A Parallel Algorithm for Generating Discrete Orthogonal Polynomials *

Ömer Eğecioğlu

Department of Computer Science,
University of California,
Santa Barbara, CA 93106

Çetin K. Koç

Department of Electrical Engineering
University of Houston
Houston, TX 77204

Abstract

A parallel algorithm that makes use of the classical three-term recursion formula to construct an orthogonal family of polynomials with respect to a discrete inner product is proposed. The algorithm requires $O(N \log N)$ parallel arithmetic steps on a distributed-memory multiprocessor with $N+1$ processors to construct the polynomials $p_i(x)$ for $0 \leq i \leq N$. If hypercube topology is assumed, the algorithm can be implemented with the additional overhead of $O(N \log N)$ routing steps. In this case the implementation is quite simple, requiring only scalar single node broadcast and accumulation procedures together with a Gray code mapping. The limited processor version of the algorithm requires $O(\frac{N^2}{p} + N \log p)$ arithmetic and $O(N \log p)$ routing steps on a hypercube with $p \leq N+1$ nodes. We present some experimental results obtained on an Intel cube.

Key Words: Discrete orthogonal polynomials, parallel algorithms, distributed-memory multiprocessor, hypercube.

*A portion of this work was presented in the Fourth Conference on Hypercube Concurrent Computers and Applications, Monterey, California, March 6–8, 1989.

1 Introduction

We consider generation of polynomials $\{p_0(x), p_1(x), \ldots, p_N(x)\}$ orthogonal with respect to a non-degenerate discrete bilinear form

$$< u(x), v(x) > = \sum_{j=0}^{N} w_j u(x_j) v(x_j) \ , \tag{1}$$

where $p_i(x)$ is a monic polynomial of degree i. In most applications the weights w_j are positive for $0 \leq j \leq N$, and (1) defines an inner-product on polynomials of degree $\leq N$ on the node points $x_0, x_1, \ldots, x_N$. The orthogonal polynomials $p_i(x)$ for $0 \leq i \leq N$ can be generated using the classical *three-term recursion* formula [11]

$$p_{i+1}(x) = (x - \alpha_i) p_i(x) - \beta_i p_{i-1}(x) \ \text{ for } \quad 0 \leq i \leq N-1 \tag{2}$$

with

$$p_{-1}(x) = 0 \ \text{ and } \quad p_0(x) = 1 \tag{3}$$

where α_i and β_i are constants determined as

$$\alpha_i = \frac{< x p_i(x), p_i(x) >}{< p_i(x), p_i(x) >} \ , \quad \beta_i = \frac{< p_i(x), p_i(x) >}{< p_{i-1}(x), p_{i-1}(x) >} \ . \tag{4}$$

Orthogonal polynomials with respect to a discrete bilinear form have applications in rational interpolation, least-squares polynomial approximation, and smoothing of nonlinear functions [4, 3, 9, 7, 1, 2]. In some applications it is preferable to use orthogonal polynomials than to solve a set of linear equations. This is mainly due to numerical stability problems of the underlying linear systems. For example, the normal equations arising from weighted least-squares polynomial approximation problems result in an ill-conditioned system. Thus, it becomes inevitable to use discrete orthogonal polynomials for least-squares polynomial approximation problems [3, 7].

Consider the $(N+1) \times (N+1)$ matrix $P = [P_{ij}]$ of the values of the polynomials $p_i(x)$ at the node points x_j for $0 \leq i, j \leq N$, i.e.,

$$P_{ij} = p_i(x_j) \ \text{ for } \quad 0 \leq i, j \leq N \ . \tag{5}$$

The entries of P can be computed by specializing the three-term recursion (2) by putting $x = x_j$ for $j = 0, 1, \ldots, N$. Furthermore, (2) induces a doubly-indexed recursion on the coefficients of the polynomials $p_i(x)$ for $0 \leq i \leq N$ directly. More precisely, let $A = [A_{ik}]$ be the $(N+1) \times (N+1)$ matrix in which the *ith* row consists of the coefficients of the polynomial $p_i(x)$, i.e.,

$$p_i(x) = \sum_{k=0}^{i} A_{ik} x^k \ . \tag{6}$$

Then A is a lower triangular matrix with unit diagonal whose elements satisfy the recursion

$$A_{i+1,k} = A_{i,k-1} - \alpha_i A_{ik} - \beta_i A_{i-1,k} \text{ for } \quad 0 \le k \le i \le N \tag{7}$$

induced by (2). In (7) we take

$$A_{i,-1} = A_{-1,k} = 0 \text{ for } \quad 0 \le i, k \le N-1 \ . \tag{8}$$

In the generation of the coefficients of the polynomials $p_i(x)$ for $0 \le i \le N$ using the recursion (7), the values of the polynomials $p_i(x)$ at the node points x_j (i.e. P_{ij}) are also required at each step to compute quantities α_i and β_i. Thus it is necessary to generate the values and the coefficients in tandem. This can be done by iterating first the recursion (2) for the values and then the recursion (7) for the coefficients.

The procedure given in Figure 1 (PROCEDURE TTR) computes the entries of matrices P and A using recursions (2) and (7) together with the initial conditions (3) and (8). At the end of PROCEDURE TTR, the coefficients of the polynomials $p_i(x)$ are A_{ik} for $0 \le i, k \le N$, and the values $p_i(x_j)$ are P_{ij} for $0 \le i, j \le N$. Note that $A_{ik} = 0$ for $k > i$.

Lemma 1 *The number of sequential arithmetic steps required by PROCEDURE TTR to compute the entries of matrices P and A is $O(N^2)$.*

Proof In step 1, the computation of γ_0, θ_0, and α_0 requires $2N+1$, N, and 1 arithmetic steps, respectively. In step 2, first we compute the P_{1j}'s using $N+1$ steps. Then γ_1, θ_1, α_1, and β_1 are computed using a total of $5N+5$ arithmetic operations. Similarly, step 3 requires $(N-1)(9N+9)$ arithmetic steps. The computation of A_{ik} in step 4 requires $3(N-1)+4(1+2+\cdots+N-1) = 2N^2+N-5$ arithmetic steps. Thus a total of

$$11N^2 + 10N - 6 = O(N^2) \tag{9}$$

arithmetic steps are required by PROCEDURE TTR . □

2 The Parallel Algorithm

In this section we consider the implementation of PROCEDURE TTR on a message-passing multiprocessor. In this architecture, each processor has its own local physical memory and a point-to-point interprocessor communication network provides a mechanism for communication between processors. The first generation message-passing multiprocessors adopt a store-and-forward communication mechanism and most commonly a hypercube topology. Second generation multiprocessors have more advanced communication mechanisms utilizing crossbar switches at the nodes and the system can be assumed to be fully connected for most practical purposes.

Since the communication overhead has a great impact on the performance of an algorithm, the communication pattern of the parallel algorithm should be carefully designed to reduce the communication complexity [5, 6, 10]. We give an efficient implementation of PROCEDURE TTR on the hypercube multiprocessor. This implementation makes use of hypercube spanning trees for data accumulation and broadcast, and uses a Gray code mapping to provide neighboring virtual labels for frequently communicating processors.

Following the widely accepted nomenclature, we assume that in the hypercube multiprocessor a node can send a data item to one of its d neighbors by issuing SEND $(X, target\_node)$ where X is the item and *target_node* is the node to which the data is being sent. The target node receives the data by executing RECEIVE (X). A SEND/RECEIVE pair constitutes a routing step. We will record the number of parallel routing steps for the algorithms presented. A more detailed analysis of the communication requirements of the presented algorithms is in section 4.

To implement PROCEDURE TTR on a hypercube with $2^d = N + 1$ nodes we use the following two procedures:

1. PROCEDURE BROADCAST (X) broadcasts data item X, which is initially located in node $2^d - 1$, to all nodes. This algorithm makes use of a hypercube spanning tree [5, 10] rooted at node $2^d - 1$. The spanning tree of a 3-cube rooted at node 7 is illustrated in Figure 2.

2. PROCEDURE ACCUMULATE (X_j) computes the sum $S = \sum_{j=0}^{2^d-1} X_j$ where X_j is a data item initially located in node j for $0 \leq j \leq 2^d - 1$. After the execution of PROCEDURE ACCUMULATE(X_j), the sum S can be found in node $2^d - 1$. This procedure also uses the spanning tree of the cube rooted at node $2^d - 1$.

The procedures BROADCAST and ACCUMULATE are given in Figure 3. An inspection shows that PROCEDURE BROADCAST uses d parallel routing steps to complete if X is a single floating-point number. Similarly PROCEDURE ACCUMULATE takes d parallel routing and d parallel floating-point addition steps.

These routines allow us to implement steps 1,2, and 3 of PROCEDURE TTR on the hypercube. The implementation of step 4 is a little more subtle and requires a *Gray code*, G, of the *node_id*'s of the processors. For our purposes, it suffices to take G as a permutation of the set of *node_id*'s $\{0, 1, 2, \ldots, 2^d - 1\}$ such that the Hamming distance between $G(k)$ and $G(k+1)$ is 1 for $0 \leq k \leq 2^d - 2$ [8].

In the implementation of PROCEDURE TTR on the hypercube, we perform the computation of the *kth* column of matrix A in the processor whose *node_id* is $G(k)$. Hence $A_{i+1,k}$ is computed at

processor $G(k)$ for $i = 0, 1, \ldots, N-1$. Note that the computation of $A_{i+1,k}$ requires quantities $A_{i,k-1}$, A_{ik}, and $A_{i-1,k}$ as well as α_i and β_i. After the values of the polynomials P_{ij} are computed, the vectors α and β are readily available at every node. The terms A_{ik} and $A_{i-1,k}$ themselves are located in node $G(k)$. The only term that is missing from the node $G(k)$ for the computation of $A_{i+1,k}$ is $A_{i,k-1}$. Since this element is located in processor $G(k-1)$, which is one of the neighbors of processor $G(k)$, we have fast access to this element. Thus by making use of the Gray code, we make sure that adjacent columns of the A matrix are computed by neighboring nodes in the hypercube.

In the implementation of PROCEDURE TTR on the hypercube, we assume for now that $N+1 = 2^d$ and for $0 \leq j \leq N$, processor j initially contains the initial data $x_0, x_1, \ldots, x_N$ together with w_j. PROCEDURE TTR_CUBE is given in Figure 4.

Theorem 1 *PROCEDURE TTR_CUBE computes the entries of matrices P and A using $O(N \log N)$ parallel arithmetic operations and $O(N \log N)$ routing steps on a hypercube with $N+1$ nodes.*

Proof By counting the number of arithmetic steps required for the execution of ACCUMULATE and BROADCAST, and all of the remaining steps involved in PROCEDURE TTR_CUBE, we find the number of arithmetic operations as

$$2N \log(N+1) + 13N + 2 \log(N+1) - 5 = O(N \log N) \ . \tag{10}$$

Similarly in PROCEDURE TTR_CUBE, the subprocedures ACCUMULATE and BROADCAST are called $4 + 2(N-1)$ and $3 + 2(N-1)$ times, respectively. Also, for the computation of $A_{i+1,k}$, we execute SEND/RECEIVE pairs for $N-1$ times. Since each such call to PROCEDURE ACCUMULATE and BROADCAST incurs $d = \log(N+1)$ routing steps, the total number of routing steps is

$$4N \log(N+1) + N + 3 \log(N+1) - 1 = O(N \log N) \ . \tag{11}$$

Thus the total number of arithmetic and routing steps required by PROCEDURE TTR does not exceed $O(N \log N)$. □

In the above analysis we have not considered the initial loading of the cube, i.e., the loading of the data from the host processors to all nodes of the hypercube. This can be achieved by first sending the data x_i, w_i for $i = 0, 1, \ldots, N$ from the host processor to a particular node of the cube, for example to node 0. This will take $2(N+1)$ routing steps. Node 0 then proceeds to broadcast the data to all nodes using PROCEDURE BROADCAST. This step takes an additional $2(N+1)d$ routing steps. Thus, the initial loading of the cube also requires $O(N \log N)$ routing operations, and does not increase the order of the running time of PROCEDURE TTR_CUBE.

3 Partitioning Large Problems

Here we analyze the more realistic case in which the number of processors p available on the hypercube does not match the size of the input, i.e., $p \neq N+1$. The most interesting situation is when $p < N+1$, since there should be no difficulty when $p > N+1$ nodes are available. One can simply use a subset of these nodes to implement PROCEDURE TTR_CUBE. Thus, we consider the case $p < N+1$. For simplicity assume that p divides $N+1$, i.e., $pm = N+1$ for some $m > 1$. We partition the matrix P in a very simple manner: the first m columns are computed at node 0, the second m columns at node 1, etc. The partitioning of A is similar: first m columns are computed at processor $G(0)$, the second at processor $G(1)$, the third at node $G(2)$, and so on.

This partitioning scheme allows us to compute the values and the coefficients of the orthogonal polynomials in an efficient manner.

Theorem 2 *PROCEDURE TTR_CUBE computes the entries of the matrices P and A using $O(\frac{N^2}{p} + N \log p)$ parallel arithmetic and $O(\frac{N^2}{p} + N \log p)$ routing steps on a hypercube with $p < N+1$ processors.*

Proof First we consider the limited processor implementation of PROCEDURE ACCUMULATE and PROCEDURE BROADCAST. We partition $N+1$ elements to be summed such that each node contains m elements where $pm = N+1$. First we perform sequential summation at each node simultaneously which will take $m-1$ arithmetic steps to complete. Then we use a binary tree addition procedure to sum these sum blocks of each node to find the total sum. This step will take $\log p$ arithmetic and $\log p$ routing steps. Thus, PROCEDURE ACCUMULATE takes $m - 1 + \log p$ arithmetic operations and $\log p$ routing operations to find the sum of $N+1$ elements distributed on p nodes.

Now we consider limited processor implementation of PROCEDURE BROADCAST, and its use in PROCEDURE TTR_CUBE. We note at each step in PROCEDURE TTR_CUBE, an element (either α_i or β_i) is broadcast to all the nodes on the cube. Since we have only $p < N+1$ processors available, this procedure will take $\log p$ routing steps to complete. For example, we compute $P_{1j} = x_j - \alpha_0$ at processor q for $j = qm + 0, qm + 1, \ldots, qm + m - 1$. Thus for the computation of P_{1j} for $qm < j < qm + m - 1$ we need to send α_0 from the node $p-1$ (at which α_0 is initially computed using PROCEDURE ACCUMULATE) to all nodes $q = 0, 1, \ldots, p-2$ which will take $\log p$ routing steps using the spanning tree in Figure 2.

Thus for the computation of the entries of the matrix P we count the number of arithmetic and routing operations, and find $9Nm + 2N \log p + 2 \log p - 1$ and $4N \log p + 3 \log p$, respectively.

Note that we have partitioned the matrix A such that the first m columns are computed at the processor $G(0)$, the second group is computed at the node $G(1)$, and so on. Thus the element $A_{i,qm+k}$

is computed at the node $G(q)$ for $0 \leq k \leq m-1$ and for $i = 0, 1, \ldots, N$.

The analysis can be greatly simplified by considering only those operations which processor $G(0)$ has to perform. Since the elements $A_{i,0}, A_{i,1}, \ldots, A_{i,m-1}$ are computed at the node $G(0)$, there is no need to perform SEND/RECEIVE operations for $i = 0, 1, \ldots, m-1$. For $i = m, m+1, \ldots, N$, the processor $G(q)$ will send $A_{i,qm+m-1}$ to processor $G(q+1)$ for the computation of $A_{i+1,(q+1)m}$ which is the first element to be computed at this processor. It follows from this observation that the computation of the entries of A will take $N - m$ parallel routing steps.

Also at the first step processor $G(0)$ computes quantities $A_{i,0}, A_{i,1}, \ldots, A_{i,m-1}$ for $i = 0, 1, \ldots, m-1$ except for A_{10} and A_{ii}, $0 \leq i \leq m-1$. Each of these operations takes 4 arithmetic steps in the light of (7). The total number of arithmetic operations for this step becomes $4(1 + 2 + \cdots + m) - 4 - 4m = 2m^2 - 2m + 4$. The elements that follow, $A_{i,k}$ for $m \leq i \leq N$ and $0 \leq k \leq m-1$, are computed using locally available data. Therefore this step takes $(N + 1 - m)4m$ arithmetic steps.

Thus we observe that the total number of arithmetic and routing operations for the computation of the entries A becomes $4Nm - 2m^2 + 2m - 4$ and $N - m$, respectively.

Hence, PROCEDURE TTR_CUBE takes

$$13Nm + 2N\log p + 2\log p + 2m - 2m^2 - 5 \quad = \quad O(\frac{N^2}{p} + N\log p) \tag{12}$$

parallel arithmetic steps, and

$$4N\log p + N + 3\log p - m \quad = \quad O(N\log p) \tag{13}$$

parallel routing steps to compute the entries of matrices P and A on a hypercube with $N+1$ nodes. □

Note that the routing steps given by (13) do not take into account the initial loading of the data, which requires an additional $2(N+1) + 2(N+1)\log p$ routing operations.

4 Efficiency Analysis

We define τ_{comp} as the time required to perform a floating-point operation, and τ_{comm} the time required to transfer a floating-point number to a neighboring node for the system under consideration. Using equations (9), (10), and (11) and taking the initial loading step into account, we find the sequential and the parallel time required by PROCEDURE TTR in terms of the above parameters as

$$T_{seq} \quad = \quad \left(11N^2 + 10N - 6\right)\tau_{comp} \ , \tag{14}$$

$$\begin{aligned} T_{par} \quad = \quad & \left(13Nm + 2N\log p + 2\log p + 2m - 2m^2 - 5\right)\tau_{comp} \\ & + (6N\log p + 3N + 5\log p - m + 2)\,\tau_{comm} \ , \end{aligned} \tag{15}$$

where $m = (N+1)/p$. The efficiency of an implementation is a function of the input size $N+1$, the number of processors p, and also the parameter $\tau = \tau_{comm}/\tau_{comp}$. It is well known that this parameter is very crucial in evaluating the performance of multiprocessor systems. Usually $\tau \gg 1$, e.g. our experiments indicated that $\tau > 25$ on the first generation Intel hypercube. Figure 5 shows the efficiency of the parallel algorithm presented as a function of $N+1$ for $\tau = 1, 5, 10, 50$ and for $p = 8$. Thus we see that the efficiency of our particular implementation of PROCEDURE TTR on the cube approaches 0.80 when τ is close to 1 and $N \gg p$.

We have implemented PROCEDURE TTR on a first generation Intel cube with 8 nodes (Intel iPSC/d3 hypercube running XENIX 286 R3.4 and iPSC Software R3.1) and also performed experiments, similar to those mentioned in [6], to measure τ_{comp} and τ_{comm}. The experiments indicated that $\tau_{comp} \approx 0.058$ milliseconds (if the floating-point operation is taken to be multiplication, addition, or subtraction) and $\tau_{comm} \approx 1.48$ milliseconds, which implies that $\tau \approx 25.5$. The first generation Intel cube uses the store-and-forward communication scheme which is very slow. This seems to be the fundamental reason for performance degradation. The timing results are shown in Table 1 for values of $8 \leq N+1 \leq 72$. Using formulae (14) and (15) we also tabulate the estimated efficiency of PROCEDURE TTR. The table shows that our analysis of efficiency is rather conservative; on a real machine, one can obtain higher speedup and efficiency than the estimation suggests.

5 Conclusions

We have presented a sequential algorithm and its parallel version to construct an orthogonal family of polynomials with respect to a given discrete inner product. The algorithm makes use of the classical three-term recursion formula to generate both the coefficients (matrix P) and the values (matrix A) of the orthogonal polynomials $p_i(x)$ for $0 \leq i \leq N$.

The sequential version requires $O(N^2)$ arithmetic operations to construct the matrices P and A. The parallel version of the algorithm requires $O(N \log N)$ parallel arithmetic steps on a distributed-memory multiprocessor with $N+1$ processors. If hypercube topology is assumed, then the algorithm can be implemented with an additional overhead of $O(N \log N)$ routing steps. Thus in effect, the parallel algorithm in this case constructs each orthogonal polynomial using $O(\log N)$ parallel arithmetic and $O(\log N)$ communication steps. The implementation for the hypercube is quite simple, requiring only scalar single node broadcast and accumulation procedures together with a standard Gray code mapping for efficient communication.

A straightforward partitioning scheme of the input allows for the implementation of the algorithm in the limited processor case. The limited processor version of the algorithm is shown to require

$O(\frac{N^2}{p} + N \log p)$ arithmetic and $O(N \log p)$ routing steps on a hypercube with $p \leq N + 1$ nodes.

Finally, experimental results obtained on a first generation Intel cube with 8 nodes indicate better performance parameters of the limited processor version of the parallel algorithm then our theoretical estimates suggest.

References

[1] Ö. Eğecioğlu and Ç. K. Koç. A fast algorithm for rational interpolation via orthogonal polynomials. *Mathematics of Computation*, 53(187):249–264, July 1989.

[2] Ö. Eğecioğlu and Ç. K. Koç. Parallel rational interpolation. *Intern. J. Computer Math.*, 32:217–231, 1990.

[3] G. E. Forsythe. Generation and use of orthogonal polynomials for data-fitting with a digital computer. *Journal of SIAM*, 5(2):74–88, 1957.

[4] F. B. Hildebrand. *Introduction to Numerical Analysis.* McGraw-Hill, 1956.

[5] S. L. Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *Journal of Parallel and Distributed Computing*, 4:133–172, 1987.

[6] O. A. McBryan and E. F. Van de Velde. Hypercube algorithms and implementations. *SIAM Journal on Scientific and Statistical Computing*, 8(2):s227–s227, March 1987.

[7] A. Ralston and P. Rabinowitz. *A First Course in Numerical Analysis.* McGraw-Hill, 1985.

[8] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms.* Prentice-Hall, 1977.

[9] T. J. Rivlin. *An Introduction to the Approximation of Functions.* Dover Publications, Inc., 1969.

[10] Y. Saad and M. H. Schultz. Topological properties of hypercubes. *IEEE Transactions on Computers*, 37(7):867–872, July 1988.

[11] G. Szego. *Orthogonal Polynomials.* American Mathematical Society, 1959.

Figure 1. Computation of P_{ij} and A_{ik} for $0 \le i, j, k \le N$.

PROCEDURE TTR

Input: x_j, w_j for $0 \le j \le N$

Output: P_{ij} and A_{ik} for $0 \le i, j, k \le N$

Step 1. Set $P_{0j} = 1$ for $0 \le j \le N$ and $\beta_0 = 0$, and compute

$$\gamma_0 = \sum_{j=0}^{N} w_j x_j \ , \quad \theta_0 = \sum_{j=0}^{N} w_j \ , \quad \text{and} \quad \alpha_0 = \frac{\gamma_0}{\theta_0} \ .$$

Step 2. Set $P_{1j} = x_j - \alpha_0$ for $0 \le j \le N$, and compute

$$\gamma_1 = \sum_{j=0}^{N} w_j x_j P_{1j}^2 \ , \quad \theta_1 = \sum_{j=0}^{N} w_j P_{1j}^2 \ ,$$

$$\alpha_1 = \frac{\gamma_1}{\theta_1} \ , \quad \beta_1 = \frac{\theta_1}{\theta_0} \ .$$

Step 3. For $1 \le i \le N-1$ compute

$$P_{i+1,j} = (x_j - \alpha_i) P_{ij} - \beta_i P_{i-1,j} \ \text{ for } \ 0 \le j \le N$$

$$\gamma_{i+1} = \sum_{j=0}^{N} w_j x_j P_{i+1,j}^2 \ , \quad \theta_{i+1} = \sum_{j=0}^{N} w_j P_{i+1,j}^2 \ ,$$

$$\alpha_{i+1} = \frac{\gamma_{i+1}}{\theta_{i+1}} \ , \quad \beta_{i+1} = \frac{\theta_{i+1}}{\theta_i} \ .$$

Step 4. Set $A_{ii} = 1$ for $0 \le i \le N$, and $A_{ik} = 0$ for $0 \le i < k \le N$. Set $A_{10} = -\alpha_0$. For all $1 \le k < i \le N-1$ compute

$$A_{i+1,k} = A_{i,k-1} - \alpha_i A_{ik} - \beta_i A_{i-1,k}$$

Figure 2. A hypercube spanning tree rooted at node $(111)_2$.

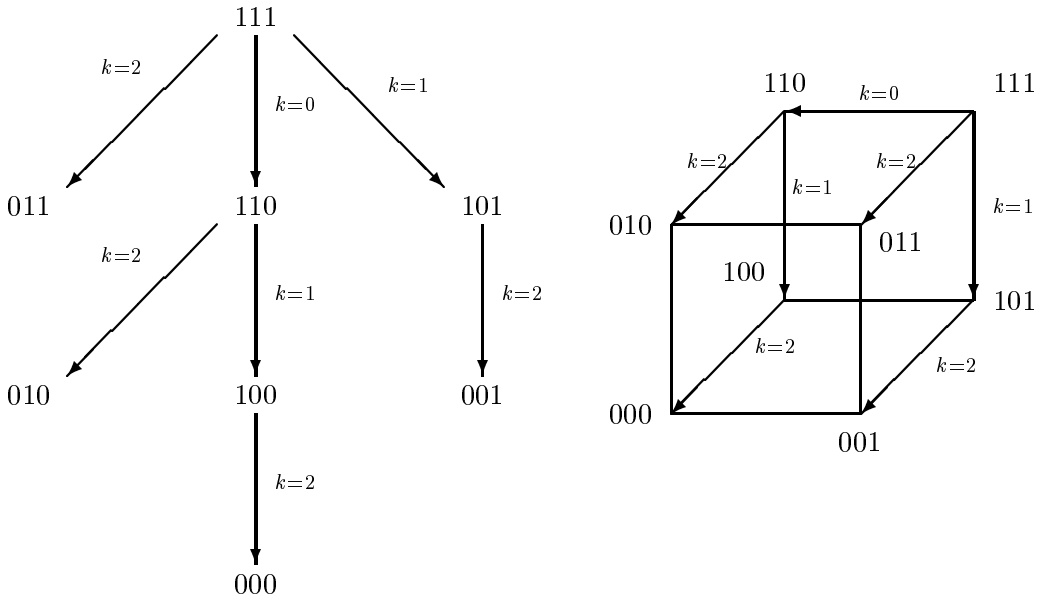

Figure 3. The procedures for broadcasting and accumulation.

PROCEDURE BROADCAST(X)

INPUT: X at node $2^d - 1$

OUTPUT: X at all nodes

$j = node\_id$

FOR $k = 0$ TO $d - 1$ DO

BEGIN

IF $j \geq 2^d - 2^k$ THEN

SEND$(X, j - 2^k)$

IF $2^d - 2^k - 1 \geq j \geq 2^d - 2^{k+1}$ THEN

RECEIVE(X)

END FOR

END BROADCAST

PROCEDURE ACCUMULATE(X_j)

INPUT: X_j at node j for $0 \leq j \leq 2^d - 1$

OUTPUT: $\sum_{j=0}^{2^d-1} X_j$ at node $2^d - 1$

$j = node\_id$

FOR $k = d - 1$ TO 0 DO

BEGIN

IF $2^d - 2^k - 1 \geq j \geq 2^d - 2^{k+1}$ THEN

SEND$(X, j + 2^k)$

IF $j \geq 2^d - 2^k$ THEN

RECEIVE$(TempX)$

$X = X + TempX$

END FOR

END ACCUMULATE

Figure 4. Computation of the entries of P and A on a hypercube.

PROCEDURE TTR_CUBE
INPUT: $x_0, \ldots, x_N$ and w_j at node j
OUTPUT: P_{ij} at node j, and A_{ik} at node $G(k)$ for $0 \leq i \leq N$.
$j = node\_id$
γ_0 =ACCUMULATE$(w_j x_j)$; θ_0 =ACCUMULATE(w_j)
IF $j = 2^d - 1$ THEN $\alpha_0 = \dfrac{\gamma_0}{\theta_0}$
BROADCAST(α_0)
$P_{1j} = x_j - \alpha_0$
γ_1 =ACCUMULATE$(w_j x_j P_{1j}^2)$; θ_1 =ACCUMULATE$(w_j P_{1j}^2)$
IF $j = 2^d - 1$ THEN $\alpha_1 = \dfrac{\gamma_1}{\theta_1}$, $\beta_1 = \dfrac{\theta_1}{\theta_0}$
BROADCAST(α_1) ; BROADCAST(β_1)
FOR $i = 1$ TO $N - 1$ DO
BEGIN
$P_{i+1,j} = (x_j - \alpha_i)P_{ij} - \beta_i P_{i-1,j}$
γ_{i+1} =ACCUMULATE$(w_j x_j P_{i+1,j}^2)$; θ_{i+1} =ACCUMULATE$(w_j P_{i+1,j}^2)$
IF $j = 2^d - 1$ THEN $\alpha_{i+1} = \dfrac{\gamma_{i+1}}{\theta_{i+1}}$, $\beta_{i+1} = \dfrac{\theta_{i+1}}{\theta_i}$
BROADCAST(α_{i+1}) ; BROADCAST(β_{i+1})
END FOR
$k = G^{-1}(node\_id)$
IF $k = 0$ THEN $A_{00} = 1$
IF $k > 0$ THEN $A_{0k} = 0$
IF $k = 0$ THEN $A_{10} = -\alpha_0$; $A_{11} = 1$
IF $k > 1$ THEN $A_{1k} = 0$
FOR $i = 1$ TO $N - 1$ DO
BEGIN
IF $k = i$ THEN $A_{ik} = 1$
IF $k > i$ THEN $A_{ik} = 0$
IF $k = 0$ THEN $A_{i+1,0} = -\alpha_i A_{i0} - \beta_i A_{i-1,0}$
IF $0 \leq k \leq i - 1$ THEN SEND$(A_{ik}, G(k+1))$
IF $1 \leq k \leq i$ THEN
BEGIN
RECEIVE$(temp_A_{i,k-1})$
$A_{i+1,k} = temp\_A_{i,k-1} - \alpha_i A_{ik} - \beta_i A_{i-1,k}$
END IF
END FOR
END TTR_CUBE

Figure 5. Efficiency of PROCEDURE TTR.

Table 1. Time and efficiency of PROCEDURE TTR on an Intel 3-cube

	Time (ms)		Efficiency	
N+1	$p = 1$	$p = 8$	Measured	Estimated
8	45	160	0.035	0.018
16	185	455	0.051	0.037
24	415	655	0.079	0.055
32	740	920	0.101	0.073
40	1160	1400	0.104	0.089
48	1675	1180	0.177	0.105
56	2285	1535	0.186	0.121
64	2995	1775	0.211	0.136
72	3815	2035	0.234	0.150