CONCURRENCY: PRACTICE AND EXPERIENCE, VOL. 10(2), 73–90 (FEBRUARY 1998)

A technique for overlapping computation and communication for block recursive algorithms

S. K. S. GUPTA[1]*, C.-H. HUANG[2], P. SADAYAPPAN[2] AND R. W. JOHNSON[3]

[1]*Department of Computer Science, Colorado State University, Ft. Collins, CO 80523, U.S.A.*

[2]*Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210, U.S.A.*

[3]*Department of Computer Science, St. Cloud State University, St. Cloud, MN 56301, U.S.A.*

SUMMARY

This paper presents a design methodology for developing efficient distributed-memory parallel programs for block recursive algorithms such as the fast Fourier transform (FFT) and bitonic sort. This design methodology is specifically suited for most modern supercomputers having a distributed-memory architecture with a circuit-switched or wormhole routed mesh or a hypercube interconnection network. A mathematical framework based on the tensor product and other matrix operations is used for representing algorithms. Communication-efficient implementations with effectively overlapped computation and communication are achieved by manipulating the mathematical representation using the tensor product algebra. Performance results for FFT programs on the Intel Paragon are presented. ©1998 John Wiley & Sons, Ltd.

Concurrency: Pract. Exper., Vol. **10**(2), 73–90 (1998)

1. INTRODUCTION

To achieve a high performance on distributed-memory machines, algorithms should be tailored to the machine's hardware, such as the network topology and the node processor's architecture. Many modern supercomputers have a distributed-memory architecture with either mesh or hypercube topology. For example, the Intel iPSC/860 is a hypercube interconnection network and the Intel Paragon uses a mesh topology. Wormhole routing has been adopted by new generation parallel computers such as the Intel Paragon and the Cray T3D. Such networks are known to be insensitive to the routing distance if communication is contention-free. In this paper, we present a framework for designing and implementing block recursive algorithms for wormhole routed or circuit-switched machines with mesh or hypercube topology. Many important algorithms such as the fast Fourier transform (FFT) and the bitonic sort belong to the class of block recursive algorithms. The class of block recursive algorithms is similar to the class of ascend and descend algorithms[1]. Block recursive algorithms are used in many application areas such as digital signal processing[2,3], image processing[4], linear system design[5] and statistics[6].

In this paper, we use a mathematical representation of algorithms which is based on tensor products and other matrix operations. The tensor product framework has been used to design and implement high performance algorithms to compute the discrete Fourier

*Correspondence to: S. K. S. Gupta, Department of Computer Science, Colorado State University, Ft. Collins, CO 80523, U.S.A.

Contract grant sponsor: DARPA; Contract grant numbers: 60NANB1D1151, 60NANB1D1150.

CCC 1040–3108/98/020073–18$17.50

Received 9 September 1995
Revised 20 November 1996

transform (DFT)[7,8] and the matrix multiplication[9,10] on shared-memory vector multiprocessors. The significance of the tensor product lies in its ability to model both the computational structures occurring in block recursive algorithms, as well as the underlying hardware structures, like the interconnection networks[11,12]. This, coupled with the availability of a methodology for translating algorithms expressed using the tensor product and other matrix operations into parallel/vector codes for shared-memory vector multiprocessors[7,10] and distributed-memory MIMD machines[13,14], provides a basis for designing and implementing block recursive algorithms on various supercomputers.

We use tensor products and its algebraic properties to devise communication-efficient algorithms. Furthermore, to reduce the impact of communication overhead, the computation is overlapped with the communication. This is achieved by: (i) organizing the required communication into contention-free phases; (ii) partitioning the computation immediately before and after the communication step; and (iii) interleaving the pre- and postcomputation partitions with the communication phases in a manner which preserves the data dependency. This overlapping scheme is illustrated by developing an efficient implementation for the FFT algorithm.

Recently, message strip-mining has been developed by Wakatani and Wolfe for hiding communication cost[15]. Similar to our scheme, message strip-mining overlaps the pre/postcomputation with the communication for data redistribution. However, it does so by dividing the communication and computation into several blocks and skewing them in a pipelined manner. Message strip-mining increases the message start-ups. In contrast, our overlapping technique introduces no extra message startup overhead. It only partitions the pre- and postcomputation and schedules the communication and computation so that *effective* overlap of computation and communication is achieved. Proper orchestrating of the communication and computation is the key to our overlapping scheme. This we achieve by introducing no extra synchronization overhead.

In the context of FFT, many researchers have developed techniques for overlapping computation and communication. In [16], Johnsson *et al.* present fine-grained pipelined FFT algorithms for hypercube connected SIMD machines. FFT algorithms for both vector multiprocessors with shared memory and hypercube are presented by Swarztrauber in [17]. In [18], Agarwal *et al.* present a 3D FFT algorithm which overlaps computation with the communication. Their scheme is similar to message strip-mining: the 2D FFT computation on 2D slabs of the 3D array is overlapped with the communication for redistributing the 3D array from, say, (*,*,block) to (block,*,*) distribution. The overlapping scheme presented in this paper can be used in a straightforward manner, without adding any extra overhead. In [19], Agarwal *et al.* have presented a high-performance parallel algorithm for 1D FFT using its multidimensional formulation. Our overlapping technique for block recursive algorithms, in particular FFT, is also based on the multidimensional formulation (recursive tensor product factorization) of the computation. Agarwal *et al.* use the multidimensional formulation to break the local computation on each node into smaller FFTs which fit into the cache memory, whereas we use it to break the local computation into independent smaller FFTs which can be overlapped with the communication. Hence, the combination of both the schemes can be used to get both high node performance and effective overlapping of computation and communication.

The paper is organized as follows. In Section 2, we present a brief introduction to the theory of tensor products and the tensor product formulation of FFT algorithms. We describe how to map tensor product formulas to distributed-memory programs in Section 3.

The design goals for our overlapping scheme are described in Section 4. In Section 5, we develop a technique for overlapping computation with communication. Performance results are presented in Section 6, and conclusions are provided in Section 7.

2. INTRODUCTION TO TENSOR PRODUCT

In this Section, we present the theory of tensor products[20,21] and illustrate how algorithms can be expressed in the tensor product framework.

2.1. Preliminaries

Let $A^{m,n}$ be an $m \times n$ matrix and $B^{p,q}$ be a $p \times q$ matrix. The *tensor product* $A^{m,n} \otimes B^{p,q}$ is the block matrix obtained by replacing each element $a_{i,j}$ by the matrix $a_{i,j}B$. The following properties of tensor products can be readily established:

1. $A \otimes B \otimes C = A \otimes (B \otimes C) = (A \otimes B) \otimes C$
2. $(A \otimes B)(C \otimes D) = AC \otimes BD$
3. $A^{m,n} \otimes B^{p,q} = (A^{m,n} \otimes I_p)(I_n \otimes B^{p,q}) = (I_m \otimes B^{p,q})(A^{m,n} \otimes I_q)$
4. $\prod_{i=0}^{n-1}(I_m \otimes A_i) = I_m \otimes (\prod_{i=0}^{n-1} A_i)$,

where I_n represents the $n \times n$ identity matrix, A, B, C and D are matrices, $\prod_{i=0}^{n} A_i$ denotes $A_n \cdots A_0$, and all the involved matrix products are well-defined. An application of a computation matrix A to a vector v will be denoted by $A(v)$.

Stride permutations, such as shuffle permutation and matrix transposition, are frequently used in this framework. The application of *stride permutation* L_n^{mn} to a vector X^{mn} results in a vector Y^{mn} which is defined as

$$Y^{mn} = L_n^{mn}(X^{mn}) = \begin{bmatrix} X^{mn}(0 : mn-1 : n) \\ X^{mn}(1 : mn-1 : n) \\ \vdots \\ X^{mn}(m-1 : mn-1 : n) \end{bmatrix}$$

i.e. the first m elements of Y^{mn} are elements of X^{mn} at stride n starting with element 0, the next m elements are elements of X^{mn} at stride n starting with element 1, and so on. Note that if X^{mn} is a row-major order linear representation of a $m \times n$ matrix, $X^{m,n}$, then Y^{mn} is a row-major order representation of its transpose $(X^{m,n})^{\mathrm{T}}$. The following properties of the stride permutation can be easily established:

1. $(L_n^{mn})^{-1} = L_m^{mn}$
2. $L_{st}^{rst} = L_s^{rst} L_t^{rst}$.

We can view tensor product operations of the form $I_r \otimes A_c$ and $B_r \otimes I_c$ as operations on two-dimensional arrays. Let $n = r \times c$, $X^n \in C^n$ and $X^{r,c} \in C^{r,c}$ be such that $X^{r,c}(k,j) = X^n(kc + j)$, i.e. X^n is a row-major representation of $X^{r,c}$. Then

$$Y^n = (I_r \otimes A_c)(X^n) \equiv Y^{r,c} = X^{r,c} A_c^T \quad \text{and} \quad Y^n = (B_r \otimes I_c)(X^n) \equiv Y^{r,c} = B_r X^{r,c}$$

2.2. Fast Fourier transform

A *block recursive algorithm* is obtained from a recursive tensor product factorization of a computation matrix. For example, FFT algorithms are derived from tensor product factorization of the DFT matrix. The algorithms obtained from tensor product factorization are computationally more efficient than those that directly use the unfactorized matrix. For example, computing the DFT of a vector of size N by directly multiplying it by an $N \times N$ DFT matrix F_N requires $\mathrm{O}(N^2)$ operations compared to only $\mathrm{O}(N \log N)$ operations using an FFT algorithm.

In general, if $N = rc$ we have the following tensor product factorization for F_N:

$$F_{rc} = (F_r \otimes I_c) T_c^{rc} (I_r \otimes F_c) L_r^{rc}$$

where $T_c^{rc} = \mathrm{diag}[I_c, D_c^{rc}, (D_c^{rc})^2, \ldots, (D_c^{rc})^{r-1}]$ such that $D_c^{rc} = \mathrm{diag}(1, \omega, \ldots, \omega^{c-1})$ and ω is the Nth primitive root of unity. Note that $T_r^{rc} = L_c^{rc} T_c^{rc} L_r^{rc}$. Furthermore, the scaling of vector x^{rc} by the diagonal twiddle factor matrix T_c^{rc}: $y^{rc} = T_c^{rc} x^{rc}$ can be viewed as a pairwise multiplication of two matrices: $y^{r,c} = F_{rc}(0 : r-1, 0 : c-1) * x^{r,c}$, where $*$ denotes the pairwise multiplication operation. That is to say,

$$y^{r,c}(i,j) = F_{rc}(i,j) \times x^{r,c}(i,j),\ 0 \le i < r \quad \text{and} \quad 0 \le j < c$$

By recursively applying the above factorization, we get the Cooley–Tukey decimation-in-time FFT algorithm. For example, the tensor product formula for the radix-2 Cooley–Tukey FFT is

$$F_{2^n} = \left[\prod_{i=1}^{n} (I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}})(I_{2^{n-i}} \otimes T_{2^{i-1}}^{2^i}) \right] R_{2^n}$$

where $R_{2^n} = \prod_{i=1}^{n} (I_{2^{i-1}} \otimes L_2^{2^{n-i+1}})$. R_{2^n} permutes the input sequence into bit-reversed order.

The tensor product formulations of various FFT algorithms are presented in [7,8]. Some other examples of block recursive algorithms are Strassen's matrix multiplication[10], convolution[2] and fast sine/cosine transforms[8]. Ascend/descend algorithms[1] are a special case of block recursive algorithms.

3. DEVELOPING PROGRAMS FOR DISTRIBUTED-MEMORY MACHINES

This Section describes how communication-efficient distributed-memory parallel programs can be synthesized from tensor product formulation of an algorithm. Efficient programs can be synthesized from tensor product formulas by exploiting its regular structure. For example, $Y^{mp} = (I_m \otimes A^{p,n})(X^{mn})$ can be interpreted as m copies of $A^{p,n}$ acting in parallel on m disjoint segments of X^{mn}:

doall $i = 0, m-1$
  $Y^{mp}(ip : ip + p - 1) = A^{p,n}[X^{nm}(in : in + n - 1)]$
enddoall

Once the algorithm has been expressed using tensor products, efficient implementations of the algorithm can be obtained by exploiting the algebraic properties of the matrix

representation. The most frequently used algebraic property is the commutation rule for the tensor product, which involves the stride permutation. The *commutation rule* can be stated as follows:

$$A^{m,q} \otimes B^{p,n} = L^{pm}_{m}(B^{p,n} \otimes A^{m,q}) L^{nq}_{n}.$$

Consider the implementation of $\mathcal{F} \equiv Y^{mp} = (A^{p,n} \otimes I_m)(X^{nm})$. Using the commutation rule, it can be determined that

$$L^{mp}_{m}(Y^{mp}) = (I_m \otimes A^{p,n})[L^{mn}_{m}(X^{nm})]$$

Hence, $\mathcal{F}$ can be implemented as follows:

doall $i = 0, m-1$
$\quad Y^{mp}(i : mp-1 : m) = A^{p,n}[X^{nm}(i : mn-1 : m)]$
enddoall

On distributed-memory machines, data-parallelism is exploited by distributing the shared data across the local memories of interconnected processors. During execution, communication is needed when a processor requires data from another processor's local memory. In most distributed-memory machines, it is important to reduce the communication overhead to achieve high performance.

The common data distributions used are the block, cyclic and block-cyclic distributions[22]. A block-cyclic distribution $cyclic(b)$ of X^N on P processors partitions the array into blocks of b consecutive elements and then maps these blocks to the P processors in a cyclic manner. The $block$ and $cyclic$ distributions are equivalent to $cyclic(\lceil N/P \rceil)$ and $cyclic(1)$, respectively. For example, consider the block and cyclic distribution of X^{mn} on m processors. Under the block distribution, X^{mn} is split into m segments consisting of n consecutive elements and the ith segment $X^{mn}[in : (i+1)n - 1]$ is assigned to processor i, whereas under the cyclic distribution X^{mn} is partitioned into m segments: $X(i : mn : m)$, $0 \leq i < m$, and $X(i : mn : m)$ is assigned to processor i. Under any block-cyclic distribution of X^{mn} on m processors, n elements are allocated to each processor, which will be denoted by x^n.

For implementing a tensor product formula on a distributed-memory machine, the first step is to identify data distributions which can be used to implement it in a communication-free manner. If no such distribution exists, then data distributions which lead to the least communication overhead should be chosen. If different components of a formula are communication-free with respect to different data distributions, then the formula can be implemented with communication-free phases separated by data redistribution steps.

It can be easily established that the tensor product formula $Y^{mp} = (I_m \otimes A^{p,n})(X^{mn})$ can be implemented without any communication using m processors if both X^{mn} and Y^{mp} have a block distribution and each node processor performs the local computation $y^p = A^{p,n}(x^n)$. However, if we choose block distributions for the input and output, then the tensor product formula $\mathcal{F} \equiv Y^{mp} = (A^{p,n} \otimes I_m)(X^{mn})$ will require communication (unless $A^{p,n}$ is a diagonal matrix). The data distributions appropriate for $\mathcal{F}$ can be obtained by noting that

$$L^{mp}_{m}(Y^{mp}) = (I_m \otimes A^{p,n})[L^{mn}_{m}(X^{nm})]$$

which can be implemented in a communication-free manner if $L^{mn}_{m}(X^{nm})$ and $L^{mp}_{m}(Y^{mp})$ have a block distribution. Next consider the block distribution of the vector obtained by

applying L_m^{mn} to X^{mn}. This will result in allocation of $X^{mn}(0 : mn-1 : m)$ to the first processor, $X^{mn}(1 : mn-1 : m)$ to the second processor and finally $X^{mn}(m-1 : mn-1 : m)$ to the last processor. This corresponds exactly to a cyclic distribution of X^{mn} on m processors.* Hence, $\mathcal{F}$ can be implemented in a communication-free manner if both the input and output arrays have a cyclic distribution on m processors.

Next consider the following tensor product formula which represents the core computation in the Cooley–Tukey FFT:

$$F_{2^n} = \prod_{i=1}^{n} (I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}})$$

$Z^{2^n} = F_{2^n}(X^{2^n})$ can be implemented on a 2^q-processor ($2q \leq n$) distributed-memory machine as follows:

1. $Y = [I_{2^q} \otimes \prod_{i=1}^{n-q} (I_{2^{n-i-q}} \otimes F_2 \otimes I_{2^{i-1}})](X)$
2. $L_{2^q}^{2^n}(Z) = \{I_{2^q} \otimes [\prod_{i=n-q+1}^{n} (I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1-q}})]\} L_{2^q}^{2^n}(Y)$.

This implies that if both the input and the output arrays are distributed block-wise on 2^q processors, then the computation for the first part can be performed without any need for interprocessor communication. Each processor will perform $y^{2^{n-q}} = [\prod_{i=1}^{n-q} (I_{2^{n-i-q}} \otimes F_2 \otimes I_{2^{i-1}})](x^{2^{n-q}})$ in its local memory. Similarly, the second part of the computation can be implemented in a communication-free manner by distributing Z^{2^n} and Y^{2^n} in a cyclic fashion on 2^q processors. Hence, the computation can be implemented as follows: initially distribute array X and Y block-wise and array Z cyclically, perform the first part of the computation, change the distribution of Y to a cyclic distribution, and finally perform the second part of the computation. Communication is only required to redistribute array Y from a block to cyclic distribution. Assuming that $n \geq 2q$, this requires an all-to-all communication, where each processor sends $y^{2^{n-q}}[j2^{n-2q} : (j+1)2^{n-2q} - 1]$ to processor j. Assuming the time to communicate a message of size m is $t_s + mt_p$, where t_s is the message latency and t_p is the link transfer time per data element, the communication overhead is $2^q(t_s + 2^{n-2q}t_p) = 2^q t_s + 2^{n-q}t_p$. On the other hand, if the distribution of the arrays were fixed to the same distribution, then it can be determined easily that such an implementation will require q communication steps with the total communication cost of $q(t_s + 2^{n-q}t_p) = qt_s + q2^{n-q}t_p$. For any fixed value of the machine parameters t_s and t_p and sufficiently large problem size, it can be seen that the program using a redistribution will have a lower communication cost than that using a fixed distribution[23].

A more communication-efficient implementation for the FFT can be obtained by embedding the bit-reversal permutation in the FFT computation. This can be done by using the following four-step algorithm, which we will later use in the paper[24]:

$$F_{rc} = (F_c \otimes I_r) L_c^{rc} T_c^{rc} (F_r \otimes I_c) \tag{1}$$

This algorithm is better explained by using the two-dimensional view of the 1D FFT computation. The computation $Y^{rc} = F_{rc}X^{rc}$ can be performed as follows:

- $X^{r,c} = F_r X^{r,c}$

*In general, a block distribution of $(L_m^{nm/b} \otimes I_b)(X^{nm})$ corresponds to a $cyclic(b)$ distribution of X^{mn} on m processors.

- $X^{r,c} = F_n(0 : r-1 : 0 : c-1) * X^{r,c}$, where $*$ denotes pairwise matrix multiplication. This is equivalent to performing $X^{rc} = T_c^{rc} X^{rc}$
- $X^{c,r} = (X^{r,c})^{\mathrm{T}}$
- $Y^{c,r} = F_c X^{c,r}$.

According to the above algorithm, $Y^{rc} = F_n X^{rc}$ can be computed on a distributed-memory machine as follows:

1. Distribute $X^{r,c}$ as $(*, cyclic)$. This corresponds to $cyclic$ distribution of X^{rc}. Each processor can apply F_r to each column allocated to it. No communication is needed.
2. Each processor can perform the scaling of its columns locally.
3. Transpose the array. The transposition will require communication.
4. The last step can be performed locally similar to the first step.

To understand the communication required for the transposition, consider that $X^{r,c}$ has its initial distribution as $(*, cyclic)$ on P processors. Each processor will have c/P columns of size r. The local elements of a processor can be considered to be elements of an $r \times c/P$ array. This array can be partitioned along the first dimension into P parts so that we have P blocks of size $r/P \times c/P$. Then a block transposition algorithm can be used, in which a processor first transposes its ith block locally and sends it to processor i. This communication pattern corresponds to a personalized all-to-all communication or complete exchange. Hence the communication requirement for the four-step algorithm is the same as that for the above redistribution based algorithm, with the added advantage that the initial and final distribution are the same and no extra bit-reversal step is required (provided that an ordered FFT algorithm is used along each dimension). We use this two-dimensional view of the 1D FFT to develop the overlapping scheme described in Section 5.

4. DESIGN GOALS

In this Section we identify the goals which motivated the design of the overlapping scheme presented in the next Section.

4.1. Communication efficiency

The primary goal is to devise an overlapping scheme which is communication-efficient. We do this by using the multidimensional-view-based block recursive algorithm to incorporate the overlap of computation and communication. In such a view, the computation can be viewed as being performed in alternating computation and communication phases. In each computation phase the computation is performed along a particular dimension of the array and the communication involves all-to-all personalized communication for redistributing/transposing the array. Such a computation and communication pattern is seen in many algorithms besides a 1D FFT – for example, as in the multidimensional FFT[18] and ADI (alternating direction implicit) methods[15]. As shown in the case of a 1D FFT, programs based on such a program model are usually preferred when the problem size N is much greater than the machine size P, i.e. $N >> P$. This condition is usually true when solving problems on modern distributed-memory machines, such as the Intel Paragon and Cray T3D. Furthermore, modern machines have high-bandwidth low-latency communication networks, which favor programs with lower overall communication volume

but higher message count over those having higher communication volume but lower message count.

4.2. Effective overlapping

Once the algorithm has been made communication-efficient, one way to further reduce the impact of communication overhead is to overlap computation with communication. This can be done only when the receiver of a message has some computation to perform, which is independent of the data in the message. For the program structure shown above, this is possible when the precomputation and the postcomputation on each node can be performed in a blocked manner, so that pre- and postcomputations on a block of data can be overlapped with the communication of some other block of data. Suppose that a processor P_q sends to processors P_r, P_s and P_t blocks of data B_r, B_s and B_t, and receives from them blocks of data B'_r, B'_s and B'_t, respectively. Then, the precomputation for B_s and postcomputation for B_r on processor P_q can be overlapped with communication from P_s to P_q as shown in Figure 1.

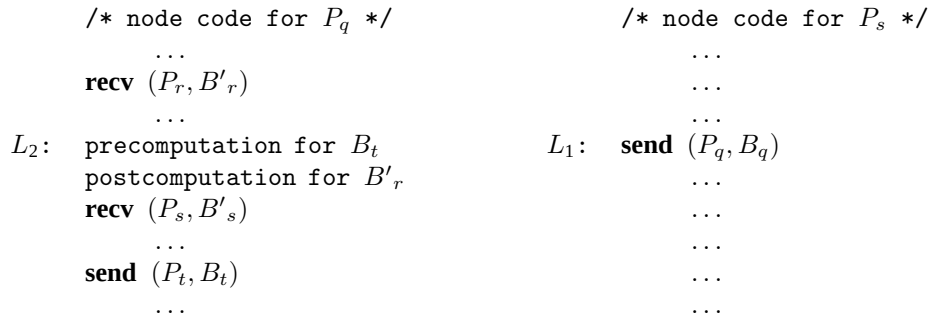

Figure 1. Overlapping communication and computation using nonblocking communication primitives

This overlapping would be *effective* only when the send from P_s to P_q (statement labeled L_1) takes place approximately at the same time as P_q starts precomputation for B_t (statement labeled L_2). If this send occurs much before statement L, then the message would already have reached P_q and there would be little or no overlap of computation with the communication. On the other hand, if the message from P_s to P_q is sent very late, say after the precomputation and postcomputation has already been performed on P_q, then also there would be no overlap of the computation and communication. So, one of the design requirements is to organize the computation and communication on each node such that the computation is effectively overlapped with the transmission of messages.

4.3. Memory efficiency

The third design goal is to minimize the memory requirement. In the overlapping scheme, we would like to use as little extra memory space as possible. For example, note that in the above example processor P_q can store the block B'_s in the memory space occupied by

B_s, if B_s is sent before receiving B'_s and the data in B_s is not required on processor P_q thereafter. Hence, one of the requirements on the way communication and computation are interleaved is that the data received by a processor can be stored in the memory space which is occupied by data already sent out. However, this should be done in an orderly fashion so that the indexing of data does not become complicated.

4.4. Adaptable to various target machines

Until now we have not taken into consideration the target machines characteristics. However, to synthesize communication-efficient programs, they should be tailored to the target machine's communication hardware, such as the network topology. Furthermore, the message routing used in the target machine should also be taken into consideration. Many modern distributed-memory machines have either a mesh or hypercube topology. For example, the Intel iPSC/860 has a hypercube interconnection network and the Intel Paragon uses a mesh topology. Wormhole routing has been adopted by new generation parallel computers, such as the Intel Paragon and Cray T3D. Such networks are known to be insensitive to the routing distance if the communication is contention-free.

A redistribution may involve an all-to-many communication pattern, which when performed naively may result in several messages using the same link simultaneously. In a distributed-memory machine which uses circuit switching (like the Intel iPSC/860) or wormhole routing (like the Intel Paragon), such link contentions result in severe performance degradation. Hence, such dense communication patterns are usually performed in several phases, each phase being contention-free. The idea is that if the phases are performed one after another in a synchronized manner, then the entire communication would be contention-free. Hence, the fourth requirement for the overlapping scheme is that it should be amenable to such communication schemes which are used to minimize contention.

5. TECHNIQUE FOR OVERLAPPING COMPUTATION WITH COMMUNICATION

In this Section, we describe an overlapping technique which meets the design goals mentioned in the previous Section. We would use FFT for illustrating the technique. Furthermore, we would describe the technique for machines with hypercube topology. Similar techniques can be applied to machines with other topologies.

5.1. Communication phases

We have seen that, when performing FFT, the communication for the redistribution is an all-to-all personalized communication, called complete exchange. In this communication pattern, each of the P processors in the network has equal size but different messages to be delivered to each of the $P-1$ processors. Without loss of generality, we would assume that a processor also sends a message to itself.

Many algorithms have been developed to perform complete exchange efficiently on meshes and hypercubes. These can be classified into direct algorithms and indirect (or store-and-forward) algorithms. We will only use direct complete exchange algorithms. In a direct algorithm, to perform a complete exchange on a wormhole-routed or circuit-switched network, the communication is scheduled in several contention-free steps.

On a hypercube, the complete exchange can be performed in P steps using the pairwise exchange algorithm[25]. At each step each processor exchanges a message with its partner. The following is pseudocode for the pairwise exchange algorithm:

do i = 1, P-1
  $mypal = xor(myid, i)$
  send (P_{mypal}, B_{mypal})
  recv (P_{mypal}, B_{mypal})
enddo

where $myid$ is the processor index of the processor executing the above code and $xor()$ is the bitwise exclusive-OR operation. On each processor, the data to be sent out can be viewed as P blocks of data: $B_0, \ldots, B_{P-1}$. If N is the size of the input array, then the size of each block is N/P^2. Note that each processor P_s has the data for itself in B_s. Figure 2 shows the blocks of data exchanged between two processors P_s and P_r in the step i, where i is such that $s = \text{xor}(r, i)$ and $r = \text{xor}(s, i)$.

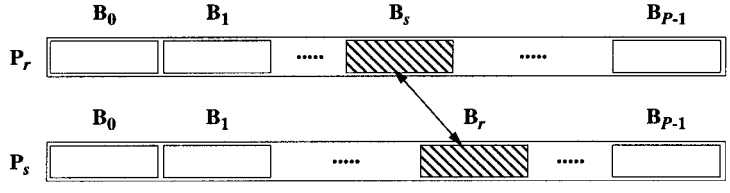

Figure 2. Blocks exchanged between a pair of processors

Since at every step each processor sends out a packet of size N/P^2 and receives a packet of the same size, the communication cost for a complete exchange on a hypercube would be

$$CC_{\text{hyper}} = (P-1)[t_s + (N/P^2)t_p] \approx (P-1)t_s + (N/P)t_p \tag{2}$$

On a $\sqrt{P} \times \sqrt{P}$ mesh, the complete exchange requires $P^{3/2}/4$ phases[6]. Each phase takes $[t_s + (N/P^2)t_p]$ time. Therefore, the communication cost for a complete exchange on a mesh would be

$$CC_{\text{mesh}} = (P^{3/2}/4)[t_s + (N/P^2)t_p] = (P^{3/2}/4)t_s + (N/4\sqrt{P})t_p \tag{3}$$

To overlap computation with the communication for the complete exchange, the computation can be organized so that the precomputation for the block to be sent in step $i+1$ and the post-computation for the block received in step $i-1$ can be overlapped with communication for the block sent in step i. We next describe how to partition the pre- and postcomputation to achieve such overlap.

5.2. Partitioning computation

In this Section, we describe the partitioning of computation using a multidimensional view of one-dimensional input and output arrays.

In the following, we show how the computation partitioning can be done by viewing the FFT as being performed on a 4-dimensional $P \times C \times R \times P$ array, where $N = RCP^2$.

To explain the overlapping technique, we use the following form of the FFT algorithm:

$$F_{rc} = (F_r \otimes I_c) T_c^{rc} L_r^{rc} (F_c \otimes I_r) \tag{4}$$

Let $r = R \times P$ and $c = C \times P$. We can apply the above formula recursively to express F_c in terms of F_C and F_P, and F_r in terms of F_R and F_P. Then, $X^{rc} = F_{rc}(X^{rc})$ can be computed as follows:

1. $X^{rc} = (T_P^{CP} \otimes I_R \otimes I_P)(L_C^{CP} \otimes I_R \otimes I_P)(F_P \otimes I_C \otimes I_R \otimes I_P)(X^{rc})$
2. $X^{rc} = (F_C \otimes I_P \otimes I_R \otimes I_P)(X^{rc})$
3. $X^{rc} = L_r^{rc}(X^{rc})$
4. $X^{rc} = T_c^{rc}(X^{rc})$
5. $X^{rc} = (F_R \otimes I_P \otimes I_C \otimes I_P)(X^{rc})$
6. $X^{rc} = (F_P \otimes I_R \otimes I_C \otimes I_P)(T_R^{RP} \otimes I_C \otimes I_P)(L_P^{RP} \otimes I_C \otimes I_P)(X^{rc})$.

Let X^{rc} have a $cyclic$ distribution on P processors. Since

$$L_r^{rc} = (L_{RP}^{RPC} \otimes I_P)(I_{RC} \otimes L_P^{P^2})(I_C \otimes L_R^{RP} \otimes I_P) \tag{5}$$

the computation on each node p is as follows:

1. $x^{RCP} = (T_P^{CP} \otimes I_R)(L_C^{CP} \otimes I_R)(F_P \otimes I_C \otimes I_R)(x^{RCP})$
2. $x^{RCP} = (F_C \otimes I_P \otimes I_R)(x^{RCP})$

3(a). $x^{RCP} = (I_C \otimes L_R^{RP})(x^{RCP})$

3(b). communication for $X^{rc} = (I_{RC} \otimes L_P^{P^2})(X^{rc})$

3(c). $x^{RCP} = L_{RP}^{RPC}(x^{RCP})$

4. $x^{RP,C} = F_n(0 : RP - 1, p : CP - 1 : P) * x^{RP,C}$
5. $x^{RCP} = (F_R \otimes I_P \otimes I_C)(x^{RCP})$
6. $x^{RCP} = (F_P \otimes I_R \otimes I_C)(T_R^{RP} \otimes I_C)(L_P^{RP} \otimes I_C)(x^{RCP})$.

Here x^{RCP} is the local array on each processor, corresponding to the global array X^{rc}. Communication is only required in step 3(b). The communication required is a complete exchange. The scaling by twiddle factors in step 4 is expressed using a two-dimensional view to show the twiddle factors used on each processor. We now organize the computation before and after this communication step so that they can be appropriately partitioned to achieve sufficient overlap of the computation with the communication. The goal is to avoid any use of extra memory space.

If the local array x^{RCP} is viewed as consisting of P blocks of data, then one way to avoid any use of extra memory space is to use the memory locations occupied by the block of data sent out at step i to store the data received at step i. This is possible when the sending of the data in a block is done before the receiving of the data which will later occupy it.

To achieve this in the above computation, the local data movement in step 3(a) should be moved to before step 2 and the local data movement in step 3(c) should be moved to after step 5. Steps 2 and 3(a) can be interchanged by noting that

$$(I_C \otimes L_R^{RP})(F_C \otimes I_P \otimes I_R) \quad = \quad (F_C \otimes I_R \otimes I_P)(I_C \otimes L_R^{RP}) \tag{6}$$

Also, the order of steps $3(c)$, 4 and 5 can be interchanged by noting that performing

$$x^{RP,C} = F_n(0 : RP-1, p : CP-1 : P) * L_{RP}^{RPC}(x^{C,RP}) \tag{7}$$

is the same as performing

$$x^{RP,C} = L_{RP}^{RPC}[F_n^T(0 : RP-1, p : CP-1 : P) * x^{C,RP}] \tag{8}$$

and $(F_R \otimes I_P \otimes I_C) L_{RP}^{RPC} = L_{RP}^{RPC}(I_C \otimes F_R \otimes I_P)$. Therefore, we have the following computation steps for F_{rc}:

1. $x^{RCP} = (T_P^{CP} \otimes I_R)(L_C^{CP} \otimes I_R)(F_P \otimes I_C \otimes I_R)(x^{RCP})$
2. $x^{RCP} = (I_C \otimes L_R^{RP})(x^{RCP})$
3. $x^{RCP} = (F_C \otimes I_R \otimes I_P)(x^{RCP})$
4. communication for $X^{rc} = (I_{RC} \otimes L_P^{P^2})(X^{rc})$
5. $x^{C,RP} = F_n^T(0 : RP-1, p : CP-1 : P) * x^{C,RP})$
6. $x^{CRP} = (I_C \otimes F_R \otimes I_P)(x^{CRP})$
7. $x^{RP,C} = L_{RP}^{RPC}(x^{C,RP})$
8. $x^{RCP} = (F_P \otimes I_R \otimes I_C)(T_R^{RP} \otimes I_C)(L_P^{RP} \otimes I_C)(x^{RCP})$.

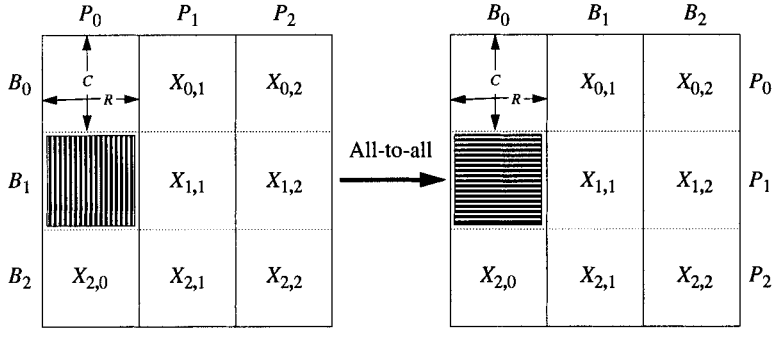

Figure 3. Two dimensional view of the FFT computation

Figure 3 shows the two-dimensional view of the FFT computation performed on three processors. In general, the array X^{rc} is viewed as a two dimensional array $X^{c,r}$. As $r = RP$, the *cyclic* distribution of X^{rc} corresponds to a *cyclic* distribution of $X^{c,r}$ along its second dimension. The elements $X^{rc}(p : rc-1 : P)$ on processor p under the cyclic distribution are viewed as P $(C \times R)$ data blocks: $X_{0,p}$, $X_{1,p}$, ..., $X_{P-1,p}$. In step 4, processor p sends block $X_{q,p}$ to processor q. Before sending block $X_{q,p}$ to processor q the computation corresponding to step 3 is performed. This amounts to performing FFT along the columns of the block. After this block is received by processor q, it performs the computation corresponding to steps 5 and 6, by scaling each data element of the block and then performing FFT along the rows of the block. The computation corresponding to step 3 will be called the $precomputation$ and the computation corresponding to steps 5 and 6 will be called the $postcomputation$. The computations in steps 1 and 2 can be viewed as

the preprocessing steps and the computation in steps 7 and step 8 can be viewed as the postprocessing steps.

5.3. Interleaving computation partitions with communication

We now describe the overlapping scheme for the hypercube. A similar scheme can be used for distributed-memory machines with other topologies such as a mesh. We use the following notation to describe the overlapping scheme:

1. B_k: represents the kth block on a processor, say p. Before the communication it contains the data elements of $X_{k,p}$ and after the communication step it holds the data elements of $X_{p,k}$
2. $precomp(B_k)$: performs the computation corresponding to $(F_C \otimes I_R)$ on B_k
3. $postcomp(B_k)$: performs the computation corresponding to scaling followed by $(I_C \otimes F_R)$ on B_k.

The pseudocode for the overlapping scheme is shown in Figure 4. It uses buffered non-blocking send and blocking receive communication primitives. Each processor performs the preprocessing steps and the precomputation on its own block, and the block which will be sent out in the first communication step performs $P-1$ communication steps, followed by a postcomputation step and the postprocessing steps. During the first $P-2$ communication steps, the precomputation of the block to be sent in the following communication step and the postcomputation of the block which was received in the previous communication step is overlapped with the communication of the block sent in the current communication step. In the last communication step, only the postcomputation for the block of data received in step $P-2$ is overlapped with communication.

preprocessing steps
$postpal = xor(myid, 1)$
$precomp(B_{myid})$
$precomp(B_{postpal})$
do $i = 1, P-2$
    $mypal = xor(myid, i)$
    $prepal = xor(myid, i-1)$
    $postpal = xor(myid, i+1)$
    send B_{mypal} to $mypal$
    $precomp(B_{postpal})$
    $postcomp(B_{prepal})$
    recv B_{mypal} from $mypal$
enddo
$postcomp(B_{mypal})$
send B_{P-1} to $P-1$
$postcomp(B_{P-2})$
recv B_{P-1} from $P-1$
$postcomp(B_{P-1})$
postprocessing steps

Figure 4. The overlapping scheme.

6. PERFORMANCE RESULTS

In this section, we present the performance results for the implementation of FFT programs developed based on the methodology presented in this paper.

6.1. Performance analysis

We now present analytical performance modeling for the above overlapping scheme. Let $T_{\text{fft}}(M)$ be the time for computing an FFT of size M on one processor. We use the following estimate for $T_{\text{fft}}(M) = 5M\log(M)t_c$, where t_c is the time to perform one floating-point operation. The execution time for the implementation with no computation overlapped with the communication can be estimated as follows.

$$\begin{aligned} T_{\text{no}} &= T_{\text{comp}} + T_{\text{comm}} \\ &= 5N[\log(N)/P]t_c + (P-1)t_s + (P-1)(N/P^2)t_p \end{aligned} \tag{9}$$

The execution time for the implementation with the computation overlapped with the communication can be estimated as follows:

$$\begin{aligned} T_{\text{o}} &= 2RCT_{\text{fft}}(P) + 2RT_{\text{fft}}(C) + (P-1)t_s \\ &\quad +(P-2)\max[RT_{\text{fft}}(C) + CT_{\text{fft}}(R), RCt_p] \\ &\quad + \max[CT_{\text{fft}}(R), RCt_p] + CT_{\text{fft}}(R) \end{aligned} \tag{10}$$

Assuming $R = C$,

$$\begin{aligned} T_{\text{o}} &= 10(N/P)\log(P)t_c + 15(N/P^2)\log(C)t_c \\ &\quad +(P-1)t_s + N/P^2\max[5\log(C)t_c, t_p] \\ &\quad +(P-2)(N/P^2)\max[10\log(C)t_c, t_p] \end{aligned} \tag{11}$$

Only the time corresponding to the message propagation can be overlapped with the computation. According to (11), the propagation time is completely overlapped with the computation when the following condition is satisfied:

$$t_p = 10\log(C)t_c, \quad \text{i.e.} \quad t_p/t_c = 5\log(N/P^2) \tag{12}$$

If N is a power of two and $P = 2^p$, then we get

$$n_{\text{peak}} = 2p + t_p/(5t_c) \tag{13}$$

where maximum performance benefits are achieved for $N = 2^{n_{\text{peak}}}$. The value of $T_{\text{no}}/T_{\text{o}}$ when the above condition is satisfied is

$$\begin{aligned} T_{\text{no}}/T_{\text{o}} &\approx 1 + (P-1)(N/P^2)t_p/\{5N[\log(N)/P]t_c + (P-1)t_s\} \\ &\approx 1 + t_p/[5\log(N)t_c + (P^2/N)t_s] \end{aligned} \tag{14}$$

Table 1. Execution times (s) on the Intel Paragon system for $P = 256$ (d = dedicated mode, i = interactive mode).

		No overlap				Overlap			
$\log_2(N)$	d/i	T_{no}	T_{comp}	T_{comm}	$T_{\text{comp}}/T_{\text{comm}}$	T_{o}	Actual $T_{\text{no}}/T_{\text{o}}$	Predicted $T_{\text{no}}/T_{\text{o}}$	$T_{\text{comm}}/(T_{\text{o}} - T_{\text{comp}})$
16	d	0.039	0.007	0.032	0.2	0.035	1.10	1.04	1.12
	i	0.037	0.006	0.031	0.2	0.034	1.11	1.01	1.13
17	d	0.042	0.009	0.033	0.3	0.041	1.02	1.06	1.03
	i	0.042	0.010	0.032	0.3	0.039	1.06	1.03	1.09
18	d	0.050	0.017	0.033	0.5	0.048	1.05	1.05	1.08
	i	0.050	0.019	0.031	0.5	0.047	1.05	1.01	1.08
19	d	0.059	0.026	0.033	0.8	0.057	1.04	1.04	1.07
	i	0.059	0.026	0.033	0.8	0.054	1.10	1.04	1.20
20	d	0.073	0.039	0.034	1.2	0.068	1.08	1.05	1.20
	i	0.074	0.040	0.034	1.2	0.065	1.14	1.05	1.36
21	d	0.102	0.067	0.035	1.9	0.098	1.05	1.05	1.14
	i	0.104	0.068	0.036	1.9	0.093	1.12	1.05	1.47
22	d	0.153	0.116	0.037	3.2	0.145	1.06	1.04	1.30
	i	0.156	0.119	0.037	3.3	0.143	1.09	1.04	1.52
23	d	0.280	0.220	0.060	3.7	0.259	1.08	1.12	1.53
	i	0.278	0.219	0.059	3.7	0.254	1.10	1.11	1.71
24	d	0.561	0.479	0.082	5.9	0.529	1.06	1.10	1.66
	i	0.560	0.479	0.081	5.9	0.528	1.06	1.10	1.64
25	d	1.228	1.099	0.129	8.6	1.163	1.06	1.09	2.02
	i	1.254	1.100	0.154	7.1	1.162	1.08	1.11	2.47
26	d	2.357	2.095	0.262	8.0	2.193	1.07	1.11	2.67
	i	2.369	2.105	0.264	8.0	2.183	1.09	1.11	3.42
27	d	4.818	4.289	0.529	8.1	4.431	1.09	1.12	3.73
	i	4.813	4.279	0.534	8.0	4.420	1.09	1.12	3.84

Table 2. Execution times (sec.) on the Intel Paragon system for $P = 512$

	No overlap				Overlap			
$\log_2(N)$	T_{no}	T_{comp}	T_{comm}	$T_{\text{comp}}/T_{\text{comm}}$	T_{o}	Actual $T_{\text{no}}/T_{\text{o}}$	Predicted $T_{\text{no}}/T_{\text{o}}$	$T_{\text{comm}}/(T_{\text{o}} - T_{\text{comp}})$
18	0.080	0.013	0.067	0.20	0.073	1.10	1.08	1.12
19	0.084	0.019	0.065	0.30	0.081	1.04	1.05	1.05
20	0.102	0.035	0.067	0.51	0.098	1.04	1.06	1.06
21	0.121	0.053	0.068	0.77	0.114	1.06	1.06	1.11
22	0.149	0.079	0.070	1.13	0.137	1.09	1.06	1.22
23	0.209	0.137	0.072	1.90	0.196	1.07	1.05	1.23
24	0.355	0.239	0.116	2.07	0.308	1.15	1.18	1.67
25	0.629	0.478	0.151	3.16	0.562	1.12	1.17	1.80
26	1.242	0.991	0.251	3.95	1.105	1.12	1.18	2.21
27	2.830	2.251	0.579	3.89	2.423	1.17	1.22	3.36
28	5.607	4.302	1.305	3.30	4.581	1.22	1.29	4.69

6.2. Experimental performance results

This Section presents the performance results for an FFT on the Intel Paragon, which is a wormhole routed mesh.

The program which used overlapping of computation and communication was developed using the overlapping scheme above. A pairwise exchange algorithm was used to perform complete exchange. On a mesh the pairwise exchange algorithm is not contention-free; however, it has been found to perform better than other contention-free schemes proposed in the literature[27]. Intel's assembly coded node FFT routines were used to perform local DFTs.

Tables 1 and 2 give the performance results for the Intel Paragon on 256 and 512 processors, respectively. We report in the last column of the Tables the ratio $T_{\text{comm}}/(T_{\text{o}} - T_{\text{comp}})$, which gives the effective speedup in communication due to overlapping. The overall speedup in the execution time is given by the ratio $T_{\text{no}}/T_{\text{o}}$. Further, in Table 1, the timing results in both the interactive mode and dedicated mode are reported. Table 2 reports only the timing results taken in the dedicated mode.* The timing results reported are the averages of the timings obtained by performing the forward FFT followed by the inverse FFT.

The following observations can be made from the timing results:

1. On the 256 processor up to 12%, and on the 512 processor up to 22%, improvement in running time is obtained by using the overlapping scheme.
2. The ratio of execution times (in the dedicated mode) for the nonoverlapped routine to the overlapped routine is within $\pm 6\%$ of the predicted value for $T_{\text{no}}/T_{\text{o}}$ [using (14)†].

7. CONCLUSION

In this paper we have presented a design methodology based on tensor products to develop efficient implementations for block recursive algorithms in which computation is effectively overlapped with communication. Overlapping of computation and communication was achieved by interleaving the computation before and after a data redistribution with the communication for redistribution. This method incurs no extra communication overhead and is suitable for machines with a wormhole routed mesh or a hypercube interconnection network. This overlapping technique can also be used in numerous scientific computations in which block recursive computations are performed along different dimensions of a multidimensional array. This methodology was illustrated by developing an efficient implementation for the FFT on the Intel Paragon. Substantial communication speedups were obtained on these machines by using our overlapping scheme.

ACKNOWLEDGEMENTS

We are grateful to the Center for Advance Computing Research at Caltech for providing access to the Intel Paragon. This work was supported in part by DARPA: order number 7898, monitored by NIST under grant number 60NANB1D1151; and order number 7899, monitored by NIST under grant number 60NANB1D1150. A preliminary version of this

*The programs on 512 nodes could be run only in the dedicated mode.

†For each N, T_{comp} and T_{comm} components of T_{no} were used to compute the effective values of t_c and t_p, respectively. These were then plugged into (14) along with the value for t_s to obtain the expected speedup.

paper appeared in ICPADS 94[28]. We thank the referees for helping us to improve the quality of the paper.

REFERENCES

1. F. P. Preparata and J. Vuillemin, The cube-connected cycles: a versatile network for parallel computation, *Commun. ACM*, **24**(5), 300–309 (1981).
2. J. Granta, M. Conner and R. Tolimieri, Recursive fast algorithms and the role of the tensor product, *IEEE Trans. Signal Process.* **40**(12), 2921–2930 (1992).
3. P. A. Regalia and S. K. Mitra, Kronecker products, unitary matrices and signal processing applications, *SIAM Rev.*, **31**(4), 586–613 (1989).
4. G. X. Ritter and P. D. Gader, Image algebra techniques and parallel image processing, *J. Parallel Distrib. Comput.* **4** 7–44 (1987).
5. J. W. Brewer, 'Kronecker products and matrix calculus in system theory', *IEEE Trans. Circuits Syst.* **25** 772–781 (1978).
6. F. A. Graybill, *Matrices, With Applications in Statistics*, Wadsworth International Group, Belmont, CA, 1983.
7. 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**(4), 450–500 (1990).
8. C. Van Loan, *Computational Frameworks for the Fast Fourier Transform SIAM*, (1992).
9. C.-H. Huang, J. R. Johnson and R. W. Johnson, 'A tensor product formulation of Strassen's matrix multiplication algorithm', *Appl. Math Lett.* **3**(3), 67–71 (1990).
10. C.-H. Huang, J. R. Johnson and R. W. Johnson, Generating parallel programs from tensor product formulas: A case study of Strassen's matrix multiplication algorithm, in *Proc. Int'l Conf. Parallel Processing 1992* Aug. 1992, pp. 104–108.
11. S. D. Kaushik, S. Sharma and C.-H. Huang, An algebraic theory for modeling multistage interconnection networks, *J. Inf. Sci. Eng.* **9**, 1–26 (1993).
12. S. D. Kaushik, S. Sharma, C.-H Huang, J. R. Johnson, R. W. Johnson and P. Sadayappan, An algebraic theory for modelling direct interconnection networks, in *Supercomputing '92*, Nov. 1992, pp. 488–497.
13. S. K. S. Gupta, *Synthesizing communication-efficient distributed-memory parallel programs for block recursive algorithms, Ph.D. thesis*, The Ohio State University, March 1995.
14. S. K. S. Gupta, C.-H Huang, P. Sadayappan and R. W. Johnson, A framework for synthesizing distributed-memory programs for block recursive algorithms, *J. Parallel Distrib. Comput.* **34**(2), 137–153 (1996).
15. A. Wakatani and M. Wolfe, A new approach to array redistribution: Strip mining redistribution, in *Proc. Parallel Architecture and Languages Europe (PARLE '94)*, LNCS v.817, 1984, pp. 323–335.
16. S. L. Johnsson, M. Jacquemin and R. L. Krawitz, 'Communication efficient multi-processor FFT', *J. Comput. Phys.*, **102**(2), 381–397 (1992).
17. P. N. Swarztrauber, 'Multiprocessor FFT's', *Parallel Comput.* **5**, 197–210 (1987).
18. R. C. Agarwal, F. G. Gustavson and M. Zubair, 'An efficient parallel algorithm for the 3-D FFT NAS parallel benchmark', in *Proc. Scalable High Performance Computing Conference*, 1994, pp. 129–133.
19. R. C. Agarwal, F. G. Gustavson and M. Zubair, 'A high-performance parallel algorithm for 1-d FFT', in *Supercomputing '94* 1994, pp. 34–40.
20. A. Graham, *Kronecker Products and Matrix Calculus: With Applications*, Ellis Horwood Limited, 1981.
21. R. A. Horn and C. R. Johnson, *Topics in Matrix Analysis*, Cambridge University Press, Cambridge, 1991.
22. High Performance Fortan Forum, 'High Performance Fortran language specification, version 1.0', *Technical Report CRPC-TR92225*, Rice University, 1993.
23. S. K. S. Gupta, C.-H. Huang, P. Sadayappan and R. W. Johnson, 'Implementing fast Fourier transforms on distributed-memory multiprocessors using data redistributions', *Parallel Processing Letter*, **4**(4), 477–488 (1994).

24. D. Bailey, 'FFTs in external or hierarchical memory', *J. Supercomput.* **4**, 23–35 (1990).
25. S. H. Bokhari, 'Complete exchange on the iPSC', *Technical Report 91-4*, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA, U.S.A., (1991).
26. D. S. Scott, Efficient all-to-all communication patterns in hypercube and mesh topologies, in *IEEE Distributed Memory Computing Conference*, 1991, pp. 398–403.
27. R. Thakur, A. Choudhary and G. Fox, 'Complete exchange on the CM-5 and Touchstone Delta', *J. Supercomput.* **8**, 305–328 (1995).
28. S. K. S. Gupta, C.-H. Huang, R. W. Johnson and P. Sadayappan, Communication-efficient implementation of block-recursive algorithm on distributed-memory machines, in *Proc. Int'l Conf. on Parallel and Distributed Systems*, Dec. 1994, pp. 113–118.

