

# Efficient Overlapped FFT Algorithms for Hypercube-Connected Multicomputers\*

Cevdet Aykanat and Argun Derviř

Department of Computer Engineering, Bilkent University  
06533 Bilkent, Ankara, Turkey

## Abstract

*In this work, we propose parallel FFT algorithms, for medium-to-coarse grain hypercube-connected multicomputers, which are more elegant and efficient than the existing ones. The proposed algorithms achieve perfect load-balance for the efficient simplified-butterfly scheme, minimize the communication overhead by decreasing both the number and the volume of concurrent communications. Communication and computation cannot be overlapped easily due to the strong data dependencies in the FFT algorithm. In this paper, we propose a restructuring for the FFT algorithm which enables overlapping each communication with one fifth of the local computations involved in a stage. Two of the proposed parallel FFT algorithms achieve overlapping by exploiting this restructuring while using the efficient table-lookup scheme for complex coefficients. The proposed algorithms are implemented on an Intel's 32-node iPSC/2 hypercube multicomputer. High efficiency values are obtained even for small size FFT problems.*

*KEYWORDS : FFT, Parallel Computing, Multicomputer, Hypercube, Perfect Load Balance, Overlapping Communication and Computation.*

## 1 Introduction

The *Fast Fourier Transform (FFT)* algorithm formulated by Cooley and Tukey in 1965 [5], provides an efficient method for the analysis, design and the implementation of *Digital Signal Processing (DSP)* algorithms. The extent to which these algorithms can be performed in real time has in many cases limited by the rate at which the *FFT* algorithm can be executed. The high performance requirement for real time implementation of these algorithms led to the design of special purpose hardware. An extensive research has been conducted to implement efficient *FFT* algorithms on vector processors [6, 11], and general purpose parallel architectures with shared memory [1, 4, 10] and distributed memory [12, 13, 15].

The purpose of this paper is to investigate the efficient parallelization of one-dimensional *FFT* algorithm on medium-to-coarse grain, distributed-memory, message-passing architectures (multicomputers) implementing the hypercube interconnection topology. In order to achieve speedup on such architectures, the algorithm must be designed so that both computations and data can be distributed to the processors with local memories in such a way that computational tasks can be run in parallel, balancing the computational loads of the processors. Communication between processors to exchange data must also be considered as part of the algorithm. One

---

\*This work is partially supported by Intel Supercomputer Systems Division under Grant SSD100791-2 and Turkish Science and Research Council (TUBITAK) under Grant EEEAG-5

important factor in designing parallel algorithms is *granularity*. Granularity depends on both the application and the parallel machine. In a parallel machine with high communication latency, the algorithm should be structured so that large amounts of computations are done between successive communication steps. Another important factor is the ability of the parallel system to *overlap* communication and computation. In order to exploit this property of the parallel system, the algorithm must be structured so that processors have independent local computations to perform after initiating communication operations. In this work, all these points are considered in designing efficient parallel *FFT* algorithms for hypercube-connected multicomputers.

In the literature, there is a lot of theoretical work done on the parallelization of *FFT*. However, there are only a few parallel *FFT* algorithms proposed and implemented for hypercube-connected multicomputers. Walker [12] proposed various parallel *FFT* algorithms for MIMD architectures including the hypercube-connected multicomputers without any experimental results. First two parallel algorithms proposed in [12] are mainly for fine-grain architectures which involve fragmentary message passing. The second algorithm overlaps communication with computation. The third parallel algorithm eliminates fragmentary message passing and requires only  $2d$  concurrent exchange communication steps each with a volume of  $N/2P$  *FFT* points for the parallelization of an  $N$ -point *FFT* on a  $d$ -dimensional hypercube with  $P = 2^d$  processors. The fourth algorithm achieves parallelization by performing only two global communications. However, each global communication involves  $d$  concurrent exchange communication steps each with a volume of  $N/2P$  *FFT* points on a hypercube architecture.

Walton [13] proposed an algorithm similar to Walker's third algorithm. In his algorithm, each processor exchanges  $N/2P$  points with one of its neighbors at two points, once to obtain the data for the next stage of the transform, and again to send the partially transformed data "home". Nevertheless, the parallel complexity of his algorithm is equal to that of Walker's.

These algorithms [12, 13] achieve perfect load-balance only for the *basic-butterfly* (Fig. 1(a)) scheme which requires two complex multiplications per butterfly. However, the *simplified-butterfly* (Fig. 1(b)) scheme is much more efficient since it requires only one complex multiplication per butterfly. Figure 1 shows clearly the redundant multiplication which is discarded in the simplified-butterfly scheme. To our knowledge, only Zhu [15] has proposed a parallel *FFT* algorithm which achieves perfect load-balance for the simplified-butterfly scheme. His scheme also minimizes both the number and the volume of concurrent communications to  $d$  and  $d(N/2P)$ , respectively. His scheme performs the communications in the first  $d$  stages of the *FFT* computations and overlaps communications with the computation of the complex coefficients (complex exponentiations). However, in most of the *DSP* applications,  $N$ -point *FFT* is applied successively to  $N$ -point input data sets, for a fixed  $N$ . In such applications, the computation of complex coefficient values as they are needed, will be extremely inefficient. In general,  $N/2$  complex coefficient values can be computed once and stored in a table (during preprocessing), so that they are accessed by simple table-lookup operations during successive *FFT* computations.

In this work, we propose and present more efficient and elegant parallel *FFT* algorithms for medium-to-coarse grain hypercube-connected multicomputers. The proposed algorithms preserve the best features of the existing work in the literature as follows: (i) use simplified-butterfly scheme, (ii) achieve perfect load-balance,

(iii) exploit the efficient table-lookup scheme for handling complex coefficients, (iv) allow only nearest-neighbor communications, (v) minimize the number of concurrent communications to  $d$  by eliminating fragmentary message passing, (vi) minimize the volume of communication in each concurrent exchange step to  $N/2P$ , (vii) overlap communications with one half of the addition/subtraction operations during the  $d$  concurrent exchange communication steps. The proposed algorithms achieve these nice features by using a simple yet efficient mapping scheme and a restructuring for the *FFT* algorithm which enables overlapping communication with computation.

Different strategies exist for the computation of *FFT* [5, 7, 8, 14]. The *FFT* scheme chosen for parallelization is radix-2, Cooley-Tukey scheme using the decimation-in-time decomposition [5]. Section 2 presents and discusses the computational structure of this *FFT* scheme. Parallelization of the chosen *FFT* scheme is discussed in Section 3. The mapping scheme proposed to achieve perfect load-balance is presented in Section 3.1. Section 3.2 presents two different schemes proposed to overlap communication with computation. Section 4 presents the implementation results and the relative experimental performance evaluation of the proposed algorithms on a 32-node iPSC/2 hypercube multicomputer.

## 2 The Sequential FFT Algorithm

The computational flow graph of radix-2, Cooley-Tukey *FFT* scheme using the decimation-in-time decomposition is illustrated in Fig. 2 for a ( $N = 32$ )-point *FFT* computation. In this scheme, the input is in *bit-reversed* order and the output is in *normal* order. The numbers in the normal decimal order used in Fig. 2 to illustrate the output sequence also denote the indices of the corresponding *FFT* points used for *in-place* computations. As is seen in Fig. 2, each stage of the computation takes a set of  $N$  complex numbers and transforms them into another  $N$  complex numbers by performing  $N/2$  simplified-butterfly computations. This process is repeated  $n = \lg_2 N$  times, resulting in the computation of the desired discrete Fourier transform in normal order. The simplified-butterfly computations required at stage  $k$  of an  $N$ -point *FFT* is,

$$temp = W_N^r \times X_k(q); X_{k+1}(p) = X_k(p) + temp; X_{k+1}(q) = X_k(p) - temp \quad (1)$$

where  $q = p + 2^k$ ,  $W_N^r = e^{-j(\frac{2\pi}{N})r}$ , for  $k = 0, 1, \dots, n-1$ . That is, at stage  $k$ , simplified-butterfly computations are performed on partially transformed pairs separated by  $2^k$ . Hence, stage  $k$  consists of  $b_k = N/2^{k+1}$  blocks  $B_k^0, B_k^1, \dots, B_k^{b_k-1}$  where each block contains  $2^{k+1}$  consecutive *FFT* points. The consecutive  $2^{k+1}$  *FFT* points in each *FFT* block constitute the  $p, q$  points of all  $2^k$  *FFT* butterflies in that block. First and second halves of each block constitute the  $p$  and  $q$  points, respectively, of the *FFT* butterflies in that block. Note that, the block size is  $b_0 = 2$  at the first stage and doubles at each successive stage and reaches  $b_{n-1} = 2^n = N$  at the last stage. Two consecutive blocks  $B_{k-1}^{2^i}$  and  $B_{k-1}^{2^i+1}$  of stage  $k-1$  constitute the  $p$  and  $q$  halves, respectively, of the block  $B_k^i$  of the next stage  $k$ , for  $i = 0, 1, \dots, b_k-1$ .

The pseudo-code for an  $N$ -point *FFT* algorithm is given in Prog. 1. The *SEQFFTk* function shown in this program performs the in-place computation of  $N/2$  simplified-butterflies required at stage  $k$ . The *SEQFFTk* function is invoked  $n = \lg_2 N$  times to compute the complete  $N$ -point *FFT*. The outer *for-loop* in *SEQFFTk* function iterates  $N/2^{k+1}$  times to identify  $N/2^{k+1}$  consecutive *FFT* blocks. The inner *for-loop* iterates  $2^k$  times to identify and perform the computations associated with the  $2^k$  *FFT* butterflies in the respective block.

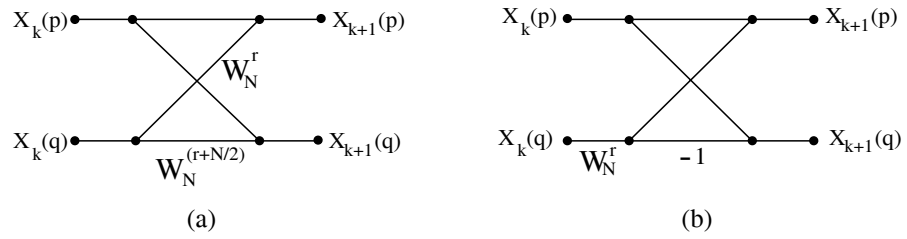


Fig. 1: Computational flow graphs for (a) basic-butterfly, (b) simplified-butterfly.

Fig. 2: Computational flow graph for a 32-point *FFT* and its *static* mapping on a hypercube with 4-processors.

```

/* Input in bit-reversed order in X[0 ... N-1] */
/* Output in normal order in X[0 ... N-1] */
n := lg2 N;
for k := 0 to n-1 do
  Call SEQFFTk (X, Wfac, N, k);
/* Performs N/2 butterfly computations over bit k */
SEQFFTk (X, Wfac, N, k)
  for i := 0 to (N/2k+1) - 1 do
    for j := 0 to 2k - 1 do
      p := i × 2k+1 + j;
      q := p + 2k;
      temp := Wfac × X(q);
      X(q) := X(p) - temp;
      X(p) := X(p) + temp;
    end SEQFFTk
end SEQFFTk

```

Prog. 1: Sequential N-pt FFT algorithm

As is seen in Eq. 1 and Prog. 1, one complex multiplication and two complex addition/subtraction operations are required in a simplified-butterfly computation. Since  $N/2$  butterfly computations are performed at each stage, the *FFT* algorithm requires  $(N/2) \lg_2 N$  and  $N \lg_2 N$  complex multiplication and addition/subtraction operations, respectively. Hence,  $N$ -point *FFT* requires  $2N \lg_2 N$  and  $3N \lg_2 N$  real multiplication and addition/subtraction operations, respectively. The complexity of the sequential *FFT* can be expressed as,

$$T_{P1} = \left[ 5N t_{calc} \right] \lg_2 N \quad (2)$$

where  $t_{calc}$  is the time taken by the floating point multiplication, addition and subtraction operations. The computations of the complex coefficients (*Wfac*) are not involved in the given complexity analysis since they are performed only once during the preprocessing.

### 3 Parallel FFT Algorithms

Computational structure of the *FFT* algorithm is very suitable for parallelization on multicomputers implementing the hypercube interconnection topology. The distribution of data and computations is straightforward for medium-to-coarse *grain* parallelism whenever the number of *FFT* points,  $N = 2^n$ , is greater than or equal to the number of processors,  $P = 2^d$ , in the hypercube. Successive processors in the decimal ordering are assigned the consecutive slices of the X-array with each slice containing equal number of,  $M = N/P$ , consecutive *FFT* points. In this scheme, each processor is responsible for carrying out the complete in-place computations required for the *FFT* points assigned to itself. The horizontal dashed-lines in Fig. 2 illustrate this straightforward mapping for a 32-point *FFT* data and computations on a 2-dimensional hypercube, with each processor assigned  $M = 8$  *FFT* points.

Computational interdependencies in a particular stage of an *FFT* algorithm are confined within butterflies belonging to that stage. Furthermore, *FFT* butterflies are confined within consecutive blocks of size  $2^{k+1}$  at stage  $k$ . Note that, block size increases as  $2, 4, 8, \dots, 2^{n-d}$  during the first  $n-d$  stages  $k = 0, 1, 2, \dots, n-d-1$ . Thus, *FFT* blocks and hence *FFT* butterflies are not fragmented during the first  $n-d$  stages since the straightforward mapping scheme assigns consecutive  $M = N/P = 2^{n-d}$  *FFT* points in blocks to consecutive processors

in decimal ordering. Hence, no interprocessor communication is required during the first  $n - d$  stages. However, in the last  $d$  stages,  $FFT$  blocks and hence  $FFT$  butterflies are fragmented between processors thus necessitating interprocessor communication.

In the  $(n - d)^{th}$  stage  $k = n - d - 1$ , each processor  $P_i$  computes only a single  $FFT$  block  $B_{n-d-1}^i$  of size  $M$  for  $i = 0, 1, \dots, P - 1$ . During the following  $d$  stages  $k = n - d, n - d + 1, \dots, n - 1$ , consecutive  $FFT$  blocks of size  $2^1 M, 2^2 M, \dots, 2^d M$ , are fragmented across consecutive processor blocks containing  $2^1, 2^2, \dots, 2^d$  consecutive processors in decimal ordering. Note that, each processor block constitutes a lower dimensional hypercube called here *subcube*. Hence, during the last  $d$  stages,  $k = n - d, n - d + 1, \dots, n - 1$ ,  $FFT$  blocks are fragmented across  $\ell = 1, 2, \dots, d$  dimensional subcubes, respectively, where  $\ell = k - (n - d) + 1$ . The fragmentation of an individual  $FFT$  block of size  $2^\ell M$  across an  $\ell$ -dimensional subcube is such that  $i^{th}$  processors (for  $i = 0, 1, \dots, 2^{\ell-1} - 1$ ) in the first and second halves ( $(\ell - 1)$ -dimensional subsubcubes) of that subcube hold  $M$   $p$ -points and  $M$   $q$ -points, respectively, of the  $2^{\ell-1} M$  butterflies involved in that block. Hence, in the last  $d$  stages,  $p$  and  $q$  points of  $FFT$  butterflies separated by  $2^{n-d}, 2^{n-d+1}, \dots, 2^{n-1}$ , are assigned to neighbor processors whose decimal indices differ by  $2^0, 2^1, \dots, 2^{d-1}$  respectively. Thus, one concurrent pairwise exchange communication step is required just before the computations involved at each stage during the last  $d$  stages in order to exchange  $p$  values with  $q$  values, and vice versa, of the fragmented butterflies. Note that, all processors should involve in these pairwise exchange communication operations at each stage during the last  $d$  stages. The volume of concurrent information exchange between each processor pair is  $N/P$   $FFT$  points where each  $FFT$  point consists of a complex floating-point word.

A pseudo-code for the node program of the parallel  $FFT$  algorithm is given in Prog. 2. The local variable *mynode* is assumed to contain the index of the respective node processor. A C-like notation is used in Prog. 2 and hereafter to represent the *for-loops* whenever needed. The first and second *for-loops* in Prog. 2, accumulate the summations over the first  $(n - d)$  and the last  $d$  bits by performing the computations corresponding to the fragmented and unfragmented butterflies, respectively. The first *for-loop* involves no interprocessor communication. The variable  $\ell$  inside the second *for-loop* denotes the dimension of the subcubes across which  $FFT$  blocks of size  $2^\ell M$  are fragmented. The processor indices whose  $(\ell - 1)^{th}$  bit is “0” and “1” identify the processors in the first and second halves (subsubcubes) of these subcubes, and these two types of processors hold  $M$   $p$ -points and  $M$   $q$ -points of the fragmented butterflies, respectively. Hence,  $M$ -point pairwise data exchanges are performed concurrently across the channel  $(\ell - 1)$  by each processor issuing a *send/receive* message pair at each iteration of the second *for-loop*. Here, channel  $(\ell - 1)$  refers to the set of  $P/2$  communication links connecting processor pairs whose indices differ only in the  $(\ell - 1)^{th}$  bit. In Prog. 2, *csend* and *crecv* denote *synchronous (blocking)* communication primitives which achieve the transmission and receive of the communication packets. As is seen in Prog. 2, this scheme introduces a local storage overhead of size  $M = N/P$  due to the local receive buffer *XRB*. Note that, the size of the local  $X$ -array is  $N/P$ .

The straightforward mapping scheme avoids fragmentary message passing and assigns interacting  $FFT$  sub-block pairs to the neighbor processor pairs of the hypercube. However, this scheme has two major drawbacks. First, partially transformed  $N/P$   $FFT$  points have to be exchanged between processor pairs at each stage of the last  $d$  stages. Second, perfect load-balance is disturbed during the last  $d$  stages. These two drawbacks can explicitly be seen when Prog. 2 is analyzed. At each iteration of the second *for-loop*, one half of the processors

```

/* Computations over the first (n - d) bits: no communication phase */
n := lg2 N; d := lg2 P; M := N/P; m := lg2 M;
for k := 0 to n - d - 1 do
  Call SEQFFTk (X, Wfac, M, k)
/* Computations over the next (last) d bits: d exchange communication phase */
for k := n - d to n - 1 do
  ℓ = k - (n - d) + 1; dnode := mynode ⊕ 2ℓ-1;
  if ((ℓ-1)th bit of mynode is 1) then do
    for q := 0 to M - 1 do
      X(q) := Wfac × X(q);
    csend from (X(q): q = 0, 1, ..., M - 1) to dnode;
    crecv into (XRB(p): p = 0, 1, ..., M - 1) from dnode;
    for (q := p := 0; q < M; q++, p++) do
      X(q) := XRB(p) - X(q);
  else /* (ℓ-1)th bit of mynode is 0 */
    csend from (X(p): p = 0, 1, ..., M - 1) to dnode;
    crecv into (XRB(q): q = 0, 1, ..., M - 1) from dnode;
    for (p := q := 0; p < M; p++, q++) do
      X(p) := X(p) + XRB(q);

```

Prog. 2: Parallel *FFT* algorithm.

hold only the updated values for the  $p$ -points, whereas the other half hold only the updated values for the  $q$ -points of the butterflies. Since only the updated values of the  $q$ -points of the butterflies have to be multiplied by the complex coefficients, those processors holding  $N/P$   $q$ -points perform  $N/P$  complex multiplications while the other processors wait idle for receiving the multiplication results from those processors. Hence, the parallel complexity of this scheme is,

$$T_{P2} = \left[ \frac{5N}{P} t_{calc} \right] \lg_2 \frac{N}{P} + \left[ \frac{8N}{P} t_{calc} \right] \lg_2 P + \left[ t_{su} + \frac{N}{P} t_{tr} \right] \lg_2 P \quad (3)$$

Here,  $t_{su}$  denotes the message *set-up* time overhead, and  $t_{tr}$  denotes the time taken for the transmission of a complex floating-point word ( $2 * 4$  bytes) between two neighbor processors.

### 3.1 Perfect Load Balance

The straightforward mapping scheme used in Prog. 2 already maintains perfect load-balance during the first  $(n - d)$  stages since all processors hold equal number(s) of unfragmented *FFT* blocks. Hence, this mapping is maintained during the first  $(n - d)$  stages of the parallel algorithm proposed in this subsection. As is indicated earlier, one half of the processors hold only updated values for the  $p$ -points and the other half hold only updated values for the  $q$ -points of the butterflies during the last  $d$ -stages. This *static* mapping scheme is altered at the beginning of each stage of the last  $d$  stages. At the very beginning of each stage, each processor holding updated values for  $N/P$   $q$ -points exchanges one half of its  $q$ -points with one half of the  $p$ -points of its neighbor processor which holds all the  $p$ -points of its butterflies at that stage and vice versa. This exchange operation is not only the exchange of data values to be used at that stage. In fact, processors effectively exchange the responsibility of the further *FFT* computations associated with those exchanged *FFT* points. This scheme maintains a single unfragmented *FFT* subblock of size  $M = N/P$  at each processor during the last  $d$  stages.

Fig. 3: Dynamic mapping of 32-point *FFT* data and computations on a hypercube with 4-processors.

This *dynamic* mapping scheme is illustrated in Fig. 3 for a 32-point *FFT* on a 4-processor hypercube. The pseudo-code for the node-program of the proposed parallel *FFT* algorithm is given in Prog. 3. We only present the last  $d$  concurrent exchange communication phase of Prog. 3 since the initial mapping scheme and the first *for-loop* is exactly similar to Prog. 2. As is seen in Fig. 3 and Prog. 3, each processor exchanges either the first half or the second half of its local  $X$ -array, in-place, by simply checking the  $(\ell-1)^{th}$  bit of its processor index, where  $(\ell-1)$  denotes the channel across which the exchange operation is to be performed at that stage. Due to the *dynamic* mapping scheme, each processor performs simplified-butterfly computations on local  $p$  and  $q$  pairs separated by  $2^{m-1} = N/2P$  after the exchange operations at each stage of the last  $d$  stages. Hence, in this scheme, each processor holds equal number of  $p$  and  $q$  points which form unfragmented *FFT* subblocks of size  $M = N/P$  points (consisting of  $M/2$   $p$ -points and  $M/2$   $q$ -points) after the exchange operation. Each processor performs equal number of  $(N/2P)$  complex multiplications thus achieving a perfect load-balance.

```

/* Computations over the last  $d$  bits:  $d$  exchange communication phase */
for  $k := n - d$  to  $n - 1$  do
   $\ell := k - (n - d) + 1$ ;  $dnode := mynode \oplus 2^{\ell-1}$ ;
  if  $((\ell - 1)^{th}$  bit of  $mynode$  is 1) then do
    csend from  $(X(q): q = 0, 1, \dots, M/2 - 1)$  to  $dnode$ ;
    crecv into  $(X(p): p = 0, 1, \dots, M/2 - 1)$  from  $dnode$ ;
  else /*  $(\ell - 1)^{th}$  bit of  $mynode$  is 0 */
    csend from  $(X(p): p = M/2, M/2 + 1, \dots, M - 1)$  to  $dnode$ ;
    crecv into  $(X(q): q = M/2, M/2 + 1, \dots, M - 1)$  from  $dnode$ ;
  for  $(p := 0, q := M/2; p < M/2; p++, q++)$  do
     $temp := Wfac \times X(q)$ ;
     $X(q) := X(p) - temp$ ;
     $X(p) := X(p) + temp$ ;

```

Prog. 3: Parallel *FFT* algorithm with perfect load-balance ( $d$  concurrent exchange communication phase).

This scheme requires no extra storage for *send/receive* buffers since *send/receive* operations are performed in-place from/to the local  $X$ -array. Furthermore, the volume of concurrent communication at each exchange communication step is reduced by a factor of two (from  $N/P$  to  $N/2P$  complex floating-point words). Hence, the parallel complexity of the proposed scheme is,

$$T_{P3} = \left[ \frac{5N}{P} t_{calc} \right] \lg_2 N + \left[ (t_{su} + \alpha \frac{N}{2P} t_{tr}) + \frac{N}{2P} t_{copy} \right] \lg_2 P \quad (4)$$

The *set-up* times ( $t_{su}$ ) of the mutual *send* operations are overlapped in an exchange operation. The overlap of mutual data *transmissions* ( $(N/2P)t_{tr}$ ) between a pair of processors is feasible only when two physical links are present between neighbor processors as in *iPSC/2*. However, the internal hardware architecture of an individual *iPSC/2* processor is such that internal bus conflicts occur due to the outgoing and incoming long messages during an *exchange* operation. Hence, a complete overlap cannot be achieved in *iPSC/2* during the mutual data *transmission* phase of the *exchange* operation. The performance of the exchange operation can be modeled as  $(t_{su} + \alpha(N/2P)t_{tr})$  on *iPSC/2*, where  $\alpha$  is measured to vary between 1.3 and 1.6 with varying incoming/outgoing message size [2]. Note that,  $\alpha = 1$  corresponds to complete overlap.

In Prog. 3, each processor issues a *synchronous receive* just after the *synchronous send* operation. Due to the perfect load-balance, communicating processor pairs perform the synchronous send operations concurrently. A synchronous send operation returns the control back to node program only after the outgoing message leaves the indicated send area in the local  $X$ -array. Whenever an incoming message begins to arrive to a destination processor, it does not find a pending receive, hence it is copied to a temporary system buffer by the node operating system NX. Later, whenever the receive operation is issued by the node program, that message is copied from the temporary system buffer to the indicated half of the local  $X$ -array. Hence, late issue of the receive operation introduces a block copy overhead  $(N/2P)t_{copy}$  where  $t_{copy}$  represents the time taken to copy a single complex floating-point word from the system buffer to the indicated half of the  $X$ -array. Note that, such a receive overhead is not included in the parallel time complexity model given in Eq. 3 for Prog. 2. In Prog. 2, due to the lack of load balance, communicating processor pairs do not initiate send operations concurrently. The model in Eq. 3 is given for the bottleneck processors which stay idle, waiting for the multiplication results from their neighbor processors. These processors always issue early synchronous receives thus avoiding the receive overhead.

As is seen in Fig. 3, the output results are slightly scrambled (in  $N/2P$  blocks) in this scheme due to the proposed *dynamic* mapping scheme. In most *DSP* applications a sequence of *DSP* blocks are applied consecutively on a set of input data. A proper output/input interface between successive *DSP* blocks can always be maintained, if the output data order of a particular *DSP* block is disturbed for the sake of efficiency. For example, in most *DSP* applications, a set of time domain sample data is transformed into the frequency domain by applying an *FFT DSP* block. Then, a sequence of frequency domain operations are performed. The results are transformed back to the time domain by applying an inverse *FFT DSP* block. The sequence of frequency domain computations including the inverse *FFT* block can easily be modified to operate on the disturbed output order of the previous *FFT* block. Hence, the order of input and output data of individual *DSP* blocks does not bring any inefficiency to the overall application.

### 3.2 Overlapping Communication with Computation

There are strong data dependencies in the *FFT* algorithm. As is seen in Prog. 3, the update of each butterfly necessitates the communication of either its  $p$  or  $q$  point, during the last  $d$ -stages of the algorithm. Hence, as is also indicated in [12], the *FFT* algorithm differs from local and spatially decomposable problems such as Finite Difference and Finite Element problems. In such problems, communications associated with the boundary points can easily be overlapped with the update of interior data points [2, 12]. Thus, communication and computation in the *FFT* algorithm cannot be overlapped easily. In this section, we propose a restructuring for the *FFT* algorithm which enables overlapping by pipelining the communication of the following stage by the local computations of the current stage, as much as possible. Based on this restructuring, we present two schemes which overlap each communication with one fifth of the local computations involved in a stage.

#### 3.2.1 Scheme 1: Asynchronous Send and Synchronous Receive

The pseudo-code for the node program of the parallel *FFT* algorithm which overlaps communication and computation is given in Prog. 4. The initial *static* mapping for the first  $(n - d)$  stages and the *dynamic* mapping for the last  $d$ -stages are similar to the scheme given in Prog. 3. However, as is seen in Prog. 4, the first *for-loop* iterates only  $(n - d - 1)$  times which is one less than the iteration count in Prog. 3. Then, at each iteration (computation stage) of the second *for-loop*, each processor initiates the *send* portion of the exchange operation required at the following stage. Each processor classifies its computational tasks at each stage into two categories: those updates to be *sent* to its neighbor processor in the following stage and other updates to be kept local for the following stage. Then, each processor first performs the computations associated with those points required by the neighbor processor in the next stage. Hence, each processor first performs  $N/2P$  complex multiplications associated with its local  $N/2P$   $q$ -points. Then, each processor updates either the values of its local  $p$ -points or  $q$ -points simply by checking the  $\ell^{\text{th}}$  bit of its processor index. Here,  $\ell$  denotes the channel across which the exchange operation is to be performed in the next stage. Upon completion of these  $N/2P$  updates, each processor issues an *asynchronous (non-blocking) send* (*isend* in Prog. 4) to initiate the transmission of the updated  $N/2P$  *FFT*-point values to the neighbor processor. After initiating the *send* operation, each processor completes the local computations associated with that stage by updating the other half of its local *FFT* points that will be kept local for the following stage. Upon completion of the second type updates each processor issues a *synchronous receive* to complete the already initiated exchange.

```

/* Computations over the first  $(n - d - 1)$  bits: no communication phase */
 $n := \lg_2 N$ ;  $d := \lg_2 P$ ;  $M := N/P$ ;  $m := \lg_2 M$ ;
for  $k := 0$  to  $n - d - 2$  do
  Call SEQFFT $k(X, Wfac, M, k)$ 
/* Computations over the next  $d$  bits: from  $n - d - 1$  to  $n - 2$  */
/*  $d$  concurrent exchange communication phase */
for  $k := n - d - 1$  to  $n - 2$  do
   $\ell := k - (n - d) + 1$ ;  $dnode := mynode \oplus 2^\ell$ ;
  if ( $\ell^{\text{th}}$  bit of mynode is 1) then do
    for ( $p := 0, q := M/2; p < M/2; p++, q++$ ) do
       $X(q) := Wfac \times X(p)$ ;
       $XSB(p) := X(p) + X(q)$ ;
    isend from ( $XSB(p): p = 0, 1, \dots, M/2 - 1$ ) to dnode;
    for ( $q := M/2, p := 0; q < M; q++, p++$ ) do
       $X(q) := X(p) - X(q)$ ;
    crecv into ( $X(p): p = 0, 1, \dots, M/2 - 1$ ) from dnode;
    msgwait on isend;
  else /*  $\ell^{\text{th}}$  bit of mynode is 0 */
    for ( $q := M/2, p := 0; q < M; q++, p++$ ) do
       $X(q) := Wfac \times X(q)$ ;
       $XSB(q - M/2) := X(p) - X(q)$ ;
    isend from ( $XSB(q): q = 0, 1, \dots, M/2 - 1$ ) to dnode;
    for ( $p := 0, q := M/2; p < M/2; p++, q++$ ) do
       $X(p) := X(p) + X(q)$ ;
    crecv into ( $X(q): q = M/2, M/2 + 1, \dots, M - 1$ ) from dnode;
    msgwait on isend;
/* Computations over the last bit: no communication */
Call SEQFFT $(X, Wfac, M, m - 1)$ ;

```

Prog. 4: Overlapped parallel *FFT* algorithm (Scheme 1) with perfect load-balance.

In Prog. 4, the first inner *for-loop* of the second outer *for-loop* computes the updates to be sent to the neighbor processor while storing the intermediate multiplication results into the second half of the local  $X$ -array to be reused in the second inner *for-loop*. Each processor stores these local update results into a send buffer ( $XSB$  array) of size  $N/2P$  since it does not need these results in further *FFT* computations. Note that, updated local  $p$  points sent from the local buffer  $XSB$  will be the  $q$  points of the following stage, and vice versa. After initiating the asynchronous send operation, the second inner *for-loop* computes the local updates to be kept local for the following iteration and stores them either into the first or the second half of the local  $X$ -array depending on the type of updates,  $p$  or  $q$  point updates, respectively. Hence,  $N/2P$  complex addition/subtraction operations in the second inner *for-loop* are overlapped with communication. Thus, communication is overlapped with one fifth of the computations involved in a stage. The receive portion of the exchange operations can be done in-place into the local  $X$ -array since *synchronous receive* is issued after the completion of the overall computations associated with that stage. Hence, the local storage overhead is only  $N/2P$  due to the local send buffer  $XSB$ . The only computational overhead is the loop overhead since two *for-loops* are required instead of one. The number of floating-point computations is exactly equal to that of Prog. 3. Thus, the parallel complexity of the proposed algorithm is,

$$T_{P4} = \left[ \frac{5N}{P} t_{calc} \right] \lg_2 \frac{N}{P} + \left[ \frac{4N}{P} t_{calc} \right] \lg_2 P + \left[ Max \left\{ \frac{N}{P} t_{calc}, (t_{su} + \alpha \frac{N}{2P} t_{tr}) \right\} + \frac{N}{2P} t_{copy} \right] \lg_2 P \quad (5)$$

Hence, complete overlap of communication can be achieved for sufficiently large  $N/P$ , where,

$$(N/P) t_{calc} \geq t_{su} + (N/2P) t_{tr} \quad (6)$$

```

/* Computations over the  $d$  bits from  $n-d-1$  to  $n-2$  */
/*  $d$  concurrent exchange communication phase */
pstart := 0;  qstart := M/2;
for  $k := n-d-1$  to  $n-2$  do
   $\ell := k - (n-d) + 1$ ;   $dnode := mynode \oplus 2^\ell$ ;
  if ( $\ell$  is even) then  $rpctr = M$  else  $rpctr = 3M/2$ ;
  irecv into  $(X(i): i = rpctr, rpctr+1, \dots, rpctr+M/2-1)$  from  $dnode$ 
   $pptr := pstart$ ;   $qptr := qstart$ ;
  if ( $\ell^{th}$  bit of  $mynode$  is 1) then
    for ( $p := 0, q := M/2; p < M/2; p++, q++, pptr++, qptr++$ ) do
       $X(q) := Wfac \times X(qptr)$ ;
       $XSB(p) := X(pptr) + X(q)$ ;
    isend from  $(XSB(p): p = 0, 1, \dots, M/2-1)$  to  $dnode$ ;
    for ( $q := M/2, pptr := pstart; q < M; q++, pptr++$ ) do
       $X(q) := X(pptr) - X(q)$ ;
    msgwait on isend;  msgwait on irecv;
     $pstart := rpctr$ ;   $qstart := M/2$ ;
  else /*  $\ell^{th}$  bit of  $mynode$  is 0 */
    for ( $q := M/2; q < M; q++, pptr++, qptr++$ ) do
       $X(q) := Wfac \times X(qptr)$ ;
       $XSB(q-M/2) := X(pptr) - X(q)$ ;
    isend from  $(XSB(q): q = 0, 1, \dots, M/2-1)$  to  $dnode$ ;
    for ( $p := 0, q := M/2, pptr := pstart; p < M/2; p++, q++, pptr++$ ) do
       $X(p) := X(pptr) + X(q)$ ;
    msgwait on isend;  msgwait on irecv;
     $pstart := 0$ ;   $qstart := rpctr$ ;
  /* Computation over the last bit: no communication */
   $pptr := pstart$ ;   $qptr := qstart$ ;
  for ( $p := 0, q := M/2; p < M/2; p++, q++, pptr++, qptr++$ ) do
     $temp := Wfac \times X(qptr)$ ;
     $X(q) := X(pptr) - temp$ ;
     $X(p) := X(pptr) + temp$ ;

```

Prog. 5: Overlapped parallel *FFT* algorithm (Scheme 2) with perfect load-balance (computations over the last  $d+1$  bits).

### 3.2.2 Scheme 2: Asynchronous Send and Asynchronous Receive

As is seen in Prog. 4, only the *send* portions of the exchange communications are overlapped with one fifths of the local computations. Note that, in Prog. 4, each processor issues *synchronous receive* after initiating *asynchronous send* operation and performing  $N/2P$  complex addition/subtraction operations. Due to the perfect load-balance, communicating processor pairs initiate the asynchronous send operations concurrently. Hence, there are no pending receives in the destination processors for the incoming messages. The last term of the communication component in Eq. 5 accounts for the receive overhead due to the late issue of the synchronous receives. However, if there is a pending receive for an incoming message, then it is directly copied into the indicated receive buffer. Hence, it is also possible to overlap the receive portion by issuing an early *asynchronous receive*. In this section, we present a second scheme which overlaps both *send* and *receive* portions of the exchange communications with local computations. Prog. 5 illustrates the pseudo-code for the node program of this scheme. We only present the pseudo-code for the computations over the last  $d+1$  bits since the initial mapping and the first-loop is same as that of Prog. 4.

In the  $d$  concurrent exchange phase of Prog. 5, each processor issues an asynchronous receive at the beginning of each iteration in order to provide a pending receive for the incoming message. In this scheme, a local

Table 1: Relative performance features of the proposed parallel *FFT* algorithms Progs. 2, 3, 4 and 5.

	Prog. 2	Prog. 3	Prog. 4	Prog. 5
Local Storage Overhead	$N/P$	<i>none</i>	$N/2P$	$3N/2P$
Concurrent Comm. Volume	$d(N/P)$	$d(N/2P)$	$d(N/2P)$	$d(N/2P)$
Perfect Load Balance	<i>no</i>	<i>yes</i>	<i>yes</i>	<i>yes</i>
Comm./Comp. Overlap	<i>no</i>	<i>no</i>	<i>send</i>	<i>send/receive</i>

$X$ -array of size  $2N/P$  is used compared to  $N/P$  in Progs. 2, 3, and 4. The third and fourth quarters (each of size  $N/2P$ ) of the local  $X$ -array are used as two consecutive receive buffers. As is seen in Prog. 5, incoming messages are received either into the first or the second buffer in a cyclic manner according to  $\ell$  being *even* or *odd*, respectively. This switching receive buffer scheme is chosen to avoid the contamination of the message received in the previous iteration by the incoming message expected in the current iteration. In Prog. 5, the local variables  $pstart$  and  $qstart$  are used to inform the following iteration about the two particular quarters of the local  $X$ -array which will contain the updated  $p$  and  $q$  points, respectively. The local variables  $pptr$  and  $qptr$  are used to index the local  $X$ -array for accessing updated  $p$  and  $q$  points. In the first inner *for-loop* of the first outer *for-loop*, the results of the local updates to be sent to the neighbor processor are stored into the send buffer  $XSB$ , whereas the multiplication results are temporarily stored into the second quarter of the local  $X$ -array to be reused in the second inner *for-loop*. In the second inner *for-loop*, results of the local updates to be kept local for the following iteration are either stored into the first or the second quarter of the local  $X$ -array depending on the type of updates,  $p$  or  $q$  point updates, respectively.

The size of  $XSB$ -array is  $N/2P$  as in the case of Prog. 4. Hence this scheme introduces an extra local storage overhead of size  $N/P$  due to the two receive buffers in the second half of the  $X$ -array, compared to Prog. 4. On the other hand, the computational overhead introduced compared to Prog. 3 is as same as in Prog. 4, only an extra *for-loop*. The number of floating point computations associated in each stage is exactly equal to those of Progs. 3 and 4. Thus, the parallel complexity of the proposed algorithm is,

$$T_{P5} = \left[ \frac{5N}{P} t_{calc} \right] \lg_2 \frac{N}{P} + \left[ \frac{4N}{P} t_{calc} \right] \lg_2 P + \left[ \text{Max} \left\{ \frac{N}{P} t_{calc}, \left( t_{su} + \alpha \frac{N}{2P} t_{tr} \right) \right\} \right] \lg_2 P \quad (7)$$

As is seen in Eq. 7, receive overhead is avoided by the early issue of the *asynchronous receive*.

Table 1 illustrates the relative performance features of the proposed parallel *FFT* algorithms. The local storage overhead in Table 1 denotes the extra local storage requirement for send/receive operations in addition to the local  $X$ -array of size  $N/P$ . The parallel performance of these algorithms are expected to increase with increasing program index as is also verified experimentally in the following section.

## 4 Experimental Results

All programs presented in this paper have been coded in C language and run on a 32-node iPSC/2 hypercube multicomputer for various data sizes,  $256 \leq N = 2^n \leq 64K$ . Figure 4, illustrates the variation of percent performance improvement of Prog. 3 compared to Prog. 2 during the  $d$  exchange communication phase. As is seen in Fig. 4, Prog. 3 outperforms Prog. 2 as expected since it achieves perfect load-balance and reduces the volume of communication by a factor of two compared to Prog. 2. As is also seen in Fig. 4, percent performance improvement of Prog. 3 compared to Prog. 2 increases with increasing data size.

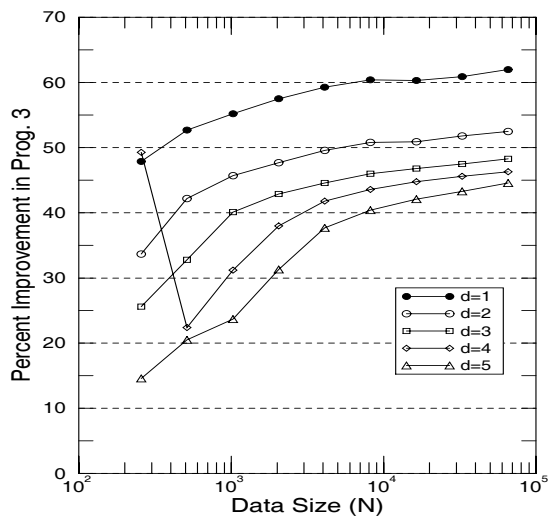


Fig. 4: Percent improvement curves for Prog. 3, over Prog. 2, during  $d$  exchange communication phase.

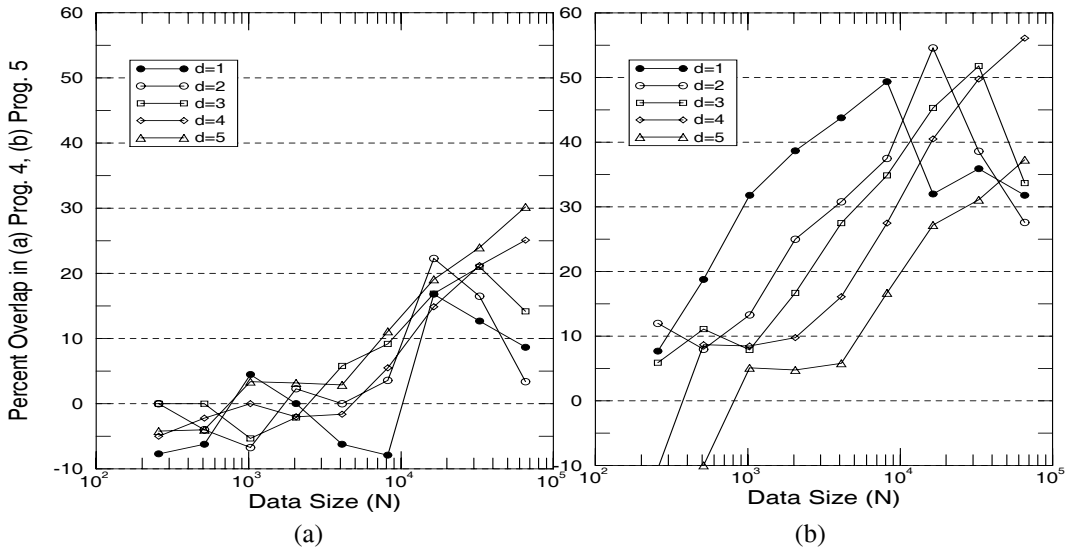


Fig. 5: Percent overlap curves for (a) Prog. 4, (b) Prog. 5, compared to Prog. 3.

Figures 5(a) and 5(b) display the variation of percent overlap achieved in Progs. 4 and 5, respectively. Total communication times, and overlapped communication times in Progs. 4 and 5 are computed by running Prog. 3 without invoking *csend* and *crecv* communication routines and subtracting these timings from the original execution timings of Progs. 3, 4 and 5, respectively. Percent overlap is then computed by dividing overlapped communication times by total communication times. As is seen in Figs. 5(a) and 5(b), percent overlap increases with increasing data size as expected. Note that, *for-loop* overhead is also included in these timings. For small data sizes, the amount of computation is not large enough to achieve complete overlap of the communication (see Eq. 6). Hence, negative percent overlap values are obtained for small data sizes. As is also seen in Figs. 5(a) and 5(b), percent overlap values begin to decrease slightly after reaching a maximum value at large data sizes ( $16K \leq N/2P \leq 32K$ ). This decrease is closely related to the variation of  $\alpha$  for those large data sizes. Comparison of Figs. 5(a) and 5(b) shows that Prog. 5 achieves higher percent overlap values than Prog. 4 for  $N/P \geq 32$ . Maximum overlap values of 23% and 54% are obtained in Progs. 4 and 5, respectively, in spite of the *for-loop* overheads. Also, as is indicated in [3], complete overlap cannot be achieved due to the internal architecture of an individual iPSC/2 processor.

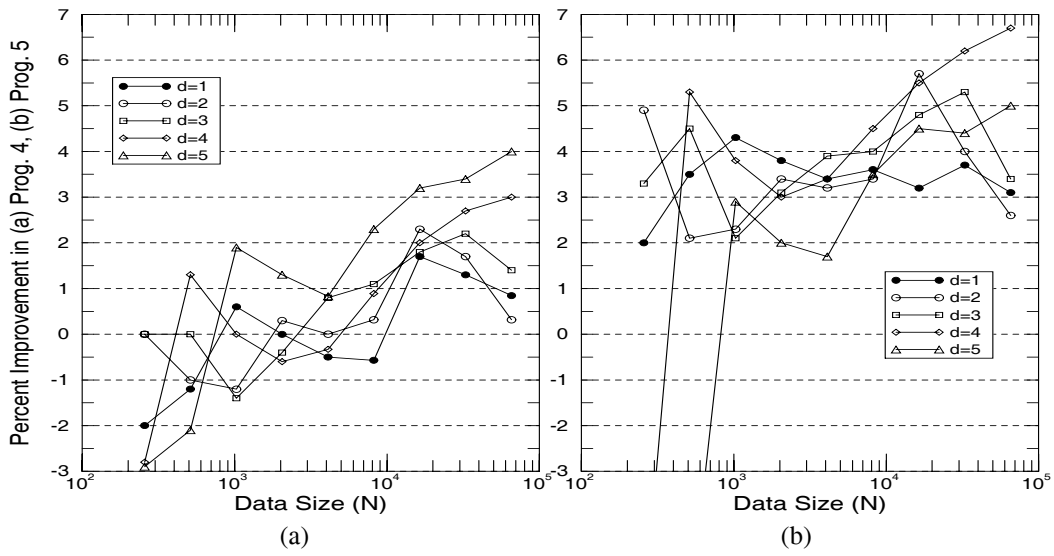


Fig. 6: Percent improvement curves for (a) Prog. 4, (b) Prog. 5, over Prog. 3, during  $d$  exchange communication phase.

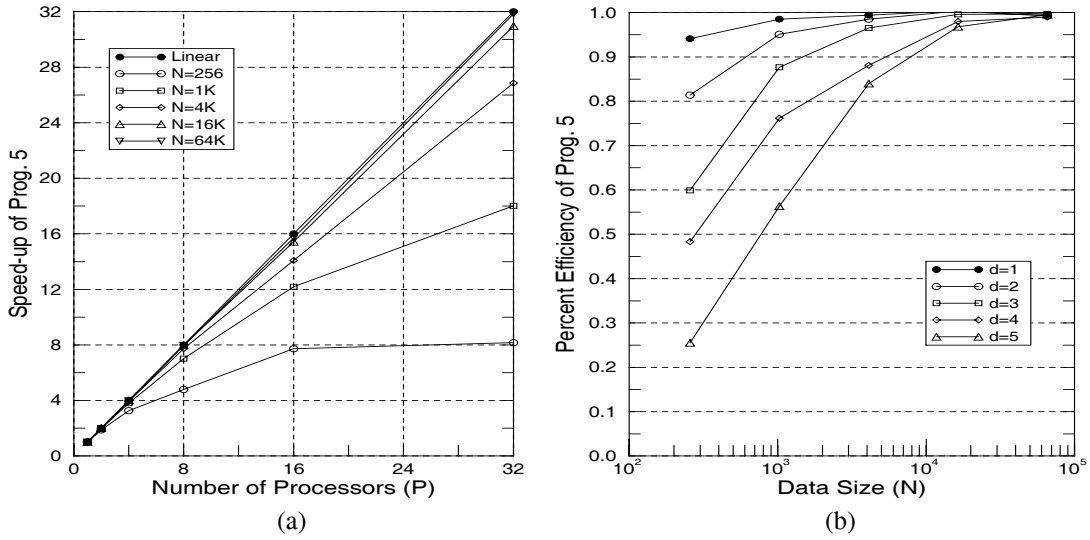


Fig. 7: (a) Speedup, (b) Efficiency curves for Prog. 5.

Figures 6(a) and 6(b) display the variation of percent performance improvement of Progs. 4 and 5 compared to Prog. 3, respectively, during the  $d$  exchange communication phase. Comparison of Figs. 6(a) and 6(b) shows that Prog. 5 performs better than Prog. 4 for  $N/P \geq 32$ . As is seen in these figures, the variation of percent performance improvement is very similar to the variation of overlap curves in Figs. 5(a) and 5(b) as expected. Programs 4 and 5 give better performance results than Prog. 3 for  $N/P \geq 8K$  and  $N/P \geq 32$ , respectively. These overlapped programs do not perform better than Prog. 3 for small granularities due to the *for-loop* overhead and the insufficient amounts of local computations for overlapping communications (see Eq. 6). The relative performances of the overlapped algorithms are expected to be much higher on larger dimensional hypercubes and architectures which enable complete overlap.

Figures 7(a) and 7(b) show speed-up and efficiency curves for Prog. 5. As is seen in Fig. 7(a), almost linear speed-up is achieved for  $N \geq 4K$  and  $P \leq 32$ . As is seen, in Fig. 7(b), efficiency remains over 94% when  $N/P \geq 128$  FFT points are mapped to an individual processor of the hypercube.

## 5 Conclusion

We have proposed more elegant and efficient, three parallel *FFT* algorithms for medium-to-coarse grain hypercube-connected multicomputers. All the proposed parallel *FFT* algorithms achieve perfect load-balance by exploiting the efficient simplified-butterfly and the table-lookup schemes. The first proposed parallel *FFT* algorithm (Prog. 3) achieves perfect load-balance, reduces the volume of concurrent communication by a factor of two, and doesn't introduce any memory overhead. The second proposed parallel *FFT* algorithm (Prog. 4) overlaps only the send portions of the exchange communications with one fifths of the local computations. This method introduces an extra send buffer of length  $N/2P$ . The third proposed parallel *FFT* algorithm (Prog. 5) overlaps both send and receive portions of the exchange communications with one fifths of the local computations. This method further introduces an extra receive buffer of length  $N/P$ .

## References

- [1] A. Averbuch, E. Babber, B. Gordinssky and Y. Meclan, "A parallel FFT on an MIMD machine", in *Proceedings of the ICPP*, Vol.3, pp.63-70, 1989.
- [2] C. Aykanat, F. Ozguner, F. Ercal, and P. Sadayappan, "Iterative Algorithms for Solution of Large Sparse Systems of Linear Equations on Hypercubes," in *IEEE Transactions on Computers*, Vol 37, no. 12, pp. 1554-1568, December 1988.
- [3] L. Bomans, and D. Roose, "Benchmarking the iPSC/2 hypercube multiprocessor," in *Concurrency : Practice and Experience*, Vol. 1(1), pp. 3-18, September 1989.
- [4] W.L. Briggs, I.B. Hart, R.A. Sweet and A. O'Gallagher, "Multiprocessor FFT methods", in *SIAM J. Sci. Stat. Comput.* Vol.1, No.1, pp.27-42, January 1987.
- [5] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. Comput.*, Vol. 19, pp. 297-301, April 1965.
- [6] B. Fornberg, "A vector implementation of the Fast Fourier Transform", in *Math. Comput.*, 36, pp.189-191, 1981.
- [7] G. Goertzel, "An Algorithm for the evaluation of Finite Trigonometric Series", *Amer. Math. Monthly*, 65, pp.34-35, 1968.
- [8] I.J. Good "The relationship between two fast fourier transform", *IEEE Transactions on Computers*, C-20, pp.310-317, 1971.
- [9] A.V. Oppenheim "Digital Signal Processing," (*Prentice Hall International*, 1985).
- [10] P.N. Swarztrauber, "Multiprocessor FFTs", in *Parallel Computing* Vol.5, pp. 197-210, 1987.
- [11] P.N. Swarztrauber, "FFT algorithms for vector computers", in *Parallel Computing* 1, pp. 45-63, 1984.
- [12] D. W. Walker, "Portable Programming within a Message-Passing Model: the FFT as an Example," in *Third Conference on Hypercube Concurrent Computers and Applications*, Pasadena, CA, pp. 1438-1450, January 1988.
- [13] S. R. Walton, "Performance of the One-Dimensional Fast Fourier Transform on the Hypercube," in *Second Conference on Hypercube Multiprocessors*, Knoxville, pp. 530-535, Sept. 1986.
- [14] S. Winograd "On Computing the Discrete Fourier Transform", *Math. Comp.*, 32, pp.175-199, 1978.
- [15] J.P. Zhu, "An Efficient FFT algorithm on Multiprocessors with Distributed Memory," in *The Fifth Distributed Memory Computing Conference*, Vol. 1(2), pp 358-363, January 1990.