IMPLEMENTING FAST FOURIER TRANSFORMS ON DISTRIBUTED-MEMORY MULTIPROCESSORS USING DATA REDISTRIBUTIONS[1]

S. K. S. GUPTA, C.-H. HUANG, and P. SADAYAPPAN
Department of Computer and Information Science, The Ohio State University
Columbus, OH 43210-1277, U.S.A.

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

Implementations of various fast Fourier transform (FFT) algorithms are presented for distributed-memory multiprocessors. These algorithms use data redistribution primitives to achieve data communication. The goal is to optimize communication cost by using a minimum number of redistribution steps. Performance results on an Intel iPSC/860 system are presented.

Keywords: fast Fourier transform, parallel algorithm, distributed-memory multiprocessor, High Performance Fortran, data distribution, communication optimization.

1. Introduction

This paper presents distributed-memory parallel programs for the fast Fourier transform (FFT) [1, 2] which use data redistribution primitives for data communication. In a distributed-memory multiprocessor like the Intel iPSC/860 and the Thinking Machines CM5, shared data is distributed across the local memories of interconnected processors. Various data distribution schemes are used to distribute data among processors. One of the most common distributions used are the block, cyclic, and block-cyclic distribution. These distributions are also proposed in High Performance Fortran (HPF) [3] for massively parallel computing. On a distributed-memory multiprocessor, communication is needed when a processor requires data from another processor's local memory. In most distributed-memory multiprocessors the cost of communicating a data value is considerably larger than the cost of a primitive arithmetic computation on the data element. It is therefore important to reduce the number of communication steps if high performance is to be achieved. Choosing the right initial distributions for shared data arrays is important in reducing communication overhead. One way to further reduce communication overhead is to use efficient aggregate communication primitives. Redistribution is one such communication primitive. As a redistribution may involve a many-to-many communication, the cost of a single data redistribution can be expected to be considerably higher than a one-to-one communication primitive. However, by redistributing the shared data at some intermediate step in the computation, it is sometimes possible to eliminate a large number of subsequent communication steps by localizing the remaining compu-

[1]This work was supported in part by ARPA, order number 7899, and monitored by NIST under grant number 60NANB1D1150. Submitted to Parallel Processing Letters.

tation. This may lead to a substantial reduction in the overall communication overhead. Furthermore, using a redistribution primitive for communication will lead to a portable implementation by abstracting away the details of the target machine such as the network topology. In this paper, we demonstrate that FFT programs with efficient communication can be developed by using only redistribution primitives. For simplicity of presentation, only radix-2 decimation-in-time FFT algorithms are developed. Performance results on an Intel iPSC/860 system show that real benefits can be achieved by such a scheme.

The paper is organized as follows. Section 2 describes the semantics of block-cyclic distributions. In Section 3, we present a distribute-memory node program for the Cooley-Tukey FFT. Section 4 and Section 5 present implementations of the Pease FFT and the Stockham FFT. Performance results for FFT programs on an Intel iPSC/860 system are given in Section 6. Conclusions are included in Section 7.

2. Semantics of Data Distribution

Various data distributions of an array correspond to mapping certain bits of the address (binary representation of the array index) of an element to represent the processor address [4]. One such class of data distribution is block-cyclic distributions. A block-cyclic distribution partitions an array into equal sized blocks of consecutive elements and then maps them to the processors in a cyclic manner. The elements mapped to a processor are stored in the increasing order of their indices in its local memory. We will use the following convention to express the block-cyclic distribution of a linear array $A(0 : N-1)$ on P processors:

> $A(cyclic(b))$: the block size is b and element i, $0 \leq i < N$, is on processor $(i \ div \ b) \ mod \ P$ at local index $(i \ mod \ b) \ + \ (i \ div \ (P * b)) * b$.

The local array $A'(0 : \lceil N/P \rceil - 1)$ will contain elements of array A mapped to a processor. $A(block)$ and $A(cyclic)$ will be used to express block and cyclic distributions of array A. These distributions are equivalent to $cyclic(\lceil N/P \rceil)$ and $cyclic(1)$, respectively.

Suppose an array of size 2^n is block-cyclically distributed on 2^p processors. Then the p bits required for the processor address is chosen from the n bits used for the array indexing. If $b_{n-1}{}^a$ is the address of an array element, then $cyclic(2^b)$ corresponds to choosing $b_{b+p-b+1}$ as its processor address. As the array elements are stored in increasing order of their indices in each processor's local memory, the local address of element b_{n-1} on processor $b_{b+p-b+1}$ is $b_{n-b+p+1}b_{b-1}$. We will identify a bit b_i used for the processor indexing in a global address as $\hat{b}_i$. For example, the global address for the index b_{n-1} would be $\hat{b}_{n-n-p+1}b_{n-p-1}$ for a linear array of size 2^n distributed as $block$. It would be $b_{n-p+1}\hat{b}_{p-1}$ when it is distributed as $cyclic$. The local address and the processor address can be extracted from a global address by functions $local$ and $proc$. For example, $local(\hat{b}_{n-n-p+1}b_{n-p-1}) = b_{n-p-1}$ and $proc(\hat{b}_{n-n-p+1}b_{n-p-1}) = b_{n-n-p+1}$.

Redistribution corresponds to a remapping of the processor bits. Changing the distribution from $cyclic(2^b)$ to $cyclic(2^{b'})$ implies using bits $b_{b'+p} \cdots b_{b'+1}$ of the global address for the processor address instead of bits $b_{b+p} \cdots b_{b+1}$.

[a] The notation b_{i-j}, where $i \geq j$, denotes the bit sequence $b_i b_{i-1} \cdots b_j$ and b_{k-j}, where $k \leq j$, denotes the bit sequence $b_k b_{k+1} \cdots b_j$.

3. Cooley-Tukey FFT

For simplicity of presentation, we will assume that the input is in bit-reversed order for the Cooley-Tukey FFT and the Pease FFT. The twiddle factors will be assumed to be replicated on all processors. Furthermore, we will present only radix-2 decimation-in-time FFT algorithms. However, the techniques presented in this paper are also applicable to mixed-radix and decimation-in-frequency FFTs [2]. We will use N $(= 2^n)$ to denote the size of the FFT being performed and P $(= 2^p;\ p \leq n)$ to denote the number of processors being used to perform the FFT.

A FFT is an $O(N \log N)$ algorithm to compute the matrix-vector product:

$$B(i) = \sum_{j=0}^{N-1} \omega_N^{ij} A(j), \quad \text{where } 0 \leq i < N \text{ and } \omega_N = e^{2\pi\sqrt{-1}/N}.$$

In a 2^n point Cooley-Tukey FFT [5], there are n steps of computation. At step i, $1 \leq i \leq n$, the following *butterfly* computation is performed on elements at address $b_{n \leftarrow i+1} 0 b_{i-1 \leftarrow 1}$ and $b_{n \leftarrow i+1} 1 b_{i-1 \leftarrow 1}$:

$$\begin{aligned}
B(b_{n \leftarrow i+1} 0 b_{i-1 \leftarrow 1}) &= A(b_{n \leftarrow i+1} 0 b_{i-1 \leftarrow 1}) \\
B(b_{n \leftarrow i+1} 1 b_{i-1 \leftarrow 1}) &= \omega_N^{2^{n-i} * < b_{i-1 \leftarrow 1} >} * A(b_{n \leftarrow i+1} 1 b_{i-1 \leftarrow 1}) \\
A(b_{n \leftarrow i+1} 0 b_{i-1 \leftarrow 1}) &= B(b_{n \leftarrow i+1} 0 b_{i-1 \leftarrow 1}) + B(b_{n \leftarrow i+1} 1 b_{i-1 \leftarrow 1}) \\
A(b_{n \leftarrow i+1} 1 b_{i-1 \leftarrow 1}) &= B(b_{n \leftarrow i+1} 0 b_{i-1 \leftarrow 1}) - B(b_{n \leftarrow i+1} 1 b_{i-1 \leftarrow 1}),
\end{aligned}$$

where $< b_{k \leftarrow 1} >$ is the decimal representation of $b_{k \leftarrow 1}$. The above computation is performed on elements whose address differ only in bit b_i. This will be denoted by $b_{n \leftarrow i+1} B_i b_{i-1 \leftarrow 1}$.

A simple way to implement the Cooley-Tukey FFT on a distributed-memory multiprocessor is to use a *block* distribution for the input array A. The communication required for such a program can be determined from its trace in terms of the global address. For $i \leq n - p$, the computation can be summarized as $\hat{b}_{n \leftarrow n-p+1} b_{n-p \leftarrow i+1} B_i b_{i-1 \leftarrow 1}$, whereas, for $i > n-p$, it can be summarized as $\hat{b}_{n \leftarrow i+1} B_i \hat{b}_{i-1 \leftarrow n-p} b_{n-p-1 \leftarrow 1}$. Communication is required for steps where the butterfly is performed on elements differing in bits used for the processor index, i.e., for steps $n - p + 1$ through n. Hence, such a program requires p communication steps; at step $n - p + j$, $1 \leq j \leq p$, processor $b_{p \leftarrow 1}$ communicates with processor $b_{p \leftarrow j+1} \bar{b}_j b_{j-1 \leftarrow 1}$, where $\bar{b}_j$ is the complement of b_j. It can be similarly determined that, if A has a fixed distribution of $cyclic(2^b)$, then communication is required for steps $b + 1$ to $b + p$; the rest of the $n - p$ steps are communication-free.

If $n \geq 2p$ $(N \geq P^2)$ then by choosing the most significant p bits for the first $n - p$ steps and the least significant p bits for the last p steps to represent the processor address, we can localize the computation for all n steps. This implies that, the input array A is initially distributed using a *block* distribution and then after performing $n - p$ steps of computation, its distribution is changed to a *cyclic* distribution. The computation performed on each node before the redistribution is:

$$\begin{aligned}
B'(b_{n-p \leftarrow i+1} 0 b_{i-1 \leftarrow 1}) &= A'(b_{n-p \leftarrow i+1} 0 b_{i-1 \leftarrow 1}) \\
B'(b_{n-p \leftarrow i+1} 1 b_{i-1 \leftarrow 1}) &= \omega_N^{2^{n-i} * < b_{i-1 \leftarrow 1} >} * A'(b_{n-p \leftarrow i+1} 1 b_{i-1 \leftarrow 1}) \\
A'(b_{n-p \leftarrow i+1} 0 b_{i-1 \leftarrow 1}) &= B'(b_{n-p \leftarrow i+1} 0 b_{i-1 \leftarrow 1}) + B'(b_{n-p \leftarrow i+1} 1 b_{i-1 \leftarrow 1}) \\
A'(b_{n-p \leftarrow i+1} 1 b_{i-1 \leftarrow 1}) &= B'(b_{n-p \leftarrow i+1} 0 b_{i-1 \leftarrow 1}) - B'(b_{n-p \leftarrow i+1} 1 b_{i-1 \leftarrow 1}).
\end{aligned}$$

After a *block* to *cyclic* redistribution, a node with processor identifier pid performs:

$$B'(b_{n-1-i+1}0b_{i-1-p+1}) = A'(b_{n-1-i+1}0b_{i-1-p+1})$$
$$B'(b_{n-1-i+1}1b_{i-1-p+1}) = \omega_N^{2^{n-i}*(2^p*<b_{i-1-p+1}>+pid)} * A'(b_{n-1-i+1}1b_{i-1-p+1})$$
$$A'(b_{n-1-i+1}0b_{i-1-p+1}) = B'(b_{n-1-i+1}0b_{i-1-p+1}) + B'(b_{n-1-i+1}1b_{i-1-p+1})$$
$$A'(b_{n-1-i+1}1b_{i-1-p+1}) = B'(b_{n-1-i+1}0b_{i-1-p+1}) - B'(b_{n-1-i+1}1b_{i-1-p+1}).$$

In general, $\lceil n/(n-p) \rceil$ redistribution steps would be required for any $n \geq p$. If the initial distribution of the input array is to preserved, then one extra redistribution is required at the end of the computation.

4. Pease FFT

Pease proposed an FFT algorithm suitable for parallel processing [6]. In this algorithm, the array of intermediate results is permuted (using the perfect shuffle) after the butterfly computation at each step so that the two data elements involved in a butterfly computation at the next step are moved adjacent to each other. The following computation is performed at each step i, $1 \leq i \leq n$, of the Pease FFT:

$$A(b_{n-1-2}1) = \omega_N^{2^{n-i}*<b_{n-1-n-i+1}>} * A(b_{n-1-2}1)$$
$$B(b_{n-1-2}0) = A(b_{n-1-2}0) + A(b_{n-1-2}1)$$
$$B(b_{n-1-2}1) = A(b_{n-1-2}0) - A(b_{n-1-2}1)$$
$$A(b_1 b_{n-1-2}) = B(b_{n-1-2}b_1).$$

A direct implementation of the Pease FFT algorithm on a distributed-memory multiprocessor would be to distribute array A according to some block-cyclic distribution. Suppose array A is initially distributed using a *block* distribution. The shuffle permutation at each step, in terms of the global address, can be expressed as:

$$A(\hat{b}_1\hat{b}_{n-1-n-p+2}b_{n-p+1-1-2}) = B(\hat{b}_{n-1-n-p+1}b_{n-p-1-1}).$$

Note that $proc(\hat{b}_{n-1-n-p+1}b_{n-p-1-1}) \neq proc(\hat{b}_1\hat{b}_{n-1-n-p+2}b_{n-p+1-1-2})$, which implies that communication is needed at every step. This is true for any distribution of the input array. To avoid communication at each step, we need to modify the Pease FFT algorithm. We now present a modified Pease FFT algorithm which requires a data redistribution, after every $n-p$ steps. When $n \geq 2p$, only a single redistribution is required.

In the modified algorithm, a shuffle permutation is performed only on the local address bits. For the first $n-p$ steps, the shuffle permutation is performed on the least significant $n-p$ bits. This preserves bits $b_{n-1-n-p+1}$ in the most significant p address positions. Similarly, for the last p steps, the shuffle permutation is performed only on the most significant p bits; preserving bits b_{p-1-1} in the least significant p address positions. This modification permits a use of a *block* distribution for the first $n-p$ steps and a *cyclic* distribution for the last p steps, with the desirable property that communication is only needed for a redistribution after step $n-p$. The computation performed on each node before the redistribution is:

$$A'(b_{n-p-1-2}1) = \omega_N^{2^{n-i}*<b_{n-p-1-n-p-i+1}>} * A'(b_{n-p-1-2}1)$$
$$B'(b_{n-p-1-2}0) = A'(b_{n-p-1-2}0) + A'(b_{n-p-1-2}1)$$
$$B'(b_{n-p-1-2}1) = A'(b_{n-p-1-2}0) - A'(b_{n-p-1-2}1)$$
$$A'(b_1 b_{n-p-1-2}) = B'(b_{n-p-1-2}b_1).$$

The computation performed on each node after a *block* to *cyclic* redistribution is:

$$
\begin{aligned}
A'(b_{n\leftarrow n-p+2}1b_{n-p\leftarrow p+1}) &= \omega_N^{2^{n-i}*\alpha(j,pid)} * A'(b_{n\leftarrow n-p+2}1b_{n-p\leftarrow p+1}) \\
B'(b_{n\leftarrow n-p+2}0b_{n-p\leftarrow p+1}) &= A'(b_{n\leftarrow n-p+2}0b_{n-p\leftarrow p+1}) + A'(b_{n\leftarrow n-p+2}1b_{n-p\leftarrow p+1}) \\
B'(b_{n\leftarrow n-p+2}1b_{n-p\leftarrow p+1}) &= A'(b_{n\leftarrow n-p+2}0b_{n-p\leftarrow p+1}) - A'(b_{n\leftarrow n-p+2}1b_{n-p\leftarrow p+1}) \\
A'(b_1b_{n\leftarrow n-p+2}b_{n-p\leftarrow p+1}) &= B'(b_{n\leftarrow n-p+2}b_1b_{n-p\leftarrow p+1}),
\end{aligned}
$$

where $j = i - n + p$ and $\alpha(j, pid) = 2^{n-p}*<b_{n-1\leftarrow n-j+1}> +2^p*<b_{n-2p\leftarrow p+1}> +pid$. A trace of the modified Pease FFT algorithm in terms of the initial global address bit pattern is shown below. Bits $b_{n\leftarrow n-p+1}$ are fixed to represent the processor address for the first $n-p$ steps and $b_{p\leftarrow 1}$ are fixed to represent the processor address for the last p steps of the computation. Steps ia and ib are the computation and the permutation phase at step i.

Step	global ad.	proc ad.	local ad.
$1a$	$b_{n\leftarrow n-p+1}b_{n-p\leftarrow 2}B_1$	$b_{n\leftarrow n-p+1}$	$b_{n-p\leftarrow 1}$
$1b$	$b_{n\leftarrow n-p+1}b_1b_{n-p\leftarrow 2}$	$b_{n\leftarrow n-p+1}$	$b_1b_{n-p\leftarrow 2}$
$2a$	$b_{n\leftarrow n-p+1}b_1b_{n-p\leftarrow 3}B_2$	$b_{n\leftarrow n-p+1}$	$b_1b_{n-p\leftarrow 2}$
$2b$	$b_{n\leftarrow n-p+1}b_{2\leftarrow 1}b_{n-p\leftarrow 3}$	$b_{n\leftarrow n-p+1}$	$b_{2\leftarrow 1}b_{n-p\leftarrow 3}$
...	...	...	...
$(n-p)b$	$b_{n\leftarrow n-p+1}b_{n-p\leftarrow 1}$	$b_{n\leftarrow n-p+1}$	$b_{n-p\leftarrow 1}$
$redist.$	$b_{n\leftarrow p+1}b_{p\leftarrow 1}$	$b_{p\leftarrow 1}$	$b_{n\leftarrow n-p+1}b_{n-p\leftarrow p+1}$
$(n-p+1)a$	$b_{n\leftarrow n-p}B_{n-p+1}b_{n-p\leftarrow p+1}b_{p\leftarrow 1}$	$b_{p\leftarrow 1}$	$b_{n\leftarrow n-p+1}b_{n-p\leftarrow p+1}$
$(n-p+1)b$	$b_{n-p+1}b_{n\leftarrow n-p+2}b_{n-p\leftarrow p+1}b_{p\leftarrow 1}$	$b_{p\leftarrow 1}$	$b_{n-p+1}b_{n\leftarrow n-p+2}b_{n-p\leftarrow p+1}$
...	...	...	...
nb	$b_{n\leftarrow n-p+1}b_{p\leftarrow 1}$	$b_{p\leftarrow 1}$	$b_{n\leftarrow n-p+1}$

The Korn-Lambiotte FFT [7], which was developed for vector processing, can be similarly implemented on a distributed-memory vector multiprocessor. The Korn-Lambiotte FFT performs a shuffle permutation at the beginning of each step so that a fixed vector length of 2^{n-1} is achieved. Using a technique similar to the one used for the Pease FFT, the Korn-Lambiotte FFT can be modified so that only a single redistribution is required when $n \geq 2p$. Further, a fixed vector length of 2^{n-p-1} is achieved on each node.

5. Stockham FFT

In the Stockham FFT [8], at step i, $1 \leq i \leq n$, a permutation corresponding to a right cyclic shift of the most significant i address bits is performed on the input vector. The input vector is then multiplied by appropriate twiddle factors and a butterfly operation is performed with the lower-half and upper-half of the input vector as its two inputs. The Stockham FFT has the bit-reversal permutation implicitly embedded in its computation and therefore does not require the initial bit-reversal needed in the Cooley-Tukey and Pease FFT. The following computation is performed at step i:

$$
\begin{aligned}
B(b_{n-i+1}b_{n\leftarrow n-i+2}b_{n-i\leftarrow 1}) &= \omega_N^{2^{n-i}*\beta(i)} * A(b_{n\leftarrow n-i+2}b_{n-i+1}b_{n-i\leftarrow 1}) \\
A(0b_{n\leftarrow n-i+2}b_{n-i\leftarrow 1}) &= B(0b_{n\leftarrow n-i+2}b_{n-i\leftarrow 1}) + B(1b_{n\leftarrow n-i+2}b_{n-i\leftarrow 1}) \\
A(1b_{n\leftarrow n-i+2}b_{n-i\leftarrow 1}) &= B(0b_{n\leftarrow n-i+2}b_{n-i\leftarrow 1}) - B(1b_{n\leftarrow n-i+2}b_{n-i\leftarrow 1}),
\end{aligned}
$$

where $\beta(i) =< b_{n-i+1} > * < b_{n\leftarrow n-i+2} >$.

To implement the Stockham FFT on a 2^p processor distributed-memory multiprocessor, it can be determined that if we choose the least significant p bits for the processor address, i.e., *cyclic* distribution, then the first $n-p$ steps are communication-free. This is because the least significant p bits are not affected by the permutation performed in these steps. However, communication will be required for the remaining p steps. Redistributing the input array to any other distribution will also require p steps of communication. In order to use redistributions to eliminate subsequent communication steps, we modify the Stockham FFT algorithm.

Now assuming that $n \geq 2p$, a trace of the modified algorithm in terms of the initial global address bit pattern is shown below. Steps ia and ib are the permutation and the computation phase at step i.

Step	global ad.
$1a$	$b_{n\leftarrow p+1}\hat{b}_{p\leftarrow 1}$
$1b$	$B_n b_{n-1\leftarrow p+1}\hat{b}_{p\leftarrow 1}$
...	...
$(n-p)a$	$b_{p+1\rightarrow n}\hat{b}_{p\leftarrow 1}$
$(n-p)b$	$B_{p+1}b_{p+2\rightarrow n}\hat{b}_{p\leftarrow 1}$
$redist.$	$b_{p+1\rightarrow n-p}\hat{b}_{n-p+1\rightarrow n}b_{p\leftarrow 1}$
$perm.$	$b_{p+1\rightarrow n-p}b_{p\leftarrow 1}\hat{b}_{n-p+1\rightarrow n}$
$(n-p+1)a$	$b_{p\rightarrow n-p}b_{p-1\leftarrow 1}\hat{b}_{n-p+1\rightarrow n}$
$(n-p+1)b$	$B_p b_{p+1\rightarrow n-p}b_{p-1\leftarrow 1}\hat{b}_{n-p+1\rightarrow n}$
...	...
na	$b_{1\rightarrow n-p}\hat{b}_{n-p+1\rightarrow n}$
nb	$B_1 b_{2\rightarrow n-p}\hat{b}_{n-p+1\rightarrow n}$

Assuming that the global arrays A and B both have an initial *cyclic* distribution, the computation performed on each node for the first $n-p$ steps is:

$$B'(b_{n-i+1}b_{n\leftarrow n-i+2}b_{n-i\leftarrow p+1}) = \omega_N^{2^{n-i}*\beta(i)} * A'(b_{n\leftarrow n-i+2}b_{n-i+1}b_{n-i\leftarrow p+1})$$
$$A'(0b_{n\leftarrow n-i+2}b_{n-i\leftarrow p+1}) = B'(0b_{n\leftarrow n-i+2}b_{n-i\leftarrow p+1}) + B'(1b_{n\leftarrow n-i+2}b_{n-i\leftarrow p+1})$$
$$A'(1b_{n\leftarrow n-i+2}b_{n-i\leftarrow p+1}) = B'(0b_{n\leftarrow n-i+2}b_{n-i\leftarrow p+1}) - B'(1b_{n\leftarrow n-i+2}b_{n-i\leftarrow p+1}),$$

where $\beta(i) =< b_{n-i+1} > * < b_{n\leftarrow n-i+2} >$. After performing this computation, each node copies $A'(j)$ to $B'(j)$, $0 \leq j < 2^{n-p}$. The distribution of array B is then changed to $cyclic(2^p)$ (step $redist.$ in the trace above). Subsequently, an address-bit permutation, which corresponds to exchanging bits $b_{n-p+1\rightarrow n}$ with $b_{p\leftarrow 1}$, is performed (step $perm.$ in the trace above). This permutation is performed by copying $B(b_{p+1\rightarrow n-p}\hat{b}_{n-p+1\rightarrow n}b_{p\leftarrow 1})$ to $A(b_{p+1\rightarrow n-p}b_{p\leftarrow 1}\hat{b}_{n-p+1\rightarrow n})$. As,

$$proc(b_{p+1\rightarrow n-p}\hat{b}_{n-p+1\rightarrow n}b_{p\leftarrow 1}) = proc(b_{p+1\rightarrow n-p}b_{p\leftarrow 1}\hat{b}_{n-p+1\rightarrow n})$$

and

$$local(b_{p+1\rightarrow n-p}\hat{b}_{n-p+1\rightarrow n}b_{p\leftarrow 1}) = local(b_{p+1\rightarrow n-p}b_{p\leftarrow 1}\hat{b}_{n-p+1\rightarrow n}),$$

no communication is needed for this copying. Assuming $j = i-n+p$, the following computation is performed locally on each node for the remaining p steps:

$$B'(b_{2p-j+1}b_{n\leftarrow 2p-j+2}b_{2p-j\leftarrow p+1}) = \omega_N^{2^{n-i}*\gamma(j,pid)} * A'(b_{n\leftarrow 2p-j+2}b_{2p-j+1}b_{2p-j\leftarrow p+1})$$
$$A'(0b_{n\leftarrow 2p-j+2}b_{2p-j\leftarrow p+1}) = B'(0b_{n\leftarrow 2p-j+2}b_{2p-j\leftarrow p+1}) + B'(1b_{n\leftarrow 2p-j+2}b_{2p-j\leftarrow p+1})$$
$$A'(1b_{n\leftarrow 2p-j+2}b_{2p-j\leftarrow p+1}) = B'(0b_{n\leftarrow 2p-j+2}b_{2p-j\leftarrow p+1}) - B'(1b_{n\leftarrow 2p-j+2}b_{2p-j\leftarrow p+1})$$

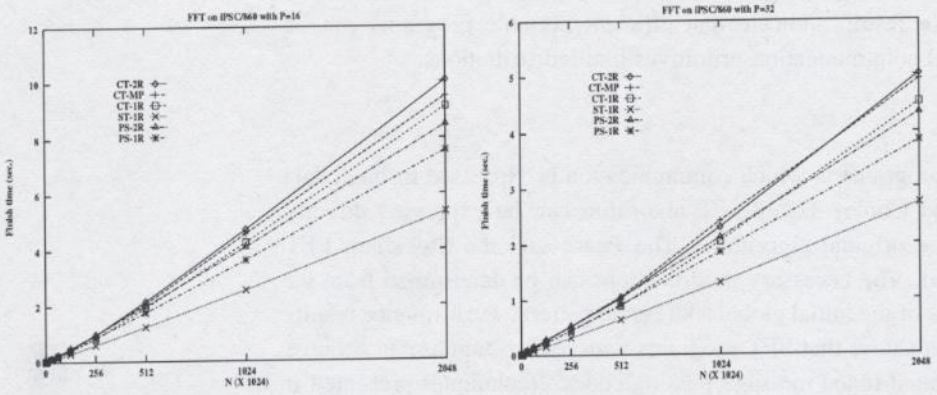

Fig. 1. FFT on iPSC/860 with P=16 and P=32.

where $\gamma(j, pid) =< b_{2p-j+1} > *(2^{p+j-1} * b_{n \leftarrow 2p+1} + 2^{j-1} * pid + b_{2p \leftarrow 2p-j+2})$.

As the distribution of array A remains unchanged, no final redistribution is needed to preserve the same distribution for the input and the output. Hence, the Stockham FFT can be performed with only one redistribution. Furthermore, the Stockham FFT does not require a bit-reversal permutation. This leads to a very communication-efficient distributed-memory implementation of the Stockham FFT.

6. Performance Results

We now present performance measurements for the Cooley-Tukey, Pease, and Stockham FFT implemented using redistributions on an Intel iPSC/860 system. The iPSC/860 is a distributed-memory hypercube. Using the one-to-one communication primitives in the iPSC's communication library, we implemented redistribution routines. Timings were measured using the millisecond node timer `mclock`. The achieved performance is compared with that of an explicit message-passing code (CT-MP) for the Cooley-Tukey FFT that uses $log(P)$ nearest-neighbor communication steps [9].

The Cooley-Tukey FFT requires communication only during the redistribution step where the distribution of array A is changed from *block* to *cyclic*. If the initial distribution of array A is to be preserved, then another redistribution will be required at the end of the program. Two versions of the Cooley-Tukey FFT were implemented, one using only a single redistribution step (CT-1R) and another one performing an additional *block* to *cyclic* redistribution (CT-2R). Similarly for the Pease FFT, two versions were implemented: PS-1R and PS-2R. The program for the Stockham FFT (ST-1R) requires communication only for redistributing array B from a *cyclic* to a $cyclic(2^p)$ distribution. In all programs except ST-1R, there is an initial bit-reversal permutation step. The implementation for the bit-reversal permutation uses one redistribution. Figure 1 shows performance results for data sizes ranging from 1K to 2M, on subcubes of size 16 and 32.

It can be observed that CT-2R performs nearly as well as or better than CT-MP.

CT-1R performs better than both CT-2R and CT-MP. PS-1R and PS-2R perform better than all the Cooley-Tukey programs. ST-1R has the best performance among all the FFT implementations. These results indicate that efficient portable programs can be implemented using high-level communication primitives like redistributions.

7. Conclusions

We have presented FFT programs in which communication is expressed through data redistribution primitives. The Cooley-Tukey FFT algorithm can be expressed directly without having to modify the original algorithm. The Pease and the Stockham FFT algorithm need to be modified. The necessary modifications can be determined from the trace of the algorithm in terms of the initial global address bit pattern. Performance results on an Intel's iPSC/860 system shows that FFT programs with redistributions can achieve performance comparable to hand-tuned message passing codes. Techniques presented in this paper have been used to develop efficient HPF programs for the FFT [10].

References

1. J. R. 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*, **9** (1990) 449–500.
2. C. Van Loan, *Computational frameworks for the fast Fourier transform.* SIAM, 1992.
3. High Performance Fortran Forum, High Performance Fortran language specification, Technical Report CRPC-TR92225, Rice University, 1993.
4. S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan, A methodology for the generation of data distributions to optimize communication, *Proc. Fourth IEEE Symposium on Parallel and Distributed Processing*, Dec. 1992, 436–441.
5. J. W. Cooley and J. W. Tukey, An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation.* **19** (1965) 297–301.
6. M. C. Pease, An adaptation of the fast Fourier transform for parallel processing, *J. ACM*, **15(2)** (1968) 252–264.
7. D. G. Korn and J. J. Lambiotte, Computing the fast Fourier transform on a vector computer, *Mathematics of Computation*, **33** (1979) 977–992.
8. T. G. Stockham, High speed convolution and correlation, *Proc. Spring Joint Computer Conf., AFIPS*, 1966, 229–233.
9. P. N. Swarztrauber, Multiprocessor FFT's, *Parallel Computing*, **5** (1987) 197–210.
10. S. K. S. Gupta, C.-H. Huang, R. W. Johnson, and P. Sadayappan, Programming FFT in High Performance Fortran. *Proc. Int'l Conf. on Parallel and Distributed Systems*, Taiwan, 1993, To appear.