Spring 1993

Fast Fourier Transforms on Distributed Memory Parallel Machines

Anshu Dubey
Old Dominion University

Recommended Citation
Dubey, Anshu. "Fast Fourier Transforms on Distributed Memory Parallel Machines" (1993). Doctor of Philosophy (PhD), dissertation, Computer Science, Old Dominion University, DOI: 10.25777/jkyf-4c89
https://digitalcommons.odu.edu/computerscience_etds/105

FAST FOURIER TRANSFORMS ON DISTRIBUTED MEMORY PARALLEL MACHINES

by

Anshu Dubey
B.Tech, Indian Institute of Technology, New Delhi, India
M.S. Auburn University, Auburn, Alabama

A Dissertation submitted to the Faculty of Old Dominion University in
Partial Fulfillment of the Requirement for the Degree of

DOCTOR OF PHILOSOPHY
in
COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
May, 1993

Approved by:

Dr. Chester E. Grosch (Advisor)

Dr. Mohammad Zubair (Advisor)

Dr. Larry W. Wilson

Dr. Tom L. Jackson

ABSTRACT

FAST FOURIER TRANSFORMS ON DISTRIBUTED MEMORY PARALLEL MACHINES

Anshu Dubey

Old Dominion University, 1993

One issue which is central in developing a general purpose subroutine on a distributed memory parallel machine is the data distribution. It is possible that users would like to use the subroutine with different data distributions. Thus there is a need to design algorithms on distributed memory parallel machines which can support a variety of data distributions. In this dissertation we have addressed the problem of developing such algorithms to compute the Discrete Fourier Transform (DFT) of real and complex data. The implementations given in this dissertation work for a class of data distributions commonly encountered in scientific applications, known as the block scattered data distributions. The implementations are targeted at distributed memory parallel machines. We have also addressed the problem of rearranging the data after computing the FFT. For computing the DFT of complex data, we use a standard Radix-2 FFT algorithm which has been studied extensively in parallel environment. There are two ways of computing the DFT of real data that are known to be efficient in serial environments: namely (i) the real fast Fourier transform (RFFT) algorithm, and (ii) the fast Hartley transform (FHT) algorithm. However, in distributed memory environments they have excessive communication overhead. We restructure the RFFT and FHT algorithms to reduce this overhead. The restructured RFFT and FHT algorithms are then used in the generalized implementations which work for block scattered data distributions. Experimental results are given for the restructured RFFT and the FHT algorithms on two parallel machines; NCUBE-7 which is a Hypercube MIMD machine and AMT DAP-510 which is a Mesh SIMD machine. The performances of the FFT, RFFT and FHT algorithms with block scattered data distribution were evaluated on Intel iPSC/860, a Hypercube MIMD machine.

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my gratitude to a number of people. First and foremost, my thanks are due to Dr. Grosch and Dr. Zubair for supervising this work, and their unfailing help and encouragement. Dr. Jackson and Dr. Wilson, with their incisive comments, have helped in improving the quality of this dissertation considerably. I would also like to thank Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA for providing access to 32 node Intel iPSC/860 machine. Thanks are also due to the Systems Group at Old Dominion University for keeping the "very quirky" parallel machines running most of the time. I have no words to express my gratitude to my parents, whose vision and dedication inspired me to reach this goal, and to my husband whose sunny outlook on life has helped me through the sometimes stormy waters of graduate school.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Parallel Computation Issues

In developing parallel algorithms for a problem, there are various architectural issues that confront us. The architecture of the machine may be coarse grain (a few powerful processors), or fine grain (a large number of very simple processors). The machine may be SIMD, where every processor in the machine works in lock step with all other processors, or it may be MIMD, where every processor does its own share of work by executing its local code. The memory may be shared by all processors, or each processor may have its own local memory, or there may be a combination of local and shared memory. The interconnection networks differ amongst machines and the interprocessor communication may also be brought about in different ways. All of these factors play an important role in developing algorithms for a parallel machine.

Parallel algorithms usually have more overhead than their sequential counterparts. The overhead may be due to several reasons. If the work is not evenly distributed to all the processors, then the overhead is due to some processors remaining idle for periods of time. Sometimes the overhead may be caused by processors duplicating their computations. In SIMD machines some of the overhead may be due to steps which require different operations at various processors. In MIMD machines it may be caused by the synchronization problem. In distributed memory machines one factor which contributes significantly to the overhead is the interprocessor communication. These are just some of the causes of overhead and they are not unconnected. Reducing one type of overhead may cause another type to increase. For instance a good load distribution among processors may cause much more internode communication and vice versa. Or minimizing replication of computations in different processors may cause some processors to remain idle. A good parallel algorithm must take into account the trade-offs in overhead and find a good balance between them. It should be aimed at decreasing the overall computation time rather than full processor utilization or equal load distribution or other such issues.

1.2 Fast Fourier Transform

Fourier transforms are an important ingredient of mathematical analysis. The discrete version of the Fourier transform, known as the DFT, plays an important role in numerical analysis, with applications such as: digital filtering, calculation of auto- and cross-correlation, the solution of partial differential equations etc. The computation of the DFT from its definition takes $O(n^2)$ time for an input sequence of length n. The fast Fourier transform (FFT) algorithm computes the transform of an n-component sequence in $O(n \log n)$ time. It was first introduced by Cooley and Tukey in 1965 [21]. The FFT algorithm made techniques based on Fourier transform attractive for many applications. A large number of variants of the original Cooley and Tukey algorithm have been proposed since 1965 [24].

The standard FFT algorithm computes the DFT of a sequence of complex data. In many applications, such as the solution of PDE's, we need to compute DFT of real data only. For such applications one can use the standard FFT algorithm by taking the imaginary part of the input to be zero. However, such an approach leads to a lot of redundant computation, since the DFT of real data can be divided into two halves which are complex conjugates of each other. The real fast Fourier Transform algorithm (RFFT) uses this property to reduce computation [7, 44]. Alternatively, one can use the fast Hartley transform (FHT) algorithm [12] for computing the Fourier transform of real data. The FHT algorithm provides advantage over the FFT algorithm by eliminating all complex arithmetic. A comparative study of algorithms for computing DFT of real data can be found in [42]. In most applications the RFFT algorithm is slightly faster than the FHT algorithm. However, where both forward and reverse transforms are needed, the FHT algorithm is the more attractive one since it involves identical computation for both forward and inverse transforms.

Even with the FFT algorithm only very limited real life problems can be solved on conventional machines. To solve even the moderately sized problems, one has to use the so called "high performance computers." There are two classes of such high performance machines. The machines in the first class use higher clock rates and other technological advances along with carefully designed architecture and software support to achieve high computation speeds. The second class of high performance machines achieve high speeds through parallelism. A parallel machine typically has a number of identical processing units. The total work is divided into smaller tasks and these tasks are distributed among the processing units which execute them in parallel. The parallel machines can also use some of the features of the first class of high performance machines to achieve higher speeds. The main advantage of the high computation speed machines over parallel machines is that achieving good performance is transparent to the user. Part of this advantage comes from optimizing compilers while the other part comes through the use of highly optimized subroutine libraries. In order to get good performance out of a parallel machine the users have to be aware of the architecture of the machine. Further, there are very few parallel subroutine libraries available to the users. Considering the potential of parallel machines for high performance computing, it is desirable to look into the issues related to programming them and providing the kind of software support that is available on other high performance machines.

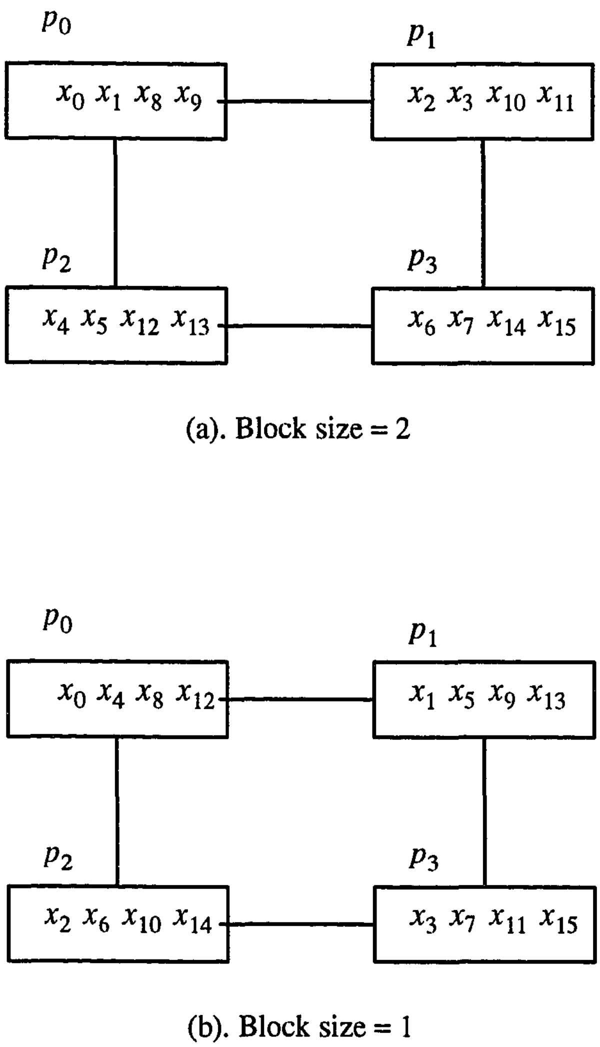
1.3 Data Distributions

To develop a general purpose subroutine on a distributed memory parallel machine one has to address the issue of data distribution in addition to all the issues mentioned earlier in this section [36]. It is possible that different users may wish to use the routine with different data distributions. Typically, users determine their data distribution based on the over all application requirements, which could vary from user to user. Thus, it is extremely important to design schemes on distributed memory parallel machines which can support a variety of data distributions.

There are two possible approaches to this problem. The first one is to design a scheme for a specific data distribution which gives optimal performance, along with a set of basic communication subroutines to convert a user supplied data distribution to the specific data distribution. This approach has the problem of rearranging the user data initially which is quite costly on distributed memory parallel machines. The second approach is to design a scheme which works well for arbitrary data distributions. The second approach is obviously extremely difficult to achieve. A compromise between these two extremes is to design algorithms that support a class of data distributions. A common set of data distributions, referred to as block scattered distributions, has been identified by Walker and Dongarra [49] as very useful for distributed memory parallel machines. Block scattered distributions encompass the two most common data distributions; the linear data distribution and the scattered data distribution. For a one dimensional data set, a block scattered distribution is specified by the block size. The data are divided into a set of equal sized blocks. A block j is mapped to node (j mod p), where p is the number of nodes. For example, two data distributions for a one dimensional array of 16 data values on a 4 node machine with two different block sizes are shown in Figure 1.1.

1.4 Overview

In this dissertation we present algorithms for computing DFT of real and complex data that work for block scattered data distributions. These algorithms work for all block sizes without requiring any initial redistribution of data. For computing DFT of complex data the FFT algorithm is used. The FFT algorithm is well suited for most parallel environments. It is possible to distribute work among


p_0
$x_0\ x_1\ x_8\ x_9$
p_1
$x_2\ x_3\ x_{10}\ x_{11}$
p_2
$x_4\ x_5\ x_{12}\ x_{13}$
p_3
$x_6\ x_7\ x_{14}\ x_{15}$
p_0
$x_0\ x_4\ x_8\ x_{12}$
p_1
$x_1\ x_5\ x_9\ x_{13}$
p_2
$x_2\ x_6\ x_{10}\ x_{14}$
p_3
$x_3\ x_7\ x_{11}\ x_{15}$

(a). Block size = 2

(b). Block size = 1

Figure 1.1: Block scattered data distributions for two block sizes.

processors such that the load is balanced and no processor is idle. The internode communication pattern is also very regular and does not cause excessive overhead [14, 18, 26, 29, 34, 47]. However, the RFFT and FHT algorithms do not work very well on distributed memory parallel machines. Their communication patterns result in excessive communication overhead, which may even offset their computational advantage on distributed memory parallel machines [37, 39]. We present a restructuring of the RFFT and FHT algorithms which eliminates their excessive communication overhead, while retaining their computational advantage. The algorithms for computing DFT of real data with block scattered distribution are based upon the restructured RFFT and FHT algorithms. We have also addressed the issue of rearranging data after computation such that the output data have the same distribution as the input. The motivation for rearrangement comes from problems such as solution of partial differential equations using spectral techniques which require the final data distribution to be identical to the initial one.

The dissertation is organized in six chapters including this one. Chapter 2 includes a discussion of the previous work in the area of parallel FFT algorithms. It also gives the definitions relevant to this work. The restructuring of RFFT and FHT algorithms is described in Chapter 3. The restructured FHT and RFFT algorithms, along with the FFT algorithm were implemented on two distributed memory parallel machines with different architectures. All three implementations work only for a specific data distribution. Chapter 3 also describes the results of the experiments with these implementations to verify the superiority of the FHT and RFFT algorithm over the FFT algorithm for computing DFT of real data.

An FFT implementation on a distributed memory parallel machine for block scattered data distributions with different block sizes is given in Chapter 4. Chapter 5 gives the RFFT and FHT implementations for block scattered data distributions using the restructured algorithms from Chapter 3. As with the FFT algorithm, these implementations support different block sizes. However, a minimum of 4 blocks are required per node irrespective of the block size. This requirement comes from the grouping formed in the restructured algorithms. All the algorithms are independent of the number of processors in the machine as long as the data size is greater than the number of processors. The performances of all three implementations were evaluated on the Intel iPSC/860. The conclusions and the scope for further study in this area are discussed in chapter 6.

Chapter 2

Background

2.1 Definitions

In this chapter we give the definitions and history of the FFT algorithms. A number of variants of the FFT algorithms exist in literature. We give the definitions relevant to this work only since it is not possible to include all definitions here. A discussion of the previous work by other researchers is also included in this chapter.

2.1.1 Discrete Fourier Transform

The DFT, $X(k)$, of an N-point sequence $x(r)$ is defined as,

$$X(k) = 1/N \sum_{r=0}^{N-1} x(r)e^{-j2\pi rk/N}, 0 \leq k < N, \tag{2.1}$$

where $j = \sqrt{-1}$, and N is a power of 2.

FFT Algorithm

There are two major classes of the FFT algorithms, namely; **decimation in time (DIT-FFT)** and **decimation in frequency (DIF-FFT**. The two classes of the FFT algorithm are described here briefly. (For details one can refer to [24]). For the DIT-FFT algorithm the N-point sequence $x(r)$ is divided into two $(N/2)$-point sequences $x_1(r)$ and $x_2(r)$ as the odd and even elements of $x(r)$ respectively; i.e.

$$x_1(r) = x(2r), r = 0, 1, 2, ...N/2 - 1, \tag{2.2}$$

$$x_2(r) = x(2r+1), r = 0, 1, 2, ...N/2 - 1. \tag{2.3}$$

We then recursively compute $X_1(k)$ and $X_2(k)$, the DFT's of $x_1(r)$ and $x_2(r)$ respectively. The recursion stops when the DFT of a 1-point sequence, which is the element itself, is required. The two sequences $X_1(k)$ and $X_2(k)$ are then merged to generate $X(k)$ using the following expressions

$$X(k) = X_1(k) + \omega_N^k X_2(k), \quad 0 \leq k < N/2, \tag{2.4}$$

$$X(k) = X_1(k - N/2) - \omega_N^k X_2(k - N/2), \quad N/2 \leq k < N. \tag{2.5}$$

where $\omega_N^k = e^{-j2\pi k/N}$.

For the DIF-FFT algorithm the N-point sequence $x(r)$ is divided into two halves, $x_1(r)$ and $x_2(r)$ so that the the transformed sequence can be written as

$$X(2k) = \sum_{r=0}^{(N/2-1)} [x_1(r) + x_2(r)]{\omega_N}^{2kr} \tag{2.6}$$

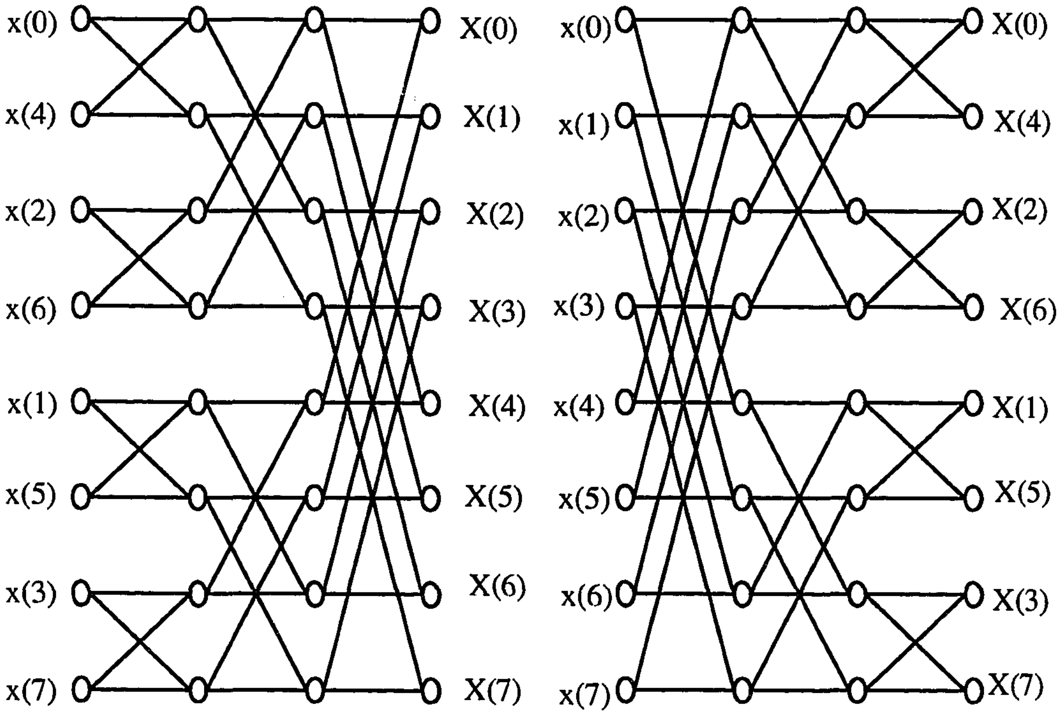
$$X(2k+1) = \sum_{r=0}^{(N/2-1)} [x_1(r) - x_2(r)]\omega_N^r{\omega_N}^{2kr}, \; k = 0, 1..N/2-1. \tag{2.7}$$

These equations represent two $N/2$ point DFT's of sequences $[x_1(r) + x_2(r)]$ and $[x_1(r) - x_2(r)]\omega_N^r$. The process is then repeatedly applied to the two subsequences. The flow graphs for the DIT FFT and DIF-FFT for input sequence of length 8 are shown in Figure 2.1. Notice that the DIT-FFT algorithm requires the input sequence to be in bit-reversed order (ordering obtained by reversing the bits in the binary representation of the data item indices) and returns the output in sequential order. The DIF-FFT algorithm requires the input in sequential order and returns the output in a bit reversed order. The basic units of computation for the FFT algorithms are butterflies shown in Figure 2.2.

RFFT Algorithm

The RFFT algorithm described here is derived from the DIT-FFT algorithm. It is different from the FFT algorithm at the merge step. In the RFFT algorithm the N-point sequence $X(k)$ is obtained from two $N/2$-point sequences $x_1(k)$ and $x_2(k)$ as follows

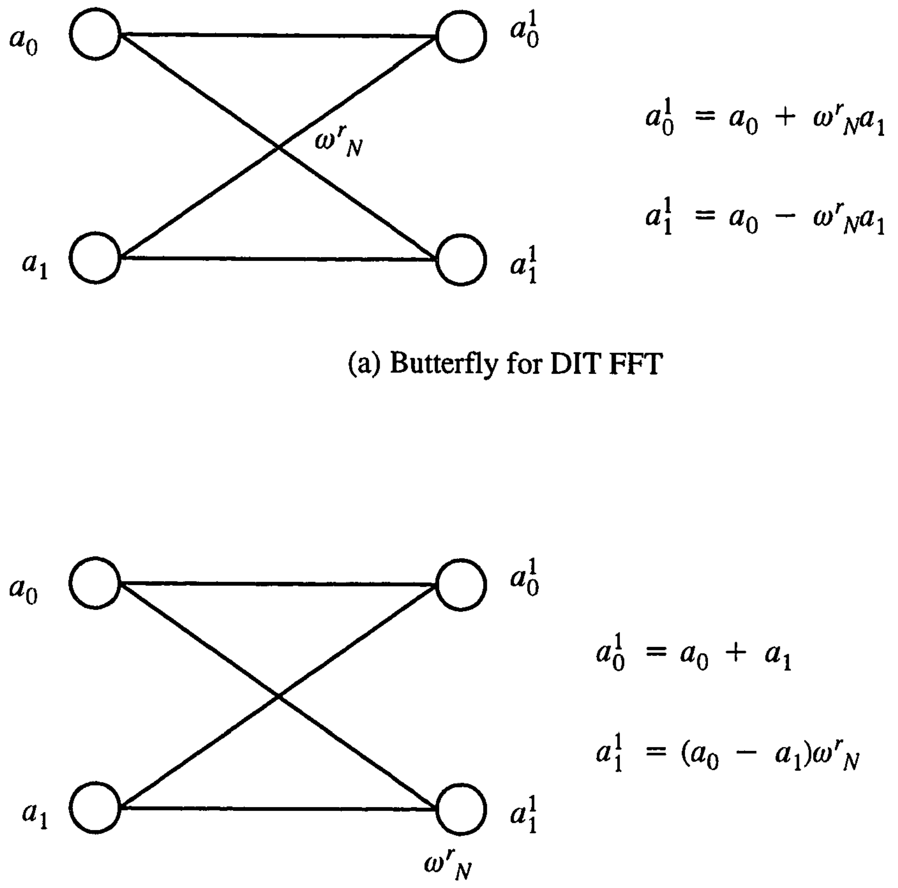
$$X(k) = X_1(k) + \omega_N^k X_2(k), \quad 0 \leq k \leq N/2, \tag{2.8}$$



(a) DIT–FFT Flowgraph.

(b) DIF–FFT Flowgraph.

Figure 2.1: Flow Graphs of 8-point DIT and DIF FFT algorithms.


a_0
a_0^1
$\omega^r{}_N$
$a_0^1 = a_0 + \omega^r{}_N a_1$
$a_1^1 = a_0 - \omega^r{}_N a_1$
a_1
a_1^1
a_0
a_0^1
$a_0^1 = a_0 + a_1$
$a_1^1 = (a_0 - a_1)\omega^r{}_N$
a_1
a_1^1
$\omega^r{}_N$

(a) Butterfly for DIT FFT

(b) Butterfly for DIF FFT

Figure 2.2: Butterflies for the FFT algorithms.

$$X(k) \quad = \quad X^*(N-k) \quad N/2 < k < N \tag{2.9}$$

where $X^*(N-k)$ is the complex conjugate of $X(N-k)$.

2.1.2 Hartley Transform

The discrete Hartley transform, $X(k)$, of an N-point sequence $x(r)$ is defined as [12]

$$X(k) = 1/N \sum_{r=0}^{N-1} x(r)\{\cos(2\pi rk/N) + \sin(2\pi rk/N)\}, 0 \leq k < N \tag{2.10}$$

The even and odd parts of the DHT are given by

$$E(k) = (X(k) + X(N-k))/2, \tag{2.11}$$

$$O(k) = (X(k) - X(N-k))/2. \tag{2.12}$$

The even and odd parts of the DHT can be combined to give real and imaginary parts of the DFT [12].

$$X(k) = E(k) - jO(k). \tag{2.13}$$

FHT Algorithm

The fast Hartley transform (FHT) differs from the FFT algorithm only at the merge step. For the FHT algorithm the merging of two sequences $X_1(k)$ and $X_2(k)$of length $N/2$ each to give a sequence of length N is given by,

$$X(k) \quad = \quad X_1(k) + X_2(k)\cos(2\pi k/N)$$

$$+X_2(N/2-k)\sin(2\pi k/N), \quad 0 \leq k < N/2, \tag{2.14}$$

$$X(k) = X_1(k-N/2) + X_2(k-N/2)\cos(2\pi k/N)$$

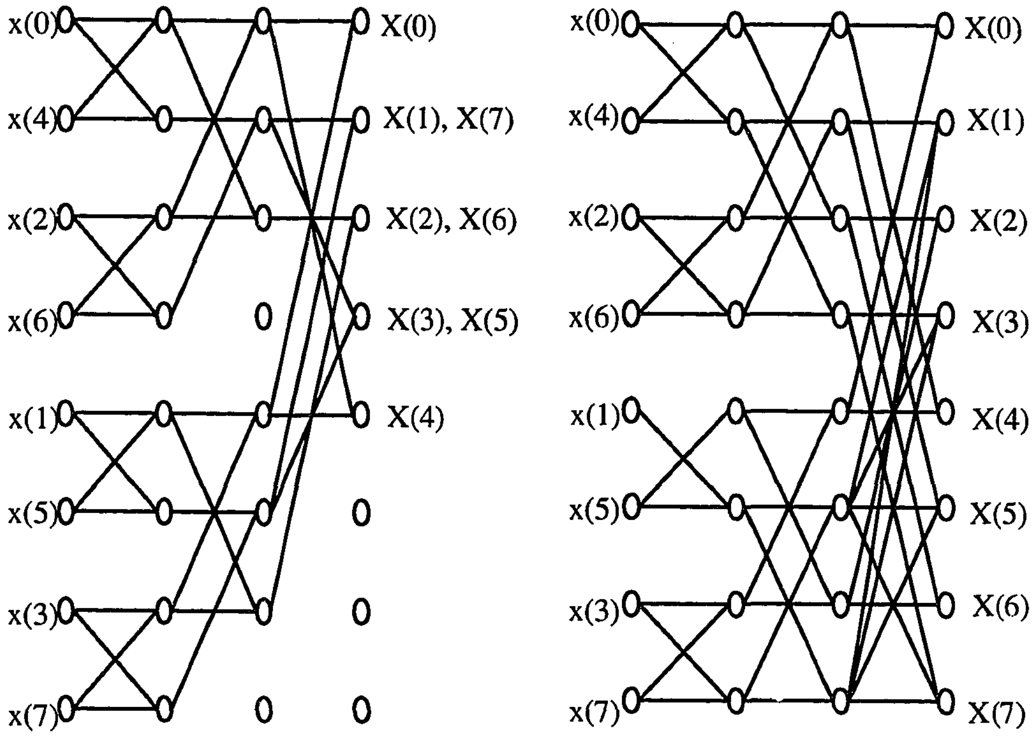
$$+X_2(N-k)\sin(2\pi k/N), \quad N/2 \leq k < N. \tag{2.15}$$

The flow graph for the RFFT and FHT algorithms are given in Figure 2.3. It can be seen from this figure that it is very difficult to identify a basic unit of computation for these algorithms.

2.2 Previous Work

Three distinct phases can be identified in the history of parallelization of the FFT algorithm. The first phase started in the late 60's and continued up to the mid 70's. Majority of the implementations reported in this phase were based on hypothetical or "paper and pen" machines [8, 9, 38]. One work even tried to match the FFT algorithm to the concept of associativity derived from memory design [50]. These early works were almost always targeted at special purpose and highly constrained architectures. Despite the lack of machines on which to test these ideas, they still made a significant contribution towards understanding the parallelism inherent in the FFT algorithm.

The second major phase started with the arrival of commercial vector processors. Korn and Lambiotte [35] discussed an implementation on CDC Star 100. They identified a major drawback of the FFT algorithm in relation to vector pro-



(a) RFFT Flowgraph

(b) FHT Flowgraph

Figure 2.3: Flow Graphs for the RFFT and FHT algorithms.

cessing. The size of vector in a single FFT computation is not uniform and can be very small in some stages. Hence it was not possible to exploit the vector processing capabilities throughout the algorithm. They suggested computation of multiple independent transforms as a way of countering this problem. Multiple independent transforms arise in a number of application for example computing multi-dimensional transforms. The corresponding elements of each data sequence to be transformed are processed as vectors in such a situation. Fornberg [23] improved upon this work and suggested evaluation and storing of the multiplication coefficients before computing the DFT. The coefficients were extracted from the table when they were needed for computations. This approach proved to be very useful when a number of DFT's of same size were to be computed at different times. Swarztrauber [43] divided the single transform into multiple transforms to exploit the capabilities of the vector processor. Agarwal and Cooley [1] identified a second major problem with efficient implementations of the FFT algorithm on vector machines, namely, cache use. As long as the data size for the FFT was smaller than the size of cache memory, the performance of the algorithm was very good. However, for data sizes bigger than the cache size, the performance deteriorated rapidly. This is because in most variants of the FFT algorithm, in every stage a data item is required for a few computations. By the time it is required in the next stage it would have been thrown out of the cache. Hence the ratio of cache miss to cache hit is fairly high. There are several other vector FFT implementations in the literature using different approaches [2, 4, 6, 17, 48].

The third phase in the development of parallel FFT algorithms has overlapped with the second phase. Some of the challenges faced in this phase are similar to those in the second phase while some others are quite different. This phase came with the advent of multiprocessor machines in the market. These machines come in a wide variety of architectures including SIMD array processors to MIMD hypercubes to massively parallel connection machines. FFT implementations reported on these machines are necessarily different from each other. Some of the earliest work in this phase has been on array processors [27, 25] and towards developing FFT processors [19, 20]. The early work on multiprocessor machines addressed the issues involved in mapping the data onto the processors. Some of these issues are: the relation between the number of data points and the number of processors, overheads of data organization when the number of data points is more than the number of processors and the degree of parallelization achievable with different data distributions [27, 25, 26]. These issues continue to be relevant for all the machines available commercially today.

Since the mid 80's, a great deal of attention has been given to FFT implementations on machines with hypercube architectures for two reasons. One of the reasons is the ease with which the FFT algorithm maps onto a hypercube when the size of the input sequence is a power of 2. The second reason is that a significant number of commercially available machines have hypercube based interconnection networks. On these machines the data communication costs are an important factor in the overall cost of computation. Johnson et.al [29, 31] discussed the computation of FFT on boolean cubes and other similar interconnection networks. Johnson et.al [30, 33] and Kamin and Adams [34] gave implementations on a connection machine. Swarztrauber [45] used index-digit permutations to address the issue of computing ordered transforms (where both the initial and final data distribution are in same order). In a later work with Tong [47], he pointed out a cyclic order data distribution which results in less communication cost than the natural order on connection machine for ordered FFT. He also gave an implementation for an arbitrary size data (not power of 2) on a hypercube [46]. Chamberlain [18] discussed computing FFT of an initial data distribution which is in a Gray code ordering rather than natural ordering. He proved it would require communication between nodes at most a distance two apart.

Relatively less attention has been given to shared memory machines. Swarztrauber [45] discussed implementations on Cray-XMP and Alliant FX-8 which are shared memory vector processors. Briggs et.al [14] discussed FFT methods on HEP computer which is an MIMD shared memory machine. More recently Averbuch et.al [3] reported the results of their implementation on an experimental shared memory machine MMX.

One can arrive at two conclusions by looking at the existing literature for the FFT algorithms. Even though a lot of attention has been given by the researchers to the parallelization of the complex FFT algorithm, the RFFT algorithm has been largely ignored. Also, there has been very little effort towards finding parallel implementations that can compute the Fourier transform of the variety of data distributions useful to the scientific community.

Chapter 3

Computing Fourier Transform of Real Data on Distributed Memory Parallel Machines

3.1 Introduction

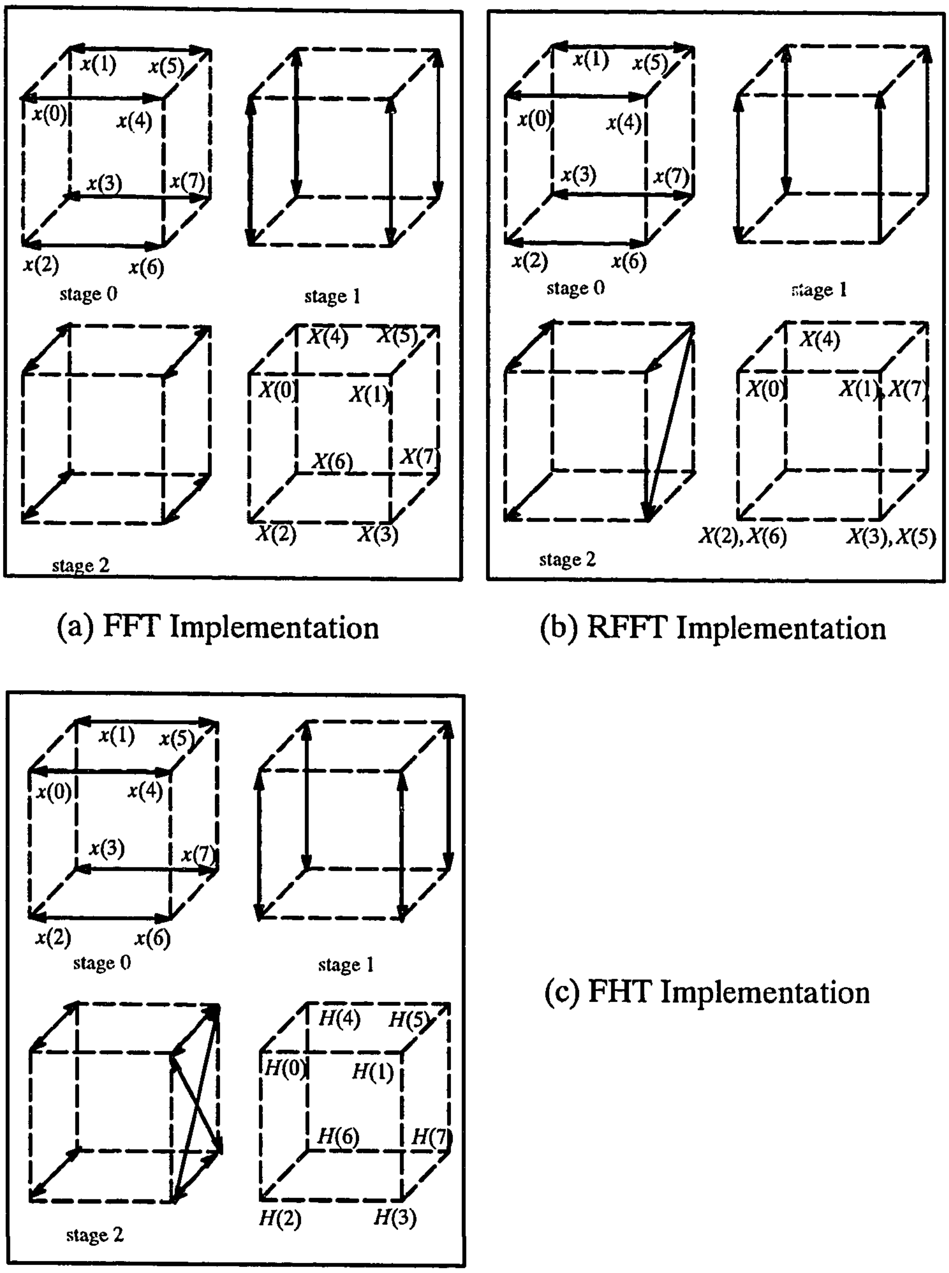
On a sequential computer it has been shown that both the RFFT and the FHT algorithms are faster than the FFT algorithm [7, 12, 42, 44]. However, it is not obvious that the same is true on parallel machines. The communication patterns of the RFFT and the FHT algorithms, which are critical to the cost of implementations on distributed memory parallel machines, are different from those of the FFT algorithm (see Figures 2.1 and 2.3). We assume that in a parallel environment a processor is assigned to each node of the flow-graph. A link between two nodes of the flow-graph represents communication between the corresponding processors. A careful observation of Figure 2.3 also indicates that for an identical data distribution, the communication pattern of the RFFT algorithm is a subset of the communication pattern of the FHT algorithm. It can be easily verified that this is in general true. The communication patterns of these algorithms are unsuitable for MIMD and SIMD machines [39, 37]. For MIMD machines these patterns require additional communication overhead. For SIMD machines there is an added disadvantage of different data movements at different processors. Hence a simple mapping of data items to processors is not likely to be efficient.

In this chapter we present a restructuring of the RFFT and FHT algorithms such that their communication patterns become similar to that of the FFT algorithm. The restructuring is such that the computational advantage of these algorithms is also retained. The restructured algorithms are based on the observation that at any stage of the RFFT and the FHT algorithms, a group of four data points uniquely determine four data points of the next stage (a similar grouping of data has been suggested before [4, 12, 15, 42] in the context of minimizing the number of arithmetic operations). This restructuring makes these algorithms suitable for most of the contemporary distributed memory parallel architectures. We tested the suitability of the restructured algorithms by implementing them on two different parallel machines. One is an MIMD distributed memory machine with a hypercube interconnection network, the NCUBE. The second one is the AMT-DAP which is an SIMD distributed memory machine with a mesh architecture. Our results indicate that implementations of the FHT and RFFT algorithms run about $25 - 40\%$ faster than the FFT algorithm on these machines.

3.2 Simple FFT-like Implementation

We have seen that the communication patterns of the FHT and the RFFT algorithms are different from those of the FFT algorithm. This necessitates a different approach for their implementation on parallel machines. We illustrate this by considering the implementations of 8-point DIT-FFT, RFFT and FHT algorithms on a 3-cube and a 4×2 mesh. The communication pattern of the FFT algorithm implementation on the cube is shown in Figure 3.1(a) and it follows directly from Figure 2.1(a). Similarly the communication patterns of the RFFT and FHT algorithms on a cube, implemented from the flow graphs in Figure 2.3 are shown in Figures 3.1(b) and 3.1(c) respectively. (An arrow between two nodes indicates a corresponding data transfer.) It can be seen from Figure 3.1 that for the RFFT and FHT algorithms, Stage 2 (computing a sequence of length 8) requires communication between two nodes which are a distance two apart (the diagonal transfer on the face of the cube shown in Figure 3.1(b) and 3.1(c)). It is easy to prove that in general if a stage is computing a sequence of length 2^m, where $m > 2$, then it requires communication between two nodes which are a distance $m - 1$ apart.

For the mesh implementation of the algorithms, the input sequence is distributed linearly along the processors. The communication patterns generated by



(a) FFT Implementation

(b) RFFT Implementation

(c) FHT Implementation

Figure 3.1: Implementations of 8-point FFT, RFFT and FHT algorithms on a 3-cube.

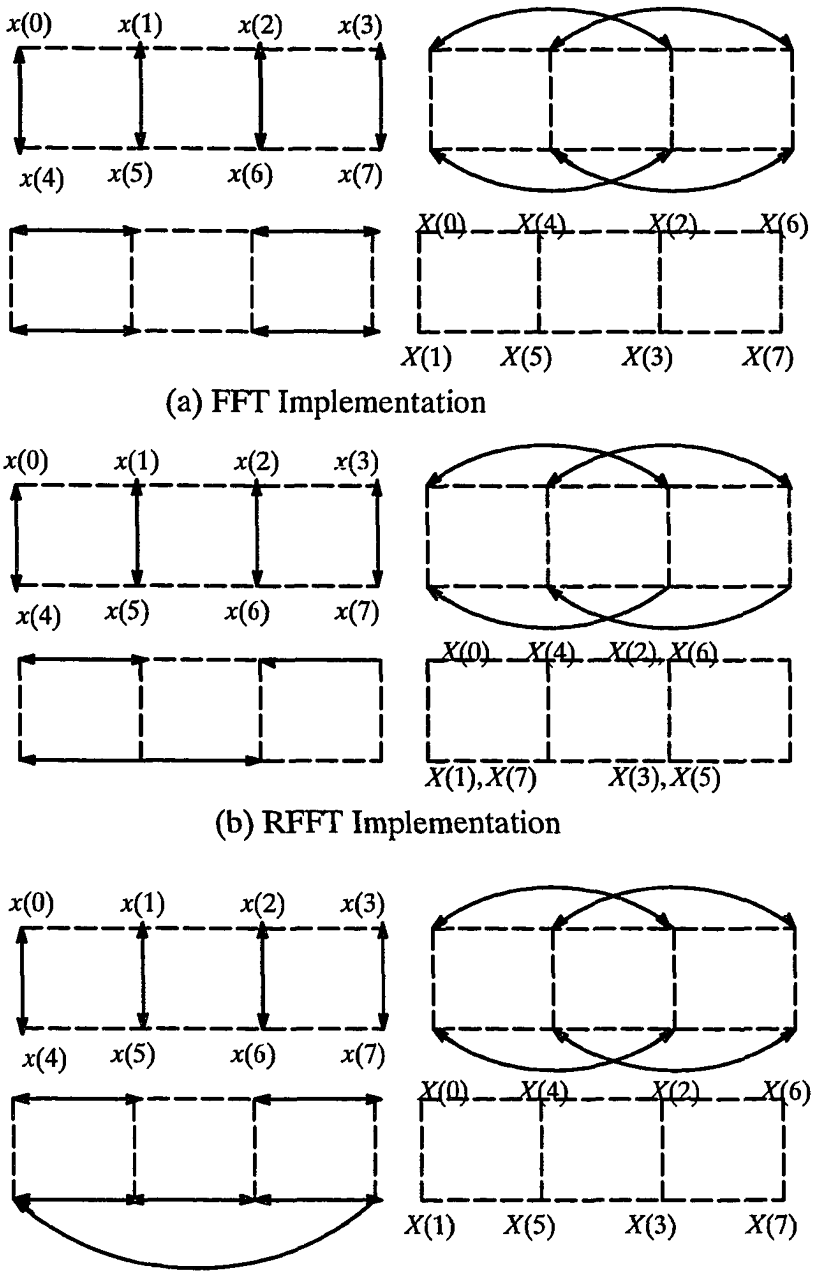
this distribution are shown in Figure 3.2. Tt is obvious from Figures 3.1 and 3.2 that the implementations with simple data mapping are not efficient for the RFFT and FHT algorithms. Since the communication pattern of the RFFT algorithm is a subset of that of the FHT algorithm, an efficient mapping of the FHT algorithm would also be efficient for the RFFT algorithm. The two restructured algorithms differ only in the computation. Hence we discuss only the restructured FHT algorithm in the next two sections.

3.3 Restructured FHT Algorithm

The restructured FHT algorithm consists of an initialization step followed by a number of stages (in general for an N-point sequence there are $\log(N) - 2$ stages) where input to a stage is a set of groups consisting of four data points each. A stage in the restructured FHT algorithm, like the FFT algorithm, merges two n-point sequences X_1 and X_2 to form a 2n-point sequence X. The output of the final stage is the DHT of the input data. The initialization step partitions the input into groups of four data points each and computes 4-point DHT for each of them, which becomes the input for stage 1. A stage consists of two phases.

Exchange Phase

Forms a new set of four data point groups by exchanging data between two groups



(a) FFT Implementation

(b) RFFT Implementation

(c) FHT Implementation

Figure 3.2: Implementations of 8-point FFT, RFFT and FHT algorithms on 4 × 2 mesh.

of the original set.

Computation Phase

Processes the four data points in each group obtained after the exchange phase to generate four new data points. These four new data points form a group for the next stage.

Before giving the details of the restructured FHT algorithm we introduce some necessary notations and define sets of four data point groups, henceforth referred to as groupings.

3.3.1 Notations and Grouping Definitions

Notations

The two n-point sequences to be merged in a stage i are denoted by X_1^i and X_2^i; and the resulting $2n$-point sequence by H^{i+1}, where $n = 2^{i+1}$. For brevity we also introduce the following notations.

$$\begin{aligned}
q_1(k) &= k \bmod n/4, \\
q_2(k) &= n/2 - k, \\
q_3(k) &= n/2 + k \bmod n/4, \\
q_4(k) &= n - k,
\end{aligned}$$

$$q_5(k) \quad = \quad k \bmod n/2.$$

For convenience, the argument k will be dropped in all the subsequent references to $q_j(k)$'s.

Grouping Definitions

Grouping G^i

A grouping, G^i is composed of groups $G^i(k), \quad 1 \leq k \leq n/4$ consisting of four elements from H^i defined by

$$G^i(k) = [H^i(q_1), H^i(q_2), H^i(q_3), H^i(q_4)] \tag{3.1}$$

Grouping G^i_1

A grouping G^i_1 is composed of groups $G^i_1(k), \quad 1 \leq k \leq n/4$ consisting of four elements from X^i_1, given by

$$G^i_1(k) = [X^i_1(q_1), X^i_1(q_2)X^i_1(q_3), X^i_1(q_4)]. \tag{3.2}$$

Grouping G^i_2

A grouping G^i_2 is composed of groups $G^i_2(k), \quad 1 \leq k \leq n/4$ consisting of four elements from X^i_2, given by

$$G^i_2(k) = [X^i_2(q_1), X^i_2(q_2), X^i_2(q_3), X^i_2(q_4)]. \tag{3.3}$$

Grouping G^i_{12}

A grouping, G^i_{12}, is defined as the result of an *exchange operation*, '<=>', between G^i_1 and G^i_2. The exchange operation <=> is such that $G^i_1(k)$ <=> $G^i_2(k)$ implies an exchange of two elements of $G^i_1(k)$ with two elements of $G^i_2(k)$. The elements exchanged depend on the value of k. For $1 \leq k < n/4$, $X^i_1(q_2)$ and $X^i_1(q_3)$ of G^i_1 are exchanged with $X^i_2(q_1)$ and $X^i_2(q_4)$ of G^i_2; and for $k = n/4$, $X^i_1(q_1)$ and $X^i_1(q_3)$ of G^i_1 are exchanged with $X^i_2(q_2)$ and $X^i_2(q_4)$ of G^i_2. As a result the following four sets define the grouping G^i_{12}.

$$[X^i_1(q_1), X^i_2(q_1), X^i_1(q_4), X^i_2(q_4)], \quad 1 \leq k < n/4$$

$$[X^i_1(q_2), X^i_2(q_2), X^i_1(q_3), X^i_2(q_3)], \quad 1 \leq k < n/4$$

$$[X^i_1(q_1), X^i_2(q_1), X^i_1(q_3), X^i_2(q_3)], \quad k = n/4$$

$$[X^i_1(q_2), X^i_2(q_2), X^i_1(q_4), X^i_2(q_4)], \quad k = n/4.$$

The new groups defined by $G^i_{12}(k)$ can also be written as

$$G^i_{12}(k) = [X_1(q_5), X_2(q_5), X_1(q_4), X_2(q_4)], \quad 1 \leq k < n/2. \tag{3.4}$$

This can be verified by observing that the above equation can be written as

$$G^i_{12}(k) = \begin{cases} [X^i_1(q_5), X^i_2(q_5), X^i_1(q_4), X^i_2(q_4)], & 1 \leq k < n/4 \\ [X^i_1(q_2), X^i_2(q_2), X^i_1(q_4), X^i_2(q_4)], & k = n/4 \\ [X^i_1(q_2), X^i_2(q_2), X^i_1(q_3), X^i_2(q_3)], & 1 \leq k < n/4 \\ [X^i_1(q_5), X^i_2(q_5), X^i_1(q_3), X^i_2(q_3)], & k = n/2 \end{cases} \tag{3.5}$$

where:

$$[X_1^i(q_2), X_2^i(q_2), X_1^i(q_4), X_2^i(q_4)], k = n/4$$

is identical to

$$[X_1^i(q_5), X_2^i(q_5), X_1^i(q_4), X_2^i(q_4)], k = n/4,$$

and

$$[X_1^i(q_2), X_2^i(q_2), X_1^i(q_3), X_2^i(q_3)], 1 \leq k < n/4$$

is identical to

$$[X_1^i(q_5), X_2^i(q_5), X_1^i(q_4), X_2^i(q_4)], n/4 < k < n/2.$$

The groupings G_1^i and G_2^i define the sequences X_1^i and X_2^i respectively. These sequences form the input for stage i of the restructured algorithm. The grouping G_{12}^i is the set of new groups formed in the exchange phase of the algorithm where four elements of every group uniquely determine four new data points. It should be noted here that this property of grouping G_{12}^i forms the basis of restructuring. The new data points are computed in the computation phase and they form the grouping G^{i+1}, which is also the output of stage i.

3.4 Algorithm

Initialization

(i) Rearrange the data in a bit reversed order.

(ii) Partition the N-point sequence, $x(n)$, into $N/4$ groups given by,

$$[x(i), x(N/2+i), x(N/4+i), x(3N/4+i)], \quad 0 \leq i < N/4$$

(iii) Compute (4-point) Hartley transforms for each of these groups. The resultant groups are the input for Stage 1.

The exchange and computation phases of a Stage i are given below.

Exchange Phase

Form $G^i_{12} through G^i_1(k) <=> G^i_2(k)$.

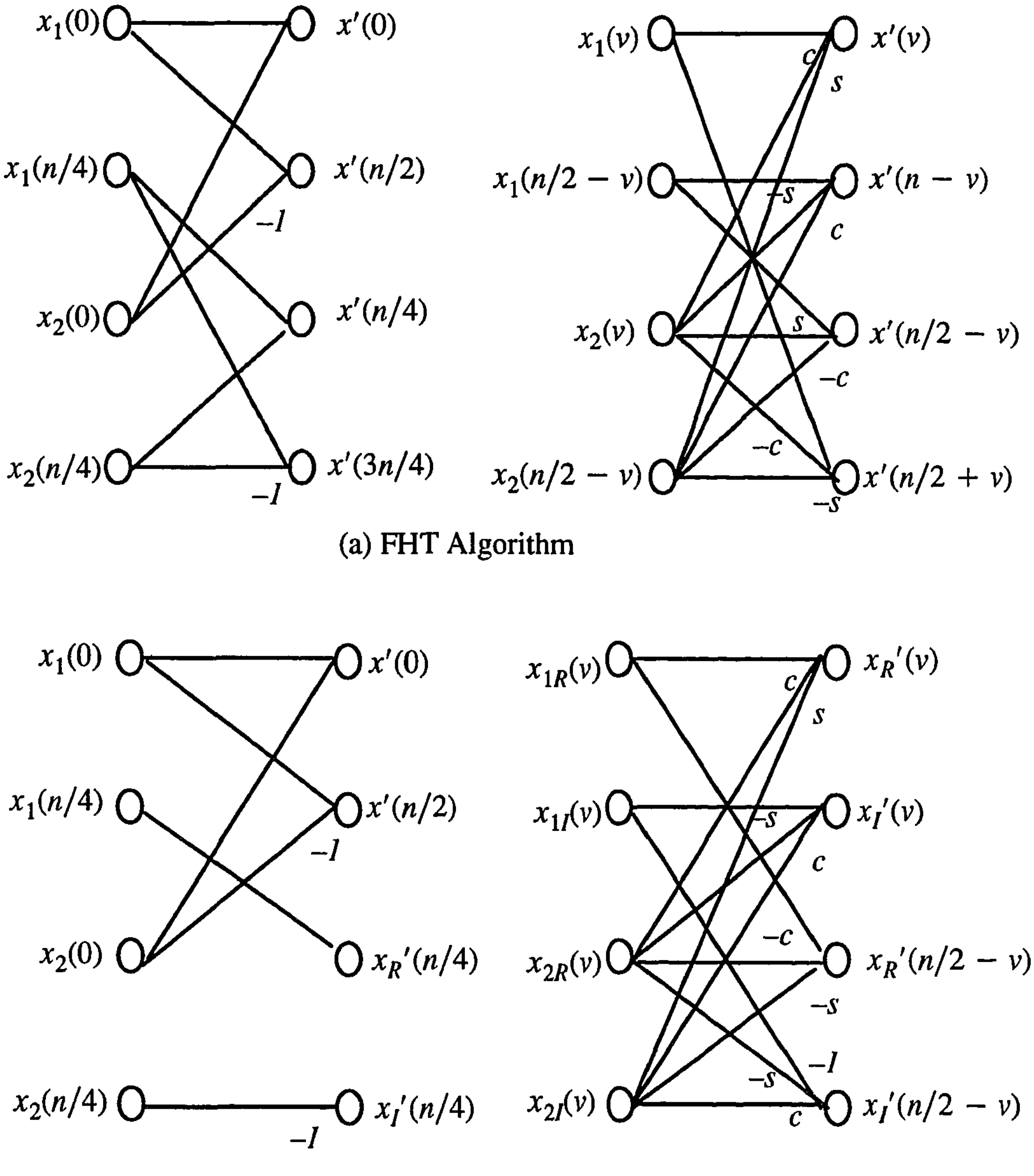
Computation Phase

Compute G^{i+1} from G^i_{12} using Eq. 2.14-2.15.

Example.

The basic computation units of the restructured FHT and RFFT algorithms are shown in Figure 3.3.

We illustrate the communication pattern of the restructured algorithm with the help of an example of 16-point FHT (Figure 3.4). For Stage 1 we assume



(a) FHT Algorithm

(b) RFFT Algorithm

Figure 3.3: Basic computation units of the restructured RFFT and FHT algorithms.

that all steps necessary to generate sequences of length 4 have been executed and the two such sequences are to be combined. It is seen from Figure 3.4 that the communication requirements of the restructured FHT algorithms are very similar to those of the FFT algorithm (see Figure 2.1).

Computing DFT from the restructured FHT.

The groupings suggested for the restructured FHT algorithm have another nice property. The four points of DHT in a group, after the final stage, are sufficient to generate the four corresponding points of the DFT. To see this consider the grouping $G^k(k)$, $k = \log N - 1$, of data points after the final stage.

$$G^k(k) = [H^k(q_1), H^k(q_2), H^k(q_3), H^k(q_4)], \quad 1 \leq k \leq N/4 \tag{3.6}$$

It is observed from Eq. 2.11-2.13 that such a group of four DHT points directly gives the four corresponding points of DFT.

$$\begin{aligned}
X(q_1) &= \{H(q_1) + H(q_4) - j[H(q_1) - H(q_4)]\}/2, \quad 1 \leq k < N/4 \\
X(q_1) &= H(q_1), \quad k = N/4 \\
X(q_2) &= \{H(q_2) + H(q_3) - j[H(q_2) - H(q_3)]\}/2, \quad 1 \leq k < N/4 \\
X(q_2) &= \{H(q_2) + H(q_4) - j[H(q_2) - H(q_4)]\}/2, \quad k = N/4 \\
X(q_3) &= \{H(q_3) + H(q_2) - j[H(q_3) - H(q_2)]\}/2, \quad 1 \leq k < N/4 \\
X(q_3) &= H(q_3), \quad k = N/4 \\
X(q_4) &= \{H(q_4) + H(q_1) - j[H(q_4) - H(q_1)]\}/2, \quad 1 \leq k < N/4
\end{aligned}$$

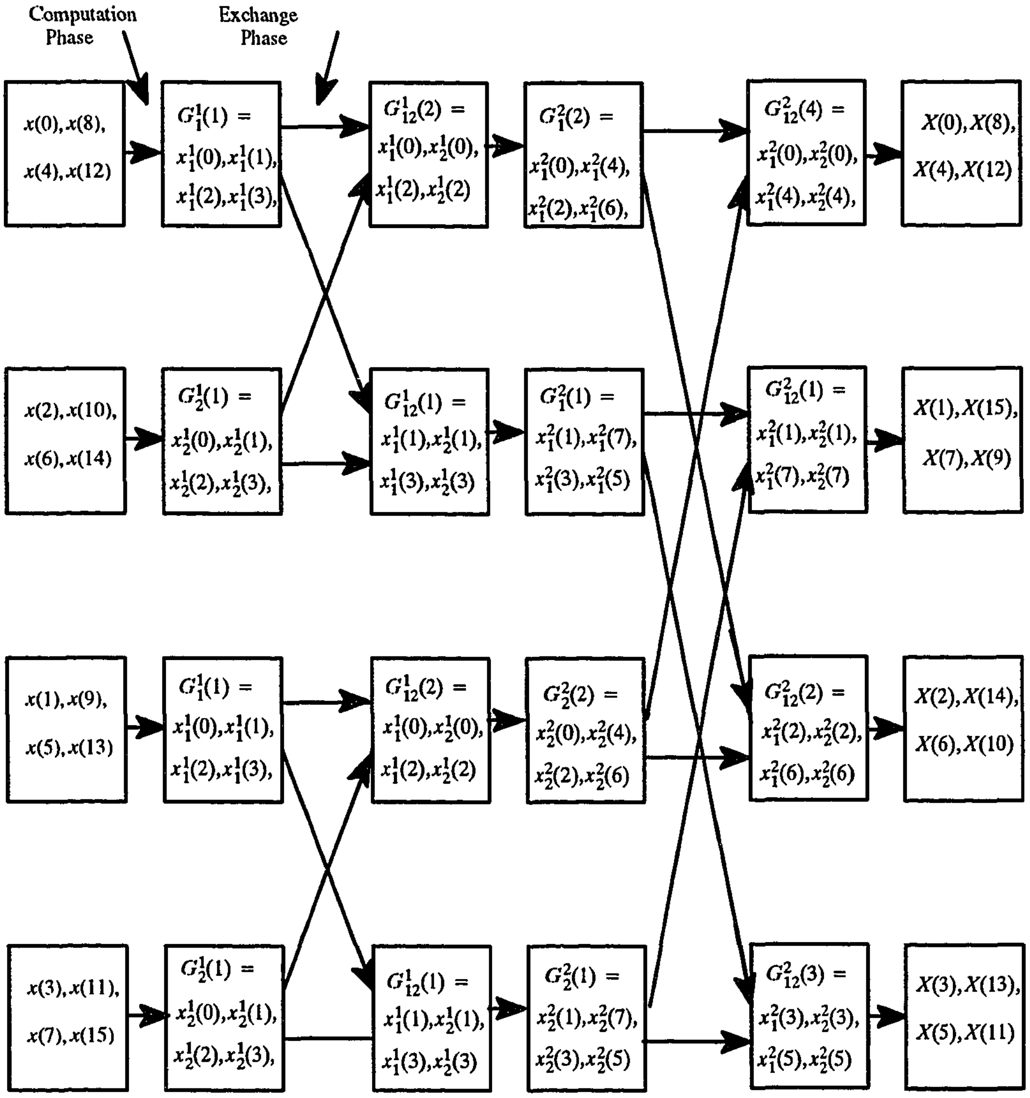

Computation Phase
Exchange Phase
$x(0), x(8), x(4), x(12)$
$G_1^1(1) = x_1^1(0), x_1^1(1), x_1^1(2), x_1^1(3),$
$G_{12}^1(2) = x_1^1(0), x_2^1(0), x_1^1(2), x_2^1(2)$
$G_1^2(2) = x_1^2(0), x_1^2(4), x_1^2(2), x_1^2(6),$
$G_{12}^2(4) = x_1^2(0), x_2^2(0), x_1^2(4), x_2^2(4),$
$X(0), X(8), X(4), X(12)$
$x(2), x(10), x(6), x(14)$
$G_2^1(1) = x_2^1(0), x_2^1(1), x_2^1(2), x_2^1(3),$
$G_{12}^1(1) = x_1^1(1), x_2^1(1), x_1^1(3), x_2^1(3)$
$G_1^2(1) = x_1^2(1), x_1^2(7), x_1^2(3), x_1^2(5)$
$G_{12}^2(1) = x_1^2(1), x_2^2(1), x_1^2(7), x_2^2(7)$
$X(1), X(15), X(7), X(9)$
$x(1), x(9), x(5), x(13)$
$G_1^1(1) = x_1^1(0), x_1^1(1), x_1^1(2), x_1^1(3),$
$G_{12}^1(2) = x_1^1(0), x_2^1(0), x_1^1(2), x_2^1(2)$
$G_2^2(2) = x_2^2(0), x_2^2(4), x_2^2(2), x_2^2(6)$
$G_{12}^2(2) = x_1^2(2), x_2^2(2), x_1^2(6), x_2^2(6)$
$X(2), X(14), X(6), X(10)$
$x(3), x(11), x(7), x(15)$
$G_2^1(1) = x_2^1(0), x_2^1(1), x_2^1(2), x_2^1(3),$
$G_{12}^1(1) = x_1^1(1), x_2^1(1), x_1^1(3), x_2^1(3)$
$G_2^2(1) = x_2^2(1), x_2^2(7), x_2^2(3), x_2^2(5)$
$G_{12}^2(3) = x_1^2(3), x_2^2(3), x_1^2(5), x_2^2(5)$
$X(3), X(13), X(5), X(11)$

Figure 3.4: Flow Graph of the restructured FHT algorithm.

$$X(q_4) = \{H(q_4) + H(q_2) - j[H(q_4) - H(q_2)]\}/2, \quad k = N/4$$

3.5 Machines

3.5.1 NCUBE

NCUBE-7 is a coarse grain MIMD distributed memory machine with a hypercube architecture. A k-dimensional hypercube is an interconnection network of 2^k nodes, each node being a processor. The nodes of a hypercube can be labeled by an integer (represented as a binary number) in the range 0 to $2^k - 1$ such that there is a direct communication link between any two processors if and only if the binary representations of their labels differ in precisely one bit. On NCUBE-7 the maximum allowable dimension is 5. However, a smaller dimension can be used depending upon the problem size. The processors in NCUBE-7 communicate with each other in message passing mode. Hence the larger the distance between the two processors communicating, the more is the time taken. There is a host processor which invokes a cube of the given dimension, and loads the program on to the corresponding nodes.

NCUBE-7 supports the regular FORTRAN and C languages for programming with a set of parallel constructs in the form of subroutine library. Some of the most important routines are :

nopen :	for host only, to open a cube of given dimension,
nloadm:	for host only, to load the programs on the nodes,
nread :	for host and node, to receive a message,
nwrite:	for host and node, to send a message.

3.5.2 DAP-510

The DAP-510 is a fine grain massively parallel computer. It is an *SIMD* machine with 1024 one-bit processors, arranged in a 32 × 32 matrix. Each processor is connected to its four nearest neighbors. The processors on the edge of the matrix have wrap around connections to the processors on the opposite edge. In addition to nearest neighbor connections, a bus system connects all the processors in each row and all the processors in each column. Each processor has its local memory of 64K bits. The whole memory can be viewed as a three dimensional array of bits, consisting of 64K *bit-planes*. A bit-plane has 1024 bits, one from each processor's local memory. Similarly, a *word-plane* has 1024 words, one from each processors local memory.

The higher level language available on the DAP is extended Fortran Plus, a parallel extension of Fortran, which also supports virtual array sizes. The most important feature of Fortran Plus is the ability to manipulate matrices and vectors as single objects. For example, two matrices can be added with a single statement, as done for scalars. The masking and selection operations are available for performing computation on selected processors. As an example, consider the Fortran code:

```
      DO 10 I = 1,32
      DO 10 J = 1,32
          IF (A(I,J) .GE. 0) GOTO 10
          A(I,J) = A(I,J) + 5
10    CONTINUE
```

This is very inefficient on a serial machine partly due to the IF construct. In the corresponding Fortran Plus statement,

```
   A(A.LT.0) = A + 5,
```

the boolean matrix *A.LT.0* is used as a mask so that only those values of A corresponding to a *TRUE* value are changed. The contrast with the sequential machines on conditional operations is important: the sequential machine performs a conditional jump, whereas the DAP will typically use activity control to perform a masked assignment. In addition to the basic functions which have been extended to take vector and matrix arguments, a large number of other functions are provided as standard. For details on these functions one can refer to the AMT DAP 510 extended Fortran-Plus Language Manual. The functions which we have used in our implementation are described below.

shec: right shifts matrix columns by a given distance.

shwc: left shifts matrix columns by a given distance.

shnc: shifts matrix rows upward by a given distance.

shsc: shifts matrix rows downward by a given distance.

All the shifts are carried out in wrap-around manner.

3.6 Implementations

3.6.1 Hypercube Implementation

The hypercube implementation of the restructured algorithm follows from Figure 3.4. For reasons of clarity we restrict our discussion to the computation of $4n$-point DHT where n is the number of nodes in the hypercube (however, the actual code for the NCUBE machine has been written for the general case). We use the following variables and constructs in describing our implementation.

Variables

$x()$: A real array for storing the input.

$h()$: A real array to store the output.

$h1()$ and $h2()$: Real arrays used for exchanging data.

$h3()$: A real temporary array.

The arrays $x()$ and $h()$ are of dimension four; and $h1()$, $h2()$, and $h3()$ are of dimension two.

node : node number.

negh : neighbor node number.

dimension : dimension of the cube.

temp : a real temporary storage.

Constructs

receive(negh,buffer): receives data from the node *negh* into the *buffer*.

send(negh,buffer): sends *buffer* data to the node *negh*.

find_negh(i,negh): finds the neighboring node *negh* for the *i*th iteration

combine(array1,array2,array3): merges data of *array1* and *array2* using Eq. 2.14 and Eq. 2.15 and stores the result in *array1* and *array3*.

Algorithm: Hartley Transform

Input: A sequence of real data of length $4N$.

Output: A transformed sequence of real data of length $4N$.

```
(* the following code is executed at all nodes *)
(* two point merge for x(1),x(2) and x(3),x(4) *)
h(1) = x(1) + x(2);
h(2) = x(1) - x(2);
```

```
h(3) = x(3) + x(4);
h(4) = x(3) - x(4);
(* four point Merge for h(1:4) *)
temp = h(1) + h(3);
h(3) = h(1) - h(3);
h(1) =temp;
temp = h(2) + h(4);
h(4) = h(2) - h(4);
h(2) =temp;
(* all merge within the node done *)

(* prepare for exchange *)
h1(1) = h(1);
h1(2) = h(4);
h2(1) = h(2);
h2(2) = h(3);

(* communication between nodes for merging *)
for i = 1 to dimension do
    (* find the node number of the Neighbor to exchange data*)
```

```
find_negh(i,negh);
if (node < negh) then
  begin
        receive(negh,h3);
        (* receive h2 for computation from the neighbor and
        store it in h3 *)
        send (negh,h2) ;
        (* and send h2 to the neighbor, completing the exchange*)
        combine(h1,h3,h2);
        (* merge h1 and h3 and return values in h1 and h2,
        such that they are in the correct place for exchange
        in the next stage *)
  end
else
  begin
        (* corresponding actions of the neighboring node *)
        send(negh,h1);
        receive (negh,h1)
        combine(h1,h2,h2)
  end
```

```
    end if
end for
```

Figure 3.5 illustrates the implementation of the restructured FHT algorithm on a 2-cube.

3.6.2 DAP Implementation

The mapping on the DAP requires at least $4 \times N^2$ data points, where $N \times N$ is the size of the mesh. The general SIMD Mesh implementation of the FHT algorithm follows from Figure 3.4. For reasons of clarity we restrict our discussion to the computation of $4 \times N \times N$-point DHT, even though the algorithm has been implemented for larger sized DHTs.

We use the following variables in describing our implementation.

Variables

A (,) in the argument of array represents $N \times N$, the size of the DAP array

hart(,,4) : A real array for FHT data.

ndist : A real scalar variable to indicate the distance between columns or rows being exchanged.

factor1(,) and factor2(,) : Real arrays to keep intermediate results.

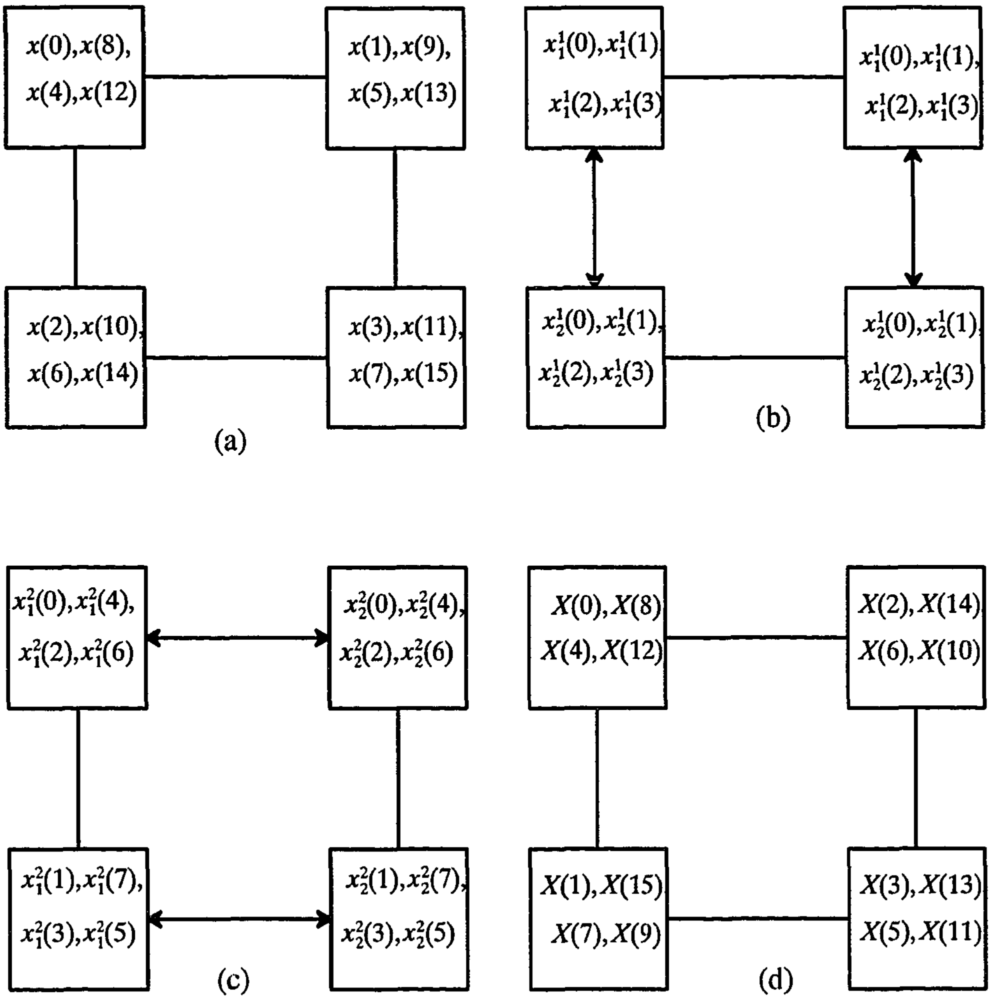


Figure 3.5: Hypercube implementation of the restructured FHT algorithm.

Algorithm: **Hartley Transform**

Input: A sequence of real data of length $4 * N^2$, distributed such that any data item $h(k)$, with $k \bmod N^2 = N(i-1) \times (j-1)$ is mapped to the processor P_{ij}. (The four data items on a processor P_{ij} are referenced by $h(i, j, 1:4)$)

Output: A Hartley sequence of real data of length $4 * N^2$, such that if a processor P_{ij} has data item $h(k)$, it also has $h(N/2-k)$, $h(N/2+k)$ and $h(N-k)$,

```
begin
    fourdht(h); { carry out four point DHT on all nodes }
for $i = 1$ to $log_2(N)$ do
  begin
  ndist = $\log_2(N)/2^{i \bmod(\log_2(N)/2)}$ { distance between columns exchanging data }
  for j = 1 and 3 do
    begin { data exchange with appropriate masks. }
    if ($i \leq \log_2(N)/2$) then { communication among columns}
      begin
      temp = shwc(hart(,,j),ndist);
```

```
            hart(,,j) = shec(hart(,,j+1),ndist);
            end;
          else { communication among rows }
            begin
            temp = shnc(hart(,,j),ndist);
            hart(,,j) = shsc(hart(,,j+1),ndist);
            end;
          hart(,,j+1) = temp;
          end;
      factor1 = hart(,,2)*cos θ + hart(,,4)*sin θ; { θ computed according to Eq. 2.14-2.15
      factor2 = hart(,,4)*cos θ - hart(,,2)*sin θ;
      hart(,,2) = hart(,,3) - factor2;
      hart(,,4) = hart(,,3) + factor2;
      hart(,,3) = hart(,,1) - factor1;
      hart(,,1) = hart(,,1) + factor1;
  end;
```

The implementation based on this mapping in shown in Figure 3.6. with 32-point data sequence on a 4 × 2 mesh.

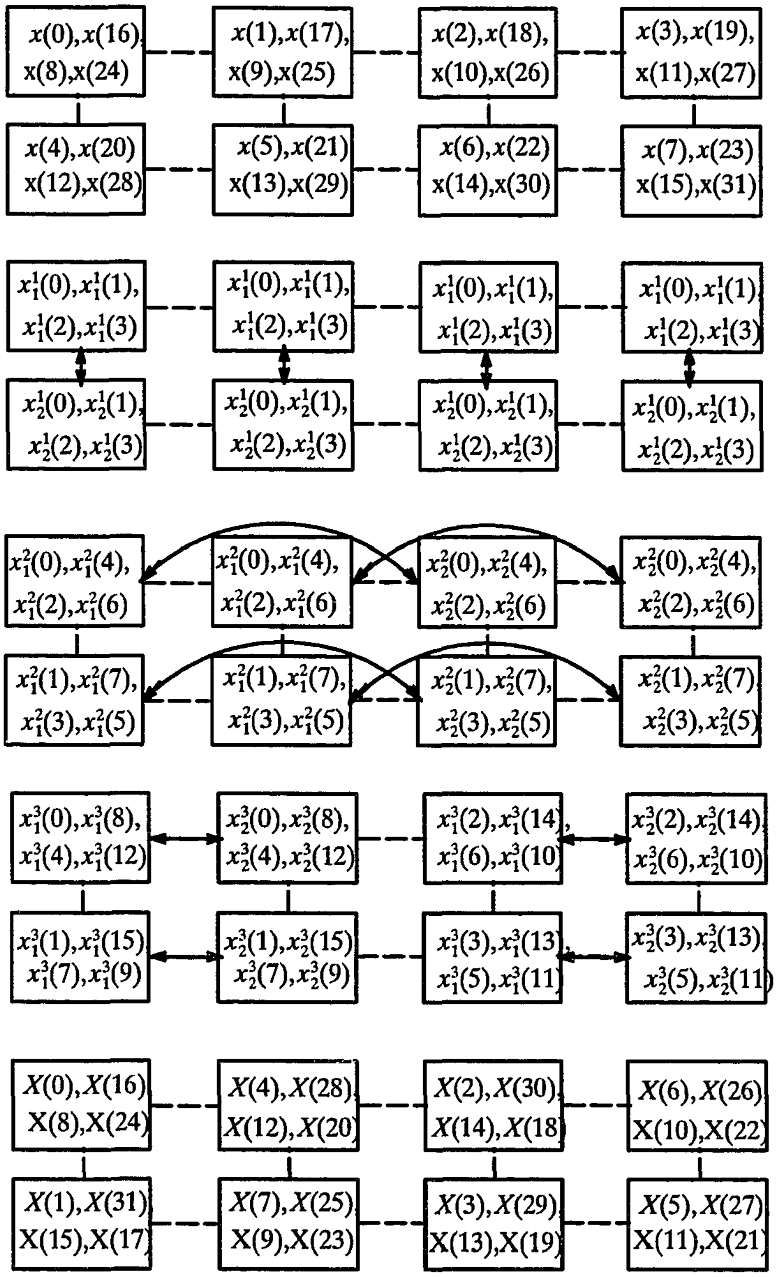

x(0),x(16), x(8),x(24)
x(1),x(17), x(9),x(25)
x(2),x(18), x(10),x(26)
x(3),x(19), x(11),x(27)
x(4),x(20) x(12),x(28)
x(5),x(21) x(13),x(29)
x(6),x(22) x(14),x(30)
x(7),x(23) x(15),x(31)
$x_1^1(0),x_1^1(1),\ x_1^1(2),x_1^1(3)$
$x_1^1(0),x_1^1(1),\ x_1^1(2),x_1^1(3)$
$x_1^1(0),x_1^1(1),\ x_1^1(2),x_1^1(3)$
$x_1^1(0),x_1^1(1)\ x_1^1(2),x_1^1(3)$
$x_2^1(0),x_2^1(1),\ x_2^1(2),x_2^1(3)$
$x_2^1(0),x_2^1(1),\ x_2^1(2),x_2^1(3)$
$x_2^1(0),x_2^1(1),\ x_2^1(2),x_2^1(3)$
$x_2^1(0),x_2^1(1),\ x_2^1(2),x_2^1(3)$
$x_1^2(0),x_1^2(4),\ x_1^2(2),x_1^2(6)$
$x_1^2(0),x_1^2(4),\ x_1^2(2),x_1^2(6)$
$x_2^2(0),x_2^2(4),\ x_2^2(2),x_2^2(6)$
$x_2^2(0),x_2^2(4),\ x_2^2(2),x_2^2(6)$
$x_1^2(1),x_1^2(7),\ x_1^2(3),x_1^2(5)$
$x_1^2(1),x_1^2(7),\ x_1^2(3),x_1^2(5)$
$x_2^2(1),x_2^2(7),\ x_2^2(3),x_2^2(5)$
$x_2^2(1),x_2^2(7)\ x_2^2(3),x_2^2(5)$
$x_1^3(0),x_1^3(8),\ x_1^3(4),x_1^3(12)$
$x_2^3(0),x_2^3(8),\ x_2^3(4),x_2^3(12)$
$x_1^3(2),x_1^3(14),\ x_1^3(6),x_1^3(10)$
$x_2^3(2),x_2^3(14)\ x_2^3(6),x_2^3(10)$
$x_1^3(1),x_1^3(15)\ x_1^3(7),x_1^3(9)$
$x_2^3(1),x_2^3(15)\ x_2^3(7),x_2^3(9)$
$x_1^3(3),x_1^3(13),\ x_1^3(5),x_1^3(11)$
$x_2^3(3),x_2^3(13)\ x_2^3(5),x_2^3(11)$
X(0),X(16) X(8),X(24)
X(4),X(28) X(12),X(20)
X(2),X(30) X(14),X(18)
X(6),X(26) X(10),X(22)
X(1),X(31) X(15),X(17)
X(7),X(25) X(9),X(23)
X(3),X(29) X(13),X(19)
X(5),X(27) X(11),X(21)

Figure 3.6: Mesh implementation of the restructured FHT algorithm.

3.7 Results and Discussion

On the NCUBE-7 we conducted a set of experiments to determine the effect of the varying the data size and the number of nodes on the overall execution time of the algorithms. The number of nodes was varied from one to thirty two and the number of data points from 128 to 32K. The results of these experiments are summarized in Tables 3.1, 3.2, 3.4 and 3.5. For ease of presentation the completion time, which is measured as the number of clock ticks, is averaged over the individual nodes. One clock tick equals 1024/(clock rate), so for a 6MHz system, one tick is approximately 0.17msec. The first column in all these tables gives the execution time on a single node in terms of clock ticks. These numbers are used as a reference for computing the speed-up as shown in the remaining columns. We define speedup as the ratio of the algorithm's execution time on one processor to the execution time on P processors. The efficiency of execution (Table 3.4,3.6) is defined as the ratio of the speedup to P, the ideal speedup.

The cost of executing these algorithms on the MIMD hypercube NCUBE-7 consists of two parts (indeed, this is true for any algorithm). The first is the cost of doing the computation and the second is the cost of communication between processors. For a given problem size, increasing the number of processors will decrease the amount and cost of computations per processor but may increase the communication cost. Depending on the relative cost of computation and communication, increasing the number of processors for a fixed problem size may result in a decrease in efficiency. Increasing the problem size for a fixed number of processors will certainly increase the computation cost in proportion. It may not, however result in a proportionate increase in communication cost. Thus there may be an increase in both the speedup and the efficiency.

Both these effects are apparent in the results presented in Tables 3.1, 3.2, 3.4 and 3.5. In all the four tables the problem size is constant along a row and the number of processors is constant along a column. From Tables 3.1 and 3.4 one can see that, proceeding along a row, the speedup is not linear with the increase in the number of processors. In fact, for the case of 128 data points, the speedup actually decreases as the number of processors is increased from 8 to 16 and from 16 to 32. It is also clear from Tables 3.1 and 3.4 that as the number of data points is increased with a fixed number of processors the speedup increases tends to saturate around a certain value. (Note that this trend will be there irrespective of the relative cost of computation to communication. If computation is very efficient compared to communication, then the saturation will occur at a lower value of speed up and vice-versa). Similar behavior is also seen in the results shown in Tables 3.2 and 3.5. For a fixed problem size, increasing the number of processors nearly always results in a decrease in the efficiency. Finally, one can see from the results of Tables 3.2 and 3.5 that there is a threshold value of number of data points per processor below which there is a drastic fall in efficiency. This threshold is clearly algorithm and machine dependent and thus is not a universal constant.

We also implemented a standard FFT algorithm to compare its performance with those of the restructured FHT and the RFFT algorithms. On the NCUBE-7, a set of experiments were conducted to measure the execution times of the three implementations for different cube sizes and different data sizes. Tables 3.3 and 3.6 summarize the results of comparison between the performances of the standard FFT algorithm and the restructured FHT and the RFFT algorithms respectively. The time taken for the execution of these programs can be attributed to two contributing factors; computation time and communication time. One can make two observations from Tables 3.3 and 3.6, namely;

1. The ratios of execution times of FFT and FHT algorithms and FFT and RFFT algorithms are uniform for most data and cube sizes. The ratios are lower when the number of data points mapped per node is below the threshold.

2. The RFFT algorithm performs slightly better than the FHT algorithm. The reason is that the computation unit of the RFFT algorithm has fewer computation than that of the FHT algorithm (see Figure 3.3).

On DAP-510 experiments were conducted for real input sequences of lengths 4K to 256K. For the sake of comparison we also implemented the FFT algorithm for the same data sizes, The results of these experiments are summarized in Table 3.7. The first column is the number of data points to be transformed. The second, third and fourth columns give the execution time in seconds for the FHT, RFFT, and FFT implementations respectively. The final two columns show the ratio of

execution times of FFT algorithms over those of the FHT and RFFT algorithms. Note that the ratios are essentially independent of the size of the data array as with the NCUBE-7. However, on DAP-510, the FHT algorithm performs better than the RFFT algorithm. The reason is the less regularity in the computation unit of the RFFT algorithm. DAP-510, being an SIMD machine, requires more instructions for irregularities.

Table 3.1: Speed-Up of the Restructured FHT Algorithm.

No of Points	1 Node Time	2 Nodes Speed-Up	4 Nodes Speed-Up	8 Nodes Speed-Up	16 Nodes Speed-Up	32 Nodes Speed-Up
128	519	1.88	3.33	5.14	5.70	4.51
256	1223	1.93	3.64	6.37	8.93	8.86
512	2822	1.96	3.78	7.11	11.81	14.85
1024	6400	1.97	3.86	7.45	13.65	20.98
2048	14321	1.98	3.89	7.63	14.66	25.57
4096	31689	1.98	3.92	7.72	15.13	28.37
8192	69460	1.98	3.93	7.77	15.34	29.82
16384	151150	1.98	3.94	7.80	15.45	30.48
32768	326724	1.99	3.94	7.83	15.52	30.75

Table 3.2: Parallelism Efficiency of the restructured FHT Algorithm.

No of Points	1 Node Time	2 Nodes Efficiency	4 Nodes Efficiency	8 Nodes Efficiency	16 Nodes Efficiency	32 Nodes Efficiency
128	519	94.0%	83.3%	64.3%	35.6%	14.1%
256	1223	96.5%	91.0%	79.6%	55.8%	27.7%
512	2822	98.0%	94.5%	88.9%	73.8%	46.4%
1024	6400	98.5%	96.5%	93.1%	85.3%	65.6%
2048	14321	99.0%	97.3%	95.4%	91.6%	79.9%
4096	31689	99.0%	98.0%	96.5%	94.6%	88.7%
8192	69460	99.0%	98.3%	97.1%	95.9%	93.2%
16384	151150	99.0%	98.5%	97.5%	96.6%	95.3%
32768	326724	99.5%	98.5%	97.9%	97.0%	96.1%

Table 3.3: Ratio of Execution Times of FFT and Restructured FHT Algorithms.

Points	1 Node	2 Nodes	4 Nodes	8 Nodes	16 Nodes	32 Nodes
128	1.185	1.207	1.205	1.202	1.109	1.035
256	1.180	1.202	1.202	1.208	1.153	1.079
512	1.176	1.197	1.201	1.227	1.197	1.132
1024	1.172	1.192	1.210	1.224	1.220	1.184
2048	1.169	1.187	1.205	1.221	1.228	1.213
4096	1.166	1.184	1.200	1.216	1.226	1.224
8192	1.165	1.180	1.195	1.210	1.219	1.230
16384	1.163	1.177	1.191	1.205	1.212	1.230
32768	1.162	1.172	1.188	1.200	1.213	1.220

Table 3.4: Speed-Up of the Restructured RFFT Algorithm.

No of Points	1 Node Time	2 Nodes Speed-up	4 Nodes Speed-Up	8 Nodes Speed-Up	16 Nodes Speed-Up	32 Nodes Speed-Up
128	478	1.84	3.21	4.83	5.25	4.16
256	1122	1.90	3.52	6.10	8.37	8.19
512	2579	1.93	3.68	6.82	11.12	13.79
1024	5840	1.95	3.77	7.23	13.04	19.66
2048	13043	1.96	3.82	7.40	14.07	24.29
4096	28824	1.96	3.84	7.51	14.59	27.06
8192	63111	1.96	3.86	7.58	14.84	28.58
16384	137191	1.97	3.87	7.67	14.98	29.33
32768	296342	1.97	3.92	7.68	15.09	29.68

Table 3.5: Parallelism Efficiency of the Restructured RFFT Algorithm.

No of Points	1 Node Time	2 Nodes Efficiency	4 Nodes Efficiency	8 Nodes Efficiency	16 Nodes Efficiency	32 Nodes Efficiency
128	478	92.0%	80.3%	60.4%	32.8%	13.0%
256	1122	95.0%	88.0%	76.3%	52.3%	25.6%
512	2579	96.5%	92.0%	85.3%	69.5%	43.1%
1024	5840	97.5%	94.3%	90.4%	81.5%	61.4%
2048	13043	98.0%	95.5%	92.5%	87.9%	75.9%
4096	28824	98.0%	96.0%	93.9%	91.2%	84.6%
8192	63111	98.0%	96.5%	94.8%	92.8%	89.3%
16384	137191	98.5%	96.8%	95.9%	93.6%	91.7%
32768	296342	98.5%	98.0%	96.0%	94.3%	92.8%

Table 3.6: Ratio of Execution Times of FFT and Restructured RFFT Algorithms.

Points	1 Node	2 Nodes	4 Nodes	8 Nodes	16 Nodes	32 Nodes
128	1.290	1.280	1.262	1.202	1.109	1.035
256	1.286	1.290	1.266	1.260	1.179	1.088
512	1.286	1.292	1.280	1.288	1.233	1.150
1024	1.284	1.292	1.295	1.300	1.277	1.212
2048	1.283	1.290	1.296	1.299	1.294	1.260
4096	1.282	1.289	1.295	1.300	1.299	1.283
8192	1.282	1.288	1.293	1.299	1.297	1.298
16384	1.281	1.287	1.292	1.304	1.295	1.303
32768	1.281	1.282	1.302	1.299	1.300	1.299

Table 3.7: Comparison of the performances of FHT and RFFT implementations with the FFT implementation on DAP-510.

No of Points	FHT Time (Sec)	RFFT Time (Sec)	FFT Time (Sec)	FFT/FHT	FFT/RFFT
4K	0.026	0.027	0.042	1.62	1.56
8K	0.063	0.066	0.104	1.65	1.58
16K	0.133	0.138	0.215	1.62	1.56
32K	0.282	0.292	0.450	1.60	1.54
64K	0.598	0.619	0.949	1.59	1.53
128K	1.269	1.324	2.028	1.60	1.53
256K	2.685	2.790	4.261	1.59	1.53

Chapter 4

An FFT Implementation for Block Scattered Data Distributions

4.1 Introduction

In a number of applications the FFT algorithm is only a part of the overall computational scheme. In such cases, the ordering of the data elements may be determined by considerations other than the requirements of the FFT algorithm. While this is not a big concern on sequential machines, it can pose a major problem on distributed memory parallel machines. The ordering of data dictated by an application may result in a data distribution which is not ideally suited for the FFT algorithm. The users in such a situation have three options. One is to design different FFT subroutines to match the data distributions for different applications. The second option is to match the data distribution of the application as closely as possible to the requirements of the FFT subroutine. And the third option is to redistribute data before a call to the FFT subroutine. Clearly none of the these options are very suitable and hence there is a need for a subroutine that can support different user data distributions.

This chapter presents an adaptation of the FFT algorithm for distributed memory machines which works for block scattered data distributions with different block sizes. Block scattered distributions have been identified as being extremely useful for scientific computations [49]. They have already been described in detail in Chapter 1. The implementation also makes the final data distribution identical to the initial one. This is important in a large number of practical engineering and scientific applications. FFT algorithms generally change the ordering of data. As a result making input and output data distributions identical involves internode communication. The performance of this scheme has been evaluated by an implementation on Intel iPSC/860.

4.2 Parallel Implementation

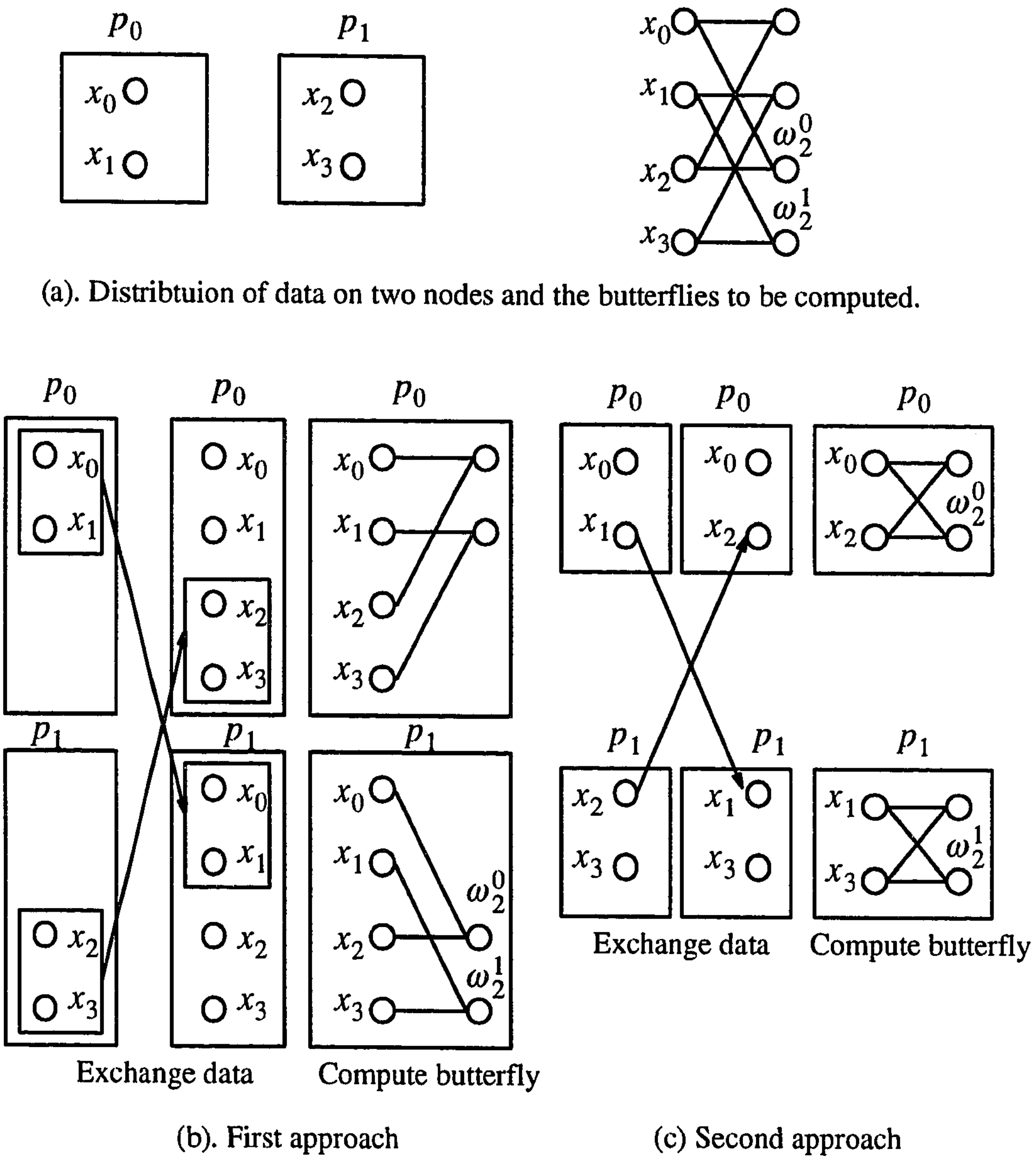
A typical implementation of the FFT algorithm on a distributed memory machine results in a sequence of butterflies at each node interspersed with internode communication. Depending upon the initial distribution, data for some of the butterflies are available locally and for others, off-node data are required. There are two approaches for computing butterflies which need off-node data. The first approach splits a butterfly between two nodes, and in the second approach a complete butterfly is computed on a node. The parallelism in the later case is achieved by distributing different butterflies on different nodes. For example, consider a simple case of computing two butterflies on a two node machine as shown in Figure 4.1(a). (These butterflies are from the DIF-FFT algorithm, which is used for this implementation.) Notice that both butterflies need off-node data. The two approaches are illustrated in Figure 4.1(b) and Figure 4.1(c), respectively. It is obvious from these figures that the first approach has certain disadvantages. These are:

High communication volume The first approach requires twice the inter-node communication volume when compared with the second approach.

Unbalanced computational load The first approach results in additional computation on some of the nodes. For example, the multiplication by ω in the computation of both the butterflies of Figure 4.1(b) is done on node P_1, unlike the second approach (See Figure 4.1(c).)

Extra storage. The first approach requires twice the storage of the second approach.

For these reasons our parallel implementation of the FFT algorithm is based on the second approach.



(a). Distribtuion of data on two nodes and the butterflies to be computed.

(b). First approach

(c) Second approach

Figure 4.1: Two approaches for computing butterflies in parallel.

4.3 Algorithm

As stated earlier, the main feature of our FFT scheme is that it works for block scattered data distribution with variable block sizes. That is, the same algorithm can be used for different data distributions without any initial rearrangement of the data. The algorithm consists of three phases: the first and the third phase compute butterflies for which the data are locally available, and the second phase computes butterflies for which off-node data are required. As a result, internode communication occurs only during the second phase. Depending upon the block size the work distribution for the first and third phases will differ. In extreme cases one of these two phases will not be executed. For a block size of 1 we need to execute only the first two phases, while for a block consisting of all the data on a node only the last two phases are executed. For all other block sizes all three phases of the algorithm are executed. Given the number of processors, the amount of work in the second phase remains constant for all block sizes. When a single block is mapped on a node, it must be treated as a special case. This is because in phase 2 of the algorithm, where off-node data are needed, the block gets divided into two sub-blocks, which is not true for other block sizes.

An FFT algorithm with data size N has $\log_2(N)$ distinct stages of computation. Each stage computes $N/2$ butterflies. In our FFT scheme, as in most other parallel FFT schemes, all the nodes participate in computing a stage by operating on different data points. The algorithm is described here by considering an N point FFT on a p-node machine, with $n = N/p$ as the number of data points mapped per node and b as the block size. The distribution of work in the three phases is as follows

1. The first $\log_2(n/b)$ stages are computed in the first phase. The butterflies in these stages require data available locally from different blocks.
2. The next $\log_2(p)$ stages are computed in the second phase. The butterflies need off-node data, hence internode communication takes place.
3. The last $\log_2(b)$ stages constitute phase three. The butterflies in phase three are computed with local data from within a block.

The algorithm has a computational kernel **dftstep** which is common to all the three phases. The kernel is common to all nodes and computes all the butterflies of a stage mapped onto a node. It assumes that the ω values (see Eq. 2.6 and 2.7) have been precomputed and arranged so that they are available in the right order as needed. A FORTRAN call to the kernel can be made as follows:

$$call\ dftstep(a, w, offset, groups, dist, wincr)$$

where the arguments are:

a Array of input sequence.

w Array of coefficients ω.

offset The distance between the two elements of a butterfly.

groups The number of similar sets of butterflies.

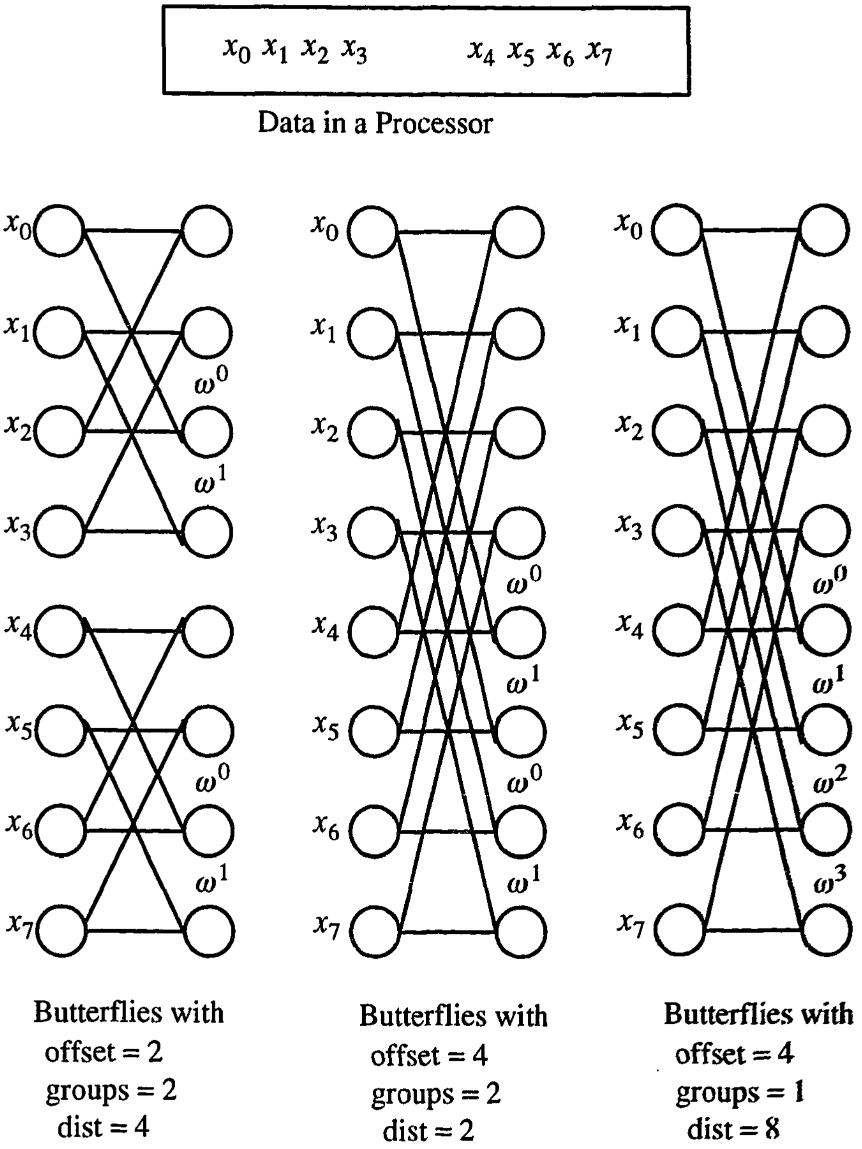
dist The distance between two groups.

wincr The stride for **w**.

The kernel **dftstep** essentially computes $n/2$ butterflies. The formation of these butterflies is determined by the three arguments **offset, groups** and **dist.** Figure 4.2 shows the difference in the formation of the sets of butterflies depending on the values of these three arguments.

The pseudo code given below describes the FFT algorithm using 'dftstep'. { This code is executed on each node }

```
begin{phase 1}
    offset = $n/2$ {distance between two points of a butterfly }
    groups = 1 {only one subgroup in the first stage }
    wincr = 1 { stride for $\omega$ }
    for $i = 1$ to $\log_2(n/b)$ do
        dftstep(a,w,offset,groups,offset*2,wincr)
        offset = offset/2
        groups = groups*2
```



Butterflies with
offset = 2
groups = 2
dist = 4

Butterflies with
offset = 4
groups = 2
dist = 2

Butterflies with
offset = 4
groups = 1
dist = 8

Figure 4.2: Formation of different sets of butterflies.

```
        end for
    end {phase 1}
    begin {phase 2 }
        offset = $n/2$
        groups = offset/b
          for $i = 1$ to $\log_2(p)$ do
          {negh is node with $d - i^{th}$ bit differing}
          {exchange half the data with negh }
          negh = $mynode \oplus 2^{d-i+1}$
          exchange(a(k),negh) {k = $n * j/2 + 1$, where $j$ is the value of $(d - i)^{th}$ bit }
          dftstep(a,w,offset,groups,b,wincr)
          if($b = n$) then {special case}
             negh = $mynode \oplus 2^d$
             exchange(a(k),negh)
          end if
        end for
    end {phase 2 }
    begin {phase 3 }
        for $i = 1$ to $\log_2(b)$ do
          groups = groups*2
```

```
            dftstep(a,w,offset,groups,offset*2,wincr)
            offset = offset/2
            wincr = 2*wincr
        end for
    end {phase 3 }
```

For the implementation on Intel iPSC/860 the call to procedure **exchange** first initiates a send and then posts a receive for incoming data. This protocol is followed due to the peculiarity of the iPSC/860 communication characteristics [41]. The communication between two nodes starts with the source node sending a probe to the destination node. The data is sent to the destination node only after it has acknowledged the probe. If both nodes want to exchange data with each other and they send out the probe at the same time, the data can be transferred to both nodes concurrently. However, if the nodes are out of step and one node starts sending its data before the other one sent its probe, then the transfer of data proceeds in only one direction. This is because each connection between nodes has only two channels, one for receiving and one for transmitting. Once a node starts sending out data it has no channel available to acknowledge the probe. Hence the second node is forced to wait until the first one finishes sending its data.

The working of the three phases is shown in Figure 4.3 with the initial data distribution of Figure 2.1(a). In this example a total of 4 stages are required. The first phase is computed in stage 1 of the algorithm. The butterflies in this stage are formed by the corresponding elements of the two blocks. The second phase is computed in stages 2 and 3 which require exchanges of data. The third phase includes stage 4, which is computed by combining the data within a block.

4.4 Rearrangement

In general, FFT algorithms generate the resultant sequence in an order different from that of the input sequence. The decimation in time algorithms have the input sequence in bit reversed order (index of a data item is obtained by reversing the binary representation of the original index) and output in natural order. The decimation in frequency algorithm, used in this implementation, has the input sequence in natural order and the output sequence in bit reversed order. The computation of a butterfly within a node also scrambles the output data distribution. As a consequence, the resultant data are in the wrong node at the wrong indices and must be redistributed. In a parallel environment redistribution almost always involves internode communication in addition to reordering within the node. The difficulties in redistributing the data can be fully appreciated by following the data movements using the binary representation of the their indices (similar to index digit permutations used in [44]). Consider a data sequence of length 2^n. The index

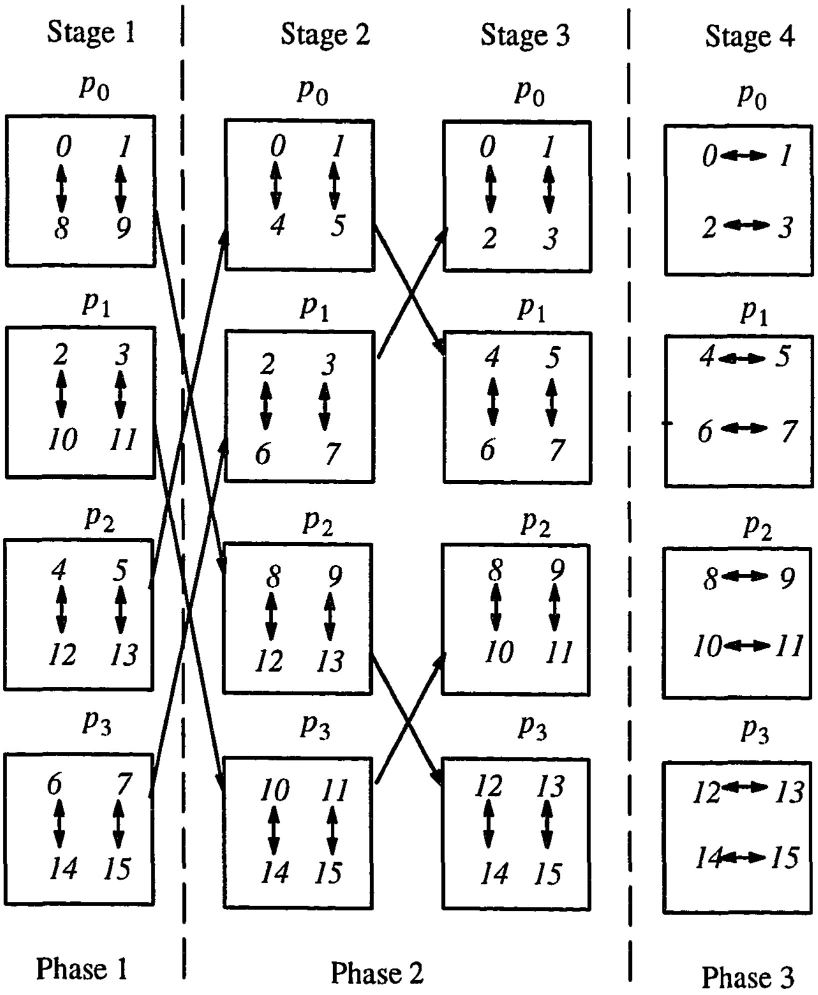


Figure 4.3: Three phases of the FFT algorithm with block scattered data distribution.

of each data item in this sequence can be uniquely represented by n bits. Let this be represented by

$$i_{n-1}i_{n-2}...i_1i_0. \tag{4.1}$$

Here it is assumed that this data is divided amidst 2^d nodes using a block scattered distribution with a block size b. In this distribution, bits $i_{b-1}i_{b-2}...i_0$ represent the offset of a data item within a block, bits $(i_{b+d-1}..i_b)$ represent the node number on which a data item is mapped and bits $(i_{n-1}..i_{b+d})$ represent the block number of a data item.

We will now see the change in the data distribution during FFT computation. To ensure the computation of a butterfly within a node, the following strategy is used for data exchange. Consider two nodes p_1 and p_2 that must exchange data with each other. Between a node pair exchanging data at a stage j in phase 2 of the algorithm, p_1 is the node with $i_{b+d-j} = 0$ and p_2 is the node with $i_{b+d-j} = 1$. Let each node have k blocks of data. Then p_1 keeps the first $k/2$ blocks and sends the last $k/2$ blocks to p_2. Whereas p_2 keeps the last $k/2$ blocks and sends the first $k/2$ blocks to p_1. In terms of binary representation of the indices this movement translates into an exchange between the most significant and $b + d - j$th bits. A sequence of a few such exchanges is shown below;

original distribution :$i_{n-1}i_{n-2}...i_{b+d}i_{b+d-1}i_{b+d-2}i_{b+d-3}...i_bi_{b-1}...i_0$,

after first exchange :$i_{b+d-1}i_{n-2}...i_{b+d}i_{n-1}i_{b+d-2}i_{b+d-3}...i_bi_{b-1}...i_0$,

after second exchange : $:i_{b+d-2}i_{n-2}...i_{b+d}i_{n-1}i_{b+d-1}i_{b+d-3}...i_b i_{b-1}...i_0$,

after third exchange : $:i_{b+d-3}i_{n-2}...i_{b+d}i_{n-1}i_{b+d-1}i_{b+d-2}...i_b i_{b-1}...i_0$.

At the end of the second phase (when all the data movement for computation is over after d exchanges) the binary representation of the data distribution is

$$i_b i_{n-2}...i_{b+d}i_{n-1}i_{b+d-2}...i_{b+1}i_{b-1}...i_0. \tag{4.2}$$

In this distribution the block number is given by $i_b i_{n-2}i_{n-3}i_{b+d}$, the node number by $i_{n-1}i_{b+d-2}...i_{b+1}$ and the offset within a block by $i_{b-1}...i_0$. This distribution remains undisturbed in the third phase. The binary representation required of the final data indices is

$$i_0 i_1...i_{n-2}i_{n-1}. \tag{4.3}$$

The redistribution process changes the data distribution of Eq. 4.2 to that of Eq. 4.3.

Eq. 4.2 and 4.3 also help in determining the number of nodes that each node must send data to and receive data from. The number of nodes is determined by comparing the bits corresponding to the node number in the two equations. This is illustrated with the help of an example of 1024 data points distributed over 16 nodes with two different block sizes, 8 and 4. For both block sizes the initial data distribution is

$$i_9 i_8 i_7 i_6 i_5 i_4 i_3 i_2 i_1 i_0, \tag{4.4}$$

and the final required data distribution is

$$i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9. \tag{4.5}$$

For block size of 8 distribution after all three phases of FFT computation is

$$i_3 i_8 i_7 i_9 i_6 i_5 i_4 i_2 i_1 i_0. \tag{4.6}$$

The node number in Eq. 4.5 is given by $i_3 i_4 i_5 i_6$ and in Eq. 4.6 by $i_9 i_6 i_5 i_4$. Notice that bits i_3, i_4 and i_5 are common to both the node numbers. Given a node number, the values of these bits are fixed. The only bit that can be varied in the destination node number is bit i_3. Hence each node will be sending data to at most two nodes. A careful consideration of the two equations also gives the fixed bits in the representation of source node numbers that a node will receive data from. In this example it is $i_x i_4 i_5 i_6$, where i_x can be any bit other than the ones included in the expression. This expression reveals that each node will receive data from at most two other nodes. For example node 6 will be sending data to nodes 3 and 11 and receiving data from the same two nodes. Node 5 on the other hand will send data to and receive data from only node 13.

For a block size of 4 the distribution after computation is

$$i_2 i_8 i_7 i_6 i_9 i_5 i_4 i_3 i_1 i_0. \tag{4.7}$$

Here the node numbers from Equations 4.5 and 4.6 are given by $i_9 i_5 i_4 i_3$ and $i_4 i_5 i_6 i_7$, respectively. The fixed bits are i_4 and i_5, and the bits that can be varied in the destination node numbers are i_6 and i_7. The binary expression for the node numbers that a node will receive data from is $i_x i_5 i_9 i_x$, where the first and the last bits in the expression are changeable. Each node therefore sends data to and receives data from at most 4 nodes. However, a comparison between the expressions for source and destination node numbers indicates that they may not be identical. Indeed, node 6 sends data to nodes 12, 13, 14 and 15 and receives data from nodes 4, 5, 12 and 13. Similarly it can be easily verified that with block sizes of 1 and 16, each node will be sending data to and receiving data from maximum 8 nodes and for all other block sizes all nodes will be sending data to each other. The last type of data exchange is known as the complete exchange.

The example given above highlights a number of problems that arise in deciding a strategy for redistribution of data after the computation is over. The most important of these are summarized here.

1. For a given data size and the number of processors the nature of the redistribution problem varies with block size (it may or may not be a complete exchange problem.)

2. For the block sizes where redistribution is not a complete exchange problem, all nodes may not send data to an equal number of nodes. More precisely, some nodes will themselves be a destination and hence will send data to one less node.

3. The bits common to the binary representation of source and destination node numbers appear in different order and in different places. The order and placing are dependent on the block size, data size and the dimension of the cube. There is no easily noticeable pattern to generalize the redistribution process.

On Intel iPSC/860 machine the situation is somewhat simpler when the redistribution problem is a complete exchange. There are several algorithms available for the complete exchange problem which carry out data transfer without contention [10, 11]. However, when the redistribution problem is not a complete exchange, it is obvious that these algorithms will result in some redundant work. Each node will be trying to send data to some nodes for which it has no data. The simplest way to counter this problem is to eliminate the redundant data "sends". In addition to redundancy there may also be the problem of contention on the network. We consider an example with 2048 data points, a 5-cube and a block size of 8. The binary representations for source and destination nodes numbers are $i_{10}i_7i_6i_5i_4$ and $i_3i_4i_5i_6i_7$, respectively. From these representations, node 2 must send data to nodes 4 and 20 and node 3 must send data to nodes 12 and 28. The routing in iPSC/860 follows the e-cube algorithm, where the next node in the route is determined by complimenting the least significant bit that does not match with the corresponding bit in the destination. In this example the respective routes for data transfer from node 2 to both of its destinations are

$$2 - 0 - 4, \tag{4.8}$$

$$2 - 0 - 4 - 20. \tag{4.9}$$

Similarly for node 3 they are

$$3 - 2 - 0 - 4 - 12, \tag{4.10}$$

$$3 - 2 - 0 - 4 - 12 - 28. \tag{4.11}$$

It is obvious from Eq. 4.8-4.11 that, irrespective of the way these transfers are scheduled, there will be contention. For these reasons one of the complete exchange algorithms from [10] has been used in the redistribution section of the code.

4.5 Experimental results

We evaluated the performance of our implementation on Intel iPSC/860 for different block sizes and for different data sizes. The Intel iPSC/860 is a distributed memory machine which can have up to 128 nodes. Internode communication is done through a hypercube interconnection network. The experiments reported here were carried out using all nodes of a 32 node machine. The results of these experiments are summarized in Figures 4.4 through 4.7. In all the figures, the quantity along the x axis is represented by its logarithm to the base 2.

Figure 4.4 summarizes the results for variation in performance in terms of Mflops per node as the data size is increased. The two lowest curves in the figure are for the two extreme block sizes. The third curve gives performance when the block size is approximately equal to the number of blocks and the top most curve gives the best performance of any block size for a given data size. The curves corresponding to the best performance, and the two extreme block sizes show a similar trend. There is a rapid increase in the performance with increase in data size in the beginning, which tends to saturate with sufficiently large data. To explain the reason for this behavior we consider the communication characteristics of the machine Intel iPSC/860 the cost of communication is determined by the equation

$$t_{comm} = 164 + 0.398\alpha + 29.9\beta \tag{4.12}$$

where α is the number of bytes in the message and β is the distance between two nodes [10]. The first term in the equation is the setup overhead. As α increases for fixed β, the fraction of total time used in the setup decreases. For small data sizes the overheads are a significant fraction of the overall execution time and bring down the performance. As the data size increases, the overheads become a smaller and smaller fraction until they become insignificant, and the performance curve reaches saturation. The curve for the mid sized blocks shows a much more curious trend. For these block sizes a node sends data to only a subset of nodes, thus saving on setup overheads, but there is contention in the rearrangement process. For smaller data sizes, the volume of data communicated is low enough for overheads to be a greater factor in determining the performance, hence it is close to the best performance. However, for large data sizes, the volume of data communication being high, contention plays a more important role and brings the performance down.

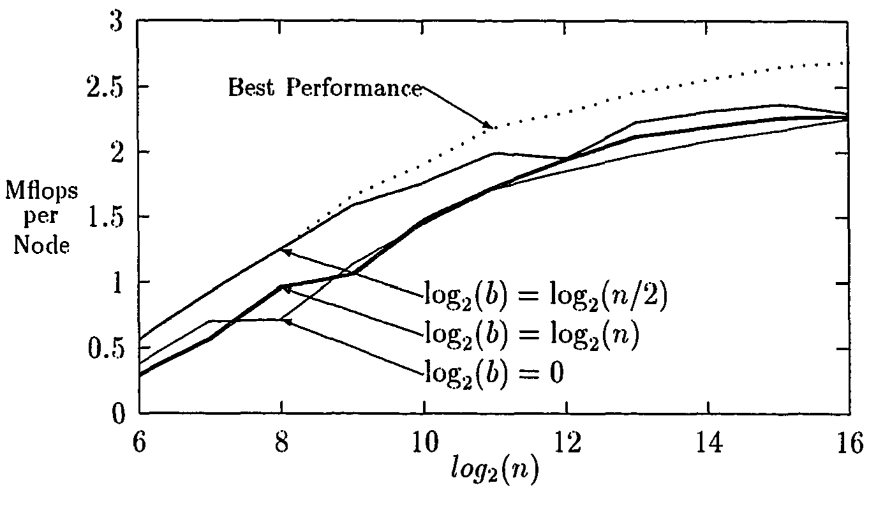


Figure 4.4: The variation in performance per node as datasize is increased.

The best performance for a given data size occurs when there is no contention in the redistribution and the number of blocks are as close as possible to the block size. This can be observed from Figure 4.5 which plots the performance per node as a function of block size for a fixed data size. The top curve in Figure 4.5 gives the performance of the FFT computation without rearrangement and the second curve gives the overall performance. These results indicate that while the performance of computational section varies by small amounts, the overall performance shows more noticeable differences. The variation in the overall performance is due to the data rearrangement. The curve for the FFT performance tend to peak in the middle. The reason for the slight variation in the FFT section performance lies in the relative distribution of work between the three phases of the algorithm. In Figure 4.6 we have plotted the effect of block size on relative distribution of work in the three phases of the computational section. As expected, the fraction of time taken by the first phase is maximum for the smallest block size and steadily decreases as the block size increases. The third phase exhibits a reverse trend. The region where both these phases have approximately equal work is also the region which shows higher performance in the computational section curve of Figure 4.5. Also notice from Figure 4.6 that the fraction of time used in the second phase remains almost constant for all block sizes. The second phase takes more time than the other two since it also involves internode communication.

The effect of data size on the communication and computation time of the computational section of the code is shown in Figure 4.7. To plot this we chose the best performance for every data size. With very small data sizes almost the entire time is taken up by the communication. As the data sizes increase, computation starts taking larger fractions of the execution time. The computation fraction

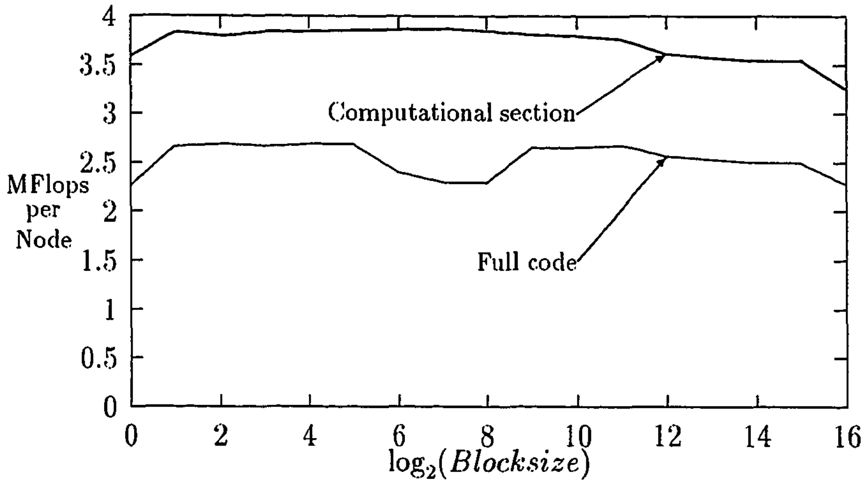


Figure 4.5: The variation in performance as block size is increased.

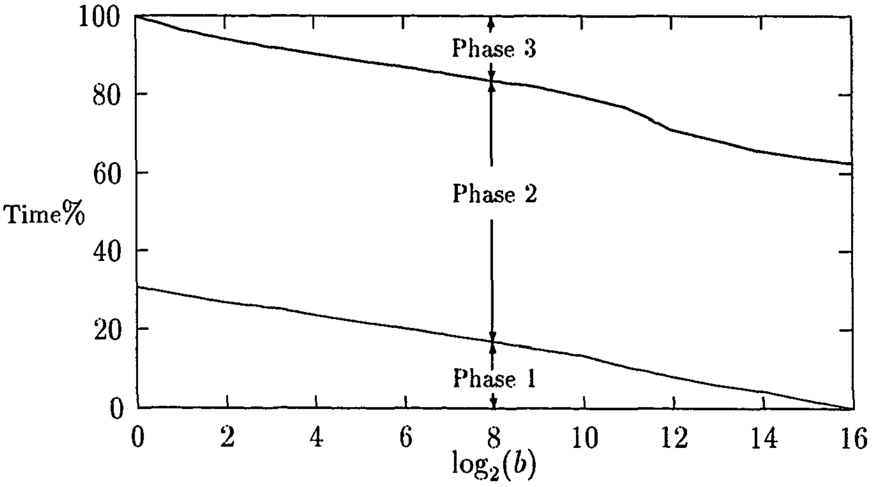


Figure 4.6: The distribution of work in the three phases with different block sizes.

tends to saturate when the data sizes become sufficiently large. To explain the reason we consider Eq. 4.12 again. Notice from Figure 4.7 that the saturation occurs for $\alpha = 2^{11}$. The value of β for the entire curve is 1. For this data size the contribution from the overhead term in the expression is about 5%.

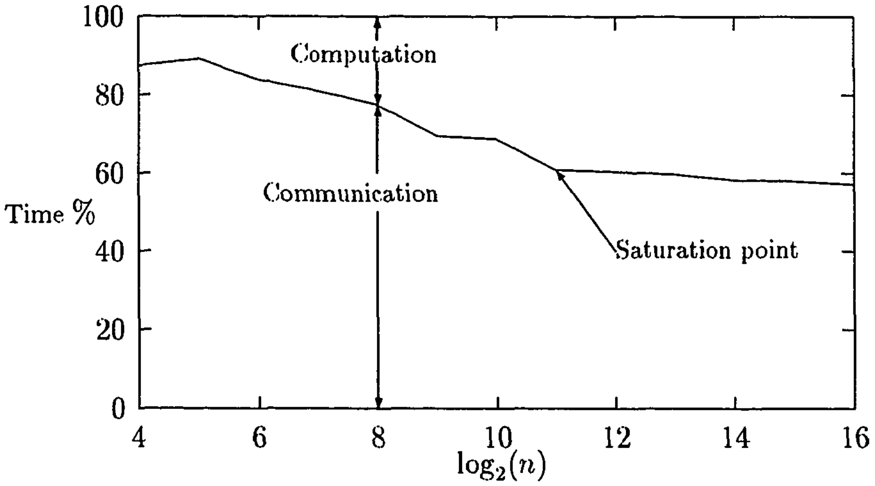


Figure 4.7: The relative contributions of computation and communication in the computational section as the datasize is varied.

4.6 Summary

In this chapter an FFT implementation was given on a distributed memory parallel machine which works for a number of data distributions commonly encountered in scientific applications. We evaluated the performance of our implementation on the Intel iPSC/860. The results of our experiments indicate that the variation in block sizes has more effect on the performance of the rearrangement section than on the computational section. In the computational section, the variation in performance is not significant enough to make the choice of block size a critical issue. On a 32 node machine we obtained a peak performance of 124 Mflops, that is 3.875 Mflops per node (This Figure does not include the initialization costs which are incurred only once for a given size FFT.) If we include the data rearrangement, the performance decreases to 2.69 Mflops per node.

Chapter 5

Fourier Transform of Block Scattered Real Data Distribution

5.1 Introduction

In the previous chapter we discussed an implementation of the FFT algorithm for block scattered data distributions. The implementation tacitly assumed the input data to be complex. In Chapter 3 efficient parallel implementations of the FHT and the RFFT algorithms for transforming real data were given. However, those implementations are efficient only for a specific data distribution. The use of these implementations without modification for other data distributions may require internode communication in the beginning. The internode communication, being very expensive on distributed memory machines, may eliminate the computational advantage of the FHT and RFFT algorithms. It may even make the use of these algorithms more expensive than the FFT algorithm. An alternative is to adapt the restructuring to the given data distribution such that no rearrangement involving internode communication is required in the beginning.

A strategy for adapting the restructuring of data to block scattered distributions is suggested in this chapter. We give implementations for the restructured RFFT and FHT algorithms on a distributed memory parallel machine for block scattered data distributions with different block sizes. The issue of data rearrangement after the computation is also addressed, as with the complex FFT implementation. The performances of these implementations were evaluated on the Intel iPSC/860 and were also compared with the performance of the FFT algorithm given in Chapter 4.

5.2 Parallel Implementation

The implementations presented in this chapter have certain features in common with the implementation of the FFT algorithm from Chapter 4. Some of these features are :

1. The algorithms work for block scattered data distribution with variable block sizes.

2. The algorithms have three phases. The data required for the first and the third phase are available locally, and the second phase needs off node data.

3. The amount of work in the first and the third phase varies with the block size, but remains constant for the second phase.

4. The distribution of work in the three phases is similar.

5. Each algorithm has a computational kernel which is common to all three phases. The kernels are different for different algorithms.

The main differences between them are :

1. The basic unit of computation for the complex FFT algorithm is a butterfly, while for RFFT and FHT algorithms it consists of four data points as shown in Figure 3.3. (We refer to it as a **group.**) Also a group with index 0 is computed differently from the other groups.

2. The first stage of these algorithms, where sequences of size 1 are combined, must be treated as a special case, since no group can be formed.

3. Unlike the FFT algorithm, these two algorithms need at least four blocks per node to be efficient. Less than four blocks per node would require initial data rearrangement with internode communication.

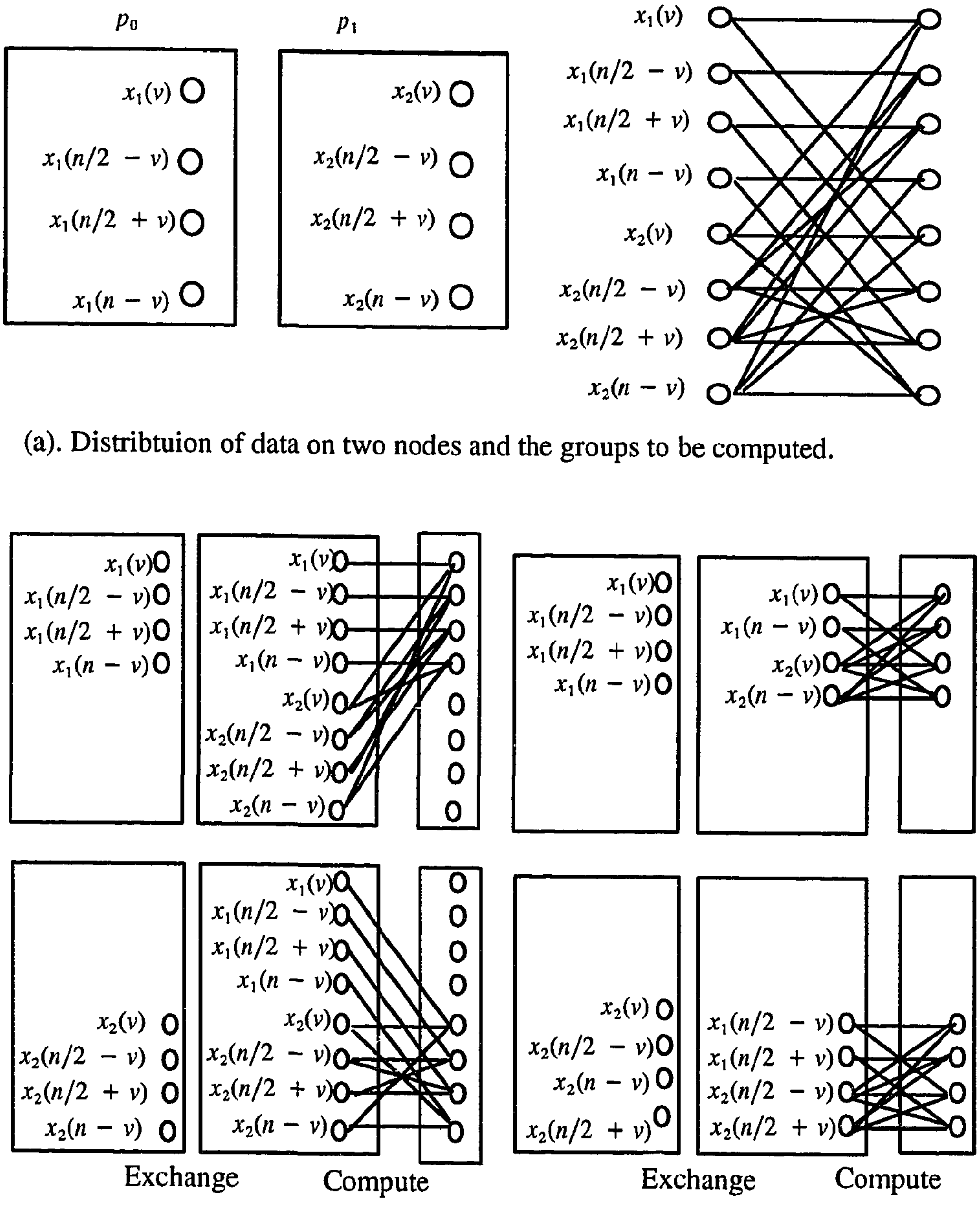
4. A block is not maintained as an entity throughout the computation in the FHT and RFFT algorithms.

5. Some data rearrangement within the node is required in all three phases of the two algorithms discussed here.

6. The data rearrangements after the computation are vastly different.

A computational group of Figure 3.3 can be computed in parallel in two ways similar to the butterfly in Chapter 4. This is illustrated in Figure 5.1. using a computational group with a non-zero index from the FHT algorithm. The computation shown in Figure 5.1(b) has the disadvantages of twice the storage area and twice the communication volume requirements of the one in Figure 5.1(c). The computational load is identical in both approaches, unlike the butterfly. However, the approach shown in Figure 5.1(b) divides a group between two processors. This results in an unsuitable data distribution for the next stage. Hence, we must use the approach shown in Figure 5.1(c).

The restructured FHT and RFFT algorithms require input data to be in bit reversed order. This could be avoided in the FFT implementation by using the decimation in frequency FFT algorithm which shifts the process of bit reversal to the end. This is not possible with the restructured FHT and RFFT algorithms since the formation of the grouping mentioned in Chapter 3 requires the data to be in bit reversed order. However, arranging the entire data in bit reversed order requires internode communication. A compromise can be reached by arranging the data within the node in a bit reversed order. Recall from Chapter 3 that the initial grouping is formed by putting four consecutive data items of the bit reversed sequence in one group. In terms of binary representation of the indices,



(a). Distribtuion of data on two nodes and the groups to be computed.

(b) A group between two nodes.

(c) A group within a node

Figure 5.1: Two ways of computing groups in parallel.

a bit reversed sequence of 2^n data points is represented as

$$i_0 i_1 ... i_{n-1}. \tag{5.1}$$

The initial grouping is formed by putting all data items which have identical values for bits $i_0 ... i_{n-3}$ in one group. To verify that the bit reversal of the data within a node allows the initial grouping to be formed, we consider a mapping of 2^n length data on 2^d nodes, with a block size of 2^b and the number of blocks on a node equal to 2^k. The data distribution is

$$i_{n-1} ... i_{b+d}, i_{b+d-1} ... i_b, i_{b-1} .. i_0, \tag{5.2}$$

where bits $i_{n-1} ... i_{b+d}$ give the block number, bits $i_{b+d-1} ... i_b$ represent the node number and bits $i_{b-1} .. i_0$ give the offset within a block. The distribution with bit reversal within a node is

$$i_0 ... i_{b-1} i_{b+d} ... i_{n-b-1}, i_{b+d-1} ... i_b, i_{n-b} ... i_{n-2} i_{n-1}; \quad k > b, \tag{5.3}$$

$$i_0 ... i_{b-1}, i_{b+d-1} ... i_b, i_{b+d} ... i_{n-1}; \quad k = b, \tag{5.4}$$

$$i_0 ... i_{k-1}, i_{b+d-1} .. i_b, i_k ... i_{b-1} i_{b+d} ... i_{n-1}; \quad k < b \tag{5.5}$$

The commas in all the equations separate the bits representing the three different quantities, namely the block number, the node number and the offset within the block. It can be easily seen from Eq. 5.3-5.5 that the conditions for forming a grouping are met as long as bits i_{n-1} and i_{n-2} are not a part of the node number which is true for $k \geq 2$. Following the argument from Section 3.3, one can see that the data for the first stage and the groups for the next $k-1$ stages are available within the node, when $k \geq 2$. The next d stages require off node data and the last b stages are again computed within the node. The internode communication during computation is carried out along decreasing order of dimension. That is, in the first stage of phase 2 nodes differing in bit i_{b+d-1} exchange data. In the next stage the nodes exchanging data with each other differ in bit i_{b+d-2} and in the last stage they differ in bit i_b.

5.3 Algorithms

As mentioned earlier, the algorithms have a computational kernel **fhtstep** which is common to all the three phases. The only difference between the kernels of the two algorithms is the set of equations for computing the groups, hence we discuss only the FHT kernel. The kernel is common to all nodes and computes all the groups of a stage mapped onto a node. It assumes that the cosine and sine values have been precomputed and arranged so that they are available in the right order as needed. A FORTRAN call to the kernel can be made as follows:

$$call\ fhtstep(a, w, len, groups, offset, dist, stride)$$

where the arguments are:

a array containing the input sequence.

w array of coefficients.

len length of the input sequence.

groups number of groups differing in the values of coefficients.

offset distance between the elements of a group.

dist distance between two successive identical groups.

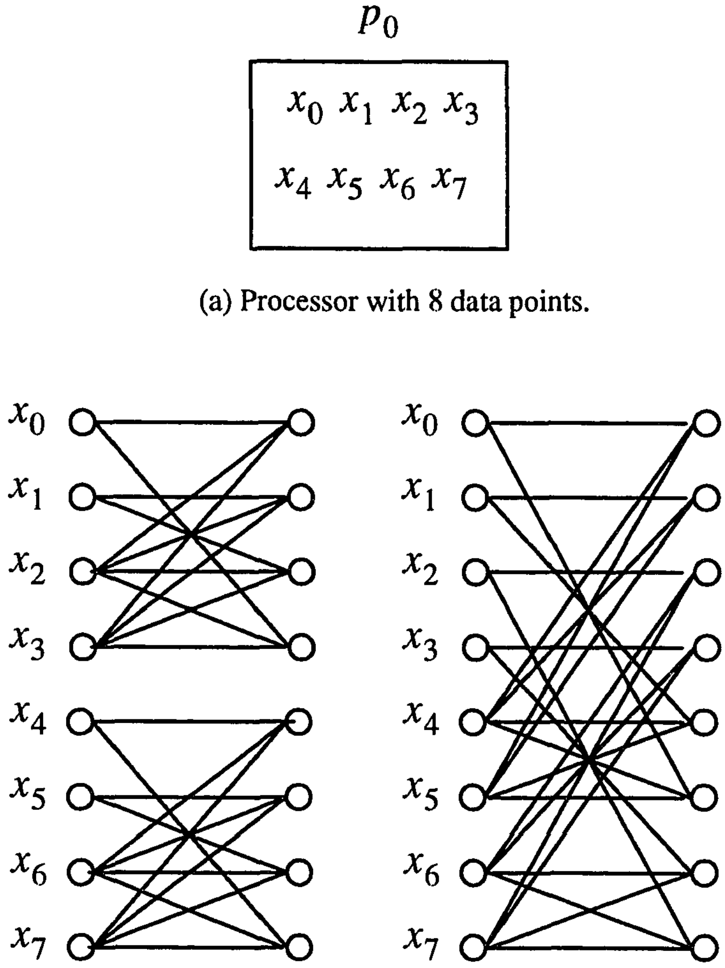
stride distance between two successive elements of **w** to be used.

The kernel **dftstep** essentially computes $n/4$ groups. The four data elements that make a group are always available in two sets. The elements in each set are consecutive to each other. Each group can be represented as consisting of elements $x(i), x(i+1), x(j), x(j+1)$. The values of i and j are determined by the arguments **offset, groups** and **dist.** Figure 5.2 illustrates this with 8 data points mapped on a processor and two different sets of values for i and j. In Figure 5.2(b) the values of i and j are 0 and 2 respectively, while in Figure 5.2(c) they are 0 and 4 respectively. The pseudo code given below describes the FFT algorithm using 'dftstep'. { This code is executed on each node }

```
begin{phase 1}
    bitreverse {rearrange data on a node in bit reverse order }
    twopoint(a) {first stage of the fht algorithm }
    offset = 2 {distance between the elements of a group }
```

```
        dist = 4 {distance between two successive identical groups }
        groups = 1 {number of distinct groups }
        stride = $b/2$ { stride for $\omega$ }
        for $i = 1$ to $\log_2(n/b) - 1$ do
            fhtstep(a,w,len,groups,offset,dist,stride)
            offset = offset*2
            dist = dist*2
            groups = groups*2
            stride = stride/2
        end for
    end {phase 1}
    begin {phase 2 }
        shuffle(a) {break each block into two subblocks shuffle them}
        offset = $n/2$
        dist = $b/2$
        groups = dist/2
        stride = 2
        for $i = 0$ to $\log_2(p) - 1$ do
            {negh is node with $d - i^{th}$ bit differing}
            {exchange half the data with negh }
            negh = $mynode \oplus 2^{d-i}$
            exchange(a(k),negh) {k = n*j/2+1, j is the value of $(d-i)^{th}$ bit }
            fhtstep(a,w,len,groups,offset,dist,stride)
        end for
    end {phase 2 }
    begin {phase 3 }
        inverse_shuffle(a) {recombine sub blocks from phase 2 }
        offset = $b$
        dist = offset*2
        for $i = 1$ to $\log_2(b)$ do
            groups = groups*2
            fhtstep(a,w,len,groups,offset,dist,stride)
            offset = offset*2
            dist = dist*2
        end for
    end {phase 3 }
```

The working of the algorithm is shown in Figure 5.3, with an example of 32 data points distributed over 4 nodes with a block size of 2. The numbers represent the indices of the data. Step (a) in Figure 5.3 shows the initial data distribution



(a) Processor with 8 data points.

(b) Set of groups with offset = 2, dist = 4, groups = 1.

(c) Set of groups with offset = 4, dist = any value, groups = 2.

Figure 5.2: Formation of different groups.

and step (b) shows the data after bit reversal within a node has been carried out. The first stage is computed without the formation of groups and is therefore not shown in the figure. The data movement of stage 2 is shown in step (c). step (d) shows the data rearrangement prior to phase 2. This step accumulates the data to be sent out from a node. It is a much less expensive method than sending data in various packets because of the communication overheads. The data exchange between the nodes in the two stages of phase 2 is shown in steps (e) and (g). The data movement with computation in the same two stages is shown in steps (f) and (h). Step(i) shows data rearrangement within a node prior to phase 3. This step ensures the simplicity and generality of the kernel. The final computational stage is shown in step (i). Notice that a block ceases to exist in step (a) of the algorithm and remains so until the end. It is restored only in the rearrangement section. None of the steps involving data rearrangement within the node are expensive relative to the overall cost.

5.4 Rearrangement

The FHT and RFFT algorithms give the output data in an ordering different from that of the input in general. A careful look at Figure 3.3 reveals that the data are displaced from their original position in computing a group. Internode communication in the second phase of the two algorithms also displaces the data. Hence,

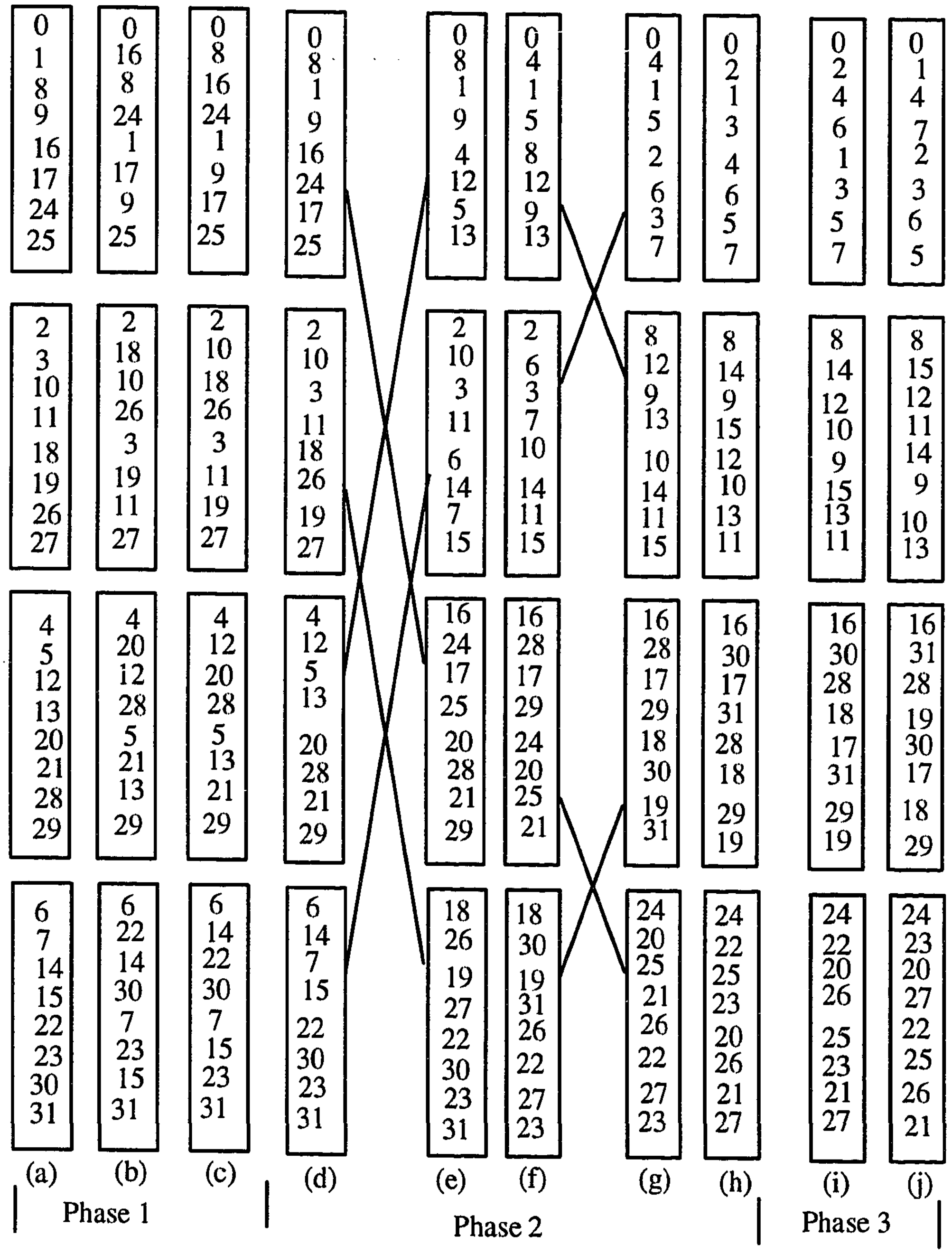


Figure 5.3: Steps in the computation of RFFT/FHT algorithms.

as with the FFT algorithm, the data must be rearranged to return the output sequence in the same distribution as the input sequence. The data movement during the computational section of the algorithm can be followed using the binary representation of their indices as with the FFT algorithm discussed in Chapter 4. Unlike the FFT algorithm, the data movement in the FHT and RFFT algorithm takes place in all three phases. The initial distribution and the distribution with bit reversal within the node have been discussed in the previous section (see Eq. 5.1, 5.3, 5.4, 5.5. Here the data movement of the distribution of Eq.5.4 only will be discussed in detail since the other two are very similar.

The first stage in the two algorithms does not have any data movement. In the subsequent $k-1$ stages which are computed within a node the data movement consists of two steps. In a stage m two sequences $x_1(r)$ and $x_2(r)$ of length 2^{m-1} are combined to form a sequence of length 2^m (see Eq. 2.8,2.9 and 2.14,2.15). The first step involves data movement in $x_2(r)$. The data items with even indices other than zero exchange places with the next data item. For example data items with indices 2 and 4 exchange places with data items with indices 3 and 5 respectively. All the elements of $x_1(r)$ and the first two elements of $x_2(r)$ remain in place. The least significant bit in the binary representation after this step is determined by

$$i_{n-m}(i_{n-m+2} + i_{n-m+3} \dots i_{n-1}) \oplus i_{n-m+1} \tag{5.6}$$

Let this expression be represented as j_{n-m+1}. In the second step the data items with even numbered indices in $x_2(r)$ exchange places with the odd numbered data items in $x_1(r)$. Hence the data items with indices 1, 3 and so on from $x_1(r)$ exchange place with data items with indices 0, 2 on so on respectively from $x_2(r)$. In binary representation this is equivalent to exchanging bit i_{n-m} with bit j_{n-m+1} determined in the first step. The data movement in the stages 2-4 with the distribution of Eq. 5.4 is shown below:

Stage 2:

$$j_{n-1} = i_{n-1}$$

$$i_0 \ldots i_{b-1}, i_{b+d-1} \ldots i_b, i_{b+d} \ldots i_{n-1} i_{n-2}$$

Stage 3:

$$j_{n-2} = i_{n-3} i_{n-1} \oplus i_{n-2}$$

$$i_0 \ldots i_{b-1}, i_{b+d-1} \ldots i_b, i_{b+d} \ldots j_{n-2} i_{n-1} i_{n-3}$$

Stage 4:

$$\begin{aligned} j_{n-3} &= i_{n-4}(j_{n-2} + i_{n-1}) \oplus i_{n-3} \\ &= i_{n-4}(i_{n-3} i_{n-1} \oplus i_{n-2} + i_{n-1}) \oplus i_{n-3} \\ &= i_{n-4}(i_{n-2} + i_{n-1}) \oplus i_{n-3} \end{aligned}$$

$$i_0 \ldots i_{b-1}, i_{b+d-1} \ldots i_b, i_{b+d} \ldots j_{n-3} j_{n-2} i_{n-1} i_{n-4}$$

At the end of stage k the distribution will be

$$i_0 \ldots i_{b-1}, i_{b+d-1} \ldots i_b, j_{b+d+1} \cdots j_{n-2} i_{n-1} i_{b+d} \tag{5.7}$$

This is the data distribution at the end of phase 1.

The shuffle step just before the beginning of the second phase breaks the already computed sequences of length 2^k into two subsequences. (Note that the number of such already computed sequences is exactly equal to the block size and their length is equal to the number of blocks.) The subsequences thus created are then forward shuffled once. This results in the following distribution

$$j_{b+d+1} i_0 \cdots i_{b-2}, i_{b+d-1} \cdots i_b, i_{b-1} j_{b+d+2} \cdots j_{n-2} i_{n-1} i_{b+d}. \tag{5.8}$$

This step is necessary since one half of each of the 2^k length sequences must be exchanged with a corresponding subsequence in another node. Shuffling of subsequences ensures that all the data to be exchanged with another node are contiguous. The data then can be sent in one step, thus encountering communication overheads only once.

The data movement in the second phase involves three steps. The first step exchanges data between the nodes and the next two steps move data for computation. The first step is identical to data movement in the second phase of the FFT algorithm discussed in the Section 3.4. In step 2, at a stage r the data movement in nodes having a value 0 in r most significant bits differs from that in the remaining nodes. In nodes where r most significant bits are 0, the even numbered data items with a value 1 in at least one of bits in the offset and the most significant bit exchange place with their next odd numbered neighbor. In nodes where at least one of the r most significant bits in the node number has a value 1, all even numbered data items with a value 1 in their most significant bit exchange places with the next odd numbered data item. In the third step, the least significant and the most significant bits exchange places. The first two stages of phase 2 are shown here to illustrate these steps.

Stage 1

$$i_{b+d-1}i_0 \ldots i_{b-2}, j_{b+d+1} \ldots i_b, i_{b-1}j_{b+d+2} \cdots j_{n-2}i_{n-1}i_{b+d}$$

$$\begin{aligned} j_{b+d} &= i_{b+d-1} \cdot (j_{b+d+1} + j_{b+d+2} \cdots + i_{n-1}) \oplus i_{b+d} \\ &= i_{b+d-1} \cdot (i_{b+d+1} + i_{b+d+2} \cdots + i_{n-1}) \oplus i_{b+d} \end{aligned}$$

$$j_{b+d}i_0 \ldots i_{b-2}, j_{b+d+1} \ldots i_b, i_{b-1}j_{b+d+2} \cdots j_{n-2}i_{n-1}i_{b+d-1}$$

Stage 2

$$i_{b+d-2}i_0 \ldots i_{b-2}, j_{b+d+1}j_{b+d}i_{b+d-3} \cdots i_b, i_{b-1}j_{b+d+2} \cdots j_{n-2}i_{n-1}i_{b+d-1}$$

$$j_{b+d-1} = i_{b+d-2} \cdot (i_{b+d} + i_{b+d+1} + j_{b+d+2} + \ldots + i_{n-1}) \oplus i_{b+d-1}$$

$$j_{b+d-1}i_0 \ldots i_{b-2}, j_{b+d+1}j_{b+d}i_{b+d-3} \ldots i_b, i_{b-1}j_{b+d+2} \cdots j_{n-2}i_{n-1}i_{b+d-2}$$

Following these steps the distribution at the end of phase 2 will be

$$j_{b+1}i_0 \ldots i_{b-2}, j_{b+d+1}j_{b+d} \cdots j_{b+2}, i_{b-1}j_{b+d+2} \ldots j_{n-2}i_{n-1}i_b \qquad (5.9)$$

At this point one inverse shuffle is carried out on the subsequences formed in the beginning of phase 2. The purpose of this step is only to maintain the simplicity of the computational kernel. With this step the data distribution at the end of phase 2 is

$$\dot{\imath}_0 \ldots \dot{\imath}_{b-1}, \dot{\jmath}_{b+d+1}\dot{\jmath}_{b+d} \cdots \dot{\jmath}_{b+2}, \dot{\jmath}_{b+1}\dot{\jmath}_{b+d+2} \cdots \dot{\imath}_{n-1}\dot{\imath}_b \tag{5.10}$$

The data movement in phase 3 of the algorithms is an extension of the data movement in phase 1. Hence at the end of the computational section of the algorithms the data distribution will be

$$\dot{\jmath}_1 \cdots \dot{\jmath}_b, \dot{\jmath}_{b+d+1}\dot{\jmath}_{b+d} \cdots \dot{\jmath}_{b+2}, \dot{\jmath}_{b+1}\dot{\jmath}_{b+d+2} \cdots \dot{\imath}_{n-1}\dot{\imath}_0 \tag{5.11}$$

When $k > b$ the final data distribution is

$$\dot{\jmath}_1 \cdots \dot{\jmath}_b, \dot{\jmath}_{b+1}\dot{\jmath}_{b+d+2} \cdots \dot{\jmath}_{n-b}, \dot{\jmath}_{b+d+1}\dot{\jmath}_{b+d} \cdots \dot{\jmath}_{b+2}, \dot{\jmath}_{n-b+1} \cdots \dot{\jmath}_{n-2}\dot{\imath}_{n-1}\dot{\imath}_0, \tag{5.12}$$

and with $k < b$ the final data distribution is

$$\dot{\jmath}_1 \cdots \dot{\jmath}_k, \dot{\jmath}_{b+d+1}\dot{\jmath}_{b+d} \cdots \dot{\jmath}_{b+2}, \dot{\jmath}_{k+1} \cdot\cdot \dot{\jmath}_{b+1}\dot{\jmath}_{b+d+2} \cdots \dot{\jmath}_{n-2}\dot{\imath}_{n-1}\dot{\imath}_0 \tag{5.13}$$

In all three equations 5.11-5.13, the expression for node number is

$$\dot{\jmath}_{b+d+1}\dot{\jmath}_{b+d} \cdots \dot{\jmath}_{b+2}. \tag{5.14}$$

The data distribution required at the end of the rearrangement section of the algorithms is

$$\dot{\imath}_0\dot{\imath}_1 \ldots \dot{\imath}_{n-2}\dot{\imath}_{n-1} \tag{5.15}$$

It is obvious from Eq. 5.14 and 5.15 that a comparison between bits appearing in source and destination node numbers does not determine the nature of rearrangement problem. Here the number of source and destination nodes is not fixed. To illustrate this we again consider the example with data size 1024 and block size 8 distributed over 4 nodes. The four bits in the node number at the end of phase 3 are

$$
\begin{aligned}
j_8 &= i_7.i_9 \oplus i_8 \\
j_7 &= i_6.(i_8 + i_9) \oplus i_7 \\
j_6 &= i_5.(i_7 + i_8 + i_9) \oplus i_6 \\
j_5 &= i_4.(i_6 + i_7 + i_8 + i_9) \oplus i_5
\end{aligned}
$$

The destination node number is given by $i_3i_4i_5i_6$ If we consider destination node 13 , then $i_6 = 1,\ i_5 = 0,\ i_4 = 1,\ i_3 = 1$. This fixes the values of bits j_6 and j_5 in the source node number. Hence node 13 will receive data from nodes 3, 7, 11 and 15. However if consider node 12, we can fix the value of only j_6 in the source node number. Hence this node will receive data from nodes 0, 1, 3, 4, 5, 8, 9, 12 and 13. For node 14 it is not possible to fix the value of any bit in the source node number, hence it will receive data from all the other nodes. Here due to the difference in the number of source nodes, contention is likely to occur. However, when none of bits can be fixed for any node number, it is possible that the exchanges are scheduled without contention. This is likely to occur for the extreme block sizes where there are no common bits in the source and destination node numbers. It must be noted here that even with all nodes exchanging data with all other nodes, the rearrangement problem may not be exact equivalent of the complete exchange problem of [11], since the amount of data being exchanged may not be the same for all nodes.

5.5 Experimental Results

The FHT and the RFFT algorithms with block scattered data distributions were implemented on the Intel iPSC/860 machine to evaluate their performance. We conducted two sets of experiments with these implementations. The first set of experiments evaluated the performance of the various sections of the codes and the variation in performance with the change in block size and data size. The second set of experiments compared the performance of the FHT and the RFFT algorithms with the FFT algorithm described in Chapter 4. The experiments were conducted on all nodes of a 32 node machine.

The results of the experiments to evaluate the performance of FHT and RFFT algorithms are summarized in Figures 5.4 through 5.11. Figures 5.4 and 5.5 show the execution times for the FHT and the RFFT algorithms respectively. Each figure has two curves showing the best and the worst time of any block size for a given data size. The curves for both the algorithms are almost identical in trend, but the FHT algorithm shows slightly better performance. The slight difference in the performance of the two algorithms is due to the rearrangement section of the code. The volume of data communication is higher in the RFFT algorithm since it involves complex numbers. Figures 5.6 and 5.7 show the variation in the execution time for a fixed data size as the block size is increased. The lower curves in both the figures show the time taken by the computational section alone. The top curves include the time taken by the rearrangement of data. The top curves show the reason for a difference between the best and the worst performances of the two algorithms. The time taken by the computational section does not vary with the block size, but the rearrangement section is affected by this variation. The rearrangement section takes a longer time when the number of block sizes is approximately equal to the number of blocks. Recall that a similar trend was observed in the FFT algorithm, where contention was responsible for deterioration in the performance. Contention is also likely to occur in the rearrangement section of the FHT and RFFT algorithms when the number of blocks is of the same order as the block size. This is because different nodes have a different number of destinations for their data. Figures 5.8 and 5.9 show the relative contribution of the three phases in the overall execution time. The relative contribution of the second phase is almost constant for both the algorithms. It is also the largest since this is the only phase involving internode communication. Figures 5.10 and 5.11 show the relative contribution of the internode communication and computation in all three phases combined. For small data sizes a very large fraction of time is taken by communication because the setup overheads constitute a significant fraction of total execution time. As the data sizes grow, the fraction contributed by the overheads reduces, thus reducing the relative contribution of the communication time. When the overheads become insignificant, the relative contribution due to communication saturates. All of these trends are identical for the two algorithms presented in this chapter and similar in to those of the FFT algorithm presented in the Chapter 4.

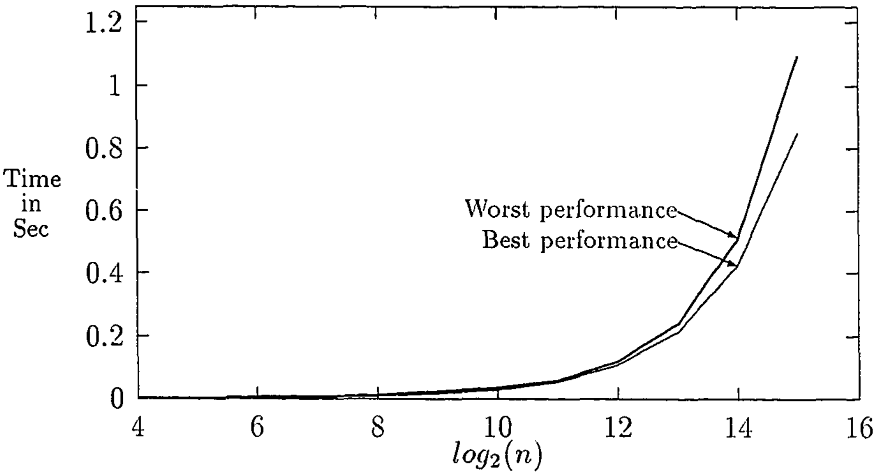


Figure 5.4: Variation in performance with data size for the FHT algorithm.

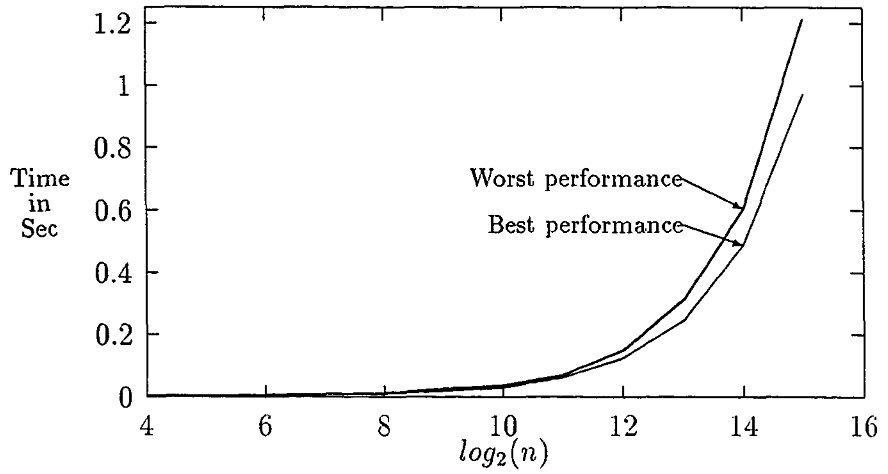


Figure 5.5: Variation in performance with data size for the RFFT algorithm.

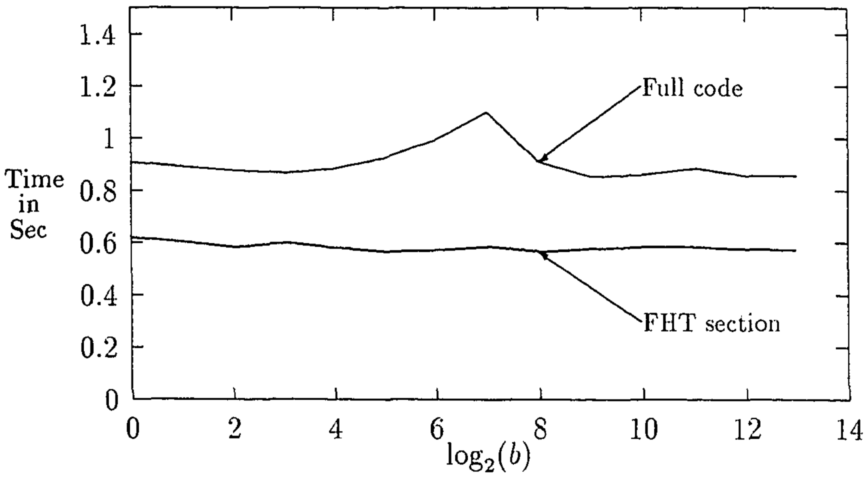


Figure 5.6: Variation in FHT Performance with block size.

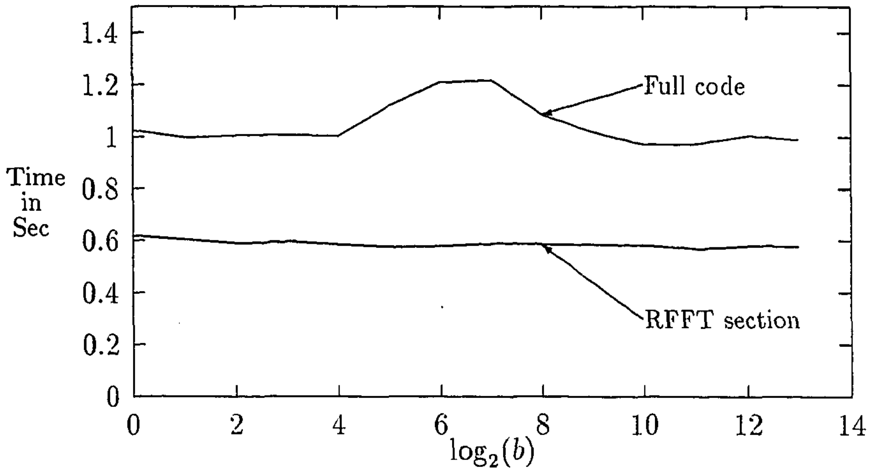


Figure 5.7: Variation in RFFT Performance with block size.

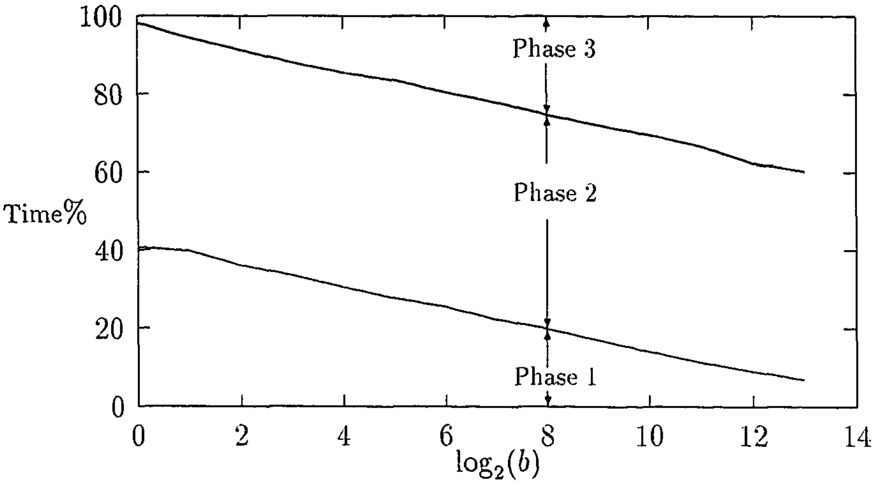


Figure 5.8: Relative contribution of the three phases of the FHT section with different block sizes.

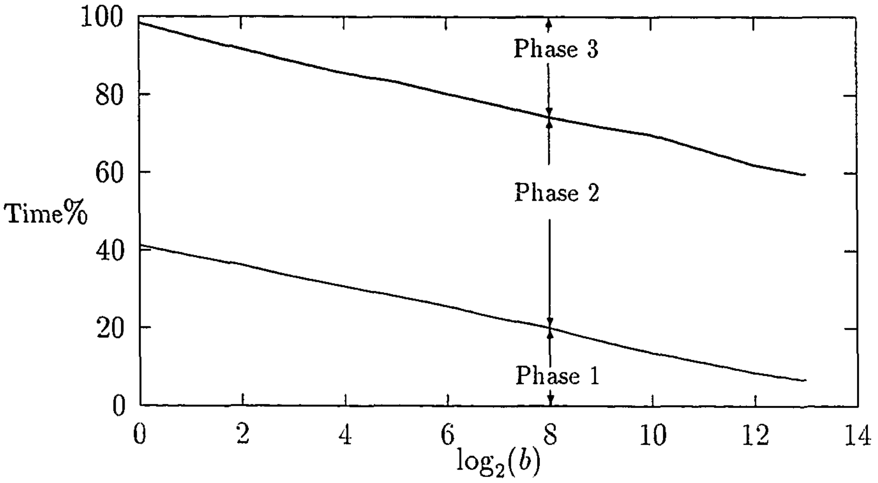


Figure 5.9: Relative contribution of the three phases of the RFFT section with different block sizes.

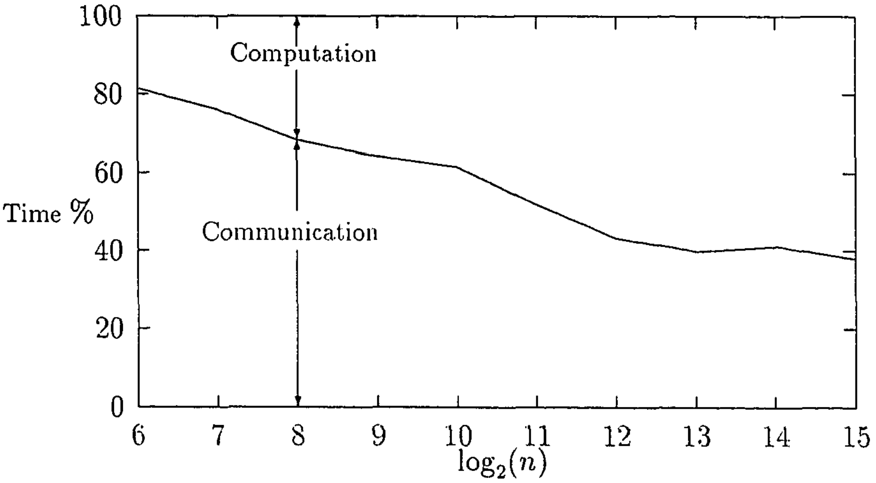


Figure 5.10: Relative contribution of communication in the computational section of the FHT algorithm.

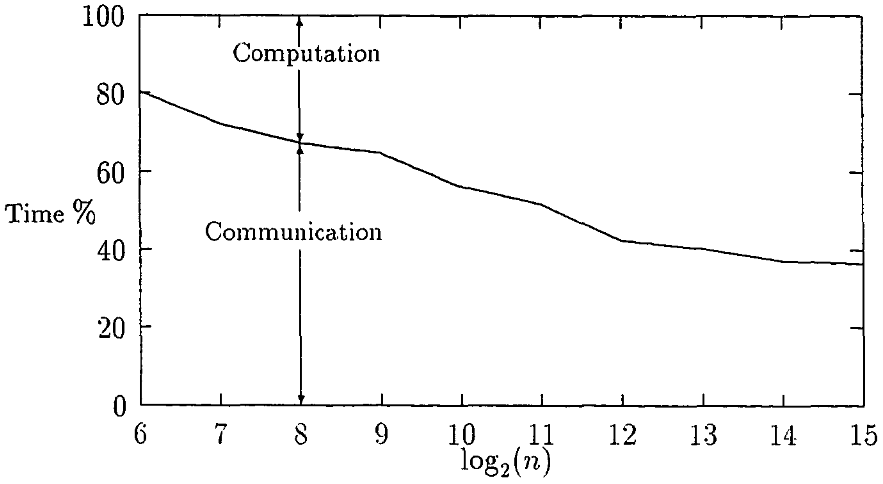


Figure 5.11: Relative contribution of communication in the computational section of the RFFT algorithm.

The results of comparison between the FHT, RFFT and FFT algorithms are shown in Figures 5.12-5.17. Figure 5.12 shows the best execution time of any block size for a given data size for all the three algorithms. Figure 5.13 shows similar curves for the worst execution times. In both the figures, FFT algorithm is the slowest and FHT algorithm is the fastest. The reason for the FFT being the slowest is that it uses complex arithmetic and has twice the communication volume of the other two algorithms in the computational section. This can be verified by observing Figure 5.14. Figure 5.14 shows the time taken by the three algorithms without including the rearrangement section. Here the FHT and RFFT algorithms have identical time, while the FFT algorithm takes longer. Same is true of the time taken by internode communication in the three algorithms as shown in Figure 5.15. The difference between the FHT and RFFT algorithm performance in Figures 5.12 and 5.13 can be attributed to the difference in the communication volume of the two algorithms in the rearrangement section. The variation in the performance of the three algorithms with block size is shown in Figure 5.16. All three curves peak in the middle, where the block sizes and the number of blocks are of the same order. The FFT algorithm also takes more time for a block size of 1, because the rearrangement for that block size is not a complete exchange and involves contention. There is also a difference in the general trend of the effect of block size on the rearrangement section of the FFT algorithm from those of the RFFT and FHT algorithms (see Figure 5.17. The curves for the RFFT and FHT algorithms have similar shape which differs from the curve for the FFT algorithm. The difference in the actual time taken by the rearrangement section of the FHT and RFFT algorithms is because of the communication volume. The data being rearranged in the FHT algorithm consist of real numbers while they are complex numbers for the RFFT algorithm. The curve for the FFT algorithm lies between the curves for the RFFT and FHT algorithms. The reason for this curious behavior can be attributed to the difference in the data distribution at the end of the computational section. At extreme block sizes, the FFT and the RFFT algorithms take approximately the same time in rearrangement. This indicates that for those block sizes the rearrangement problem for RFFT and FHT algorithms is same as the complete exchange. However, when the order of block size is the same as the number of blocks, the FFT algorithm curve is closer to the FHT algorithm curve. This indicates that for these block sizes, the data distribution of the FFT algorithm is more suitable for rearrangement than the other two algorithms. The FHT algorithm is close in its performance to the FFT algorithm because it deals with real numbers and therefore has less communication volume.

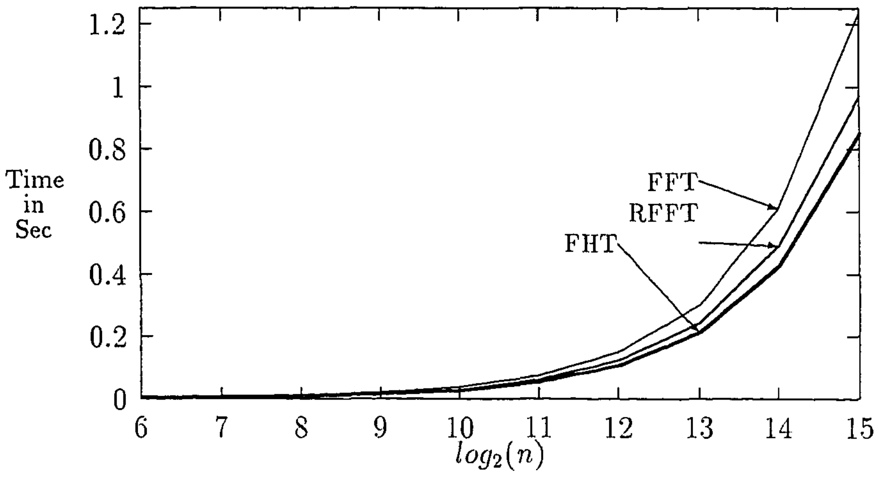


Figure 5.12: Comparison of the best performances of the FFT, the RFFT and the FHT algorithms.

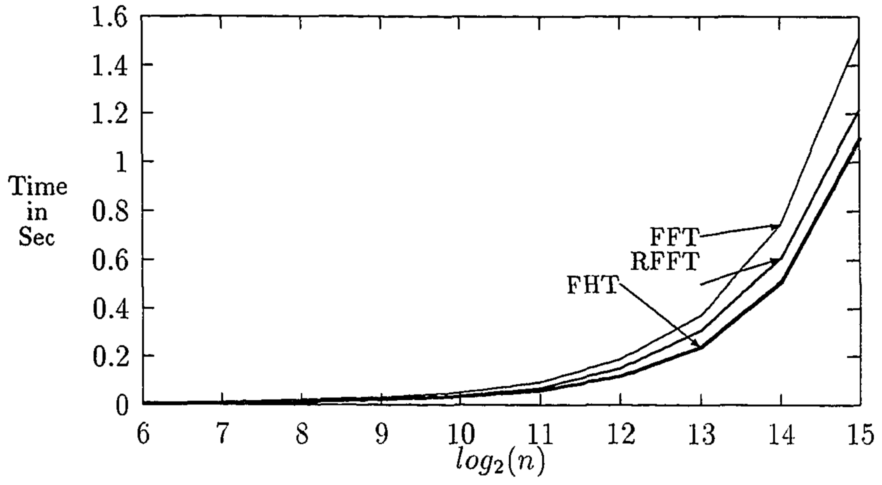


Figure 5.13: Comparison of the worst performances of the three algorithms.

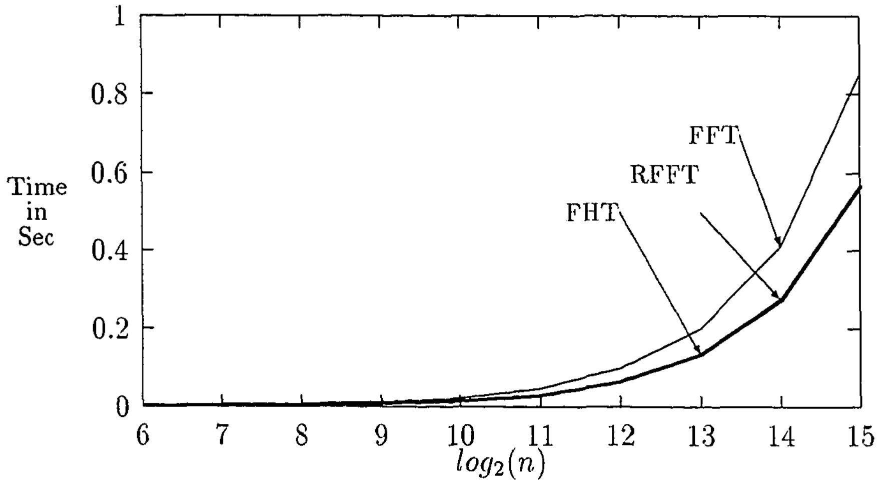


Figure 5.14: Comparison of execution times in the computational section.

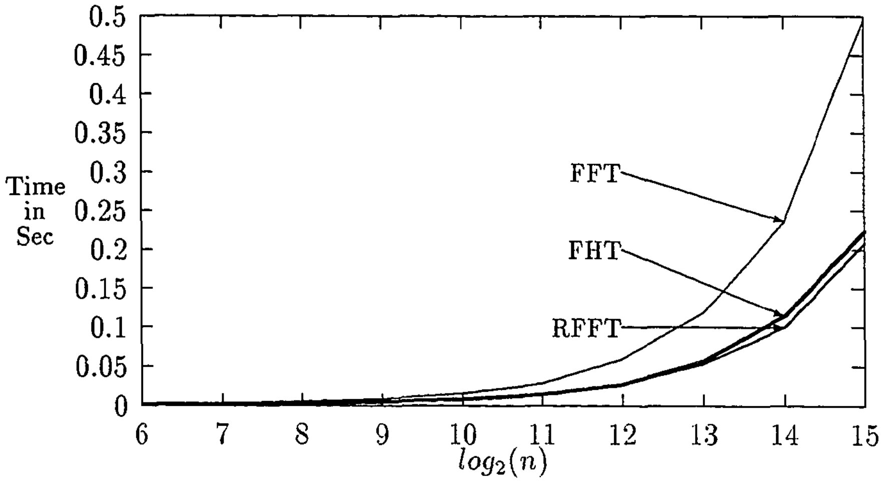


Figure 5.15: Comparison of the contribution of internode communication in the computational phase.

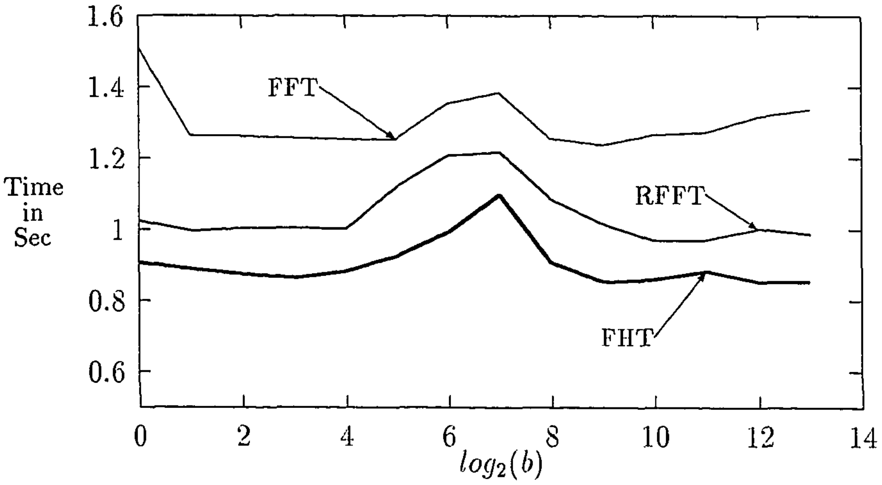


Figure 5.16: Comparison of variation in performance with block size.

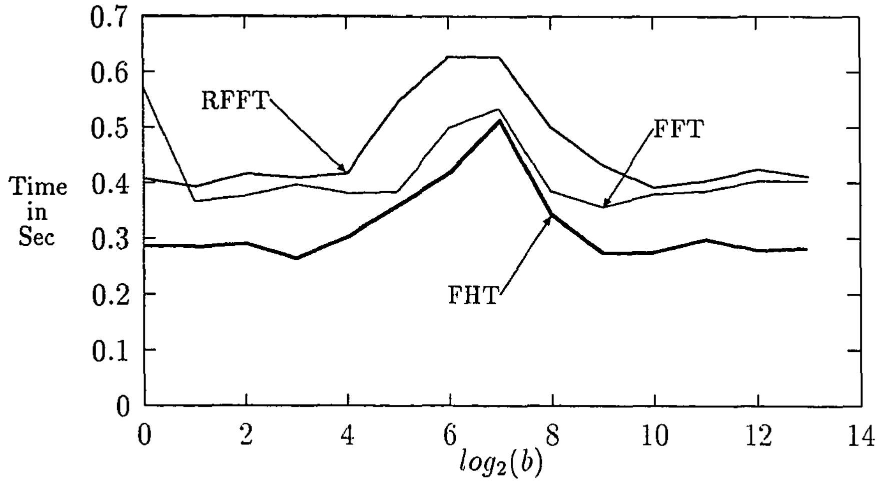


Figure 5.17: Comparison of the time taken in the rearrangement section.

5.6 Summary

In this chapter, implementations were given for the FHT and the RFFT algorithms which work for block scattered data distributions with different block sizes on distributed memory machine. A 32 node Intel iPSC/860 machine was used to evaluate these implementations. The performance of these implementations was compared with that of the FFT algorithm which also works for block scattered data distributions with different block sizes. Our experiments indicate that the use of FHT and RFFT algorithms is more efficient for computing the DFT of real data. The FHT algorithm gives the best performance of the three algorithms. Unless the transformed data are required in the complex form the FHT algorithm is the best choice. However, if the block size is such that less than 4 blocks are mapped on a node, the FFT algorithm must be used. The variation in performance with the block size is very similar in all the three algorithms and the FHT and RFFT algorithms consistently outperform the FFT algorithm. There is very little variation in the performance of the computational section of the algorithms with the block size.

Chapter 6

Summary and Future Studies

6.1 Summary

In this dissertation we presented parallel implementations for computing discrete Fourier transforms on distributed memory machines. Efficient implementations were also given for computing the DFT of real data. The implementations presented here compute DFT for block scattered data distributions with different block sizes. The block scattered data distributions are extremely useful for scientific computations and encompass the linear and scattered data distributions. These algorithms can be used without an initial data rearrangement in applications having block scattered data distributions. The only constraint is that for computing the DFT of real data, at least four blocks must be mapped on a node. The algorithms also return the output data in the same distribution as the input. Each algorithm consists of two sections; one computes the transform and the other rearranges data in the same order as the input.

There is a threshold in the size of data below which these implementations do not give a sufficiently good performance. The threshold is mainly due to the communication overheads which are very significant for small data sizes in distributed memory parallel machines. The relative contribution of the communication overheads decreases as the data size is increased. The overall performances of all three algorithms have similar trends. Their performance is worst when the number of blocks and block sizes are of the same order. The block size has very little effect on the performance of the computational section of the algorithms. However, the performance of the rearrangement section vary greatly with the block size, and account for the variation in the overall performance.

The very slight variation in the performance of the computational section is likely to be because of the variation in the execution times of different runs. The execution time for the same block and data size varies from one run to another as can be observed from Figures 6.1 and 6.2. In Figure 6.1 execution times are shown for 50 different runs of the FFT algorithm on 32 nodes with 16 data points per node and the block size of 4. Figure 6.2 is similar to Figure 6.1 except that there are 64k data points per node and the block size is 256. It can be observed from the two figures that the relative variation is larger for smaller data size, when the contribution from the communication overheads is significant. The RFFT and the FHT algorithms give almost identical performance in the computation section, which is about 1.3 times faster than the FFT algorithm. The RFFT algorithm shows the worst performance in the rearrangement section while the FHT algorithm gives the best performance, even though the two have identical data distributions. The difference is because of the difference in the volume of data being moved by the two algorithms. The difference between the RFFT and FFT performances is due to the different data distributions even though they have identical volume of data. For all block sizes, the RFFT and FHT algorithms outperform the FFT algorithm.

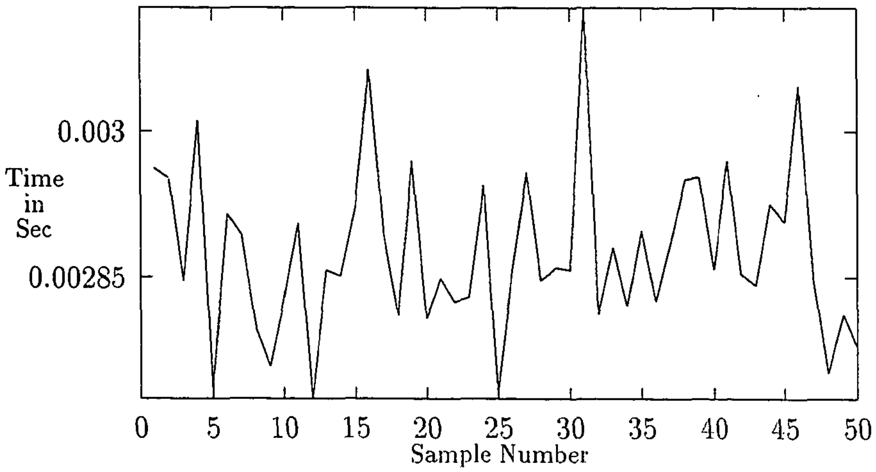


Figure 6.1: Variation in timing for different samples with 16 data points per node.

The implementations given in this work can be easily transported to other distributed memory machines with different architectures. This is especially true of the computation section because of the kernel. The computational kernels for all

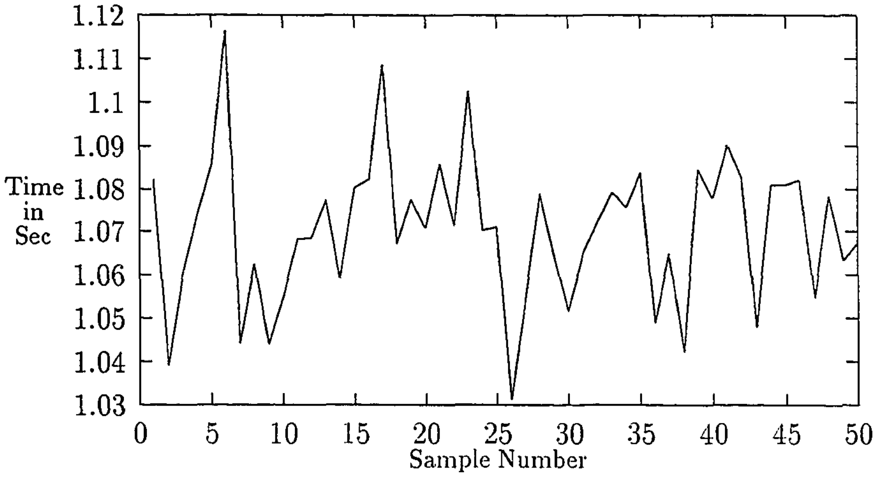


Figure 6.2: Variation in timing for different samples with 32, 678 data points per node.

three algorithms are very simple and regular in their structure and this makes the task of transporting them easy. However, the rearrangement problem is specific to a machine, especially where the data are exchanged between a subset of nodes. Much better performances can be obtained by optimizing (exploiting the architectural features such as cache, number of registers, pipelining etc. of a processor) the kernels and internode communication for a specific machine (about 25% expected for iPSC/860).

6.2 Suggestions for Further Study

The implementations suggested in this dissertation use the FFT, RFFT and the FHT algorithms when data size and block size are a power of 2. There are several variants of the FFT algorithm which work for arbitrary size data. It would be useful to find their implementations on distributed memory parallel machines which can support block scattered data distributions with arbitrary block sizes. It would also be interesting to see how the rearrangement problem varies with different architectures and different routing algorithms. Here the rearrangement problem was studied only on an MIMD hypercube based machine. The performance of the rearrangement section is likely to be affected by the interconnection network, the strategy for sending and receiving messages and the routing of messages. In addition, whether a machine is SIMD or MIMD is also likely to play a role in the rearrangement of data. It could be useful to find routing algorithms specific to the applications similar to the ones described in this work. A comparative study of the performance of these algorithms on different distributed memory architectures can also help to determine the type of machines most suited for applications involving extensive use of Fourier transforms. It would also be interesting to see the performance of multidimensional transforms on distributed memory machines. Also, there is need to study similar general implementations for the shared memory architectures.

Bibliography

[1] R.C. Agarwal and J.W. Cooley, "Fourier Transform and Convolution Subroutines for IBM 3090 Vector Facility," *IBM Journal of Research and Development*, vol. 30, No. 2, March 1986, pp. 145-162.

[2] M. Ashworth and A.G. Lyne, "A Segmented FFT algorithm for Vector Computers," *Parallel Computing*, vol. 6, 1988, pp.217-224.

[3] A. Averbuch, E. Gabber, B. Gordissky and Y. Medan, " A Parallel FFT on an MIMD Machine," *Parallel Computing*, vol. 15, 1990, pp. 61-74.

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

[5] D.H. Bailey, "A High-Performance FFT Algorithm for Vector Supercomputers," *International Journal of Supercomputing*, vol. 2, 1988, pp. 82-87.

[6] D.H. Bailey, and P.O. Frederickson, "Performance Results for Two of the NAS Parallel Benchmarks," in *Supercomputing '91.* (pp.166-173).

[7] G. D. Bergland, "A Fast Fourier Transform Algorithm for Real-valued Series," *Comm. ACM*, vol. 11, No. 10, October 1968, pp. 703-710.

[8] G.D. Bergland and D.E. Wilson, "A Fast Fourier Transform Algorithm for a global highly parallel processor," *IEEE Transactions on Audio and Electroacoustics,* vol. AU-17, No. 2, June 1969, pp. 125-127.

[9] G.D. Bergland, "A Parallel Implementation of the Fast Fourier Transform Algorithm," *IEEE Transactions on Computers,* vol. C-21, No. 4, April 1972, pp. 366-370.

[10] Shahid H. Bokhari, "Complete Exchange on the iPSC/860," *ICASE Report* No. 91-4.

[11] Shahid H. Bokhari, "Multiphase Complete Exchange on a Circuit Switched Hypercube," *ICASE Report No.* 91-5.

[12] R.N. Bracewell, *The Hartley Transform*, Oxford University Press, 1986.

[13] A. Brass and G.S. Pawley, "Two and Three Dimensional FFTs on Highly Parallel Computers," *Parallel Computing*, vol. 3, 1986, pp. 167-184.

[14] W.L. Briggs, L.B. Hart, R.A. Sweet and A. O'Gallagher, "Multiprocessor FFT Methods,", *SIAM Journal of Scientific and Statistical Computing,* vol. 8. No. 1, January 1978, pp. 10-42.

[15] O. Buneman, "Conversion of FFTs to Fast Hartley Transforms," *SIAM J. Sci. Stat. Comput.*, vol.7, No.2, April 1986.

[16] D. A. Carlson, "Using Local Memory to Boost Performance of FFT algorithms on the CRAY-2 Supercomputers," *Journal of Supercomputing*, vol. 4, 1991, pp. 345-356.

[17] D. A. Carlson, "Ultrahigh-Performance FFTs for the CRAY-2 and CRAY Y-MP Supercomputers," *The Journal of Supercomputing, vol. 6, 1992, pp. 107-116*

[18] R.M. Chamberlain, "Gray Codes, Fast Fourier Transforms and Hypercubes," *Parallel Computing* 6 (1988), pp.225-233.

[19] R.A. Collesidis, T.A. Dutton and J.R. Fisher, " An Ultra High Speed FFT Processor," *IEEE International Conference on Acoustics, Speech and Signal Processing*, April 1980, pp. 784-787.

[20] M.J. Corinthios, "The Design of a Class of Fast Fourier Transform Computers," *IEEE Transactions on Computers*, vol. C-20, 1971, pp. 617-623.

[21] J.W. Cooley and J.W. Tukey, "An Algorithm for the Calculation of Complex Fourier Series," *Mathematics of Computation,* vol. 19, 1965, pp. 297-301.

[22] J.W. Cooley, P.A.W Lewis and P.D. Welch, "The Fast Fourier Transform Algorithm: Programming Considerations in the Calculation of Sine, Cosine and Laplace Transforms," *J. Sound Vibration* 12, 1970, 315-337.

[23] B. Fornberg, "A Vector Implementation of the Fast Fourier Transform Algorithm," *Mathematics of Computation*, vol. 36, 1981, pp. 189-191.

[24] L.R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, Prentice-Hall, 1975.

[25] R.W. Hockney and C.R. Jesshope, *Parallel Computers*, Adam Higler, 1981.

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

[27] C.R. Jesshope, "The implementation of the Fast Radix 2 Transform on Array Processor," *IEEE Transactions on Computers*, vol. C-29, No. 1, January 1980, pp. 20-27.

[28] J. Johnson, R.W. Johnson, D. Rodriguez and R. Tolimieri, "A Methodology for Designing, Modifying and Implementing Fourier Transform Algorithms on Various Architectures," *Circuits, Systems and Signal Processing,* vol. 9, 1990, pp. 449-500

[29] S.L. Johnsson, C.T. Ho, M. Jacquemin and A. Ruttenburg, "Computing Fast Fourier Transform on Boolean Cubes and Related Networks," In *SPIE Advanced Algorithms and Architectures for Signal Processing II*, vol. 826, 1987, pp. 223-231

[30] L. Johnsson, R.L. Krawitz, D. MacDonald and R. Frye, "A Radix 2 FFT on the Connection Machine," *Proceedings of Supercomputing 89*, November 1989, pp. 809-819.

[31] S. L. Johnsson. M. Jacquemin and C.T. Ho, "High Radix FFT on Boolean Cube Networks," *Technical Report NA89-7*, Thinking Machines Corporation, 1989.

[32] S.L. Johnsson and R.L. Krawitz, "Communication Efficient Multiprocessor FFT," *Division of Applied Sciences Report TR-25-91,* Harvard University, 1991.

[33] S.L. Johnsson and R.L. Krawitz, "Cooley-Tukey FFT on the Connection Machine," *Parallel Computing,* vol. 18, 1992, pp. 1201-1221.

[34] R.A. Kamin III and G.B. Adams III, "Fast Fourier Transform Algorithm Design and Tradeoffs on the CM-2," in *Proceedings of the Conference on Scientific Applications on The Connection Machine*, September 1988, pp.134-160.

[35] D.G. Korn and J.J. Lambiotte Jr., "Computing the Fast Fourier Transform on a Vector Computer," *Mathematics of Computation,* vol. 33, 1979, pp. 979-992.

[36] C.L. Lawson, R.J. Hanson, and D.R. Kincaid, and F.T. Krogh, "Basic linear algebra subprogram for FORTRAN usage," *ACM Trans. Math. Soft. 5*, 3, 1979, pp308-323.

[37] O.A. Mcbryan, "Connection machine application performance," *Proceedings of the NASA-Ames Sci. Appl. of Connection Machine,* 1989.

[38] M.C. Pease, "An adaptation of the Fast Fourier Transform for Parallel Processing," *Journal of the ACM,* vol. 15, No. 2, April 1968, pp. 252-264.

[39] R.B. Pelz, "The parallel Fourier pseudospectral method," *Journal of Computational Physics*, to appear.

[40] Thomas Schmiermund and Steven R. Seidel, "A Communication Model for the Intel iPSC/2," *Computer Science Technical Report* CS-TR 90-02, Michigan Tech, Univ, April 1990.

[41] Steven R. Seidel, Ming-Horng Lee and Shivi Fotedar "Concurrent Bidirectional Communication on the Intel iPSC/860 and iPSC/2," *Computer Science Technical Report* CS-TR 90-06, Michigan Tech, Univ, November 1990.

[42] H.V. Sorensen, D.G. Jones, M.T. Heideman and C. S. Burrus, "Real-Valued Fast Fourier Transform Algorithms," *IEEE Trans. ASSP*, vol. ASSP-35, No. 6, June 1987, pp. 849-863.

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

[44] P.N. Swarztrauber, "Symmetric FFTs," *Mathematics of Computation* vol. 47, 1986, pp.323-346.

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

[46] , " Bluestein's FFT for Arbitrary N on the Hypercube," *Parallel Computing*, vol. 17, 1991, pp. 607-617.

[47] C. Tong and P.N. Swarztrauber, "Ordered Fast Fourier Transform on a Massively Parallel Hypercube Multiprocessor," *Journal of Parallel and Distributed Computing*, vol. 12, 1991, pp. 50-59.

[48] C. Temperton, "Implementation of a Prime Factor FFT Algorithm on Cray-1," *Parallel Computing*, vol. 6, 1988, pp. 99-108.

[49] D. W. Walker and J. Dongarra, "Design Issues for a Scalable Library Algebra Subroutines," *Draft Preprint, Oak Ridge National Laboratory* October 1991.

[50] M.A. Wesley, "Associative Parallel Processing for the Fast Fourier Transform," *IEEE Transactions on Audio and Electroacoustics*, vol. AU-17, No. 2, June 1969, pp. 162-165.

AUTOBIOGRAPHCAL STATEMENT

Anshu Dubey was born on January 14, 1964, at Agra in India. She received her B.Tech degree in Electrical Engineering in December 1985, from the Indian Institute of Technology (I.I.T), New Delhi. She was awarded M.S. degree in Electrical Engineering in March 1990, by Auburn University at Auburn, Alabama. She held the position of Engineer at Bharat Electronics Ltd. in Ghaziabad, India from July 1985 to November 1985. In November 1985, she joined the Center for Applied Research in Electronics, I.I.T., New Delhi, as a Senior Research Assistant. She was employed there until May 1986. In August 1986, she joined the Department of Computer Science and Engineering, I.I.T., New Delhi, as Senior Research Assistant. She was employed there until December 1987, when she left for Auburn University to join the M.S. program.