# THE FAST HARTLEY TRANSFORM ON THE HYPERCUBE MULTIPROCESSORS

Xinming Lin
Tony F. Chan
Walter J. Karplus

February 1988
CSD-880011

# The Fast Hartley Transform on the Hypercube Multiprocessors

Xinming Lin, Computer Science Department
Tony F. Chan, Mathematics Department
Walter J. Karplus, Computer Science Department
University of California, Los Angeles

## Abstract

The Fast Hartley Transform is a promising alternative to the Fast Fourier Transform when the processed data are real numbers. The hypercube implementation of the FHT is largely dependent on the way the computation is partitioned. A partitioning algorithm is presented which generates evenly-loaded tasks on each node and demands only a regular communication topology --- the Hartley graph. Mapping from the Hartley graph to the Gray graph (binary n-cube) is straightforward, since the Hartley graph has a similar structure as the Gray graph. However, the communication is not always between the nearest neighbors and thus may take some extra time. Moreover, the slowness of the communication in the presently available architectures imposes a limitation on the speedup when a large number of processors are used.

## 1. Introduction to the Fast Hartley Transform

Recall the discrete Fourier transform,

$$H_m = \sum_{n=0}^{N-1} [\cos(2*PI*m*n/N) - i \sin(2*PI*m*n/N)]\, h_n$$

$$(m = 0, 1, ..., N-1)$$

It can be seen that the Fourier transform requires complex arithmetics even when the input data are real numbers. One complex addition involves two real additions and one complex multiplication requires four real multiplications and two real additions. For improved efficiency and simplicity, the Hartley transform was developed. Its discrete form is

$$H_m = \sum_{n=0}^{N-1} [\cos(2*PI*m*n/N) + \sin(2*PI*m*n/N)]\, h_n$$

$$(m = 0, 1, ..., N-1)$$

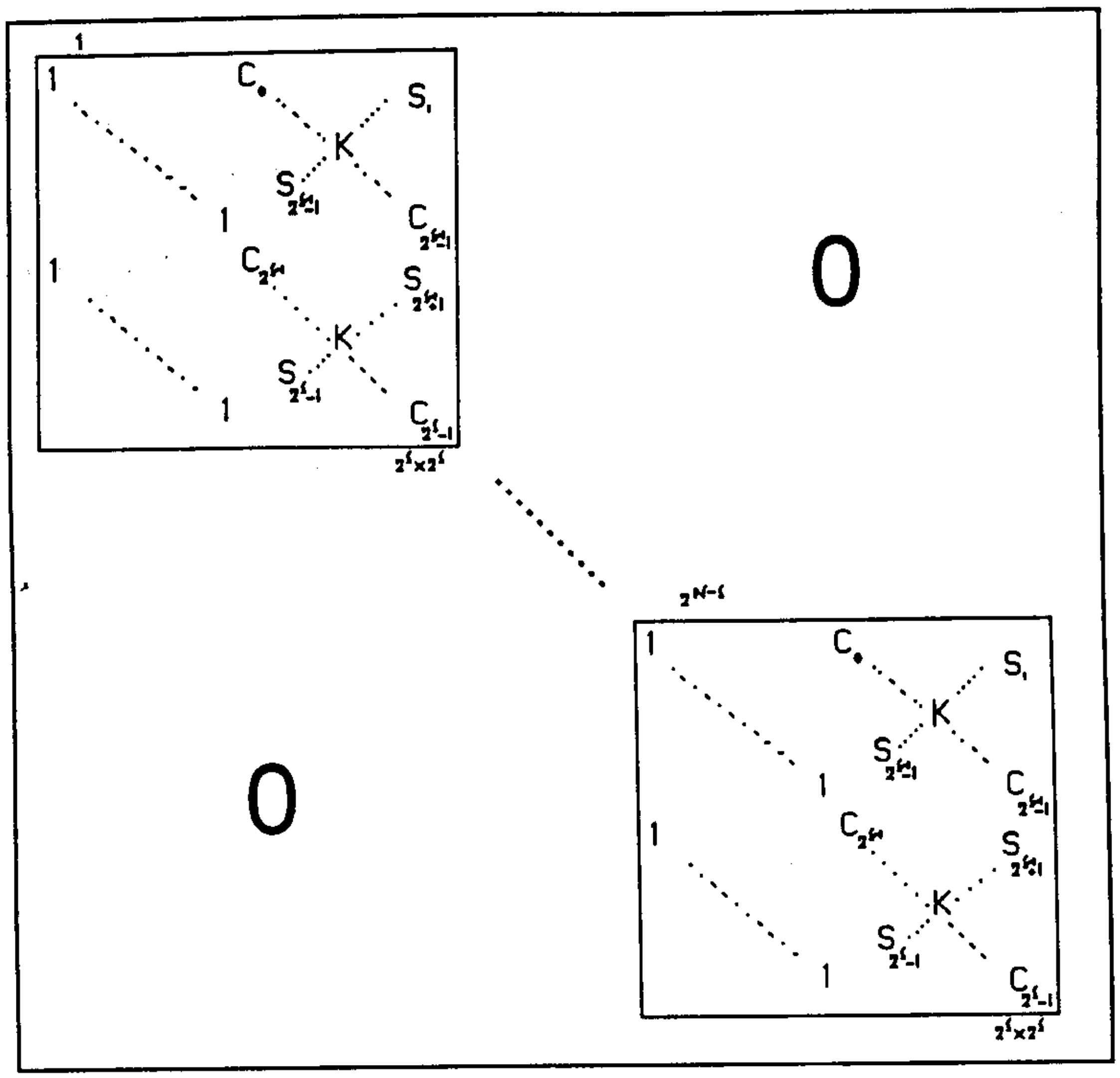
where $h_n$ are real numbers. In the Hartley transform, no complex arithmetics are required. Moreover, it has been shown that the Hartley transform has the same power as the Fourier transform in various applications [Bracewell 1986, Buneman 1986]. As in the Fourier transform, a Fast Hartley Transform (FHT) also exists and can be described in a matrix format:

$$\mathbf{H} = L_{\log(N)} \cdots L_4 L_3 L_2 L_1 P_n \mathbf{h}$$

where H= (H0, H1, H2, ..., Hn)*T* and h= (h0, h1, h2, ..., hn)*T*. **Pn** is a permutation matrix which exchanges **hx** with **hy**, where $y = i_0 i_1 \ldots i_{\log N}$ if $x = i_{\log N} \ldots i_1 i_0$. L's are matrices defined in Figure 1. As an example, the computations of the FHT of 16 data points is listed in Table 1.

***Figure 1: The Factor Matrices***

$$L_s =$$



where

$$C_i = \cos( 2 * PI * i / 2^s ),$$

$$S_i = \sin ( 2 * PI * i / 2^s ),$$

and $K_i = C_i + S_i$

(s = 1, 2, ... , N) .

**Table 1:** *Equations relating successive stages of the 16-element FHT as indexed by stage number s*

| Data | Permute | Stage 1 | Stage 2 |
|---|---|---|---|
| $F(0,0)$ | $F(0,0) \rightarrow F(0,0)$ | $F(0,0) + F(0,1) \rightarrow F(1,0)$ | $F(1,0) + F(1,2)C_0 + F(1,2)S_0 \rightarrow F(2,0)$ |
| $F(0,1)$ | $F(0,1) \rightarrow F(0,8)$ | $F(0,0) - F(0,1) \rightarrow F(1,1)$ | $F(1,1) + F(1,3)C_1 + F(1,3)S_1 \rightarrow F(2,1)$ |
| $F(0,2)$ | $F(0,2) \rightarrow F(0,4)$ | $F(0,2) + F(0,3) \rightarrow F(1,2)$ | $F(1,0) + F(1,2)C_2 + F(1,2)S_2 \rightarrow F(2,2)$ |
| $F(0,3)$ | $F(0,3) \rightarrow F(0,12)$ | $F(0,2) - F(0,3) \rightarrow F(1,3)$ | $F(1,1) + F(1,3)C_3 + F(1,3)S_3 \rightarrow F(2,3)$ |
| $F(0,4)$ | $F(0,4) \rightarrow F(0,2)$ | $F(0,4) + F(0,5) \rightarrow F(1,4)$ | $F(1,4) + F(1,6)C_0 + F(1,6)S_0 \rightarrow F(2,4)$ |
| $F(0,5)$ | $F(0,5) \rightarrow F(0,10)$ | $F(0,4) - F(0,5) \rightarrow F(1,5)$ | $F(1,5) + F(1,7)C_1 + F(1,7)S_1 \rightarrow F(2,5)$ |
| $F(0,6)$ | $F(0,6) \rightarrow F(0,6)$ | $F(0,6) + F(0,7) \rightarrow F(1,6)$ | $F(1,4) + F(1,6)C_2 + F(1,6)S_2 \rightarrow F(2,6)$ |
| $F(0,7)$ | $F(0,7) \rightarrow F(0,14)$ | $F(0,6) - F(0,7) \rightarrow F(1,7)$ | $F(1,5) + F(1,7)C_3 + F(1,7)S_3 \rightarrow F(2,7)$ |
| $F(0,8)$ | $F(0,8) \rightarrow F(0,1)$ | $F(0,8) + F(0,9) \rightarrow F(1,8)$ | $F(1,8) + F(1,10)C_0 + F(1,10)S_0 \rightarrow F(2,8)$ |
| $F(0,9)$ | $F(0,9) \rightarrow F(0,9)$ | $F(0,8) - F(0,9) \rightarrow F(1,9)$ | $F(1,9) + F(1,11)C_1 + F(1,11)S_1 \rightarrow F(2,9)$ |
| $F(0,10)$ | $F(0,10) \rightarrow F(0,5)$ | $F(0,10) + F(0,11) \rightarrow F(1,10)$ | $F(1,8) + F(1,10)C_2 + F(1,10)S_2 \rightarrow F(2,10)$ |
| $F(0,11)$ | $F(0,11) \rightarrow F(0,13)$ | $F(0,10) - F(0,11) \rightarrow F(1,11)$ | $F(1,9) + F(1,11)C_3 + F(1,11)S_3 \rightarrow F(2,11)$ |
| $F(0,12)$ | $F(0,12) \rightarrow F(0,3)$ | $F(0,12) + F(0,13) \rightarrow F(1,12)$ | $F(1,12) + F(1,14)C_0 + F(1,14)S_0 \rightarrow F(2,12)$ |
| $F(0,13)$ | $F(0,13) \rightarrow F(0,11)$ | $F(0,12) - F(0,13) \rightarrow F(1,13)$ | $F(1,13) + F(1,15)C_1 + F(1,15)S_1 \rightarrow F(2,13)$ |
| $F(0,14)$ | $F(0,14) \rightarrow F(0,7)$ | $F(0,14) + F(0,15) \rightarrow F(1,14)$ | $F(1,12) + F(1,14)C_2 + F(1,14)S_2 \rightarrow F(2,14)$ |
| $F(0,15)$ | $F(0,15) \rightarrow F(0,15)$ | $F(0,14) - F(0,15) \rightarrow F(1,15)$ | $F(1,13) + F(1,15)C_3 + F(1,15)S_3 \rightarrow F(2,15)$ |

| Stage 3 | Stage 4 |
|---|---|
| $F(2,0) + F(2,4)C_0 + F(2,4)S_0 \rightarrow F(3,0)$ | $F(3,0) + F(3,8)C_0 + F(3,8)S_0 \rightarrow F(4,0)$ |
| $F(2,1) + F(2,5)C_1 + F(2,7)S_1 \rightarrow F(3,1)$ | $F(3,1) + F(3,9)C_1 + F(3,15)S_1 \rightarrow F(4,1)$ |
| $F(2,2) + F(2,6)C_2 + F(2,6)S_2 \rightarrow F(3,2)$ | $F(3,2) + F(3,10)C_2 + F(3,14)S_2 \rightarrow F(4,2)$ |
| $F(2,3) + F(2,7)C_3 + F(2,5)S_3 \rightarrow F(3,3)$ | $F(3,3) + F(3,11)C_3 + F(3,13)S_3 \rightarrow F(4,3)$ |
| $F(2,0) + F(2,4)C_4 + F(2,4)S_4 \rightarrow F(3,4)$ | $F(3,4) + F(3,12)C_4 + F(3,12)S_4 \rightarrow F(4,4)$ |
| $F(2,1) + F(2,5)C_5 + F(2,7)S_5 \rightarrow F(3,5)$ | $F(3,5) + F(3,13)C_5 + F(3,11)S_5 \rightarrow F(4,5)$ |
| $F(2,2) + F(2,6)C_6 + F(2,6)S_6 \rightarrow F(3,6)$ | $F(3,6) + F(3,14)C_6 + F(3,10)S_6 \rightarrow F(4,6)$ |
| $F(2,3) + F(2,7)C_7 + F(2,5)S_7 \rightarrow F(3,7)$ | $F(3,7) + F(3,15)C_7 + F(3,9)S_7 \rightarrow F(4,7)$ |
| $F(2,8) + F(2,12)C_0 + F(2,12)S_0 \rightarrow F(3,8)$ | $F(3,0) + F(3,8)C_8 + F(3,8)S_8 \rightarrow F(4,8)$ |
| $F(2,9) + F(2,13)C_1 + F(2,15)S_1 \rightarrow F(3,9)$ | $F(3,1) + F(3,9)C_9 + F(3,15)S_9 \rightarrow F(4,9)$ |
| $F(2,10) + F(2,14)C_2 + F(2,14)S_2 \rightarrow F(3,10)$ | $F(3,2) + F(3,10)C_{10} + F(3,14)S_{10} \rightarrow F(4,10)$ |
| $F(2,11) + F(2,15)C_3 + F(2,13)S_3 \rightarrow F(3,11)$ | $F(3,3) + F(3,11)C_{11} + F(3,13)S_{11} \rightarrow F(4,11)$ |
| $F(2,8) + F(2,12)C_4 + F(2,12)S_4 \rightarrow F(3,12)$ | $F(3,4) + F(3,12)C_{12} + F(3,12)S_{12} \rightarrow F(4,12)$ |
| $F(2,9) + F(2,13)C_5 + F(2,15)S_5 \rightarrow F(3,13)$ | $F(3,5) + F(3,13)C_{13} + F(3,11)S_{13} \rightarrow F(4,13)$ |
| $F(2,10) + F(2,14)C_6 + F(2,14)S_6 \rightarrow F(3,14)$ | $F(3,6) + F(3,14)C_{14} + F(3,10)S_{14} \rightarrow F(4,14)$ |
| $F(2,11) + F(2,15)C_7 + F(2,13)S_7 \rightarrow F(3,15)$ | $F(3,7) + F(3,15)C_{15} + F(3,9)S_{15} \rightarrow F(4,15)$ |

## 2. The Hypercube Implementation of the FHT

Unlike in the implementation of the Fast Fourier Transform, the hypercube implementation of the Fast Hartley Transform is complicated by the unsymmetrical structure of the computations after stage 2 (refer to Table 1). Some computations have the results of two computations in the previous stage as their inputs, while some other computations have three inputs. Fortunately, the FHT exhibits a regular computational pattern, which is quite unique. The computational pattern is found by first partitioning the computations and then identifying the communication topology.

The computations in each stage are partitioned according to the following algorithm written in the C language, which takes the node number and the stage number as the input and returns the list of the computations on the node as the output.

*Algorithm 1. Partition of the Computations*

```
Partition(node, stage, nfd, np, lst)

int node;   /* the node number */
int stage;  /* the stage number */
int nfd;    /* number of data points on the node */
int np;     /* number of nodes altogether */
int lst[];  /* list of the computations on the node at
            the stage */

{
    unsigned nd;
    unsigned i, tsss, t, tt, tmp;

    tsss= ~(~0 << stage);
t= node & tsss;
tt= (node >> stage-1) << stage;
lst[0]= tt | t;

tmp= 01 << stage-1;
if (t == 0)
nd= tt | tmp;
else
if (t == tmp)
nd= tt | 0;
else
nd= tt | ((t ^ ~0) + 01) & tsss;
lst[1]= nd;

for (i= 1; i< nfd/2; i++) {
lst[2*i]= lst[2*i-2]+ np*2;
lst[2*i+1]= lst[2*i-1]+ np*2;
}
}
```

The algorithm works as follow. The node number is first written in binary format, and the computations are referred by their indices. At each stage, two basic indices are used to generate all other indices. One of the two basic indices is composed so that the s least significant bits are the s least significant bits of the node number and the **(b-s+1)** next significant bits are the **(b-s+1)** most significant bits of the node number, where **s** is the stage number and **b** = **log(np)** in which **np** is the total number of nodes. The other index is generated from the first one by *complementing* the s least significant bits. The *complement* of s bits is defined as

(i) If the s bits are all 0's, then its complement is 1 followed by **s-1** 0's, or

(ii) if the s bits are 1 followed by **s-1** 0's, then its complement is s 0's. Otherwise,

(iii) its complement is the 2's-complement of the s bits.

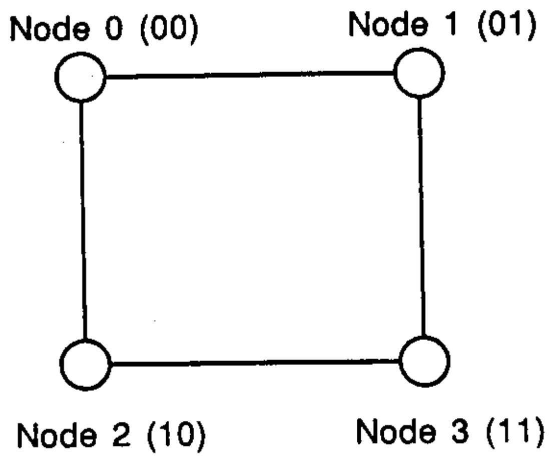
Table 2 is an example of partitioning the FHT of 16 data points onto 4 nodes.

***Table 2: Partitioning of the FHT of 16 data points onto 4 nodes***

| | stage 1 | stage 2 | stage 3 | stage 4 |
|---|---|---|---|---|
| Node 0 (00): | | | | |
| | 0000 {F(1,0)} | 0000 {F(2,0)} | 0000 {F(3,0)} | 0000 {F(4,0)} |
| | 0001 {F(1,1)} | 0010 {F(2,2)} | 0100 {F(3,4)} | 0100 {F(4,4)} |
| | 1000 {F(1,8)} | 1000 {F(2,8)} | 1000 {F(3,8)} | 1000 {F(4,8)} |
| | 1001 {F(1,9)} | 1010 {F(2,10)} | 1100 {F(3,12)} | 1100 {F(4,12)} |
| Node 1 (01): | | | | |
| | 0011 {F(1,3)} | 0001 {F(2,1)} | 0001 {F(3,1)} | 0001 {F(4,1)} |
| | 0010 {F(1,2)} | 0011 {F(2,3)} | 0111 {F(3,7)} | 0111 {F(4,7)} |
| | 1011 {F(1,11)} | 1001 {F(2,9)} | 1001 {F(3,9)} | 1001 {F(4,9)} |
| | 1010 {F(1,10)} | 1011 {F(2,11)} | 1111 {F(3,15)} | 1111 {F(4,15)} |
| Node 2 (10): | | | | |
| | 0100 {F(1,4)} | 0110 {F(2,6)} | 0010 {F(3,2)} | 0010 {F(4,2)} |
| | 0101 {F(1,5)} | 0100 {F(2,4)} | 0110 {F(3,6)} | 0110 {F(4,6)} |
| | 1100 {F(1,12)} | 1110 {F(2,14)} | 1010 {F(3,10)} | 1010 {F(4,10)} |
| | 1101 {F(1,13)} | 1100 {F(2,12)} | 1110 {F(3,14)} | 1110 {F(4,14)} |
| Node 3 (11): | | | | |
| | 0111 {F(1,7)} | 0111 {F(2,7)} | 0011 {F(3,3)} | 0011 {F(4,3)} |
| | 0110 {F(1,6)} | 0101 {F(2,5)} | 0101 {F(3,5)} | 0101 {F(4,5)} |
| | 1111 {F(1,15)} | 1111 {F(2,15)} | 1011 {F(3,11)} | 1011 {F(4,11)} |
| | 1110 {F(1,14)} | 1101 {F(2,13)} | 1101 {F(3,13)} | 1101 {F(4,13)} |

The communication topology of the example according to the partition algorithm is shown in Figure 2.

***Figure 2: Communication Topology of the example***

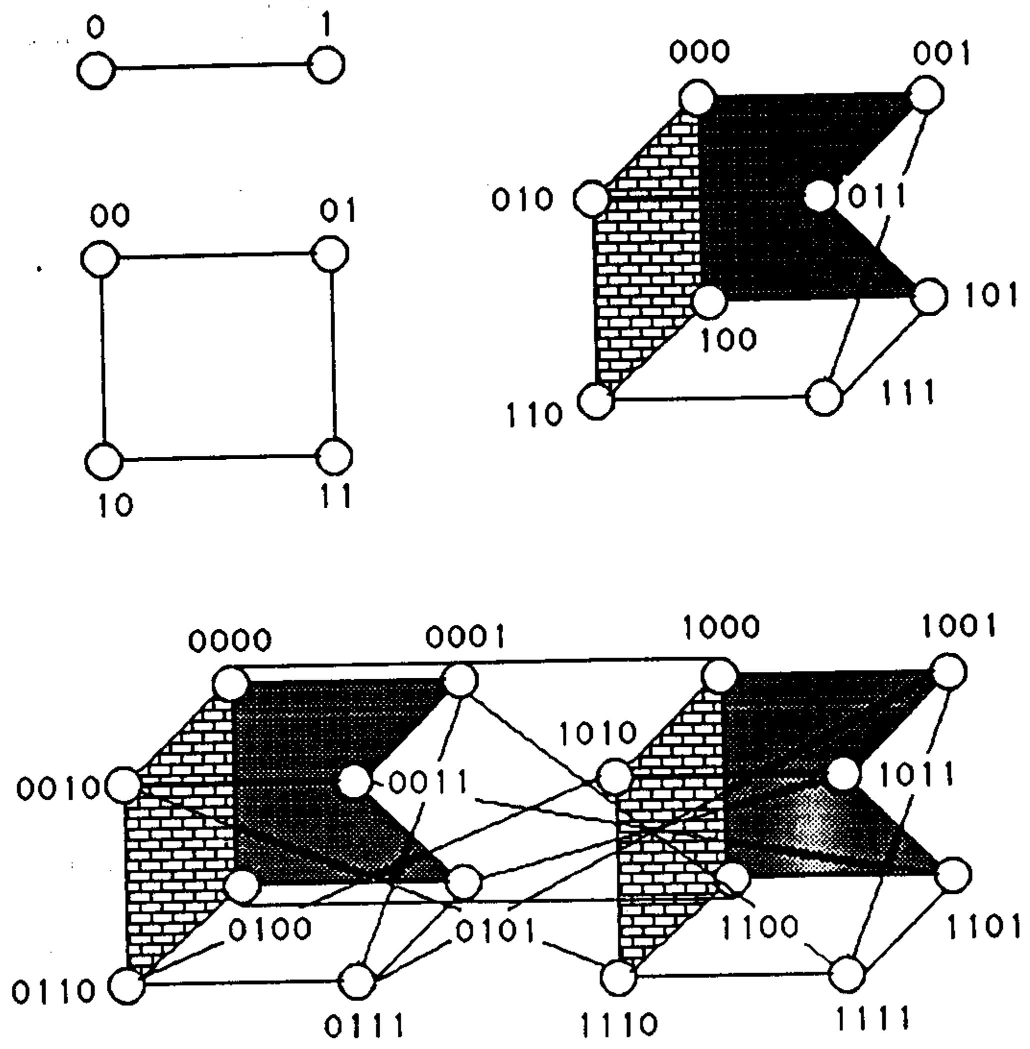


In general, the communication topology according to the partitioning algorithm in Figure 1 is a particular configuration termed the *Hartley Graph*. A Hartley graph of **np** nodes, where **np** is a power of 2, is defined as:

*Node i has a connection with node j if and only if j is a complement of i.*

As examples, the Hartley graphs of 2, 4, 8, and 16 nodes are shown in Figure 3.

***Figure 3: Hartley Graph of 1, 2, 3, and 4 Dimensions***

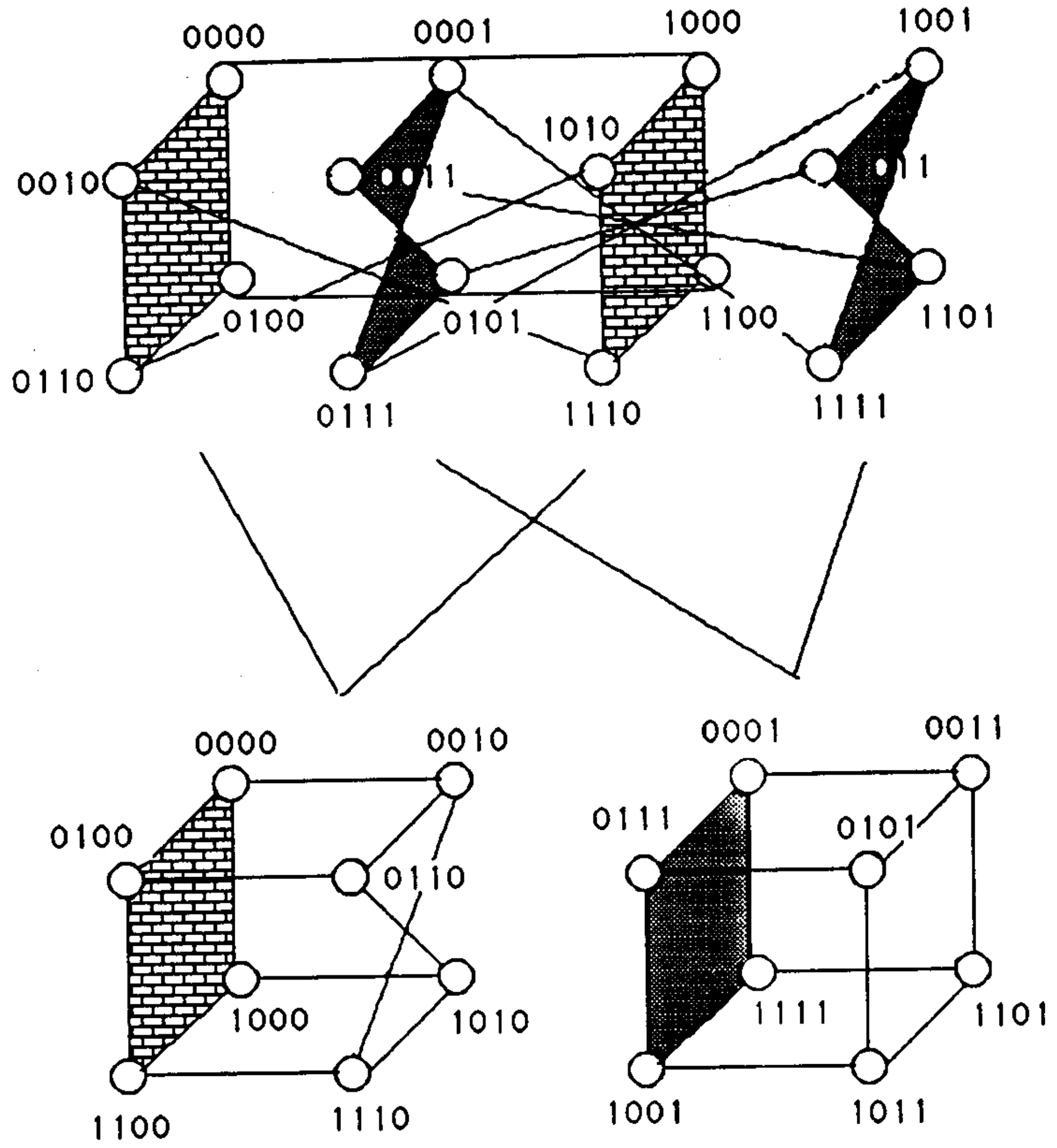


There are many properties connected with the Hartley graphs. The following is a very interesting one:

*if the connection in the first dimension is cut off, then a lower dimension Hartley graph and a lower dimension Gray graph (binary n-cube) remain.*

Figure 4 is the configuration after the first dimension connections are cut off from the 16-node Hartley graph.

***Figure 4: One Lower-Dimension Hartley Graph and One Lower-Dimension Gray Graph***



The implementation of the FHT on a hypercube multiprocessor machine is straightforward after the computational configuration in Figure 3 is determined. It is a mapping from the Hartley graph to the Gray graph (binary n-cube). The direct mapping which maps a node in the Hartley graph to the node with the same number in the Gray graph preserves much communication locality. In the Hartley graph, communication is issued along one dimension after each stage of the computation is completed, up to the number of dimensions of the graph. When the Hartley graph is mapped onto the Gray graph, some communications have to be routed along more than one dimension. The length of the longest path is the number of dimensions of the Hartley graph minus one.

An implementation of the FHT on an Intel iPSC hypercube multiprocessor machine is given in the Appendix. Its performance on different numbers of data points and on different numbers of processors is shown in Figure 5. The degraded performance when too many processors are used is due to the slow communication of the machine used, as illustrated in Figure 6, 7, and 8.

## 3. Conclusion

The Fast Hartley Transform is a promising alternative to the Fast Fourier Transform when the processed data are real numbers. The hypercube implementation of the FHT is largely dependent on the way the computation is partitioned. Our partitioning algorithm loads all nodes evenly and requires only a regular communication topology which we call the Hartley graph. Mapping from the Hartley graph to the Gray graph is straightforward, since the Hartley graph has a similar structure as the Gray graph. However, the communication is not always between the nearest neighbors and thus may require some extra time. Moreover, the slowness of the communication in the presently available architectures imposes a limitation on the speedup when too many processors are used.

## 4. Bibliography

[Bracewell 1986]: Bracewell, Ronald N.: *The Hartley Transform*, Oxford University Press, Inc. New York, N.Y., 1986

[Buneman 1986]: Buneman, Oscar: Conversion of FFT's to Fast Hartley Transforms, *SIAM Journal on Scientific and Statistical Computing,* V.7, N.2, April 1986.

*Figure 5:*

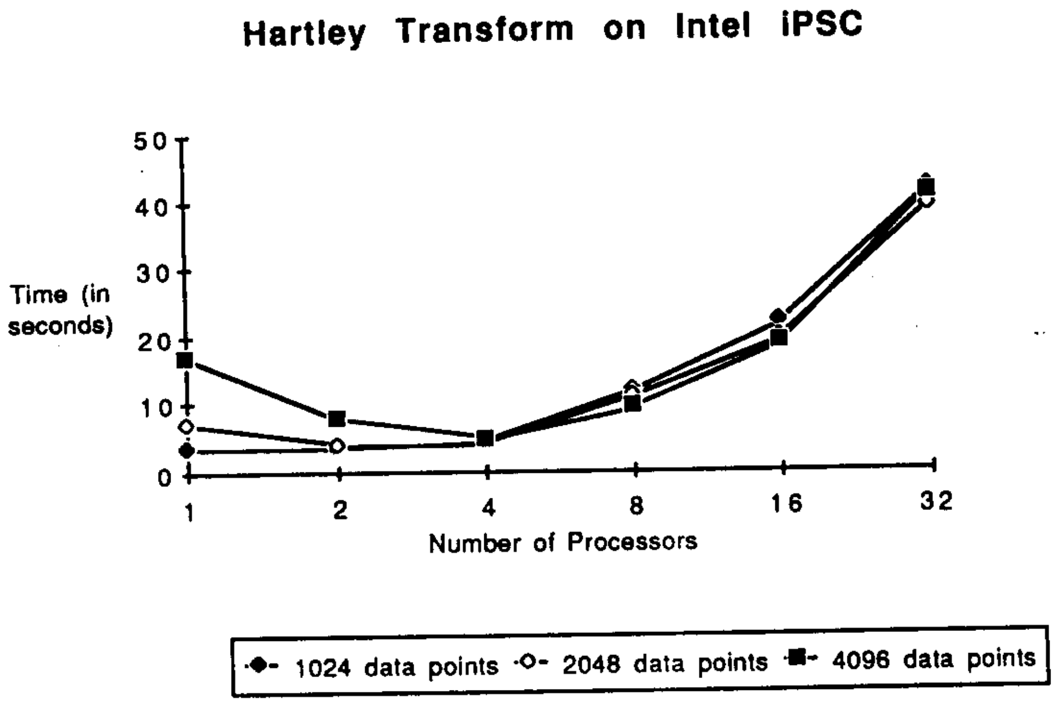


*Figure 6:*

Hartley Transform of 1024 Data Points

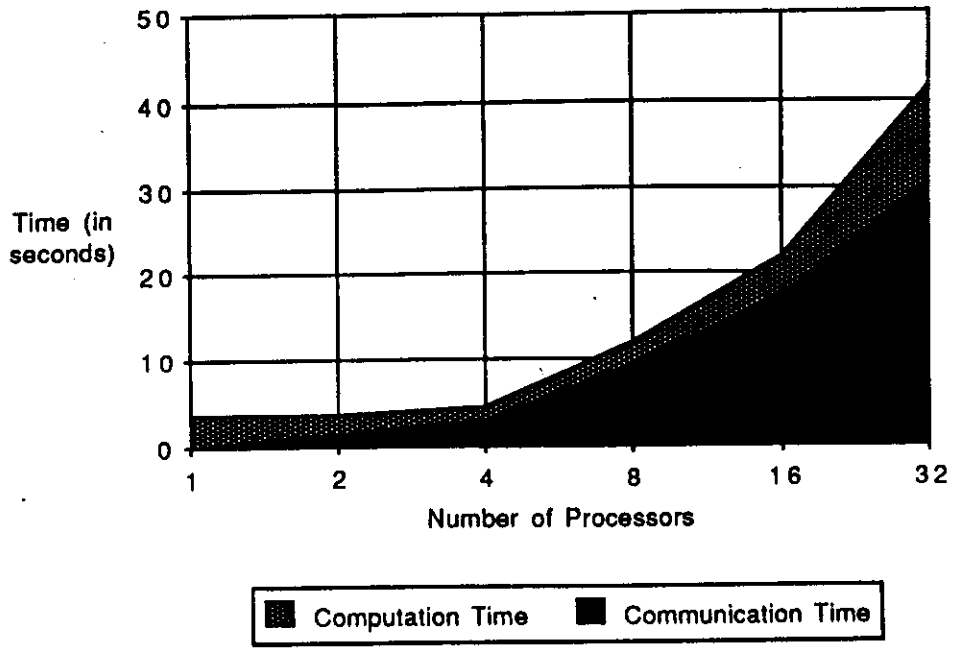


*Figure 7:*

Hartley Transform of 2048 Data Points

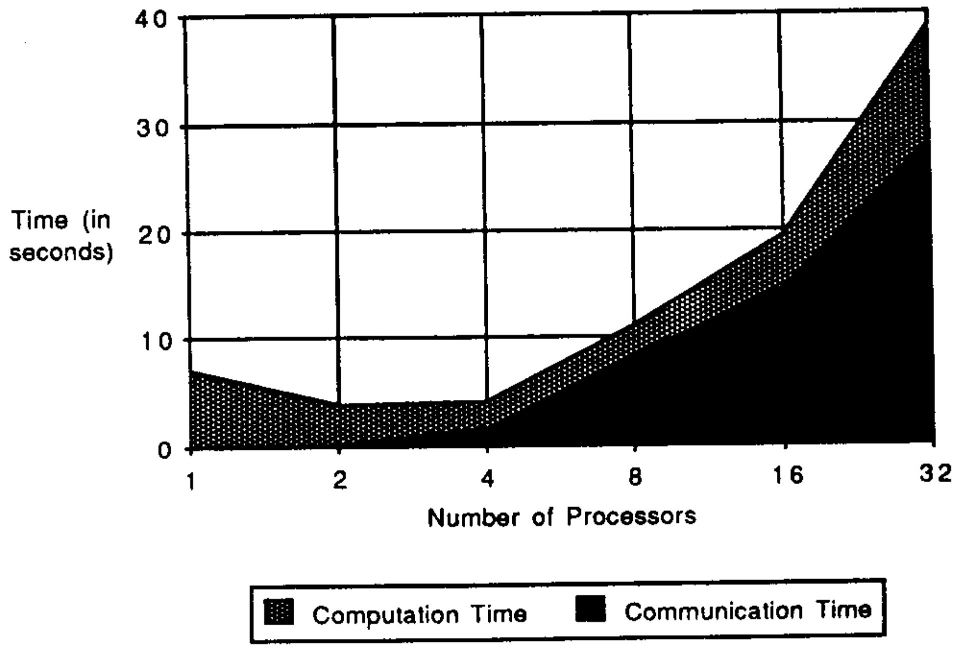
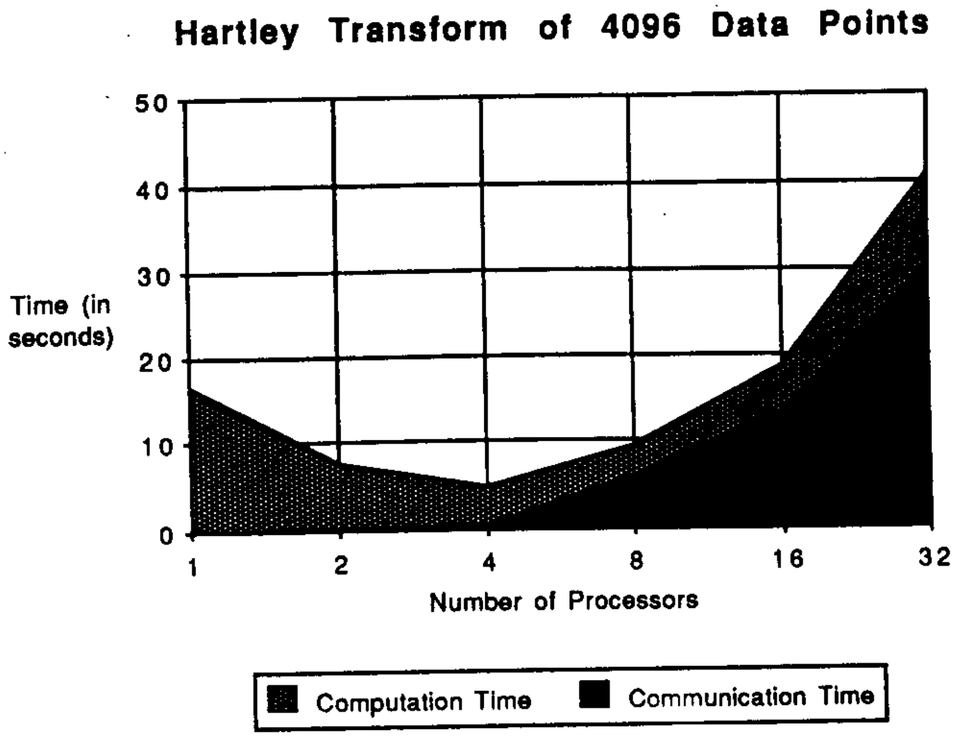


*Figure 8:*


Hartley Transform of 4096 Data Points
50
40
30
20
10
0
Time (in seconds)
1
2
4
8
16
32
Number of Processors
Computation Time
Communication Time

## Appendix

```
#include <stdio.h>
#include "/usr/ipsc/lib/chost.def"

#define MAXLEN          5000    /* maximum number of data points */
#define MLEN            3000    /* maximum number of input points */
#define LEN_TYPE        1000    /* message type for the parameters */
#define MSG_TYPE        2000    /* message type for input data */
#define PEM             4096
#define PEP             12
#define ALL_NODES       -1
#define PID             0
#define TIME_FACTOR     50.0

int     cid, type, cnt, node, pid,
        npoint,         /* number of data points */
        nproc,          /* number of processors */
        i,j, p2, t, pp,
        lst[MLEN],      /* list of the points on one node */
        perm[PEM],
        bit(),          /* function to decide the points on the node */
        permf();        /* function to calculate permutatin table */

float   a[MAXLEN],      /* input array */
        c[MAXLEN],      /* output array */
        mesg[MLEN];     /* message array */

struct  par_type {
        int nproc;      /* number of processors */
        int msg_len;    /* number of input data to each node */
        int log2N;      /* log2(N) */
        int log2P;      /* log2(P) */
} para;

FILE    *fopen(), *fp;

long    start_time,
        end_time,
        times(),
        dummy[4];

float   elapsed_time;

main() {
        printf("PLEASE WAIT WHILE I LOAD THE CUBE ...\n");
        load ("node", ALL_NODES, PID);
        cid= copen(PID);

/* input the number of points and the values */

        fp= fopen("input.data","r");
        fscanf(fp, "%d %d \n", &npoint, &nproc);
/*
        for (i= 0; i< npoint; i++)
           fscanf(fp,"%f",&a[i]);
*/
        for (i= 0; i< npoint; i++)
           a[i]= i;

        fclose(fp);
        fp= fopen("res", "w");

/* 2 to the p2's power is N */
        p2= 0;
        t= npoint;
        while ( t ^ 01 ) {
           p2++;
           t= t >> 1;
        }

/* 2 to the pp's power is nproc */
        pp= 0;
        t= nproc;
        while (t ^ 01) {
           pp++;
           t= t >> 1;
        }

/* broadcasting the number of processors and data length */

        para.nproc= nproc;
        para.msg_len= npoint/ nproc;
        para.log2N= p2;
        para.log2P= pp;
        sendmsg(cid, LEN_TYPE, &para, sizeof(para), ALL_NODES, PID);

/* calculate the permutation table */

        permf(PEM, PEP, perm);

/* interval in the perm table */

        t= 1;
        for (j= 0; j< PEP-p2; j++)
           t= t* 2;

        start_time= times(dummy);

/* send the input data to every node */

        for (i= 0; i< nproc; i++) {
           for (j= 0; j< para.msg_len/2; j++) {
                mesg[2*j+0]= a[perm[(2*i+0+j* 2*nproc)*t]];
                mesg[2*j+1]= a[perm[(2*i+1+j* 2*nproc)*t]];
           }
           sendmsg(cid, MSG_TYPE, mesg, sizeof(mesg), i, PID);
        }

/* receive and re-order the results back */

        for (i= 0; i< nproc; i++) {
           recvmsg(cid, &type, mesg, sizeof(mesg), &cnt, &node, &pid);
           bit(type, pp+1, para.msg_len, nproc, lst);
           for (j= 0; j< para.msg_len; j++)
                c[lst[j]]= mesg[j];
        }

        end_time= times(dummy);
        elapsed_time= (float) (end_time- start_time)/ TIME_FACTOR;

/* print the Hartley transform in a nice format */
/*
        for (i= 0; i< npoint;) {
           for (j= 0; (j< 4) && (i< npoint); j++)
                fprintf(fp, "%f\t",c[i++]);
           fprintf(fp, "\n");
        }
*/
```

```
        fprintf(fp, "\nThe elapsed time was: %0.4f secs.\n\n", elapsed_time);

/* release the cube */

        printf("PLEASE WAIT WHILE I CLEAN OUT THE CUBE ...\n");
        lkill(ALL_NODES, PID);
        lwaitall(ALL_NODES, PID);
        cclose(cid);
}
```

```
#include "/usr/ipsc/lib/cnode.def"
#include "/usr/include/cel287.h"

#define LEN_TYPE        1000
#define MSG_TYPE        2000
#define MLEN            3000
#define VS              2048
#define HOST            0x8000
#define PID             0
#define PI              3.1415926

int     k, k0, k1,      /* k0= 2*k+0, k1= 2*k+1, */
        t, tmp, i, j, n16, s,
        nfd,            /* number of data points to be produced */
        my_node, partner,
        cid, cnt, node, pid,
        bit(),          /* function to decide the ponits on the node */
        dest(),         /* communication destination */
        lst[MLEN];      /* list of the points on the node */

float   mesg[MLEN],
        phi, tau,
        phi1, phi2, phi3, phi4,
        c1, c2, c3, c4,
        s1, s2, s3, s4;

float   sm[VS];

struct  par_type {
        int nproc;
        int msg_len;
        int log2N;              /* log2(N) */
        int log2P;              /* log2(P) */
} para;

main() {
        cid= copen(mypid());
        my_node= mynode();

/* receive the number of processors and the input data length */

        recvw(cid, LEN_TYPE, &para, sizeof(para), &cnt, &node, &pid);

        nfd= para.msg_len;
        n16= nfd/ 2;

        if (my_node < para.nproc) {
           recvw(cid, MSG_TYPE, mesg, sizeof(mesg), &cnt, &node, &pid);

/* stage 1 */

           bit(my_node, 1, nfd, para.nproc, lst);

           if (my_node % 2 == 0)
           for (k= 0; k< n16; k++) {
              k0= 2*k+0;
              k1= 2*k+1;
              mesg[nfd+0]= mesg[k0]+ mesg[k1];
              mesg[nfd+1]= mesg[k0]- mesg[k1];
              mesg[k0]= mesg[nfd+0];
              mesg[k1]= mesg[nfd+1];
           } else
           for (k= 0; k< n16; k++) {
              k0= 2*k+0;
              k1= 2*k+1;
              mesg[nfd+0]= mesg[k0]- mesg[k1];
              mesg[nfd+1]= mesg[k0]+ mesg[k1];
              mesg[k0]= mesg[nfd+0];
              mesg[k1]= mesg[nfd+1];
           }

/* loop s */

           t= 2;
           for (s= 2; s<= para.log2P+1; s++) {

                t*= 2; /* t = 2^s */

/* comm s-1 */

           for (k= 0; k< nfd; k++)
                sm[k]= mesg[k];

           partner= dest(my_node, s-1);
           sendw(cid, my_node+10*s, sm, sizeof(sm), partner, PID);
           recvw(cid, partner+10*s, sm, sizeof(sm), &cnt, &node, &pid);

/* stage s */

           bit(my_node, s, nfd, para.nproc, lst);

           phi= 2.*PI*lst[0]/ t;
           tau= 2.*PI*lst[1]/ t;
           c1= cos(phi);
           c2= cos(tau);
           s1= sin(phi);
           s2= sin(tau);

           tmp= my_node % (t/4);
           for (k= 0; k< n16; k++) {
                k0= 2*k+ 0;
                k1= 2*k +1;

                if (tmp == 0 && my_node < partner) {
                        mesg[nfd+0]= mesg[k0]+ (c1+s1)*sm[k1];
                        mesg[nfd+1]= mesg[k0]+ (c2+s2)*sm[k1];
                        mesg[k0]= mesg[nfd+0];
                        mesg[k1]= mesg[nfd+1];
                }
                else if (tmp == 0 && my_node > partner) {
                        mesg[nfd+0]= sm[k1]+ (c1+s1)*mesg[k0];
                        mesg[nfd+1]= sm[k1]+ (c2+s2)*mesg[k0];
                        mesg[k0]= mesg[nfd+0];
                        mesg[k1]= mesg[nfd+1];
                }
                else if (tmp != 0 && my_node < partner) {
                        mesg[k0]+= c1*sm[k1]+ s1*sm[k0];
                        mesg[k1]+= c2*sm[k0]+ s2*sm[k1];
                }
                else {
                        /* tmp != 0 && my_node > partner*/
                        mesg[nfd+0]= sm[k1]+ c1*mesg[k0]+ s1*mesg[k1];
                        mesg[nfd+1]= sm[k0]+ c2*mesg[k1]+ s2*mesg[k0];
                        mesg[k0]= mesg[nfd+0];
                        mesg[k1]= mesg[nfd+1];
                }
        } /* end of for k */

        } /* end of for s */

/* > stage log2(P)+1 */

        t= 1;
        for (j= 0; j< para.log2N-(para.log2P+1); j++) {
            t*= 2;

            tau= para.nproc*2*t;
            for (k= 0; k< t/2; k++) {

              if (my_node == 0 && k == 0) {
                  phi1= 2.*PI*lst[0]/ tau;
                  phi2= 2.*PI*lst[t/2]/ tau;
                  phi3= 2.*PI*lst[t]/ tau;
                  phi4= 2.*PI*lst[t+t/2]/ tau;
              } else
              if (my_node == 0 && k != 0) {
                  phi1= 2.*PI*lst[k]/ tau;
                  phi2= 2.*PI*lst[t-k]/ tau;
                  phi3= 2.*PI*lst[t+k]/ tau;
                  phi4= 2.*PI*lst[2*t-k]/ tau;
              } else {
                  phi1= 2.*PI*lst[k]/ tau;
                  phi2= 2.*PI*lst[t-1-k]/ tau;
                  phi3= 2.*PI*lst[t+k]/ tau;
                  phi4= 2.*PI*lst[2*t-1-k]/ tau;
              }

              c1= cos(phi1);
              s1= sin(phi1);
              c2= cos(phi2);
              s2= sin(phi2);
              c3= cos(phi3);
              s3= sin(phi3);
              c4= cos(phi4);
              s4= sin(phi4);

              for (i= 0; i< nfd/(2*t); i++) {
                  tmp= i*2*t;

                  if (my_node == 0)
                     if (k == 0) {
                         mesg[nfd+0]= mesg[tmp]+ mesg[tmp+t];
                         mesg[nfd+1]= mesg[tmp+t/2]+
                                      (s2+c2)* mesg[tmp+t+t/2];
                         mesg[nfd+2]= mesg[tmp]+ c3* mesg[tmp+t];
                         mesg[nfd+3]= mesg[tmp+t/2]+
                                      (c4+s4)* mesg[tmp+t+t/2];
                         mesg[tmp]= mesg[nfd+0];
                         mesg[tmp+t/2]= mesg[nfd+1];
                         mesg[tmp+t]= mesg[nfd+2];
                         mesg[tmp+t+t/2]= mesg[nfd+3];
                     }
                     else {
                         mesg[nfd+0]= mesg[tmp+k]+ c1* mesg[tmp+t+k]
                                      + s1* mesg[tmp+2*t-k];
                         mesg[nfd+1]= mesg[tmp+t-k]+s2* mesg[tmp+t+k]
                                      + c2* mesg[tmp+2*t-k];
                         mesg[nfd+2]= mesg[tmp+k]+ c3* mesg[tmp+t+k]
                                      + s3* mesg[tmp+2*t-k];
                         mesg[nfd+3]= mesg[tmp+t-k]+s4* mesg[tmp+t+k]
                                      + c4* mesg[tmp+2*t-k];
                         mesg[tmp+k]= mesg[nfd+0];
                         mesg[tmp+t-k]= mesg[nfd+1];
                         mesg[tmp+t+k]= mesg[nfd+2];
                         mesg[tmp+2*t-k]= mesg[nfd+3];
                     }
                  else { /* my_node != 0 */
                     mesg[nfd+0]= mesg[tmp+k]+ c1* mesg[tmp+t+k]+
                                  s1* mesg[tmp+(2*t-1)-k];
                     mesg[nfd+1]= mesg[tmp+(t-1)-k]+ s2* mesg[tmp+t+k]+
                                  c2* mesg[tmp+(2*t-1)-k];
                     mesg[nfd+2]= mesg[tmp+k]+ c3* mesg[tmp+t+k]+
                                  s3* mesg[tmp+(2*t-1)-k];
                     mesg[nfd+3]= mesg[tmp+(t-1)-k]+ s4* mesg[tmp+t+k]+
                                  c4* mesg[tmp+(2*t-1)-k];
                     mesg[tmp+k]= mesg[nfd+0];
                     mesg[tmp+(t-1)-k]= mesg[nfd+1];
                     mesg[tmp+t+k]= mesg[nfd+2];
                     mesg[tmp+(2*t-1)-k]= mesg[nfd+3];
                  }
              } /* end of for i */
            } /* end of for k */
        } /* end of for j */

/* send back the results */

        sendw(cid, my_node, mesg, sizeof(mesg), HOST, PID);
    }

}
```