200–240	115	85–115	30
4	16	80	445–525	185–245	115–125	110	55

$2\log_2 P$. Each processor has $\frac{mq}{P}$ points when $n = mq$ and $P = 2^d$. Assuming that m is prime, the local FFTs cost $O(m^2 q/P)$ flops. Multiplication by twiddle factors cost another $\frac{mq}{P}$ multiplications and the computation for the distributed FFTs measures $O(mq \log_2 q/P)$ flops. Hence, letting t_{comm} be the per element data transfer rate, τ the communication start-up or latency period, and t_{comp} the per flop computational rate, a model for the Mixed-Radix 2 FFT is

$$T = m^2(\frac{q}{P})t_{comp} + \frac{mq}{P}(m + \log_2 q + 1)t_{comp} + 2\log_2 P(\tau + \frac{mq}{P}t_{comm})$$

Usually, $t_{comm} \geq t_{comp}$ and τ is several orders of magnitude greater than t_{comm}. Hence up until

$$2\log_2 P t_{comm} > \frac{t_{comp}}{P}$$

it is always better to use more processors. This is because an increased number of processors increases parallelism. Looking at the entries in Table 7.1, we can see that the local FFT portion, the computation of twiddle factors, and the factorization of m reflects linear speedup with the doubling of P, i.e., the timings for $2P$ processors is almost exactly half of the timings for P processors.

Holding n and P constant, one might ask the question of how m and q should be chosen. That is, should we always choose the smallest q? The results in Table 7.1 suggest that there should indeed be the tendency to do this, although the timings between $q = 256$ and $q = 16$ differ by only 10% for $P = 16$. Our timings are done on a hypercube without vector nodes. However we can conjecture that a smaller q would result in longer vectors of length m to be used during the distributed FFTs. In our implementation however, no use of this characteristic is made. Hence our timings show a slight increase for the timings of the smallest q value because of the increase in overhead for the factorization of m, m becoming more composite as q decreases. If this factorization time is not accounted for, i.e. the factorization of m is known and stored, then timings should decrease as q decreases.

References

L. Adams & T. Crockett [1984], "Modeling algorithm execution time on processor arrays," *Computer* 17, 38–43.

H. Ahmed, T. Natarajan & K. R. Rao [1974], "Discrete cosine transform," *IEEE Trans. Comput.* C-23, 90–93.

R. Alt [1978], "Error propagation in Fourier transforms," *Math. Comp. Simul.* 20, 37–43.

G. D. Bergland [July 1969], "A guided tour of the fast Fourier transform," *IEEE Spectrum.*

S. Bertram [Mar. 1970], "On the Derivation of the Fast Fourier Transform," *IEEE Transactions on Audio and Electroacoustics* AU-18, 55–58.

L. N. Bhuyan & D. P. Agrawal [Apr. 1984], "Generalized Hypercube and Hyperbus Structures for a Computer Network," *IEEE Trans. Comput.* C-33, 323–333.

R. E. Blahut [1985], *Fast Algorithms for Digital Signal Processing*, Addison-Wesley, Reading, Massachusetts.

P. Bois & J. Vignes [1980], "Software for evaluating local accuracy in the Fourier transform," *Math. Comp. Simul.* 22, 141–150.

R. N. Bracewell [1986], *The Hartley Transform*, Oxford University Press, New York.

W. L. Briggs [July 1987], "Further Symmetries of In-Place FFTs," *SIAM J. Sci. Statist. Comput.* 8, 644–654.

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

E. O. Brigham & R. E. Morrow [Dec. 1967], "The fast Fourier Transform," *IEEE Spectrum.*

O. Buneman [1987], "Stable On-Line Creation of Sines or Cosines of Successive Angles," *Proc. IEEE*, (to appear).

B. Buzbee [1973], "A fast Poisson solver amenable to parallel computation." *IEEE Trans. Comput.* C-22, 793–796.

R. M. Chamberlain [May 1986], "Gray Codes, Fast Fourier Transforms and Hypercubes," Chr. Michelsen Institute, CCS 86/1, Bergen, Norway.

O. W. C. Chan & E. I. Jury [Jan. 1974], "Roundoff Error in Multidimensional Generalized Discrete Transforms," *IEEE Trans. Circuits and Systems* CAS-21, 100–108.

T. F. Chan [Sept. 9, 1986], "On Gray Code Mappings for Mesh-FFTs on Binary N-Cubes," NASA Ames Research Center, RIACS Technical Report 86.17.

W. Chen, C. H. Smith & S. Fralick [Sept. 1977], "A Fast Computational Algorithm for the Discrete Cosine Transform," *IEEE Trans. Comm.* COM-25, 1004–1009.

R. J. Clarke [1985], *Transform Coding of Images*, Academic Press, New York.

W. T. Cochran, J. W. Cooley, D. L. Favin, H. D. Helms, R. A. Kaenel, W. W. Lang, G. C. Maling, Jr., D. E. Nelson, C. M. Rader & P. D. Welch [Oct. 1967], "What Is the Fast Fourier Transform?," *Proceedings of the IEEE* 55, 1664–1677.

T. Coleman & C. F. Van Loan [1987], *A Matrix Computation Handbook*, (manuscript).

D. S. P. Committee [1979], *Programs for Digital Signal Processing*, IEEE Press, New York.

J. W. Cooley, P. A. W. Lewis & P. D. Welch [1970], "The fast Fourier transform algorithm: Programming considerations in the calculation of sine, cosine and Laplace transforms," *J. Sound Vibration* 12, 315–337.

——— [June 1969], "The Finite Fourier Transform," *IEEE Transactions on Audio and Electroacoustics* AU-17, 77–85.

J. W. Cooley & J. W. Tukey [Apr. 1965], "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.* 19, 297–301.

J. W. Cooley, P. A. W. Lewis & P. D. Welch [June 1967], "Historical Notes on the Fast Fourier Transform," *IEEE Transactions on Audio and Electroacoustics* AU-15, 76–79.

——— [Mar. 1969], "The Fast Fourier Transform and Its Application," *IEEE Transactions on Education* 12, 27–34.

M. J. Corinthios [June 1971], "The design of a class of fast Fourier transform computers," *IEEE Trans. Comput.* C-20, 617–623.

J. Dollimore [1973], "Some algorithms for use with the fast Fourier transform," *J. Inst. Math. Appl.* 12, 115–117.

M. Drubin [May 1971], "Kronecker Product Factorization of the FFT Matrix," *IEEE Trans. Comput.*.

D. E. Dudgeon & R. M. Mersereau [1984], in *Multidimensional Digital Signal Processing*, Prentice-Hall, Inc., Englewood Cliffs, NJ.

D. B. Gannon & J. Van Rosendale [Dec. 1984], "On the Impact of Communication Complexity on the Design of Parallel Numerical Algorithms," *IEEE Trans. Comput.* C-33, 1180–1194.

W. M. Gentleman [1975], "Error Analysis of QR Decompositions by Givens Transformations," *Linear Algebra Appl.* 10, 189–197.

W. M. Gentleman & G. Sande [1966], "Fast Fourier Transforms—For Fun and Profit," in *1966 Fall Joint Computer Conference, AFIPS Conf. Proceedings* #29, Spartan, Washington, D.C., 563–578.

M. W. George, R. T. Ling, J. F. Mangus & W. T. Thompkins [Mar. 10-12, 1987], "Application of Computational Physics Within Northrop," *Supercomputing in Aerospace*.

W. J. Gilbert [1976], *Modern Algebra With Applications*, John Wiley and Sons, Inc., New York.

J. A. Glassman [Feb. 1970], "A generalization of the fast Fourier transform," *IEEE Trans. Comput.* C-19, 105–116.

G. H. Golub & C. F. Van Loan [1983], *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD.

I. J. Good [Mar. 1971], "The Relationship Between Two Fast Fourier Transforms," *IEEE Trans. Comput.*.

J. Gustafson [Mar. 22-25, 1987], "Ensemble FFT's as a Function of Compute Communication Ratios," in *Fast Fourier Transforms for Vector and Parallel Computers Workshop*, Charles F. Van Loan, ed., The Mathematical Sciences Institute, Cornell University, Ithaca, NY.

D. B. Harris, J. H. McClellan, D. S. K. Chan & H. W. Schuessler [1977], "Vector Radix Fast Fourier Transform," *Rec. 1977 IEEE Internat. Cof. Acoust., Speech, Signal Proc.*.

L. S. Haynes, R. L. Lau, D. P. Siewiorek & D. W. Mizell [Jan. 1982], "A Survey of Highly Parallel Computing," *Computer*.

C-T. Ho & S. L. Johnsson [Sept. 1986], "Matrix Transposition on Boolean n-cube Configured Ensemble Architectures," Yale University, Department of Computer Science, YALEU/DCS/TR-494, New Haven, CT.

R. Hockney & C. Jesshope [1981], in *Parallel Computers: Architecture, Programming and Algorithms*, Adam Hilger, Bristol.

A. K. Jain [1976], "A fast Karhunen-Loeve transform for a class of stochastic processes," *IEEE Trans. Comm.* COM-24, 1023–1029.

——— [1979], "A sinusoidal family of unitary transforms," *IEEE Trans.* PAMI-1, 356–365.

S. L. Johnsson [Feb. 1985a], "Data Permutations and Basic Linear Algebra Computations on Ensemble Architectures," Yale University, Department of Computer Science, YALEU/CSD/RR-367, New Haven, CT.

——— [Jan. 1985b], "Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures," Yale University, Department of Computer Science, YALEU/CDS/RR-361, New Haven, CT.

D. K. Kahaner [Dec. 1970], "Matrix Description of the Fast Fourier Transform," *IEEE Transactions on Audio and Electroacoustics* AU-18, 442–450.

T. Kaneko & B. Liu [Oct. 1970], "Accumulation of Round-Off Error in Fast Fourier Transforms," *J. Assoc. Comput. Mach.* 17, 637–654.

H. B. Kekre & J. K. Solanki [1978], "Comparative performance of various trigonometric unitary transforms for transform image coding," *Internat. J. Electron.* 44, 305–315.

H. Kitajima [Apr. 1980], "A Symmetric Cosine Transform," *IEEE Trans. Comput.* C-29, 317–323.

W. R. Knight & R. Kaiser [Dec. 1979], "A simple fixed point error bound for the fast Fourier transform," *IEEE Trans. Acoust. Speech Signal Process.* ASSP-27, 615–620.

C-C. J. Kuo, B. C. Levy & B. R. Musicus [July 1987], "A Local Relaxation Method for Solving Elliptic PDEs on Mesh-Connected Arrays," *SIAM J. Sci. Statist. Comput.* 8, 550–573.

B. Lui & T. Kaneko [1975], "Roundoff error in fast Fourier transforms (decimation in time)," *Proc. IEEE* 63, 991–992.

R. E. Lynch, J. R. Rice & D. H. Thomas [1964], "Direct solution of partial difference equations by tensor product methods," *Numerische Mathematik* 6, 185–199.

J. Makhoul [Feb. 1980], "A Fast Cosine Transform in One and Two Dimensions," *IEEE Trans. Acoust. Speech Signal Process.* ASSP-28, 27–34.

O. A. McBryan & E. F. Van de Velde [Mar. 1987], "Hypercube Algorithms and Implementations," *SIAM J. Sci. Statist. Comput.* 8, s277–s287.

D. Miles, P. Kinney, J. Groshong & R. Fazzari [1987], "Specification and Performance Analysis of Six Benchmark Programs for the FPS T Series," Floating Point Systems, Inc, P.O. Box 23489, Portland, OR.

D. C. Munson, Jr. & B. Liu [1981], "Floating Point error bound in the prime factor FFT," *IEEE Trans. Acoust. Speech Signal Process.* ASSP-29, 877–882.

H. J. Nussbaumer [1982], in *Fast Fourier Transform and Convolution Algorithms*, Springer-Verlag, Berlin, Heidelberg.

J. Oliver [1975], "Stable Methods for Evaluating the Points $\cos(i\pi/n)$," *J. Inst. Maths Applics* 16, 247–257.

A. V. Oppenheim & C. J. Weinstein [Aug. 1972], "Effects of Finite Register Length in Digital Filtering and the Fast Fourier Transform," *Proceedings of the IEEE* 60, 957–976.

M. C. Pease [Apr. 1968], "An Adaptation of the Fast Fourier Transform for Parallel Processing," *J. Assoc. Comput. Mach.* 15, 252–264.

I. Pitas & M. G.Strintzis [1983], "Floating point error analysis of two efficient algorithms used in the fast computation of two-dimensional DFTs," in *Signal Processing II: Theories and Applications*, H. W. Schussler, ed., North-Holland, Amsterdam, 219–222.

S. Prakash & V. V. Rao [Oct. 1982], "Vector Radix FFT Error Analysis," *IEEE Trans. Acoust. Speech Signal Process.* ASSP-30, 808–811.

W. K. Pratt [1978], *Digital Image Processing*, Wiley, New York.

W. H. Press, B. P. Flannery, S. A. Teukolsky & W. T. Vetterling [1986], *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, Cambridge.

L. R. Rabiner [1979a], "FFT Subroutines for Sequences with Spcial Properties," in *Programs for Digital Signal Processing*, Digital Signal Processing Committee, ed., IEEE Press, New York, 1.3-1–1.3-22.

——— [June 1979b], "On the Use of Symmetry in FFT Computation," *IEEE Trans. Acoust. Speech Signal Process.* ASSP-27, 233–239.

G. U. Ramos [Oct. 1971], "Roundoff Error Analysis of the Fast Fourier Transform," *Math. Comp.* 25, 757–768.

G. K. Rivard [June 1977], "Direct Fast Fourier Transform of Bivariate functions," *IEEE Trans. Acoust. Speech Signal Process.* ASSP-25, 250–52.

Y. Saad & M. H. Schultz [June 1985], "Topological Properties of Hypercubes," Research Report YALEU/DCS/RR-389.

——————[Mar. 1986], "Data Communication in Parallel Architectures," Research Report YALEU/DCS/RR-461.

——————[Sept. 1985], "Data Communication in Hypercubes," Research Report YALEU/DCS/RR-428.

C. L. Seitz [Jan. 1985], "The Cosmic Cube," *Comm. ACM* 28, 22–33.

R. C. Singleton [Oct. 1967], "On computing the fast Fourier transform," *Comm. ACM* 10, 647–654.

H. Sloate [Jan. 1974], "Matrix Representations for Sorting and the Fast Fourier Transform," *IEEE Trans. Circuits and Systems* CAS-21, 109–116.

H. S. Stone [Feb. 1971], "Parallel processing with perfect shuffle," *IEEE Trans. Comput.* C-22, 153–161.

P. N. Swarztrauber [1977], "The methods of sysclic reduction, Fourier analysis and the FACR algorithm for the discrete solution of Poisson's equation on a rectangle," *SIAM Rev.* 19, 490–501.

——————[1982], "Vectorizing the FFTs," in *Parallel Computations*, G. Rodrigue, ed., Academic Press, New York, 51–81.

——————[1984a], "FFT algorithms for vector computers," *Parallel Comput.* 1, 45–63.

——————[1984b], "Fast Poisson solvers," *MAA Studies in Numerical Analysis* 24, 319–370.

P. N. Swarztrauber [1986a], "Multiprocessor FFT's," National Center for Atmospheric Research, Boulder, CO, (to appear in Parallel Comput.).

——————[July 1986b], "Symmetric FFTs," *Math. Comp.* 47, 323–346.

C. Temperton [1979], "Direct Methods for the Solution of the Discrete Poisson Equation: Some Comparisons," *J. Comput. Phys.* 31, 1–20.

——————[1980], "On the FACR(l) Algorithm for the Discrete Poisson Equation," *J. Comput. Phys.* 34, 314–329.

——————[1983], "Fast mixed-radix real Fourier transforms," *J. Comput. Phys.* 52, 340–350.

F. Theilheimer [June 1969], "A Matrix Version of the Fast Fourier Transform," *IEEE Transactions on Audio and Electroacoustics* AU-17, 158–161.

T. Thong & B. Liu [1977], "Floating point fast Fourier transform computation using double precision floating point accumulators," *ACM Trans. Math. Soft.* 3, 54–59.

N-K. Tsao [(no date given)], "The "Equivalence" of Decimation in Time and Decimation in Frequency in FFT Computations," Department of Computer Science, Wayne State University, Detroit, MI.

C. F. Van Loan [1987], *FFT's From the Matrix Point of View*, (manuscript).

S. R. Walton [Sept. 1986], "Fast Fourier Transforms on the Hypercube," Ametek Computer Research Division, Arcadia, CA.

Z-D. Wang [1981a], "Harmonic Analysis with a Real Frequency Function—I. Aperiodic Case," *Appl. Math. Comput.* 9, 53–73.

——— [1981b], "Harmonic Analysis with a Real Function of Frequency—II. Periodic and Bounded Cases," *Applied Mathematics and Computation* 9, 153–163.

——— [1981c], "Harmonic Analysis with a Real Frequency Function. III. Data Sequence," *Applied Mathematics and Computation* 9, 245–255.

——— [Aug. 1984], "Fast Algorithms for the Discrete W Transform and for the Discrete Fourier Transform," *IEEE Trans. Acoust. Speech Signal Process.* ASSP-32, 803–816.

Z-D. Wang & B. R. Hunt [1985], "The Discrete W Transform," *Appl. Math. Comput.* 16, 19–48.

C. J. Weinstein [Sept. 1969], "Roundoff Noise in Floating Point Fast Fourier Transform Computation," *IEEE Transactions on Audio and Electroacoustics* AU-17, 209–214.

P. D. Welch [June 1969], "A Fixed-Point Fast Fourier Transform Error Analysis," *IEEE Transactions on Audio and Electroacoustics* AU-17, 151–157.

P. Wiley [June 1987], "A parallel architecture comes of age at last," *IEEE Spectrum* 24, 46–50.

P. Yip & K. R. Rao [1980], "A fast computational algorithm for the discrete sine transform," *IEEE Trans. Comm.* COM-28, 304–310.

——— [1987], "On the Shift Property of DCTs and DSTs," *IEEE Trans. Acoust. Speech Signal Process.* ASSP-35, 404–406.