SIAM J. SCI. COMPUT.
Vol. 23, No. 1, pp. 271–303
© 2001 Society for Industrial and Applied Mathematics

# ON THE EFFICIENT PARALLEL COMPUTATION OF LEGENDRE TRANSFORMS*

MÁRCIA A. INDA†, ROB H. BISSELING‡, AND DAVID K. MASLEN§

**Abstract.** In this article, we discuss a parallel implementation of efficient algorithms for computation of Legendre polynomial transforms and other orthogonal polynomial transforms. We develop an approach to the Driscoll–Healy algorithm using polynomial arithmetic and present experimental results on the accuracy, efficiency, and scalability of our implementation. The algorithms were implemented in ANSI C using the BSPlib communications library. We also present a new algorithm for computing the cosine transform of two vectors at the same time.

**Key words.** orthogonal polynomials, Legendre polynomials, BSP, parallel computation, computational harmonic analysis

**AMS subject classifications.** 65T50, 65Y05, 42C15

**PII.** S1064827599355864

## 1. Introduction.

Discrete Legendre transforms (DLTs) are widely used tools in applied science, commonly arising in problems associated with spherical geometries. Examples of their application include spectral methods for the solution of partial differential equations, e.g., in global weather forecasting [3, 9], shape analysis of molecular surfaces [16], statistical analysis of directional data [18], and geometric quality assurance [17].

A direct method for computing a discrete orthogonal polynomial transform such as the DLT transform for $N$ data values requires a matrix-vector multiplication of $O(N^2)$ arithmetic operations, though several authors [2, 28] have proposed faster algorithms based on approximate methods. In 1989, Driscoll and Healy introduced an exact algorithm that computes such transforms in $O(N \log^2 N)$ arithmetic operations [13, 14, 15]. They implemented the algorithm and analyzed its stability, which depends on the specific orthogonal polynomial sequence used.

Discrete polynomial transforms are computationally intensive, so for large problem sizes the ability to use multiprocessor computers is important, and at least two papers discussing the theoretical parallelizability of the algorithm have already been written [19, 32]. We are, however, unaware of any parallel implementation of the Driscoll–Healy algorithm at the time of writing.

In this paper, we derive a new parallel algorithm that has a lower theoretical time complexity than those of [19, 32], and we present a full implementation of this

*Received by the editors May 12, 1999; accepted for publication (in revised form) February 9, 2001; published electronically June 19, 2001. Parts of this work appeared in preliminary form in *Proceedings of the ECMWF Workshop "Towards TeraComputing—The Use of Parallel Processors in Meteorology,"* W. Zwieflhofer and N. Kreitz, eds., World Scientific, River Edge, NJ, 1999, pp. 87–108. Computer time on the Cray T3E was provided by HPaC, Delft with funding by NCF, The Netherlands. Computer time on the IBM SP was provided by SARA, Amsterdam.

http://www.siam.org/journals/sisc/23-1/35586.html

†FOM Institute for Atomic and Molecular Physics (AMOLF), Kruislaan 407, 1098 SJ Amsterdam, The Netherlands (inda@amolf.nl) and Instituto de Matemática, Universidade Federal do Rio Grande do Sul, Av. Bento Gonçalves 9500, 91509-900 Porto Alegre, RS, Brazil. The work of this author was supported by a doctoral fellowship from CAPES, Brazil.

‡Department of Mathematics, Utrecht University, P.O. Box 80010, 3508 TA Utrecht, The Netherlands (Rob.Bisseling@math.uu.nl).

§Susquehanna Partners, G.P., 401 City Avenue, Suite 220, Bala Cynwyd, PA 19004 (david@maslen.net).

algorithm. Another contribution is the method used to derive the algorithm. We present a method based on polynomial arithmetic to clarify the properties of orthogonal polynomials used by the algorithm and to remove some unnecessary assumptions made in [13, 14, 15].

The remainder of this paper is organized as follows. In section 2, we describe some important properties of orthogonal polynomials and orthogonal polynomial transforms, and we present a derivation of the Driscoll–Healy algorithm. In section 3, we introduce the bulk synchronous parallel (BSP) model and describe a basic parallel algorithm and its implementation. In section 4, we refine the basic algorithm by introducing an intermediate data distribution that reduces the communication to a minimum. In section 5, we present results on the accuracy, efficiency, and scalability of our implementation. We conclude with section 6 and two appendices describing a generalization of the algorithm and the precomputation of the data needed by the algorithm.

## 2. The Driscoll–Healy algorithm.

**2.1. Orthogonal polynomials.** A sequence of polynomials $p_0, p_1, p_2, \ldots$ is said to be an *orthogonal polynomial sequence* on the interval $[-1, 1]$ with respect to the weight function $\omega(x)$, if $\deg p_i = i$, and

$$\int_{-1}^{1} p_i(x)p_j(x)\omega(x)dx = 0 \quad \text{for } i \neq j,$$
$$\int_{-1}^{1} p_i(x)^2\omega(x)dx \neq 0 \quad \text{for } i \geq 0.$$

The weight function $\omega(x)$ is nonnegative and integrable on $(-1, 1)$.

Let $x_0, \ldots, x_{N-1}$ be a sequence of distinct real numbers called *sample points*, and let $f_0, \ldots, f_{N-1}$ be a sequence of real values. Then there exists a unique polynomial $f$ of degree less than $N$ such that

$$f(x_j) = f_j, \quad j = 0, \ldots, N-1. \tag{2.1}$$

This polynomial can be obtained by Lagrangian interpolation.

The *expansion transform* corresponding to the orthogonal polynomial sequence $\{p_k\}$ computes the coefficients $c_k$ in the expansion

$$f = \sum_{k=0}^{N-1} c_k p_k, \tag{2.2}$$

where $f$ is a polynomial given by function values $f_j$ in the sample points $x_j$. (Note that we do not require any special relation between the sample points and the orthogonal polynomials.) The *inverse expansion transform* evaluates $f$ at the sample points $x_j$, and this can be done by straightforward substitution:

$$f_j = \sum_{k=0}^{N-1} c_k p_k(x_j), \quad j = 0, \ldots, N-1. \tag{2.3}$$

In matrix-vector notation, the latter transform can be written as $\mathbf{f} = \mathbf{P}\mathbf{c}$, where $\mathbf{f} = (f_0, \ldots, f_{N-1})$ and $\mathbf{c} = (c_0, \ldots, c_{N-1})$ are column vectors, and the matrix $\mathbf{P}$ is defined by $P_{jk} = p_k(x_j)$. The matrix $\mathbf{P}$ is invertible, and $\mathbf{P}^{-1}$ represents the

expansion transform. In general, $\mathbf{P}$ need not be orthogonal, and hence its inverse and transpose need not be the same.

EXAMPLE 2.1 (discrete Chebyshev transform (DChT)). *The Chebyshev polynomials of the first kind are the sequence of orthogonal polynomials defined recursively by*

$$T_{k+1}(x) = 2x \cdot T_k(x) - T_{k-1}(x), \quad T_0(x) = 1, \quad T_1(x) = x. \tag{2.4}$$

*These are orthogonal with respect to the weight function* $\omega(x) = \pi^{-1}(1-x^2)^{-\frac{1}{2}}$ *on* $[-1,1]$*, and they satisfy* $T_k(\cos\theta) = \cos k\theta$ *for all real* $\theta$*.*

*The DChT is the expansion transform for the Chebyshev polynomials at the Chebyshev points. The* Chebyshev points *are the roots of* $T_N$*, and they are given by*

$$x_j^N = \cos\frac{(2j+1)\pi}{2N}, \quad j = 0, \ldots, N-1. \tag{2.5}$$

*We denote the Chebyshev transform by a tilde. More specifically, the coefficient of* $T_k$ *in the Chebyshev expansion of a polynomial* $f$ *is denoted by* $\tilde{f}_k$*.*

*The inverse Chebyshev transform can straightforwardly be written as*

$$f_j = \sum_{k=0}^{N-1} \tilde{f}_k T_k(x_j^N) = \sum_{k=0}^{N-1} \tilde{f}_k \cos\frac{(2j+1)k\pi}{2N}, \quad j = 0, \ldots, N-1. \tag{2.6}$$

*Furthermore, it can be shown that the Chebyshev transform itself is given by*

$$\tilde{f}_k = \frac{\epsilon_k}{N} \sum_{j=0}^{N-1} f_j T_k(x_j^N) = \frac{\epsilon_k}{N} \sum_{j=0}^{N-1} f_j \cos\frac{(2j+1)k\pi}{2N}, \quad k = 0, \ldots, N-1, \tag{2.7}$$

*where*

$$\epsilon_k = \begin{cases} 1 & \text{if } k = 0, \\ 2 & \text{if } k > 0. \end{cases} \tag{2.8}$$

In this work, we will study a slightly more general transform which includes weights. Given an orthogonal polynomial sequence $\{p_k\}$, a sequence of sample points $x_0, \ldots, x_{N-1}$, and a sequence of numbers $w_0, \ldots, w_{N-1}$ called *sample weights*, we define the *discrete orthogonal polynomial transform* of a data vector $(f_0, \ldots, f_{N-1})$ to be the vector of sums $(\hat{f}_0, \ldots, \hat{f}_{N-1})$, where

$$\hat{f}_k = \hat{f}(p_k) = \sum_{j=0}^{N-1} w_j f_j p_k(x_j). \tag{2.9}$$

The matrix of the discrete orthogonal polynomial transform (2.9) for the special case with sample weights 1 is $\mathbf{P}^T$.

EXAMPLE 2.2 (discrete cosine transform (DCT)). *The DCT, or DCT-*II *in the terminology of* [35]*, is the discrete orthogonal polynomial transform for the Chebyshev polynomials, with sample weights* 1 *and with the Chebyshev points as sample points. Thus, the matrix representing the DCT is* $\mathbf{P}^T$*. Since the matrix representing the DChT is* $\mathbf{P}^{-1}$*, the DCT is the inverse transpose of the DChT. The relation is even closer: by comparing* (2.9) *for the DCT with* (2.7) *for the DChT we see that the DChT is equivalent to a DCT followed by a multiplication of the* $k$*th coefficient by* $\frac{\epsilon_k}{N}$*.*

*A DCT can be carried out in $O(N \log N)$ arithmetic operations using a fast Fourier transform (FFT)* [1, 35] *or using the recent algorithm of Steidl and Tasche* [33]. *Such an $O(N \log N)$ algorithm is called a fast cosine transform (FCT). This also provides us with a fast Chebyshev transform (FChT). We use an upper bound of the form $\alpha N \log_2 N + \beta N$ for the number of floating point operations (flops) for one FChT of size $N$, or its inverse. The lower order term is included because we are often interested in small size transforms, for which this term may be dominant.*

EXAMPLE 2.3 (DLT). *The Legendre polynomials are orthogonal with respect to the uniform weight function 1 on $[-1, 1]$, and they may be defined recursively by*

$$P_{k+1}(x) = \frac{2k+1}{k+1} x \cdot P_k(x) - \frac{k}{k+1} P_{k-1}(x), \quad P_0(x) = 1, \quad P_1(x) = x. \tag{2.10}$$

*The Legendre polynomials are one of the most important examples of orthogonal polynomials, as they occur as zonal polynomials in the spherical harmonic expansion of functions on the sphere. Our parallel implementation of the Driscoll–Healy algorithm, to be described later, focuses on the case of Legendre polynomials. For efficiency reasons, we sample these polynomials at the Chebyshev points. In this paper, we call the discrete orthogonal polynomial transform for the Legendre polynomials, with sample weights $1/N$ and with the Chebyshev points as sample points, the DLT.*

One of the important properties of orthogonal polynomials we will use is the following lemma.

LEMMA 2.4 (Gaussian quadrature). *Let $\{p_k\}$ be an orthogonal polynomial sequence for a nonnegative integrable weight function $\omega(x)$, and let $z_0^N, \ldots, z_{N-1}^N$ be the roots of $p_N$. Then there exist numbers $w_0^N, \ldots, w_{N-1}^N > 0$, such that for any polynomial $f$ of degree less than $2N$ we have*

$$\int_{-1}^{1} f(x)\omega(x)dx = \sum_{j=0}^{N-1} w_j^N f(z_j^N).$$

*The numbers $w_j^N$ are unique and are called the Gaussian weights for the sequence $\{p_k\}$.*

*Proof.* See, e.g., [10, Theorem 6.1]. □

EXAMPLE 2.5. *The Gaussian weights for the Chebyshev polynomials with weight function $\pi^{-1}(1-x^2)^{-\frac{1}{2}}$ are $w_j^N = 1/N$. So for any polynomial $f$ of degree less than $2N$ we have*

$$\frac{1}{\pi} \int_{-1}^{1} \frac{f(x)dx}{\sqrt{1-x^2}} = \frac{1}{N} \sum_{j=0}^{N-1} f(x_j^N), \tag{2.11}$$

*where $x_j^N = \cos \frac{(2j+1)\pi}{2N}$ are the Chebyshev points.*

Another property of orthogonal polynomials that we will need is the existence of a three-term recurrence relation, such as (2.4) for the Chebyshev polynomials and (2.10) for the Legendre polynomials.

LEMMA 2.6 (three-term recurrence). *Let $\{p_k\}$ be an orthogonal polynomial sequence for a nonnegative integrable weight function. Then $\{p_k\}$ satisfies a three-term recurrence relation*

$$p_{k+1}(x) = (A_k x + B_k)p_k(x) + C_k p_{k-1}(x), \tag{2.12}$$

*where $A_k, B_k, C_k$ are real numbers with $A_k \neq 0$ and $C_k \neq 0$.*

*Proof.* See, e.g., [10, Theorem 4.1]. □

The Clebsch–Gordan property follows from, and is similar to, the three-term recurrence.

COROLLARY 2.7 (Clebsch–Gordan). *Let $\{p_k\}$ be an orthogonal polynomial sequence with a nonnegative integrable weight function. Then for any polynomial $Q$ of degree $m$ we have*

$$p_k \cdot Q \in \operatorname{span}_{\mathbf{R}}\{p_{k-m}, \ldots, p_{k+m}\}.$$

*Proof.* Rewrite the recurrence (2.12) in the form $x \cdot p_k = A_k^{-1}(p_{k+1} - B_k p_k - C_k p_{k-1})$, and use induction on $m$. □

Iterating the three-term recurrence also gives a more general recurrence between polynomials in an orthogonal polynomial sequence. Define the *associated polynomials* $Q_{l,m}, R_{l,m}$ for the orthogonal polynomial sequence $\{p_l\}$ by the following recurrences on $m$, which are shifted versions of the recurrence for $p_l$. See, e.g., [4, 5].

$$\begin{array}{ll} Q_{l,m}(x) & = (A_{l+m-1}x + B_{l+m-1})Q_{l,m-1}(x) + C_{l+m-1}Q_{l,m-2}(x), \\ Q_{l,0}(x) & = 1, \quad Q_{l,1}(x) = A_l x + B_l, \\ R_{l,m}(x) & = (A_{l+m-1}x + B_{l+m-1})R_{l,m-1}(x) + C_{l+m-1}R_{l,m-2}(x), \\ R_{l,0}(x) & = 0, \quad R_{l,1}(x) = C_l. \end{array} \tag{2.13}$$

LEMMA 2.8 (generalized three-term recurrence). *The associated polynomials satisfy* $\deg Q_{l,m} = m$, $\deg R_{l,m} \le m-1$, *and for* $l \ge 1$ *and* $m \ge 0$,

$$p_{l+m} = Q_{l,m} \cdot p_l + R_{l,m} \cdot p_{l-1}. \tag{2.14}$$

*Proof.* Equation (2.14) follows by induction on $m$ with the case $m = 1$ being the original three-term recurrence (2.12). □

In the case where the $p_l$ are the Legendre polynomials, the associated polynomials should not be confused with the associated Legendre functions, which in general are not polynomials.

**2.2. Derivation of the Driscoll–Healy algorithm.** The Driscoll–Healy algorithm [13, 14] allows one to compute orthogonal polynomial transforms at any set of $N$ sample points, in $O(N \log^2 N)$ arithmetic operations. The core of this algorithm consists of an algorithm to compute orthogonal polynomial transforms in the special case where the sample points are the Chebyshev points and the sample weights are $1/N$. For simplicity we restrict ourselves to this special case, and, furthermore, we assume that $N$ is a power of 2. In Appendix A, we sketch extensions to more general problems.

Using the relation

$$f \cdot p_{l+m} = Q_{l,m} \cdot (f \cdot p_l) + R_{l,m} \cdot (f \cdot p_{l-1}), \tag{2.15}$$

derived from the three-term recurrence (2.14), we may formulate a strategy for computing all the polynomials $f \cdot p_l$, $0 \le l < N$, in $\log_2 N$ stages.

- At stage 0, compute $f \cdot p_0$ and $f \cdot p_1$.
- At stage 1, use (2.15) with $l = 1$ and $m = N/2 - 1$ or $m = N/2$ to compute
  $$f \cdot p_{\frac{N}{2}} = Q_{1,\frac{N}{2}-1} \cdot (f \cdot p_1) + R_{1,\frac{N}{2}-1} \cdot (f \cdot p_0),$$
  $$f \cdot p_{\frac{N}{2}+1} = Q_{1,\frac{N}{2}} \cdot (f \cdot p_1) + R_{1,\frac{N}{2}} \cdot (f \cdot p_0).$$
- In general, at each stage $k, 1 \le k < \log_2 N$, similarly as before, use (2.15) with $l = 2q(N/2^k)+1$, $0 \le q < 2^{k-1}$, and $m = N/2^k - 1$ or $N/2^k$, to compute the polynomial pairs

$$f \cdot p_{\frac{N}{2^k}},\ f \cdot p_{\frac{N}{2^k}+1};\ f \cdot p_{\frac{3N}{2^k}},\ f \cdot p_{\frac{3N}{2^k}+1};\ \cdots;\ f \cdot p_{\frac{(2^k-1)N}{2^k}}, f \cdot p_{\frac{(2^k-1)N}{2^k}+1}.$$

The problem with this strategy is that computing a full representation of each polynomial $f \cdot p_l$ generates much more data at each stage than is needed to compute the final output. To overcome this problem, the Driscoll–Healy algorithm uses *Chebyshev truncation operators* to discard unneeded information at the end of each stage. Let $f = \sum_{k \geq 0} b_k T_k$ be a polynomial, of any degree, written in the basis of Chebyshev polynomials, and let $n$ be a positive integer. Then the truncation operator $\mathcal{T}_n$ applied to $f$ is defined by

$$\mathcal{T}_n f = \sum_{k=0}^{n-1} b_k T_k. \tag{2.16}$$

The important properties of $\mathcal{T}_n$ are given in Lemma 2.9.

LEMMA 2.9. *Let $f$ and $Q$ be polynomials. Then the following hold.*

1. $\mathcal{T}_1 f = \int_{-1}^{1} f(x)\omega(x)dx$, *where* $\omega(x) = \pi^{-1}(1-x^2)^{-\frac{1}{2}}$.
2. *If* $m \leq n$, *then* $\mathcal{T}_m \mathcal{T}_n = \mathcal{T}_m$.
3. *If* $\deg Q \leq m \leq n$, *then* $\mathcal{T}_{n-m}(f \cdot Q) = \mathcal{T}_{n-m}[(\mathcal{T}_n f) \cdot Q]$.

*Proof.* Part 1 follows from the orthogonality of Chebyshev polynomials, as $\mathcal{T}_1 f$ is just the constant term of $f$ in its expansion in Chebyshev polynomials. Part 2 is a trivial consequence of the definition of truncation operators. For part 3 we assume that $f = \sum_{k \geq 0} b_k T_k$ is a polynomial and that $\deg Q \leq m \leq n$. By Corollary 2.7, $T_k \cdot Q$ is in the linear span of $T_{k-m}, \ldots, T_{k+m}$, so $\mathcal{T}_{n-m}(T_k \cdot Q) = 0$ for $k \geq n$. Therefore,

$$\mathcal{T}_{n-m}(f \cdot Q) = \mathcal{T}_{n-m}\left(\sum_{k \geq 0} b_k T_k \cdot Q\right) = \mathcal{T}_{n-m}\left(\sum_{k=0}^{n-1} b_k T_k \cdot Q\right) = \mathcal{T}_{n-m}[(\mathcal{T}_n f) \cdot Q]. \qquad \square$$

As a corollary of part 1 of Lemma 2.9, we see how we can retrieve the discrete orthogonal polynomial transform from the $f \cdot p_l$'s computed by the strategy above by using a simple truncation.

COROLLARY 2.10. *Let $f$ be the unique polynomial of degree less than $N$ such that $f(x_j^N) = f_j$, $0 \leq j < N$. Let $\{p_l\}$ be an orthogonal polynomial sequence. Then*

$$\hat{f}_l = \mathcal{T}_1(f \cdot p_l), \quad 0 \leq l < N,$$

*where the $\hat{f}_l$ form the discrete orthogonal polynomial transform of $f$ of size $N$ with respect to the sample points $x_j^N$ and sample weights $1/N$.*

*Proof.* This follows from the definition of discrete orthogonal polynomial transforms, the Gaussian quadrature rule (2.11) for Chebyshev polynomials applied to the function $f \cdot p_l$, and Lemma 2.9,

$$\hat{f}_l = \frac{1}{N} \sum_{j=0}^{N-1} f(x_j^N) p_l(x_j^N) = \frac{1}{\pi} \int_{-1}^{1} \frac{f(x)p_l(x)}{\sqrt{1-x^2}} dx = \mathcal{T}_1(f \cdot p_l). \qquad \square$$

The key property of the truncation operators $\mathcal{T}_n$ is the "aliasing" property (part 3 of Lemma 2.9), which states that we may use a truncated version of $f$ when computing a truncated product of $f$ and $Q$. For example, if we wish to compute the truncated product $\mathcal{T}_1(f \cdot p_l)$ with $l, \deg f < N$ then, because $\deg p_l = l$, we may apply part 3 of Lemma 2.9 with $m = l$ and $n = l + 1$ to obtain

$$\hat{f}_l = \mathcal{T}_1(f \cdot p_l) = \mathcal{T}_1[(\mathcal{T}_{l+1} f) \cdot p_l].$$

Thus we need to know only the first $l+1$ Chebyshev coefficients of $f$ to compute $\hat{f}_l$.

The Driscoll–Healy algorithm follows the strategy described at the start of this section, but it computes truncated polynomials

$$Z_l^K = \mathcal{T}_K(f \cdot p_l) \tag{2.17}$$

for various values of $l$ and $K$, instead of the original polynomials $f \cdot p_l$. The input is the polynomial $f$, and the output is $\hat{f}_l = \mathcal{T}_1(f \cdot p_l) = Z_l^1$, $0 \le l < N$.

Each stage of the algorithm uses truncation operators to discard unneeded information, which keeps the problem size down. Instead of using the generalized three-term recurrence (2.15) directly, each stage uses truncated versions. Specifically, (2.15) with $m = K-1, K$ and part 3 of Lemma 2.9 with $m = K$ and $n = 2K$ imply the following recurrences for the $Z_l^K$:

$$Z_{l+K-1}^K = \mathcal{T}_K[Z_l^{2K} \cdot Q_{l,K-1} + Z_{l-1}^{2K} \cdot R_{l,K-1}], \tag{2.18}$$

$$Z_{l+K}^K \quad = \mathcal{T}_K[Z_l^{2K} \cdot Q_{l,K} + Z_{l-1}^{2K} \cdot R_{l,K}]. \tag{2.19}$$

The algorithm proceeds in $\log_2 N + 1$ stages, as shown in Algorithm 2.1. The organization of the computation is illustrated in Figure 2.1.

**Algorithm 2.1** Polynomial version of the Driscoll–Healy algorithm.

**INPUT** $(f_0, \ldots, f_{N-1})$: Polynomial defined by $f_j = f(x_j^N)$; $N$ is a power of 2.

**OUTPUT** $(\hat{f}_0, \ldots, \hat{f}_{N-1})$: Transformed polynomial with $\hat{f}_l = \mathcal{T}_1(f \cdot p_l) = Z_l^1$.

**STAGES**

0. Compute $Z_0^N \leftarrow f \cdot p_0$ and $Z_1^N \leftarrow \mathcal{T}_N(f \cdot p_1)$.

$k$. **for** $k = 1$ **to** $\log_2 N - 1$ **do**
  $K \leftarrow \frac{N}{2^k}$
  **for** $l = 1$ **to** $N - 2K + 1$ **step** $2K$ **do**
  (a) Use recurrence (2.18) and (2.19) to compute new polynomials.
    $Z_{l+K-1}^K \leftarrow \mathcal{T}_K \left( Z_l^{2K} \cdot Q_{l,K-1} + Z_{l-1}^{2K} \cdot R_{l,K-1} \right)$
    $Z_{l+K}^K \leftarrow \mathcal{T}_K \left( Z_l^{2K} \cdot Q_{l,K} + Z_{l-1}^{2K} \cdot R_{l,K} \right)$
  (b) Truncate old polynomials.
    $Z_{l-1}^K \leftarrow \mathcal{T}_K Z_{l-1}^{2K}$
    $Z_l^K \leftarrow \mathcal{T}_K Z_l^{2K}$

$\log_2 N$. **for** $l = 0$ **to** $N - 1$ **do**
  $\hat{f}_l \leftarrow Z_l^1$

**2.3. Data representation and recurrence procedure.** To complete our description of the Driscoll–Healy algorithm, we still need to specify how to represent the polynomials in the algorithm and to describe the methods used to multiply two polynomials and to apply the truncation operators $\mathcal{T}_K$. This is done in the following subsections.

**2.3.1. Chebyshev representation of polynomials.** Truncation of a polynomial requires no computation if the polynomial is represented by the coefficients of its expansion in Chebyshev polynomials. Therefore, we use the Chebyshev coefficients $z_n^l$ defined by

$$Z_l^K = \sum_{n=0}^{K-1} z_n^l T_n \tag{2.20}$$

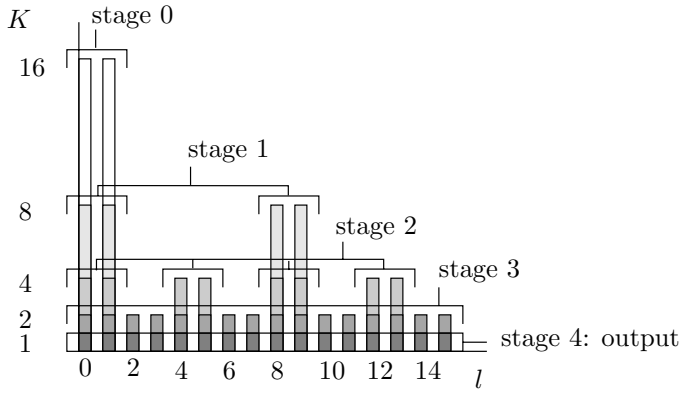


FIG. 2.1. *Computation of the truncated polynomials $Z_l^K$ for $N = 16$. Each bar represents the polynomials $Z_l^K$ for one value of $l$. The height of the bar represents the initial number of Chebyshev coefficients. At each stage, the number of coefficients is reduced as indicated by the gray scales.*

to represent all the polynomials $Z_l^K$ appearing in the algorithm. Such a representation of a polynomial is called the *Chebyshev representation*.

The input polynomial $f$ of degree less than $N$ is given as the vector $\mathbf{f} = (f_0, \ldots, f_{N-1})$ of values $f_j = f(x_j^N)$. This is called the *point-value representation* of $f$. In stage 0, we convert $Z_0^N = \mathcal{T}_N(f \cdot p_0) = f \cdot p_0$ and $Z_1^N = \mathcal{T}_N(f \cdot p_1)$ to their Chebyshev representations. For $f \cdot p_0$ this can be done by a Chebyshev transform on the vector of function values with the input values multiplied by the constant $p_0$. For $f \cdot p_1$ we also use a Chebyshev transform of size $N$, even though $f \cdot p_1$ may have degree $N$, rather than $N-1$. This poses no problem, because applying part 4 of Lemma 2.11 from the next subsection with $h = f \cdot p_1$ and $K = N$ proves that $f \cdot p_1$ agrees with $Z_1^N$ at the sampling points $x_j^N$. Stage 0 becomes the following.

Stage 0. Compute the Chebyshev representation of $Z_0^N$ and $Z_1^N$.
(a) $(z_0^0, \ldots, z_{N-1}^0) \leftarrow \text{Chebyshev}(f_0 p_0, \ldots, f_{N-1} p_0)$
(b) $(z_0^1, \ldots, z_{N-1}^1) \leftarrow \text{Chebyshev}(f_0 p_1(x_0^N), \ldots, f_{N-1} p_1(x_{N-1}^N))$

Stage 0 takes a total of $2\alpha N \log_2 N + 2\beta N + 2N$ flops, where the third term represents the $2N$ flops needed to multiply $f$ with $p_0$ and $p_1$.

**2.3.2. Recurrence using Chebyshev transforms.** To apply the recurrences (2.18) and (2.19) efficiently, we do the following.

1. Apply inverse Chebyshev transforms of size $2K$ to bring the polynomials $Z_{l-1}^{2K}, Z_l^{2K}$ into point-value representation at the points $x_j^{2K}$, $0 \le j < 2K$.
2. Perform the multiplications and additions.
3. Apply a forward Chebyshev transform of size $2K$ to bring the result into Chebyshev representation.
4. Truncate the results to degree less than $K$.

This procedure replaces the polynomial multiplications in the recurrences (2.18) and (2.19) by a slightly different operation. Because the multiplications are made in only $2K$ points, whereas the degree of the resulting polynomial could be $3K-1$, we must verify that the end result is the same. To describe the operation formally, we introduce the *Lagrange interpolation operators* $\mathcal{S}_n$ for positive integers $n$. For any polynomial $h$, the Lagrange interpolation polynomial $\mathcal{S}_n h$ is the polynomial of degree

less than $n$ which agrees with $h$ at the points $x_0^n, \ldots, x_{n-1}^n$. The important properties of $\mathcal{S}_n$ are given in Lemma 2.11.

LEMMA 2.11. *Let $g$ and $h$ be polynomials. Then the following hold.*

1. *If $\deg h < n$, then $\mathcal{S}_n h = h$.*
2. *$\mathcal{S}_n(g \cdot h) = \mathcal{S}_n((\mathcal{S}_n g) \cdot (\mathcal{S}_n h))$.*
3. *Let $m \le n$. If $\deg h \le m + n$, then $\mathcal{T}_{n-m} h = \mathcal{T}_{n-m} \mathcal{S}_n h$.*
4. *If $\deg h = n$, then $\mathcal{S}_n h = \mathcal{T}_n h$.*

*Proof.* Parts 1 and 2 are easy. To prove part 3 assume that $\deg h \le m + n$. By long division, there is a polynomial $Q$ of degree at most $m$ such that $h = \mathcal{S}_n h + T_n \cdot Q$. Applying $\mathcal{T}_{n-m}$ and using part 3 of Lemma 2.9, we obtain

$$\mathcal{T}_{n-m}\mathcal{S}_n h = \mathcal{T}_{n-m} h - \mathcal{T}_{n-m}[T_n \cdot Q] = \mathcal{T}_{n-m} h - \mathcal{T}_{n-m}[(\mathcal{T}_n T_n) \cdot Q] = \mathcal{T}_{n-m} h,$$

since $\mathcal{T}_n T_n = 0$. For part 4 we note that $\deg \mathcal{S}_n h < n$, and we use part 3 with $m = 0$ to obtain $\mathcal{S}_n h = \mathcal{T}_n \mathcal{S}_n h = \mathcal{T}_n h$. ∎

From the recurrences (2.18) and (2.19) and part 3 of Lemma 2.11 with $m = K$ and $n = 2K$, it follows that

$$Z_{l+K-1}^K = \mathcal{T}_K[\mathcal{S}_{2K}(Z_l^{2K} \cdot Q_{l,K-1}) + \mathcal{S}_{2K}(Z_{l-1}^{2K} \cdot R_{l,K-1})], \tag{2.21}$$

$$Z_{l+K}^K \quad = \mathcal{T}_K[\mathcal{S}_{2K}(Z_l^{2K} \cdot Q_{l,K}) + \mathcal{S}_{2K}(Z_{l-1}^{2K} \cdot R_{l,K})]. \tag{2.22}$$

These equations are exactly the procedure described above. The inner loop of stage $k$ of Algorithm 2.1 becomes the following.

(a) Compute the Chebyshev representation of $Z_{l+K-1}^K$ and $Z_{l+K}^K$.
$(z_0^{l+K-1}, \ldots, z_{K-1}^{l+K-1};\ z_0^{l+K}, \ldots, z_{K-1}^{l+K})$
$\leftarrow \text{Recurrence}_l^K(z_0^{l-1}, \ldots, z_{2K-1}^{l-1};\ z_0^l, \ldots, z_{2K-1}^l)$
(b) Compute the Chebyshev representation of $Z_{l-1}^K$ and $Z_l^K$.
Discard $(z_K^{l-1}, \ldots, z_{2K-1}^{l-1})$ and $(z_K^l, \ldots, z_{2K-1}^l)$

Algorithm 2.2 describes in detail the recurrence procedure, which takes $4(\alpha \cdot 2K \log_2 2K + \beta \cdot 2K) + 12K = 8\alpha K \log_2 K + (8\alpha + 8\beta + 12)K$ flops.

**2.4. Early termination.** At late stages in the Driscoll–Healy algorithm, the work required to apply the recursion amongst the $Z_l^K$ is larger than that required to finish the computation using a naive matrix-vector multiplication. It is then more efficient to use the vectors $Z_l^K$ computed so far directly to obtain the final result, as follows.

Let $q_{l,m}^n$ and $r_{l,m}^n$ denote the Chebyshev coefficients of the polynomials $Q_{l,m}$ and $R_{l,m}$, respectively, so that

$$Q_{l,m} = \sum_{n=0}^{m} q_{l,m}^n T_n, \quad R_{l,m} = \sum_{n=0}^{m-1} r_{l,m}^n T_n. \tag{2.23}$$

The problem of finishing the computation at the end of stage $k = \log_2(N/M)$, when $K = M$, is equivalent to finding $\hat{f}_l = z_0^l$ for $0 \le l < N$, given the data $z_n^l$, $z_n^{l-1}$, $0 \le n < M$, $l = 1, M+1, 2M+1, \ldots, N-M+1$. Our method of finishing the computation is to use part 1 of Lemma 2.12, which follows. Part 2 of this lemma can be used to halve the number of computations in the common case, where the polynomial recurrence (2.12) has a coefficient $B_k = 0$ for all $k$.

**Algorithm 2.2** Recurrence procedure using the Chebyshev transform.

**CALL** **Recurrence**$_l^K(\tilde{f}_0,\ldots,\tilde{f}_{2K-1};\ \tilde{g}_0,\ldots,\tilde{g}_{2K-1})$.

**INPUT** $\tilde{\mathbf{f}} = (\tilde{f}_0,\ldots,\tilde{f}_{2K-1})$ and $\tilde{\mathbf{g}} = (\tilde{g}_0,\ldots,\tilde{g}_{2K-1})$: First $2K$ Chebyshev coefficients of input polynomials $Z_{l-1}^{2K}$ and $Z_l^{2K}$; $K$ is a power of 2.

**OUTPUT** $\tilde{\mathbf{u}} = (\tilde{u}_0,\ldots,\tilde{u}_{K-1})$ and $\tilde{\mathbf{v}} = (\tilde{v}_0,\ldots,\tilde{v}_{K-1})$: First $K$ Chebyshev coefficients of output polynomials $Z_{l+K-1}^K$ and $Z_{l+K}^K$.

**STEPS**

1. Transform $\tilde{\mathbf{f}}$ and $\tilde{\mathbf{g}}$ to point-value representation.
   $(f_0,\ldots,f_{2K-1}) \leftarrow \text{Chebyshev}^{-1}(\tilde{f}_0,\ldots,\tilde{f}_{2K-1})$
   $(g_0,\ldots,g_{2K-1}) \leftarrow \text{Chebyshev}^{-1}(\tilde{g}_0,\ldots,\tilde{g}_{2K-1})$
2. Perform the recurrence.
   **for** $j = 0$ **to** $2K-1$ **do**
   $u_j \leftarrow Q_{l,K-1}(x_j^{2K})\ g_j + R_{l,K-1}(x_j^{2K})\ f_j$
   $v_j \leftarrow Q_{l,K}(x_j^{2K})\ g_j + R_{l,K}(x_j^{2K})\ f_j$
3. Transform $\mathbf{u}$ and $\mathbf{v}$ to Chebyshev representation.
   $(\tilde{u}_0,\ldots,\tilde{u}_{2K-1}) \leftarrow \text{Chebyshev}(u_0,\ldots,u_{2K-1})$
   $(\tilde{v}_0,\ldots,\tilde{v}_{2K-1}) \leftarrow \text{Chebyshev}(v_0,\ldots,v_{2K-1})$
4. Discard $(\tilde{u}_K,\ldots,\tilde{u}_{2K-1})$ and $(\tilde{v}_K,\ldots,\tilde{v}_{2K-1})$.

LEMMA 2.12.

1. *If $l \geq 1$ and $0 \leq m < M$, then*

$$\hat{f}_{l+m} = \sum_{n=0}^{m} \frac{1}{\epsilon_n}(z_n^l q_{l,m}^n + z_n^{l-1} r_{l,m}^n). \tag{2.24}$$

2. *If $p_l$ satisfies a recurrence of the form $p_{l+1}(x) = A_l x p_l(x) + C_l p_{l-1}(x)$, then*

$$q_{l,m}^n = 0 \quad \text{if } n-m \text{ is odd, and}$$
$$r_{l,m}^n = 0 \quad \text{if } n-m \text{ is even.}$$

*Proof.* Applying $\mathcal{T}_{M-m}$ to both sides of (2.15) and using part 3 of Lemma 2.9 with $n = M$ gives $Z_{l+m}^{M-m} = \mathcal{T}_{M-m}(Z_l^M \cdot Q_{l,m} + Z_{l-1}^M \cdot R_{l,m})$. Truncating again, now using $\mathcal{T}_1$, we see that $\hat{f}_{l+m} = Z_{l+m}^1$ is the constant term of the Chebyshev expansion of $Z_l^M \cdot Q_{l,m} + Z_{l-1}^M \cdot R_{l,m}$. To find this constant term expressed in the Chebyshev coefficients of $Z_l^M, Z_{l-1}^M$ and of $Q_{l,m}, R_{l,m}$, we substitute the expansions (2.20) and (2.23) and rewrite the product of sums by using the identity $T_j \cdot T_k = \frac{1}{2}(T_{|j-k|} + T_{j+k})$. For the second part, we assume that $p_l$ satisfies the given recurrence. Then $Q_{l,m}$ is odd or even according to whether $m$ is odd or even, and $R_{l,m}$ is even or odd according to whether $m$ is odd or even, which can be verified by induction on $m$. This implies that the Chebyshev expansion of $Q_{l,m}$ must contain only odd or even coefficients, respectively, and the reverse must hold for $R_{l,m}$. ∎

Assuming that the assumptions of part 2 of the lemma are valid, i.e., each term of (2.24) has either $q_{l,m}^n = 0$ or $r_{l,m}^n = 0$, and that the factor $1/\epsilon_n$ has been absorbed in the precomputed values $q_{l,m}^n$ and $r_{l,m}^n$, the total number of flops needed to compute $\hat{f}_{l+m}$ is $2m+1$.

**2.5. Complexity of the algorithm.** Algorithm 2.3 gives the sequential Driscoll–Healy algorithm in its final form. The total number of flops can be computed as follows. Stage 0 takes $2\alpha N \log_2 N + (2\beta+2)N$ flops. Stage $k$ invokes $N/(2K)$ times

the recurrence procedure, which has cost $8\alpha K \log_2 K + (8\alpha + 8\beta + 12)K$ flops, so that the total cost of that stage is $4\alpha N \log_2 K + (4\alpha + 4\beta + 6)N$ flops. Adding the costs for $K = N/2, \ldots, M$ gives $2\alpha N[\log_2^2 N - \log_2^2 M] + (2\alpha + 4\beta + 6)N[\log_2 N - \log_2 M]$ flops. In the last stage, output values have to be computed for $m = 1, \ldots, M-2$, for each of the $N/M$ values of $l$. This gives a total of $\frac{N}{M}\sum_{m=1}^{M-2}(2m+1) = NM - 2N$ flops. Summing the costs gives

$$T_{\text{Driscoll-Healy}} = N[2\alpha(\log_2^2 N - \log_2^2 M) + (4\alpha + 4\beta + 6)\log_2 N - (2\alpha + 4\beta + 6)\log_2 M + M + 2\beta]. \tag{2.25}$$

**Algorithm 2.3** Driscoll–Healy algorithm.

**INPUT** $\mathbf{f} = (f_0, \ldots, f_{N-1})$: Real vector with $N$ a power of 2.

**OUTPUT** $\hat{\mathbf{f}} = (\hat{f}_0, \ldots, \hat{f}_{N-1})$: Discrete orthogonal polynomial transform of $\mathbf{f}$.

**STAGES**

0. Compute the Chebyshev representation of $Z_0^N$ and $Z_1^N$.
   (a) $(z_0^0, \ldots, z_{N-1}^0) \leftarrow \text{Chebyshev}(f_0 p_0, \ldots, f_{N-1} p_0)$
   (b) $(z_0^1, \ldots, z_{N-1}^1) \leftarrow \text{Chebyshev}(f_0 p_1(x_0^N), \ldots, f_{N-1} p_1(x_{N-1}^N))$

$k$. **for** $k = 1$ **to** $\log_2 \frac{N}{M}$ **do**
   $K \leftarrow \frac{N}{2^k}$
   **for** $l = 1$ **to** $N - 2K + 1$ **step** $2K$ **do**
   (a) Compute the Chebyshev representation of $Z_{l+K-1}^K$ and $Z_{l+K}^K$
   $(z_0^{l+K-1}, \ldots, z_{K-1}^{l+K-1};\ z_0^{l+K}, \ldots, z_{K-1}^{l+K}) \leftarrow \text{Recurrence}_l^K(z_0^{l-1}, \ldots, z_{2K-1}^{l-1};\ z_0^l, \ldots, z_{2K-1}^l)$
   (b) Compute the Chebyshev representation of $Z_{l-1}^K$ and $Z_l^K$.
   Discard $(z_K^{l-1}, \ldots, z_{2K-1}^{l-1})$ and $(z_K^l, \ldots, z_{2K-1}^l)$

$\log_2 \frac{N}{M} + 1$. Compute the remaining values.
   **for** $l = 1$ **to** $N - M + 1$ **step** $M$ **do**
   $\hat{f}_{l-1} \leftarrow z_0^{l-1}$
   $\hat{f}_l \leftarrow z_0^l$
   **for** $m = 1$ **to** $M - 2$ **do**
   $\hat{f}_{l+m} \leftarrow z_0^l q_{l,m}^0 + z_0^{l-1} r_{l,m}^0 + \frac{1}{2}\sum_{n=1}^{m}(z_n^l q_{l,m}^n + z_n^{l-1} r_{l,m}^n)$

The optimal stage at which to halt the Driscoll–Healy algorithm and complete the computation using Lemma 2.12 depends on $\alpha$ and $\beta$ and can be obtained theoretically. The derivative of (2.25) as a function of $M$ equals zero if and only if

$$M \ln^2 2 - 4\alpha \ln M = (2\alpha + 4\beta + 6)\ln 2. \tag{2.26}$$

In our implementation $\alpha = 2.125$ and $\beta = 5$; thus the minimum is $M = 128$. In practice, the optimal choice of $M$ will depend not only on the number of flops performed, but also on the architecture of the machine used. The machine-tuned basic linear algebra subprograms (BLAS) exploit the memory hierarchy of a computer, and using the BLAS may cause a shift in the optimal value for $M$. For example, early termination can be implemented by using a level 2 BLAS operation for matrix-vector multiplication, which is more efficient than the level 1 vector operations of a straightforward implementation of the Driscoll–Healy algorithm. This will increase the optimal value for $M$.

**3. The basic parallel algorithm and its implementation.** We designed our parallel algorithm using the BSP model, which provides a simple and effective way of developing portable parallel algorithms. The BSP model does not favor any specific

computer architecture, and it includes a simple cost function that enables us to choose between algorithms without actually having to implement them.

In the following subsections, we give a brief description of the BSP model, and then we present the framework in which we develop our parallel algorithm, including the data structures and data distributions used. This leads to a basic parallel algorithm. From now on we concentrate on the DLT instead of the more general discrete orthogonal polynomial transform.

**3.1. The BSP model.** In the BSP model [34], a computer consists of a set of $p$ processors, each with a private memory, connected by a communication network that allows processors to access the memories of other processors. In the model, algorithms consist of a sequence of supersteps. In the variant of the model we use, a *superstep* is either a number of computation steps or a number of communication steps. Global synchronization barriers (i.e., places of the algorithm where all processors must synchronize with each other) precede and/or follow a communication superstep. Using supersteps imposes a sequential structure on parallel algorithms, and this greatly simplifies the design process.

A BSP computer can be characterized by four global parameters: $p$, the number of processors; $s$, the computing speed in flop/s; $g$, the communication time per data element sent or received, measured in flop time units; and $l$, the synchronization time, also measured in flop time units. Algorithms can be analyzed by using the parameters $p, g$, and $l$; the parameter $s$ just scales the time. In this work, we are able to avoid all synchronizations at the end of computation supersteps. Therefore, the time of a computation superstep is simply $w$, the maximum amount of work (in flops) of any processor. The time of a communication superstep is $hg + l$, where $h$ is the maximum number of data elements sent or received by any processor. The total execution time of an algorithm (in flops) can be obtained by adding the times of the separate supersteps. This yields an expression of the form $a + bg + cl$. For further details and some basic techniques, see [6]. BSPlib [21] is a standard library which enables parallel programming in BSP style. Available implementations are the Oxford BSP toolset [22] and the Paderborn University BSP library [8].

**3.2. Data structures and data distributions.** At each stage $k$, $1 \leq k \leq \log_2 \frac{N}{M}$, the number of intermediate polynomial pairs doubles as the number of expansion coefficients halves. Thus, at every stage of the computation, all the intermediate polynomials can be stored in two arrays of size $N$. We use an array $\mathbf{f}$ to store the Chebyshev coefficients of the polynomials $Z_l^{2K}$ and an array $\mathbf{g}$ to store the coefficients of $Z_{l+1}^{2K}$ for $l = 0, 2K, \ldots, N - 2K$ with $K = N/2^k$ in stage $k$. We also need some extra work space to compute the coefficients of the polynomials $Z_{l+K}^{2K}$ and $Z_{l+K+1}^{2K}$. For this we use two auxiliary arrays, $\mathbf{u}$ and $\mathbf{v}$, of size $N$.

The data flow of the algorithm (see Figure 3.1) suggests that we distribute all the vectors by blocks, i.e., assign one block of consecutive vector elements to each processor. This works well if $p$ is a power of two, which we will assume from now on. Since both $N$ and $p$ are thus powers of two, each processor obtains exactly $N/p$ elements. For the general case, the block distribution is defined as follows.

DEFINITION 3.1 (block distribution). *Let* $\mathbf{f}$ *be a vector of size* $N$. *We say that* $\mathbf{f}$ *is* block distributed *over* $p$ *processors if, for all* $j$, *the element* $f_j$ *is stored in* $\mathrm{Proc}(j \operatorname{div} b)$ *and has local index* $j' = j \bmod b$, *where* $b = \lceil N/p \rceil$ *is the block size.*

The precomputed data required to perform the recurrence of stage $k$ are stored in two-dimensional arrays $\mathbf{Q}$ and $\mathbf{R}$, each of size $2\log_2(N/M) \times N$. Each pair of rows

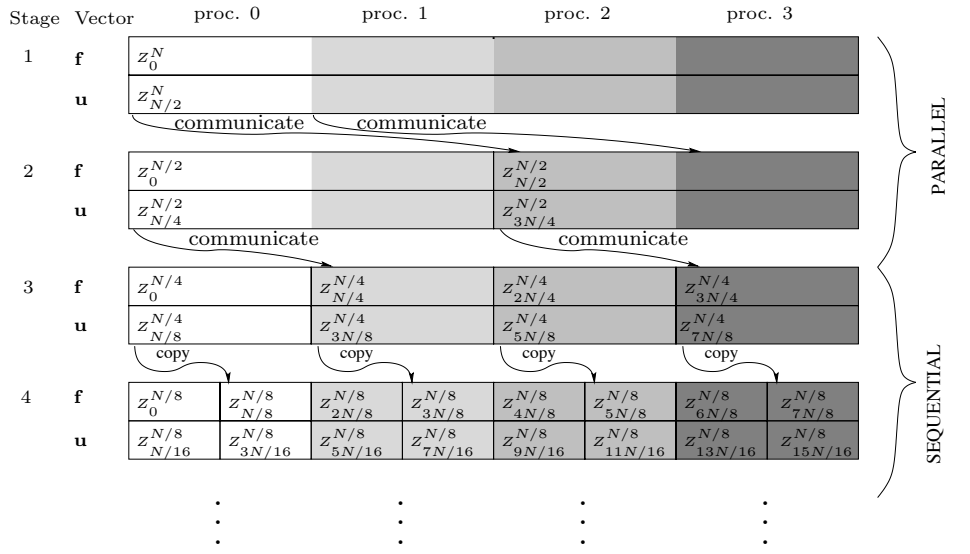


FIG. 3.1. *Main data structure and data distribution in the parallel fast Legendre transform (FLT) algorithm for $p = 4$. Arrays* **f** *and* **g** *contain the Chebyshev coefficients of the polynomials $Z_l^{2K}$ and $Z_{l+1}^{2K}$, which are already available at the start of the stage. Arrays* **u** *and* **v** *contain $Z_{l+K}^{2K}$ and $Z_{l+K+1}^{2K}$, which become available at the end of the stage. Arrays* **g** *and* **v** *are not depicted. Each array is divided into four local subarrays by using the block distribution.*

| $k$ | $K-1, K$ | proc. 0 | proc. 1 | proc. 2 | proc. 3 |
|---|---|---|---|---|---|
| 1 | 31 | $l = 0$ | | | |
| | 32 | $j = 0, \ldots, 63$ | | | |
| 2 | 15 | $l = 0$ | | $l = 32$ | |
| | 16 | $j = 0, \ldots, 31$ | | $j = 0, \ldots, 31$ | |
| 3 | 7 | $l = 0$ | $l = 16$ | $l = 32$ | $l = 48$ |
| | 8 | $j = 0, \ldots, 15$ | $j = 0, \ldots, 15$ | $j = 0, \ldots, 15$ | $j = 0, \ldots, 15$ |

FIG. 3.2. *Data structure and distribution of the precomputed data needed in the recurrence with $N = 64$, $M = 8$, and $p = 4$. Data are stored in two-dimensional arrays* **Q** *and* **R**; *one such array is shown. Each pair of rows in an array stores the data needed for one stage $k$.*

in **Q** stores data needed for one stage $k$ by

$$\begin{aligned} \mathbf{Q}[2k-2, l+j] &= Q_{l+1,K-1}(x_j^{2K}), \\ \mathbf{Q}[2k-1, l+j] &= Q_{l+1,K}(x_j^{2K}) \end{aligned} \tag{3.1}$$

for $l = 0, 2K, \ldots, N-2K$, $j = 0, 1, \ldots, 2K-1$, where $K = N/2^k$. Thus polynomials $Q_{l+1,K-1}$ are stored in row $2k-2$, and polynomials $Q_{l+1,K}$ are stored in row $2k-1$. This is shown in Figure 3.2. The polynomials $R_{l+1,K-1}$ and $R_{l+1,K}$ are stored in the same way in array **R**. Note that the indexing of the implementation arrays starts at zero. Each row of **Q** and **R** is distributed by the block distribution, i.e., $\mathbf{Q}[i,j], \mathbf{R}[i,j] \in \mathrm{Proc}(j \text{ div } \frac{N}{p})$, so that the recurrence is a local operation.

The termination coefficients $q_{l,m}^n$ and $r_{l,m}^n$ for $l = 1, M+1, 2M+1, \ldots, N-M+1$, $m = 1, 2, \ldots, M-2$, and $n = 0, 1, \ldots, m$ are stored in a two-dimensional array **T** of size $N/M \times (M(M-1)/2 - 1)$. The coefficients for one value of $l$ are stored in row $(l-1)/M$ of **T**. Each row has the same internal structure: the coefficients are stored in increasing order of $m$, and coefficients with the same $m$ are ordered by increasing $n$. (This format is commonly used to store lower triangular matrices.) By part 2 of

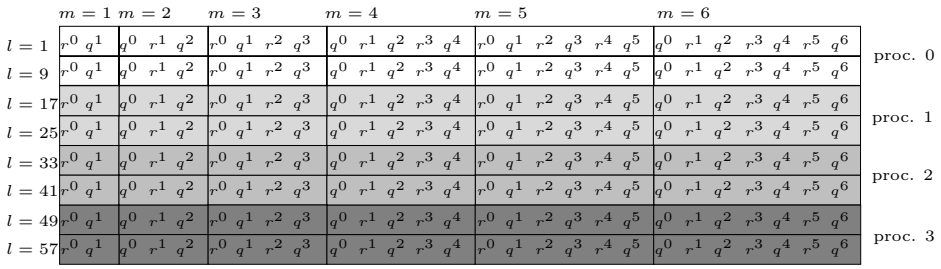


FIG. 3.3. *Data structure and distribution of the precomputed data for termination with $N = 64$, $M = 8$, and $p = 4$. The coefficients $q^n_{l,m}$ and $r^n_{l,m}$ are stored in a two-dimensional array $\mathbf{T}$. In the picture, $q^n$ denotes $q^n_{l,m}$, and $r^n$ denotes $r^n_{l,m}$.*

Lemma 2.12, either $q^n_{l,m} = 0$ or $r^n_{l,m} = 0$ for each $n$ and $m$, so we need to store only the value that can be nonzero. Since this depends on whether $n - m$ is even or odd, we obtain an alternating pattern of $q^n_{l,m}$'s and $r^n_{l,m}$'s. Figure 3.3 illustrates this data structure.

The termination stage can be kept local if $M \leq N/p$. This requires that each row of $\mathbf{T}$ is assigned to one processor, namely, to the processor that holds the subvectors for the corresponding value of $l$. Each column of $\mathbf{T}$ is in the block distribution, i.e., $\mathbf{T}[i, j] \in \text{Proc}(i \text{ div } \frac{N}{pM})$. As a result, the $N/M$ rows of $\mathbf{T}$ are distributed in consecutive blocks of rows.

**3.3. The basic parallel algorithm.** To formulate our basic parallel algorithm, we introduce the following conventions and subroutines.

*Processor identification.* The total number of processors is $p$. The processor identification number is $s$ with $0 \leq s < p$.

*Supersteps.* Labels indicate a superstep and its type: (Comp) computation superstep, (Comm) communication superstep, and (CpCm) subroutine containing both computation and communication supersteps. Global synchronizations are stated explicitly. Supersteps inside loops are executed repeatedly, though they are numbered only once.

*Indexing.* All the indices are global. This means that array elements have a unique index which is independent of the processor that owns it. This enables us to describe variables and gain access to arrays in an unambiguous manner, even though the array is distributed and each processor has only part of it.

*Vectors and subroutine calls.* All vectors are indicated in boldface. To specify part of a vector we write its first element in boldface, e.g., $\mathbf{f_j}$; the vector size is explicitly written as a parameter.

*Communication.* Communication between processors is done by using

$$\mathbf{g_j} \leftarrow \text{Put}(pid, n, \mathbf{f_i}).$$

This operation puts $n$ elements of vector $\mathbf{f}$, starting from element $i$, into processor $pid$ and stores them there in vector $\mathbf{g}$ starting from element $j$.

*Copying a vector.* The operation

$$\mathbf{g_j} \leftarrow \text{Copy}(n, \mathbf{f_i})$$

denotes the copy of $n$ elements of vector $\mathbf{f}$, starting from element $i$, to a vector $\mathbf{g}$ starting from element $j$.

*Subroutine name ending in* 2. Subroutines with a name ending in 2 perform an operation on two vectors instead of one. For example,

$$(\mathbf{f_i}, \mathbf{g_j}) \leftarrow \text{Copy2}(n, \mathbf{u_k}, \mathbf{v_l})$$

is an abbreviation for

$$\begin{aligned} \mathbf{f_i} &\leftarrow \text{Copy}(n, \mathbf{u_k}) \\ \mathbf{g_j} &\leftarrow \text{Copy}(n, \mathbf{v_l}). \end{aligned}$$

*FChT.* The subroutine

$$\text{BSP\_FChT}(s_0, s_1, p_1, sign, n, \mathbf{f})$$

replaces the input vector $\mathbf{f}$ of size $n$ by its Chebyshev transform if $sign = 1$, or by its inverse Chebyshev transform if $sign = -1$. A group of $p_1$ processors starting from processor $s_0$ work together; $s_1$ with $0 \le s_1 < p_1$ denotes the processor number within the group. The original processor number equals $s = s_0 + s_1$. For a group of size $p_1 = 1$, this subroutine reduces to the sequential FChT.

*Truncation.* The subroutine

$$\mathbf{f} \leftarrow \text{BSP\_Trunc}(s_0, s_1, p_1, n, \mathbf{u})$$

truncates two polynomials of degree less than $n$ which are stored as vectors $\mathbf{f}$ and $\mathbf{u}$ of length $n$. The subroutine copies the first half of $\mathbf{u}$ into the second half of $\mathbf{f}$. A group of $p_1$ processors starting from processor $s_0$ work together, similar to the BSP_FChT operation. For a group of size $p_1 = 1$, the subroutine reduces to a sequential truncation of one or more complete polynomials. In Figure 3.1, the truncation operation is depicted by arrows. Algorithm 3.1 describes subroutine BSP_Trunc2 which carries out two truncation operations simultaneously. In doing so, we save one synchronization.

**Algorithm 3.1** Truncation procedure for the FLT.

**CALL** $(\mathbf{f}, \mathbf{g}) \leftarrow \text{BSP\_Trunc2}(s_0, s_1, p_1, n, \mathbf{u}, \mathbf{v})$.

**DESCRIPTION**

**if** $p_1 = 1$ **then**

$1^{\texttt{Comp}}$ Sequential truncation.

$(\mathbf{f}_{\frac{\mathbf{n}}{\mathbf{2}}}, \mathbf{g}_{\frac{\mathbf{n}}{\mathbf{2}}}) \leftarrow \text{Copy2}(\frac{n}{2}, \mathbf{u}, \mathbf{v})$

**else**

$2^{\texttt{Comm}}$ Parallel truncation.

**if** $s_1 < \frac{p_1}{2}$ **then**

$(\mathbf{f}_{\mathbf{s_1}\frac{\mathbf{n}}{\mathbf{p_1}}+\frac{\mathbf{n}}{\mathbf{2}}}, \mathbf{g}_{\mathbf{s_1}\frac{\mathbf{n}}{\mathbf{p_1}}+\frac{\mathbf{n}}{\mathbf{2}}}) \leftarrow \text{Put2}(s_0 + s_1 + \frac{p_1}{2}, \frac{n}{p_1}, \mathbf{u}_{\mathbf{s_1}\frac{\mathbf{n}}{\mathbf{p_1}}}, \mathbf{v}_{\mathbf{s_1}\frac{\mathbf{n}}{\mathbf{p_1}}})$

Synchronize

**Algorithm 3.2** Basic parallel algorithm for the FLT.

**CALL** BSP_FLT$(s, p, N, M, \mathbf{f})$.

**ARGUMENTS**

$s$: Processor identification; $0 \leq s < p$.
$p$: Number of processors; $p$ is a power of 2 with $p < N$.
$N$: Transform size; $N$ is a power of 2 with $N \geq 4$.
$M$: Termination block size; $M$ is a power of 2 with $M \leq \min(N/2, N/p)$.
$\mathbf{f} = (f_0, \ldots, f_{N-1})$: Real vector of size $N$ (block distributed).

**OUTPUT** $\mathbf{f} \leftarrow \hat{\mathbf{f}}$.

**DESCRIPTION**

$1^{\texttt{Comp}}$ Stage 1: Initialization.
**for** $j = s\frac{N}{p}$ **to** $(s+1)\frac{N}{p} - 1$ **do**
$g_j \leftarrow x_j^N f_j$
$u_j \leftarrow (\mathbf{Q}[0,j] \cdot x_j^N + \mathbf{R}[0,j]) \cdot f_j$
$v_j \leftarrow (\mathbf{Q}[1,j] \cdot x_j^N + \mathbf{R}[1,j]) \cdot f_j$

$2^{\texttt{CpCm}}$ Stage 1: Chebyshev transform.
BSP_FChT2$(0, s, p, 1, N, \mathbf{f}, \mathbf{g})$
BSP_FChT2$(0, s, p, 1, N, \mathbf{u}, \mathbf{v})$

$3^{\texttt{CpCm}}$ Stage 1: Truncation.
$(\mathbf{f}, \mathbf{g}) \leftarrow$ BSP_Trunc2$(0, s, p, N, \mathbf{u}, \mathbf{v})$
**for** $k = 2$ **to** $\log_2 \frac{N}{M}$ **do**
$K \leftarrow \frac{N}{2^k}$
$p_1 \leftarrow \max(\frac{p}{2^{k-1}}, 1)$
$s_0 \leftarrow (s \text{ div } p_1)p_1$
$s_1 \leftarrow s \bmod p_1$

$4^{\texttt{Comp}}$ Stage $k$: Copy.
$(\mathbf{u}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}, \mathbf{v}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}) \leftarrow$ Copy2$(\frac{N}{p}, \mathbf{f}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}, \mathbf{g}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}})$
**for** $l = s_0\frac{N}{p}$ **to** $(s_0+1)\frac{N}{p} - \frac{2K}{p_1}$ **step** $\frac{2K}{p_1}$ **do**

$5^{\texttt{CpCm}}$ Stage $k$: Inverse Chebyshev transform.
BSP_FChT2$(s_0, s_1, p_1, -1, 2K, \mathbf{u_l}, \mathbf{v_l})$

$6^{\texttt{Comp}}$ Stage $k$: Recurrence.
**for** $j = s_1\frac{N}{p}$ **to** $s_1\frac{N}{p} + \frac{2K}{p_1} - 1$ **do**
$a1 \leftarrow \mathbf{Q}[2k-2, l+j] \cdot v_{l+j} + \mathbf{R}[2k-2, l+j] \cdot u_{l+j}$
$a2 \leftarrow \mathbf{Q}[2k-1, l+j] \cdot v_{l+j} + \mathbf{R}[2k-1, l+j] \cdot u_{l+j}$
$u_{l+j} \leftarrow a1$
$v_{l+j} \leftarrow a2$

$7^{\texttt{CpCm}}$ Stage $k$: Chebyshev transform.
BSP_FChT2$(s_0, s_1, p_1, 1, 2K, \mathbf{u_l}, \mathbf{v_l})$

$8^{\texttt{CpCm}}$ Stage $k$: Truncation.
$(\mathbf{f}, \mathbf{g}) \leftarrow$ BSP_Trunc2$(s_0, s_1, p_1, 2K, \mathbf{u_l}, \mathbf{v_l})$

$9^{\texttt{Comp}}$ Stage $\log_2 \frac{N}{M} + 1$: Termination.
**for** $l = s\frac{N}{p}$ **to** $(s+1)\frac{N}{p} - M$ **step** $M$ **do**
$\mathbf{f_l} \leftarrow$ Terminate$(l, M, \mathbf{f_l}, \mathbf{g_l})$

The basic parallel algorithm for the FLT is presented as Algorithm 3.2. At each stage $k \leq \log_2(N/M)$ of the algorithm, there are $2^{k-1}$ independent problems. For $k \leq \log_2 p$, there are more processors than problems, so the processors will have to work in groups. Each group of $p_1 = p/2^{k-1} > 1$ processors handles one subvector of size $2K$, $K = N/2^k$; each processor handles a block of $2K/p_1 = N/p$ vector components. In this case, the $l$-loop has only one iteration, namely, $l = s_0 \cdot N/p$, and the $j$-loop has $N/p$ iterations, starting with $j = s_1 \cdot N/p$, so that the indices $l+j$ start with $(s_0 + s_1)N/p = s \cdot N/p$ and end with $(s_0 + s_1)N/p + N/p - 1 = (s+1)N/p - 1$. Interprocessor communication is needed, but it occurs only in two instances:

- inside the parallel FChTs (in supersteps 2, 5, 7); see section 4;
- at the end of each stage (in supersteps 3, 8).

For $k > \log_2 p$, the length of the subvectors involved becomes $2K \leq N/p$. In that case, $p_1 = 1$, $s_0 = s$, and $s_1 = 0$, and each processor has one or more complete problems to deal with, so that the processors can work independently and without communication. Note that the index $l$ runs only over the local values $sN/p$, $sN/p+2K$, ..., $(s+1)N/p - 2K$ instead of over all values of $l$.

The original stages 0 and 1 of Algorithm 2.3 are combined into one stage and then performed efficiently as follows. First, in superstep 1, the polynomials $Z_1^N$, $Z_{N/2}^N$, and $Z_{N/2+1}^N$ are computed directly from the input vector $\mathbf{f}$. This is possible because the point-value representation of $Z_1^N = \mathcal{T}_N(f \cdot P_1) = \mathcal{T}_N(f \cdot x)$ needed by the recurrences is the vector of $f_j \cdot x_j^N, 0 \leq j < N$; see subsection 2.3.1. In superstep 2, polynomials $Z_0^N = \mathbf{f}, Z_1^N = \mathbf{g}, Z_{N/2}^N = \mathbf{u}$, and $Z_{N/2+1}^N = \mathbf{v}$ are transformed to Chebyshev representation; then, in superstep 3, they are truncated to obtain the input for stage 2.

The main loop works as follows. In superstep 4, the polynomials $Z_l^{2K}$, with $K = N/2^k$ and $l = 0, 2K, \ldots, N - 2K$, are copied from the array $\mathbf{f}$ into the auxiliary array $\mathbf{u}$, where they are transformed into the polynomials $Z_{l+K}^{2K}$ in supersteps 5–7. Similarly, the polynomials $Z_{l+1}^{2K}$ are copied from $\mathbf{g}$ into $\mathbf{v}$ and then transformed into the polynomials $Z_{l+K+1}^{2K}$. Note that $\mathbf{u}$ corresponds to the lower value of $l$, so that in the recurrence the components of $\mathbf{u}$ must be multiplied by values from $\mathbf{R}$. In superstep 8, all the polynomials are truncated by copying the first $K$ Chebyshev coefficients of $Z_{l+K}^{2K}$ into the memory space of the last $K$ Chebyshev coefficients of $Z_l^{2K}$. The same happens to polynomials $Z_{l+K+1}^{2K}$ and $Z_{l+1}^{2K}$.

The termination procedure, superstep 9, is a direct implementation of Lemma 2.12 using the data structure $\mathbf{T}$ described in subsection 3.2. Superstep 9 is a computation superstep, provided the condition $M \leq N/p$ is satisfied. This usually holds for the desired termination block size $M$. In certain situations, however, one would like to terminate even earlier, with a block size larger than $N/p$. This extension is discussed in [23].

## 4. Improvements of the parallel algorithm.

**4.1. FChT of two vectors, FChT2.** The efficiency of the FLT algorithm depends strongly on the FCT algorithm used to perform the Chebyshev transform. There exists a substantial amount of literature on this topic and many implementations of sequential FCTs are available; see, e.g., [1, 29, 30, 33]. Parallel algorithms or implementations have been less intensively studied; see [31] for a recent discussion.

In the FLT algorithm, the Chebyshev transforms always come in pairs, which led us to develop an algorithm that computes two Chebyshev transforms at the same time. The new algorithm is based on the FCT algorithm given by Van Loan [35, Algorithm 4.4.6] and the standard algorithm for computing the FFTs of two real input vectors at the same time (see, e.g., [29]). The new algorithm employs a complex FFT, which is advantageous in the sequential case because as a separate module the complex FFT can easily be replaced, for instance, by a newer, more efficient FFT. Even in the parallel case, where the parallel FFT module needs to be modified to reduce the communication cost of the FLT, we can still make use of the techniques developed for the parallel complex FFT and reuse parts of the FFT code; see subsection 5.4.

The Chebyshev transform is computed as follows. Let $\mathbf{x}$ and $\mathbf{y}$ be the input vectors of length $N$. We view $\mathbf{x}$ and $\mathbf{y}$ as the real and imaginary parts of a complex vector $(\mathbf{x} + i\,\mathbf{y})$. The algorithm has three phases. Phase 1, the packing of the input

data into an auxiliary complex vector $\mathbf{z}$ of length $N$, is a simple permutation:

$$\begin{cases} z_j & = (x_{2j} + i\, y_{2j}), \\ z_{N-j-1} & = (x_{2j+1} + i\, y_{2j+1}), \qquad 0 \le j < N/2. \end{cases} \tag{4.1}$$

In phase 2, the complex FFT creates a complex vector $\mathbf{Z}$ of length $N$:

$$Z_k = \sum_{j=0}^{N-1} z_j e^{\frac{2\pi i j k}{N}}, \qquad 0 \le k < N. \tag{4.2}$$

This phase takes $4.25N \log_2 N$ flops if we use a radix-4 algorithm [35]. Finally, in phase 3 we obtain the Chebyshev transform by

$$(\tilde{x}_k + i\tilde{y}_k) = \frac{\epsilon_k}{2N}(e^{\frac{\pi i k}{2N}} Z_k + e^{-\frac{\pi i k}{2N}} Z_{N-k}), \qquad 0 \le k < N. \tag{4.3}$$

The value $Z_N$ in (4.3) is defined as $Z_N = Z_0$ by periodic extension of (4.2). Phase 3 is efficiently performed by computing the components $k$ and $N-k$ together and using symmetry properties. The cost of phase 3 is $10N$ flops. The total cost of the FChT2 algorithm is thus $4.25N \log_2 N + 10N$, giving an average $\alpha = 2.125$ and $\beta = 5$ for a single transform.

The verification that (4.1)–(4.3) indeed produce the Chebyshev transform is best made in two steps. First, we prove that

$$e^{\frac{\pi i k}{2N}} Z_k = \sum_{j=0}^{N/2-1} [(x_{2j} + iy_{2j}) e^{\frac{\pi i k (4j+1)}{2N}} + (x_{2j+1} + iy_{2j+1}) e^{-\frac{\pi i k (4j+3)}{2N}}], \tag{4.4}$$

and

$$e^{-\frac{\pi i k}{2N}} Z_{N-k} = \sum_{j=0}^{N/2-1} [(x_{2j} + iy_{2j}) e^{-\frac{\pi i k (4j+1)}{2N}} + (x_{2j+1} + iy_{2j+1}) e^{\frac{\pi i k (4j+3)}{2N}}]. \tag{4.5}$$

Second, we add (4.4) to (4.5) and multiply the result by $\frac{\epsilon_k}{2N}$ to obtain the desired equality (2.7).

The inverse Chebyshev transform is obtained by inverting the procedure described above. The phases are performed in the reverse order, and the operation of each phase is replaced by its inverse. The cost of the inverse FChT algorithm is the same as that of the FChT algorithm.

Efficient parallelization of this algorithm requires breaking open the parallel FFT inside the FChT2 and merging parts of the FFT with the surrounding computations. We explain this process in the following subsection.

**4.2. Parallel FFT within the scope of the parallel FChT2.** The FFT is a well-known method for computing the discrete Fourier transform (4.2) of a complex vector of length $N$ in $O(N \log N)$ operations. It can concisely be written as a decomposition of the Fourier matrix $F_N$,

$$F_N = A_N \cdots A_8 A_4 A_2 P_N, \tag{4.6}$$

where $F_N$ is an $N \times N$ complex matrix, $P_N$ is an $N \times N$ permutation matrix corresponding to the so-called *bit reversal permutation*, and the $N \times N$ matrices $A_L$ are defined by

$$A_L = I_{N/L} \otimes B_L, \quad L = 2, 4, 8, \ldots, N, \tag{4.7}$$

which is shorthand for a block-diagonal matrix $\mathrm{diag}(B_L, \ldots, B_L)$ with $N/L$ copies of the $L \times L$ matrix $B_L$ on the diagonal. The matrix $B_L$ is known as the $L \times L$ *butterfly matrix*.

This matrix decomposition naturally leads to the *radix*-2 *FFT* algorithm [11, 35]. In a radix-2 FFT of size $N$, the input vector $\mathbf{z}$ is permuted by $P_N$ and then multiplied successively by all the matrices $A_L$. The multiplications are carried out in $\log_2 N$ stages, each with $N/L$ times a butterfly computation. One butterfly computation modifies $L/2$ pairs $(z_j, z_{j+L/2})$ at distance $L/2$ by adding a multiple of $z_{j+L/2}$ to $z_j$ and subtracting the same multiple.

Parallel radix-2 FFTs have already been discussed in the literature; see, e.g., [25]. For simplicity, we restrict ourselves in our exposition to FFT algorithms where $p \leq \sqrt{N}$. This class of algorithms uses the block distribution to perform the short distance butterflies with $L \leq N/p$ and the cyclic distribution to perform the long distance butterflies with $L > N/p$. Figure 4.1(a) gives an example of the cyclic distribution, which is defined as follows.

DEFINITION 4.1 (cyclic distribution). *Let* $\mathbf{z}$ *be a vector of size* $N$. *We say that* $\mathbf{z}$ *is* cyclically distributed *over* $p$ *processors if, for all* $j$, *the element* $z_j$ *is stored in* $\mathrm{Proc}(j \bmod p)$ *and has local index* $j' = j \operatorname{div} p$.

Using such a parallel FFT algorithm, we obtain a basic parallel FChT2 algorithm for two vectors $\mathbf{x}$ and $\mathbf{y}$ of size $N$.

1. PACK vectors $\mathbf{x}$ and $\mathbf{y}$ as the auxiliary complex vector $\mathbf{z}$ by permuting them, using (4.1).
2. TRANSFORM vector $\mathbf{z}$ using an FFT of size $N$.
   (a) Perform a bit reversal permutation in $\mathbf{z}$.
   (b) Perform the short distance butterflies of size $L = 2, 4, \ldots, N/p$.
   (c) Permute $\mathbf{z}$ to the cyclic distribution.
   (d) Perform the long distance butterflies of size $L = 2N/p, 4N/p, \ldots, N$.
   (e) Permute $\mathbf{z}$ to the block distribution.
3. EXTRACT the transforms from vector $\mathbf{z}$ and store them in vectors $\mathbf{x}$ and $\mathbf{y}$.
   (a) Permute $\mathbf{z}$ to put components $j$ and $N - j$ in the same processor.
   (b) Compute the new values of $\mathbf{z}$ using (4.3).
   (c) Permute $\mathbf{z}$ to block distribution, and store the result in vectors $\mathbf{x}$ and $\mathbf{y}$.

The time complexity of this basic algorithm will be reduced by a sequence of improvements as detailed in the following subsections.

**4.2.1. Combining permutations.** By breaking open the FFT phase inside the parallel FChT2 algorithm, we can combine the packing permutation (1) and the bit reversal (2(a)), thus saving one complete permutation of BSP cost $2\frac{N}{p}g + l$. The same can be done for (2(e)) and (3(a)).

**4.2.2. Increasing the symmetry of the cyclic distribution.** We can eliminate permutation (2(e))/(3(a)) completely by restricting the number of processors slightly further to $p \leq \sqrt{N/2}$ and permuting the vector $\mathbf{z}$ in phase (2(e)) from block distribution to a slightly modified cyclic distribution, the zig-zag cyclic distribution, shown in Figure 4.1(b) and defined as follows.

DEFINITION 4.2 (zig-zag cyclic distribution). *Let* $\mathbf{z}$ *be a vector of size* $N$. *We say that* $\mathbf{z}$ *is* zig-zag cyclically distributed *over* $p$ *processors if, for all* $j$, *the element* $z_j$ *is stored in* $\mathrm{Proc}(j \bmod p)$ *if* $j \bmod 2p < p$ *and in* $\mathrm{Proc}(-j \bmod p)$ *otherwise, and has local index* $j' = j \operatorname{div} p$.

In this distribution, both the components $j$ and $j + L/2$ needed by the butterfly

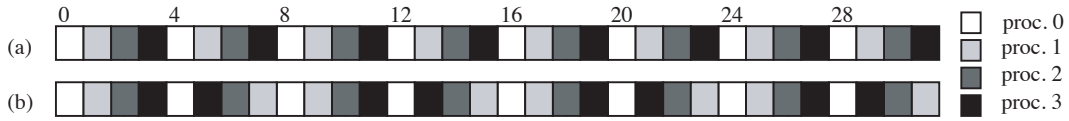


FIG. 4.1. (a) *Cyclic distribution and* (b) *zig-zag cyclic distribution for a vector of size* 32 *distributed over four processors.*

operations with $L > N/p$ and the components $j$ and $N - j$ needed by the extract operation are in the same processor; thus we avoid the permutation (2(e))/(3(a)) above, saving another $2\frac{N}{p}g + l$ in BSP costs.

**4.2.3. Reversing the stages for the inverse FFT.** To be able to apply the same ideas to the inverse transform we perform the inverse FFT by reversing the stages of the FFT and inverting the butterflies, instead of taking the more common approach of using the same FFT algorithm but replacing the powers of $e^{\frac{2\pi i}{N}}$ by their conjugates. Thus we save $6\frac{N}{p}g + 3l$, both in the Chebyshev transform and its inverse.

**4.2.4. Reducing the number of flops.** Wherever possible we take pairs of stages $A_{2L}A_L$ together and perform them as one operation. The butterflies have the form $B_{2L}(I_2 \otimes B_L)$, which is a $2L \times 2L$ matrix consisting of $4 \times 4$ blocks, each an $L/2 \times L/2$ diagonal submatrix. This matrix is a symmetrically permuted version of the radix-4 butterfly matrix [35]. This approach gives both the efficiency of a radix-4 FFT algorithm and the flexibility of treating the parallel FFT within the radix-2 framework; for example, it is possible to redistribute the data after any number of stages and not only after an even number. The radix-4 approach reduces $\alpha$ from 2.5 to 2.125. An additional benefit of radix-4 butterflies is better use of the cache memory: 34 flops are performed on a quadruple of data, instead of 10 flops on a pair of data. Thus 8.5 flops are carried out per data word loaded into the cache, instead of five flops. This effect may be even more important than the reduction in flop count.

Since we do not use the upper half of the Chebyshev coefficients computed in the forward transform, we can alter the algorithm to avoid computing them. This saves $4N$ flops in (4.3).

**4.3. Main loop.** The discussion that follows is only relevant in the parallel part of the main loop, i.e., stages $k \leq \log_2 p$, so we will restrict ourselves to these stages. Recall that in these stages a group of $p_1 = p/2^{k-1} > 1$ processors handles only one subproblem of size $2K = 2N/2^k$ corresponding to $l = s_0\frac{N}{p}$.

**4.3.1. Modifying the truncation/copy operation.** It is possible to reorganize the main loop of the FLT algorithm such that the end of stage $k$ and the start of stage $k + 1$ are merged into one, more efficient procedure. The current sequence of operations is as follows.

1. Permute from zig-zag cyclic to block distribution in stage $k$.
2. Truncate at the end of stage $k$.
3. Copy at the beginning of stage $k + 1$.
4. Permute from block to zig-zag cyclic distribution in stage $k + 1$.

In the new approach, we aim at removing permutations 1 and 4 by keeping the data in the zig-zag cyclic distribution of stage $k$ during part of stage $k + 1$ as well. In the following discussion, we treat only operations on the arrays $\mathbf{f}$ and $\mathbf{u}$ because the operations on $\mathbf{g}$ and $\mathbf{v}$ are similar. The values of $K$ and $p_1$ used in the discussion are those of stage $k$. Now assume that the second half of the $2K$ elements of $\mathbf{f_l}$ has

been discarded. Recall that the second half of $\mathbf{u_l}$ has not even been computed; see subsection 4.2.4. Therefore, $\mathbf{f_l}$ and $\mathbf{u_l}$ are now in the zig-zag cyclic distribution of $K$ elements (instead of $2K$) over $p_1$ processors. The new sequence of operations, illustrated in Figure 4.2, is as follows.

1. Prepare a working copy of the data needed at stage $k+1$.
   (a) Copy vector $\mathbf{u_l}$ of size $K$ into vector $\mathbf{u_{l+K}}$.
   (b) Copy vector $\mathbf{f_l}$ of size $K$ into vector $\mathbf{u_l}$.
2. Redistribute the data needed at stage $k+2$.
   (a) Put the first $K/2$ elements of vector $\mathbf{u_l}$ into vector $\mathbf{f_l}$ in the zig-zag cyclic distribution over the first $\frac{p_1}{2}$ processors.
   (b) Put the first $K/2$ elements of vector $\mathbf{u_{l+K}}$ into vector $\mathbf{f_{l+K}}$ in the zig-zag cyclic distribution over the next $\frac{p_1}{2}$ processors.

The synchronization at the end of the redistribution can be removed by buffering and delaying the communications until the next synchronization. The new approach reduces the BSP cost of the truncation/copy operation from $6\frac{N}{p}g + 3l$ to $\frac{N}{p}g$.

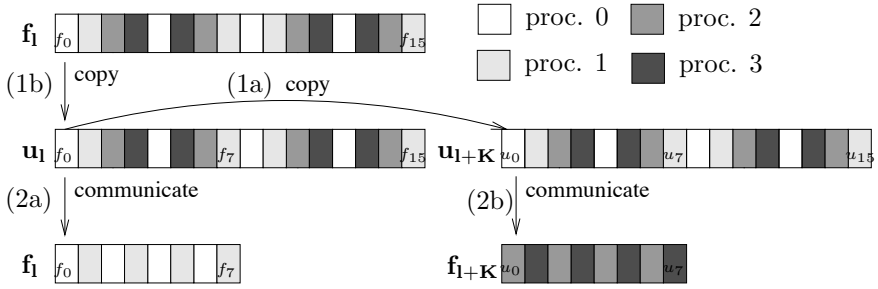


FIG. 4.2. *New truncation/copy operation of vectors* $\mathbf{f_l}$ *and* $\mathbf{u_l}$ *for* $K = 16$ *and* $p_1 = 4$.

As a result, vectors $\mathbf{u_l}$ and $\mathbf{u_{l+K}}$ contain all the data needed at stage $k+1$, and vectors $\mathbf{f_l}$ and $\mathbf{f_{l+K}}$ contain half the data needed at stage $k+2$; stage $k+1$ will produce the other half. We now verify that the operations on $\mathbf{u_l}$ and $\mathbf{u_{l+K}}$ at the start of stage $k+1$ remain local and hence do not require communication. The first operation is the inverse of operation (4.3), which acts on an array of size $K$. The pairs $(u_{l+j}, u_{l+K-j})$ and $(u_{l+K+j}, u_{l+2K-j})$ involved in this operation are indeed local. After that, the long distance butterflies of the inverse FFT have to be performed, and then the short distance ones have to be performed. The short distance butterflies, of size $L \le K/(p_1/2)$, will be done after a suitable redistribution as in the original algorithm, using the block distribution over $p_1/2$ processors. The long distance butterflies operate on pairs $(u_{l+j}, u_{l+j+L/2})$ and $(u_{l+K+j}, u_{l+K+j+L/2})$ with $L \ge 4K/p_1$. The restriction $p \le \sqrt{N/2}$ implies $p_1 \le \sqrt{K}$ and hence $2p_1 \le 2K/p_1$, which means that the period of the zig-zag cyclic distribution over $p_1$ processors does not exceed the minimum butterfly distance. As a result, the pairs involved are local.

**4.3.2. Moving the bit reversal to the precomputation.** Another major improvement is to avoid the packing/bit reversal permutation (1)/(2(a)) in the FChT2 just following the recurrence and its inverse preceding the recurrence, thus saving another $4\frac{N}{p}g + 2l$ in communication costs. This is done by storing the recurrence coefficients permuted by the packing/bit reversal permutation. This works because one permutation is the inverse of the other, so that the auxiliary vector $\mathbf{z}$ is in the same ordering immediately before and after the permutations.

**4.4. Time complexity.** After all the improvements, the total communication and synchronization cost is approximately $(5\frac{N}{p}\log_2 p)g + (2\log_2 p)l$. Only two communication supersteps remain: the zig-zag cyclic to block redistribution inside the inverse FFT, which can be combined with the redistribution of the truncation, and the block to zig-zag cyclic redistribution inside the FFT. To obtain this complexity, we ignored lower order terms and special cases occurring at the start and the end of the algorithm.

The total cost of the optimized algorithm without early termination is

$$T_{\mathrm{FLT,par}} \approx 4.25\frac{N}{p}\log_2^2 N + \left(5\frac{N}{p}\log_2 p\right) g + (2\log_2 p)\, l. \tag{4.8}$$

**5. Experimental results and discussion.** In this section, we present results on the accuracy and scalability of the implementation of the Legendre transform algorithm. We implemented the algorithm in ANSI C using the BSPlib library [21]. Our programs are completely self-contained, and we did not rely on any system-provided numerical software such as BLAS, FFTs, etc. Nonetheless, we used an optimized FFT package [27] to illustrate how to optimize the computation supersteps of the code; see section 5.4. We tested our programs using the Oxford BSP toolset [22] implementation of BSPlib running on two different machines:

1. a *Cray T3E* with up to 64 processors, each having a theoretical peak speed of 600 Mflop/s, with double precision (64-bit) accuracy of $1.0 \times 10^{-15}$;
2. an *IBM RS/6000 SP* with up to 8 processors, each having a theoretical peak speed of 640 Mflop/s, which uses the more common IEEE 754 floating point arithmetic with double precision accuracy of $2.2 \times 10^{-16}$.

To make a consistent comparison of the results, we compiled all test programs using the `bspfront` driver with options `-O3 -flibrary-level 2 -bspfifo 10000 -fcombine-puts` (on the IBM we had to add the option `-fcombine-puts-buffer 256K,8M,256K`) and measured the elapsed execution times on exclusively dedicated CPUs using the system clock.

**5.1. Accuracy.** We tested the accuracy of our implementation by measuring the error obtained when transforming a random input vector $\mathbf{f}$ with elements uniformly distributed between 0 and 1. The relative error is defined as $||\hat{\mathbf{f}}^* - \hat{\mathbf{f}}||_2/||\hat{\mathbf{f}}||_2$, where $\hat{\mathbf{f}}^*$ is the FLT and $\hat{\mathbf{f}}$ is the exact DLT (computed by (2.9), using the stable three-term recurrence (2.10) and quadruple precision); $||\cdot||_2$ indicates the $L^2$-norm.

Table 5.1 shows the relative errors of the sequential algorithm for various problem sizes using double precision except in the precomputation of the last column, which is carried out in quadruple precision. This could not be done for the Cray T3E because quadruple precision is not available there. Note, however, that it is possible to precompute the values on another computer. The results show that the error of the FLT algorithm is comparable with the error of the DLT provided that the precomputed values are accurate. Therefore, it is best to perform the precomputation in increased precision. This can be done at little extra cost because the precomputation is done only once and its cost can be amortized over many FLTs. See [19, 20] for a discussion of other techniques that can be used to obtain more accurate results.

The errors of the parallel implementation are of the same order as in the sequential case. The only part of the parallel implementation that differs from the sequential implementation in this respect is the FFT, and then only if the butterfly stages cannot be paired in the same way. Varying the termination block size between 2 and 128 also does not significantly change the magnitude of the error.

TABLE 5.1
*Relative errors for the sequential FLT algorithm. (QP indicates that the precomputation is carried out in quadruple precision.)*

| $N$ | Cray T3E | | IBM SP (IEEE 754) | | |
|---|---|---|---|---|---|
| | DLT | FLT | DLT | FLT | FLT-QP |
| 512 | $7.0 \times 10^{-14}$ | $1.4 \times 10^{-12}$ | $7.7 \times 10^{-14}$ | $4.3 \times 10^{-12}$ | $1.5 \times 10^{-14}$ |
| 1024 | $3.5 \times 10^{-13}$ | $2.1 \times 10^{-11}$ | $3.0 \times 10^{-13}$ | $3.1 \times 10^{-11}$ | $2.3 \times 10^{-13}$ |
| 8192 | $1.2 \times 10^{-11}$ | $5.4 \times 10^{-9}$ | $1.3 \times 10^{-11}$ | $3.5 \times 10^{-9}$ | $1.3 \times 10^{-11}$ |
| 65536 | $2.7 \times 10^{-10}$ | $5.5 \times 10^{-7}$ | $2.7 \times 10^{-10}$ | $9.4 \times 10^{-8}$ | $1.6 \times 10^{-10}$ |

TABLE 5.2
*Execution time (in ms) of various sequential Legendre transform algorithms.*

| $N$ | Cray T3E | | | | IBM SP | | | |
|---|---|---|---|---|---|---|---|---|
| | DLT | FLT $M = \frac{N}{2}$ | FLT $M = 64$ | FLT $M = 2$ | DLT | FLT $M = \frac{N}{2}$ | FLT $M = 64$ | FLT $M = 2$ |
| 16 | 0.013 | 0.028 | −− | 0.078 | 0.018 | 0.016 | −− | 0.041 |
| 32 | 0.050 | 0.053 | −− | 0.187 | 0.046 | 0.030 | −− | 0.100 |
| 64 | 0.219 | 0.114 | −− | 0.436 | 0.182 | 0.069 | −− | 0.239 |
| 128 | 1.199 | 0.328 | 0.328 | 1.027 | 0.729 | 0.186 | 0.186 | 0.560 |
| 256 | 5.847 | 1.394 | 1.123 | 2.576 | 3.109 | 0.612 | 0.549 | 1.297 |
| 512 | 23.497 | 5.712 | 3.340 | 6.034 | 12.483 | 2.231 | 1.568 | 3.013 |
| 1024 | 93.702 | 21.559 | 8.525 | 14.147 | 49.917 | 8.141 | 3.966 | 6.889 |

**5.2. Efficiency of the sequential implementation.** We measured the efficiency of our optimized FLT algorithm by comparing its execution time with the execution time of the direct DLT algorithm (i.e., a matrix-vector multiplication). Table 5.2 shows the times obtained by the direct algorithm and the FLT with various termination values: $M = N/2$ is the maximum termination value that our program can handle, and the resulting algorithm is similar to the seminaive algorithm [12]; $M = 64$ is the empirically determined value that makes the algorithm perform best for $N \leq 8192$ on the Cray T3E and for $N \leq 16384$ on the IBM SP (for larger values of $N$, the choice $M = 128$ gives slightly better results); $M = 2$ yields the pure FLT algorithm without early termination.

The results show that for the choice $M = N/2$ the behavior of the FLT is similar to that of the DLT. On the other extreme, the pure FLT algorithm with $M = 2$ becomes faster than the DLT algorithm at $N = 128$. Choosing the optimum value $M = 64$ further improves the performance of the FLT. This optimum is close to the theoretical optimum $M = 128$ calculated in section 2.4.

Another advantage of the FLT is its reduced need of storage space for the precomputed data. While the DLT must store $O(N^2)$ precomputed values, the FLT needs to store only $O(N \log N)$ precomputed values. Furthermore, these values can be computed in only $O(N \log^2 N)$ operations using Algorithm B.1, which is presented in Appendix B. Table 5.3 lists the storage and precomputation requirements for the DLT and the FLT for various input sizes, assuming a precomputation cost of $4N(N-2)$ for the direct transform and $12.75N \log_2^2 N + 76.25N \log_2 N$ for the fast transform; cf. (B.3).

**5.3. Scalability of the parallel implementation.** We tested the scalability of our optimized parallel implementation using our optimized sequential implementation as basis for comparison.

Tables 5.4 and 5.5 show the execution times on the Cray T3E for up to 64 proces-

TABLE 5.3
*Storage and precomputation requirements for the direct and FLT algorithms.*

| | Storage space (in words) | | Precomputation cost (in flops) | |
|---|---|---|---|---|
| $N$ | DLT | FLT ($M = 2$) | DLT | FLT ($M = 2$) |
| 16 | 256 | 96 | 896 | 8,144 |
| 64 | 4,096 | 640 | 15,872 | 58,656 |
| 256 | 65,536 | 3,584 | 260,096 | 365,056 |
| 1024 | 1,048,576 | 18,432 | 4,186,112 | 2,086,400 |

TABLE 5.4
*Execution times (in ms) for the FLT algorithm on a Cray T3E for $M = 2$ (top) and $M = 64$ (bottom).*

| $N$ | seq | $p = 1$ | $p = 2$ | $p = 4$ | $p = 8$ | $p = 16$ | $p = 32$ | $p = 64$ |
|---|---|---|---|---|---|---|---|---|
| 512 | 6.04 | 6.30 | 3.65 | 2.36 | 1.82 | 2.16 | −− | −− |
| 1024 | 14.15 | 14.42 | 8.18 | 4.19 | 3.23 | 2.71 | −− | −− |
| 8192 | 206.61 | 205.30 | 103.47 | 52.96 | 27.66 | 15.61 | 9.48 | 9.10 |
| 65536 | 4515.10 | 4518.30 | 2325.80 | 1180.00 | 569.47 | 244.02 | 126.74 | 67.17 |
| 512 | 3.37 | 3.58 | 2.29 | 1.67 | −− | −− | −− | −− |
| 1024 | 8.53 | 8.84 | 5.34 | 3.28 | 2.47 | −− | −− | −− |
| 8192 | 151.49 | 155.10 | 77.50 | 40.64 | 21.69 | 12.93 | 8.48 | 8.26 |
| 65536 | 3670.10 | 3732.80 | 1932.00 | 970.58 | 469.07 | 194.77 | 102.26 | 54.60 |

sors and the IBM SP for up to eight processors, respectively. The tables list timing results for the sequential and parallel algorithms with $p < \sqrt{N}$ and $M = 2, 64$. The table for the Cray starts at a lower value of $N$ because on the Cray parallelism is already advantageous for much smaller problem sizes. In general, the Cray T3E delivers better scalability than the IBM SP, but the execution on the IBM SP is faster. Qualitatively, this is in accordance with the BSP parameters for these machines given in Table 5.6: the parameters $g$ and $l$ are smaller for the Cray T3E, which means relatively faster communication and synchronization, whereas the parameter $s$ is larger for the IBM SP, which means faster computation.

The BSP parameters presented in Table 5.6 reflect the way we implemented the communication subroutines of our programs. Our implementations divide the communication supersteps into three phases. In phase 1, the data are locally rearranged in such a way that the elements to be sent to the same processor are packed together. In phase 2, packets of data are exchanged between the processors. In phase 3, the data are unpacked locally so that the elements get to their final destination. This scheme generally saves communication time for regular communication patterns, because the overhead of sending corresponding address information together with the actual data is drastically reduced. Furthermore, we implemented the communication subroutines using `hpputs` (high performance put operations that dispense with the use of buffers). Therefore, the $l$ and $g$ values of Table 5.6 were measured for large data packets sent by using `hpputs`. For more details, see [23, Appendix A].

Figure 5.1 shows the behavior of our algorithm in terms of flop rate per processor:

$$F(N,p) = \frac{4.25N\log_2^2 N + 34.5N\log_2 N}{p \cdot T(N,p)}. \tag{5.1}$$

Here, the numerator represents the number of flops of the basic sequential FLT algorithm with only the two main terms included and without optimizations such as early termination. This gives a convenient reference count for the FLT, similar to the

TABLE 5.5
*Execution times (in ms) for the FLT algorithm on an IBM SP for $M = 2$ (top) and $M = 64$ (bottom).*

| $N$ | seq | $p = 1$ | $p = 2$ | $p = 4$ | $p = 8$ |
|---|---|---|---|---|---|
| 8192 | 83.71 | 85.36 | 60.51 | 53.96 | 73.26 |
| 16384 | 233.89 | 242.91 | 134.45 | 99.67 | 102.77 |
| 32768 | 840.56 | 865.57 | 449.76 | 229.25 | 187.90 |
| 65536 | 2159.20 | 2201.80 | 1193.90 | 625.40 | 353.87 |
| 8192 | 56.85 | 57.81 | 48.18 | 47.30 | 70.93 |
| 16384 | 176.97 | 179.74 | 107.79 | 83.52 | 96.70 |
| 32768 | 640.49 | 651.51 | 359.49 | 187.04 | 168.59 |
| 65536 | 1729.30 | 1747.60 | 1016.10 | 540.95 | 314.75 |

TABLE 5.6
*BSP parameters measured using a modified version of the program* `bspbench` *from the package* `BSPEDUpack` [7].

| $p$ | Cray T3E ($s = 34.9$ Mflop/s) | | | | IBM SP ($s = 202$ Mflop/s) | | | |
|---|---|---|---|---|---|---|---|---|
| | $g$ | | $l$ | | $g$ | | $l$ | |
| | (flops) | ($\mu$s) | (flops) | ($\mu$s) | (flops) | ($\mu$s) | (flops) | ($\mu$s) |
| 2 | 1.14 | 0.0328 | 479 | 13.72 | 82.2 | 0.407 | 215203 | 1066 |
| 8 | 2.14 | 0.0613 | 1377 | 39.48 | 92.0 | 0.456 | 868640 | 4300 |
| 64 | 3.05 | 0.0876 | 3861 | 110.88 | −− | −− | −− | −− |

common count of $5N \log_2 N$ for the FFT. Furthermore, $T(N, p)$ denotes the execution time of the parallel FLT algorithm. Ideally, the flop rates should be high and remain constant with an increase in $p$. As already pointed out, in absolute numbers, the IBM SP delivers more Mflops per second, but the Cray T3E maintains better flop rates as a function of $p$.

Normalizing $F(N, p)$ against the flop rate of the sequential algorithm, $F^{\text{seq}}(N)$ (the entry labeled "seq" in Figure 5.1), gives the absolute efficiency of a parallel algorithm:

$$E^{\text{abs}}(N, p) = \frac{F(N, p)}{F^{\text{seq}}(N)}. \tag{5.2}$$

The absolute efficiency can be used as a measure of the scalability of the parallel algorithm. Ideally, $E^{\text{abs}}(N, p) = 1$. As a rule of thumb, efficiencies larger than 0.8 are considered very good, while efficiencies of 0.5 are reasonable. From the nearly horizontal lines of $F(N, p)$ for large $N$, it is clear that the algorithm scales very well asymptotically (i.e., when $N$ is large). The experimental results also show that on the Cray T3E with up to 64 processors a size of $N \geq 8192$ already gives reasonable to very good efficiencies, whereas on the IBM SP with up to eight processors, $N$ must at least be equal to 32768. Note that on the Cray T3E, efficiencies larger than one are observed for $N \geq 16384$. This is a well-known phenomenon related to cache size.

The DLT is often used as part of a larger spherical harmonic transform. This means that, in practical applications, many independent FLTs of small size will be performed by a group of processors that could be only slightly larger than the number of transforms. For this reason, it is important that the FLT algorithm scales well for small $N$ and $p$. Indeed, our algorithm already delivers reasonable to good efficiencies on the Cray T3E with up to eight processors for $N$ as small as 512; on the IBM SP, larger problem sizes are needed.

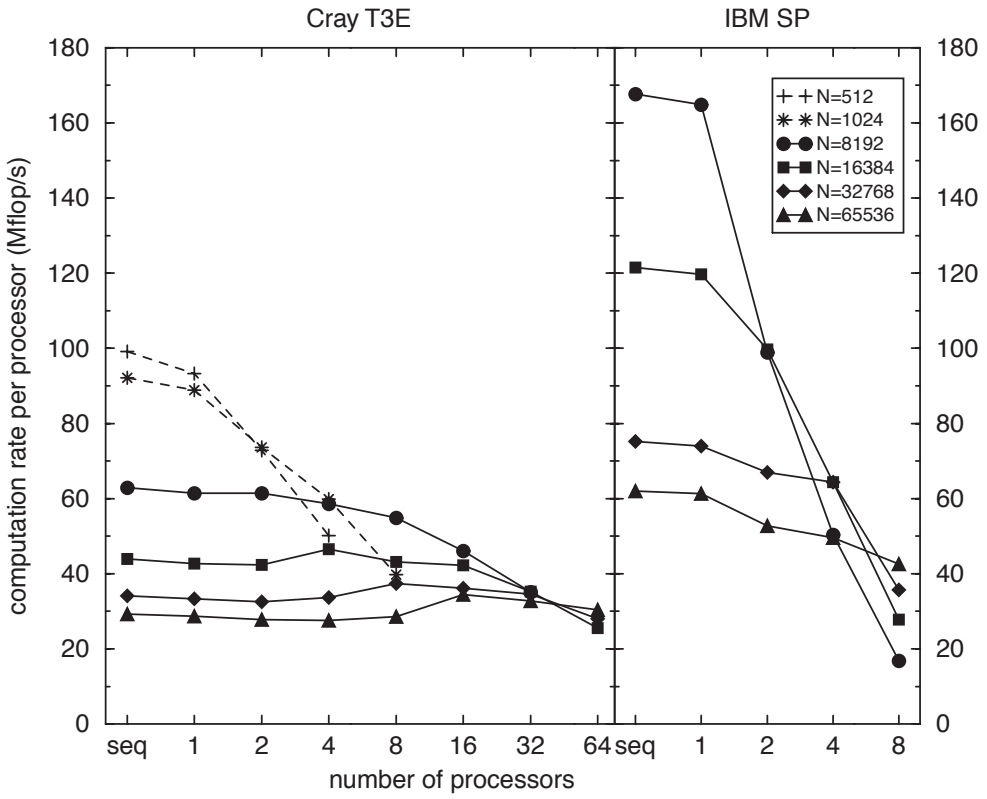


FIG. 5.1. *Mflop/s rate per processor of the FLT algorithm with termination parameter* $M = 64$.

**5.4. Further optimizations.** It is easy to modify our FLT algorithm to enable the use of complete FFTs instead of our own butterfly routines. In this subsection, we discuss the necessary modifications and demonstrate the possible gains by comparing our plain implementation with an optimized version that uses Ooura's FFT package `fft4g.c` [27]. This competitive FFT is on average 2.6 times faster than our butterflies and our FFT.

The sequential stages of our parallel algorithm involve complete FFTs, provided we do not move the bit-reversal permutation to the precomputation; see section 4.3.2. These FFTs can readily be replaced by highly optimized versions. Similarly, the short distance butterflies of the parallel stages can be replaced by a local bit-reversal permutation followed by a complete local FFT of size $N/p$. In stages 2 to $\log_2 p$, the extra permutation can be moved to the precomputation, so that it comes for free. (It is even possible to skip the extra permutation of the first stage; see [23, Section 2.3.2].)

Figure 5.2 shows the effect on the Cray T3E of optimizing the computation supersteps of the FLT algorithm. For small $N$, the main savings are already obtained by terminating early, while for large $N$ they are achieved by using complete, faster FFTs. We also observe that computation is dominant for large values of $N/p$, which leaves in principle plenty of room for improvement of the FLT by optimizing the computation supersteps. For small $N/p$, however, such optimization will not have much

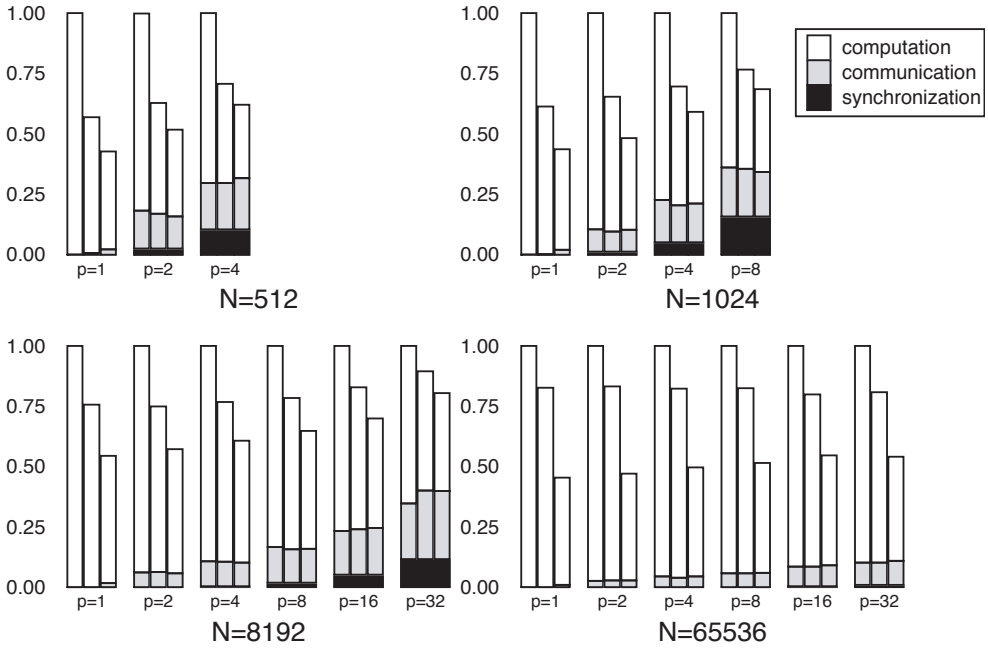


FIG. 5.2. *Gains achieved by optimizing the computational supersteps of the parallel FLT. Vertical bars represent the time of an FLT of length $N$ executed on $p$ processors of a Cray T3E, normalized with respect to our plain implementation with $M = 2$. The first bar of each triple represents the plain implementation with $M = 2$; the second bar represents the plain implementation optimized by early termination with $M = 64$; and the third bar represents the highly optimized version (with $M = 64$ and Ooura's FFT). Each bar is split into three parts, representing computation, communication, and synchronization time.*

effect because the dominant cost is that of communication and synchronization; we believe that these costs have already been reduced to the minimum.

Optimizing the FFTs of the sequential stages and the short distance butterflies of the parallel stages already covers most of the $O(N \log^2 N)$ computation operations of the algorithm. The only parts not yet optimized are as follows: the recurrence and pack/extract operations of the FChTs, with a total of $O(N \log N)$ flops, which can be carried out using level 1 BLAS; the $O(NM)$ termination routine which can be based on level 2 BLAS; and the long distance butterflies with a total of $O(N \log^2 p)$ flops. Since $p \ll N/p$ in practice, the long distance butterfly stages have a relatively small cost; if, however, $p \approx N/p$, this cost becomes significant. Fortunately, the long distance butterfly stages in the zig-zag cyclic distribution can also be carried out using FFTs at the expense of an extra $O(N \log p)$ flops and some local permutations. To do so, we first need to permute the vector to move the even elements to the front and then apply to both vector halves the method described in [23, sections 2.4 and 2.6], which performs cyclically distributed long distance butterflies using FFTs.

**6. Conclusions and future work.** As part of this work, we developed and implemented a sequential algorithm for the DLT, based on the Driscoll–Healy algorithm. This implementation is competitive for large problem sizes. Its complexity $O(N \log^2 N)$ is considerably lower than the $O(N^2)$ matrix-vector multiplication algorithms which are still much in use today for the computation of Legendre transforms.

Its accuracy is similar, provided the precomputation is performed in increased precision. The new algorithm is a promising approach for compute-intensive applications such as weather forecasting.

The main aim of this work was to develop and implement a parallel Legendre transform algorithm. Our experimental results show that the performance of our parallel algorithm scales well with the number of processors for medium to large problem sizes. The overhead of our parallel program consists mainly of communication, and this is limited to two redistributions of the full data set and one redistribution of half the set in each of the first $\log_2 p$ stages of the algorithm. Two full redistributions are already required by an FFT and an inverse FFT, indicating that our result is close to optimal. Our parallelization approach was first to derive a basic algorithm that uses block and cyclic data distributions and then to optimize this algorithm by removing permutations and redistributions wherever possible. To facilitate this we proposed a new data distribution, which we call the zig-zag cyclic distribution.

Within the framework of this work, we also developed a new algorithm for the simultaneous computation of two Chebyshev transforms. This is useful in the context of the FLT because the Chebyshev transforms always come in pairs, but such a double FChT also has many applications in its own right, as does the corresponding double FCT. Our algorithm has the additional benefit of easy parallelization. Our FFT, FChT, and FLT programs will be made available in the public domain as the package BSPFTpack, which can be obtained through the same Webpage as BSPEDUpack [7].

We view the present FLT as a good starting point for the use of fast Legendre algorithms in practical applications. However, to make our FLT algorithm directly useful in such applications, further work must be done: an inverse FLT must be developed; the FLT must be adapted to the more general case of the spherical harmonic transform where associated Legendre functions are used (this can be done by changing the initial values of the recurrences of the precomputed values and multiplying the results by normalization factors); and alternative choices of sampling points must be made possible. Lesur and Gubbins [26] have studied the accuracy of the generalization of the FLT to the spherical harmonic transform, and they found numerical instabilities for higher order transforms. Future research should investigate how techniques such as precomputation in quadruple precision and our method of truncation improve the accuracy in the general case. Driscoll, Healy, and Rockmore [15] have already shown how a variant of the Driscoll–Healy algorithm may be used to compute Legendre transforms at any set of sample points (see Appendix A), though the set of points chosen affects the stability of the algorithm.

**Appendix A. Related transforms and algorithms.**

The derivation of the Driscoll–Healy algorithm given in section 2 depends only on the properties of truncation operators $\mathcal{T}_n$ given in Lemma 2.9 and on the existence of an efficient algorithm for applying the truncation operators. In particular, Lemmas 2.9 and 2.11 hold as stated when the weight function $\omega(x) = \pi^{-1}(1-x^2)^{\frac{1}{2}}$ is changed, when the truncation operators are defined using a polynomial sequence which is orthogonal with respect to the new weight function and which starts with the polynomial 1, and when the Lagrange interpolation operators are defined using the roots of the polynomials from the sequence. In theory, this can be used to develop new algorithms for computing orthogonal polynomial transforms, though with different sample weights $w_j$. In practice, however, the existence of efficient Chebyshev and cosine transform algorithms makes these the only reasonable choice in the definition of the truncation operators. This situation may change with the advent of other fast

transforms.

Theoretically, the basic algorithm works, with minor modifications, in the following general situation. We are given operators $\mathcal{T}_n^r$ for $1 \le n \le r$ such that the following hold.

1. $\mathcal{T}_n^r$ is a mapping from the space of polynomials of degree less than $2r$ to the space of polynomials of degree less than $n$.
2. If $m \le n \le r$, then $\mathcal{T}_m^n \mathcal{T}_n^r = \mathcal{T}_m^r$.
3. If $\deg Q \le m \le n \le r$, then $\mathcal{T}_{n-m}^r(f \cdot Q) = \mathcal{T}_{n-m}^n [(\mathcal{T}_n^r f) \cdot Q]$.

The problem now is, given an input polynomial $f$ of degree less than $N$, to compute the quantities $\mathcal{T}_1^N(f \cdot p_l)$ for $0 \le l < N$, where $\{p_l\}$ is a sequence of orthogonal polynomials. This problem may be treated using the same algorithms as in section 2, but with the truncation operators $\mathcal{T}_n$ replaced by $\mathcal{T}_n^r$, where $r \le N$ depends on the stage of the algorithm. Using $r = N$ retrieves our original algorithm. The generalized algorithm uses the quantities $Z_l^K = \mathcal{T}_K^N(f \cdot p_l)$, and the recurrences in this context are

$$Z_{l+K-1}^K = \mathcal{T}_K^{2K}[Z_l^{2K} \cdot Q_{l,K-1} + Z_{l-1}^{2K} \cdot R_{l,K-1}], \tag{A.1}$$

$$Z_{l+K}^K \quad = \mathcal{T}_K^{2K}[Z_l^{2K} \cdot Q_{l,K} + Z_{l-1}^{2K} \cdot R_{l,K}], \tag{A.2}$$

cf. (2.18) and (2.19).

This generalization of our approach may be used to derive the original algorithm of Driscoll and Healy in the exact form it was presented [13, 14], which uses the cosine transforms in the points $\cos(j\pi/K)$. For more details, see [24].

Driscoll, Healy, and Rockmore [15] describe another variant of the Driscoll–Healy algorithm that may be used to compute the Legendre transform of a polynomial sampled at the Gaussian points, i.e., at the roots of the Legendre polynomial $P_N$. Their method replaces the initial Chebyshev transform used to find the polynomial $Z_0^N$ in Chebyshev representation by a Chebyshev transform taken at the Gaussian points. Once $Z_0^N$ has been found in Chebyshev representation, the rest of the computation is the same.

The Driscoll–Healy algorithm can also be used for input vectors of arbitrary size, not only powers of two. Furthermore, at each stage, we can split the problem into an arbitrary number of subproblems, not only into two. This requires that Chebyshev transforms of suitable sizes are available.

## Appendix B. The precomputed data.

In this appendix we describe algorithms for generating the point values of $Q_{l,m}, R_{l,m}$ used in the recurrence of the FLT algorithm and for generating the coefficients $q_{l,m}^n, r_{l,m}^n$ used in its termination stage.

LEMMA B.1. *Let $l \ge 1$, $j \ge 0$, and $k \ge 1$. Then the associated polynomials $Q_{l,m}, R_{l,m}$ satisfy the recurrences*

$$Q_{l,j+k} = Q_{l+k,j} Q_{l,k} + R_{l+k,j} Q_{l,k-1}, \tag{B.1}$$

$$R_{l,j+k} = Q_{l+k,j} R_{l,k} + R_{l+k,j} R_{l,k-1}. \tag{B.2}$$

*Proof.* The proof is by induction on $j$. The proof for $j = 0$ follows immediately from the definition (2.13), since $Q_{l+k,0} Q_{l,k} + R_{l+k,0} Q_{l,k-1} = 1 \cdot Q_{l,k} + 0 = Q_{l,k}$ and similarly for $R_{l,k}$. The case $j = 1$ also follows immediately from the definition. For

$j > 1$, we have

$$\begin{aligned} Q_{l+k,j}Q_{l,k} + R_{l+k,j}Q_{l,k-1} &= \left[Q_{l+k+j-1,1}Q_{l+k,j-1} + R_{l+k+j-1,1}Q_{l+k,j-2}\right] Q_{l,k} \\ &\quad + \left[Q_{l+k+j-1,1}R_{l+k,j-1} + R_{l+k+j-1,1}R_{l+k,j-2}\right] Q_{l,k-1} \\ &= Q_{l+k+j-1,1}\left[Q_{l+k,j-1}Q_{l,k} + R_{l+k,j-1}Q_{l,k-1}\right] \\ &\quad + R_{l+k+j-1,1}\left[Q_{l+k,j-2}Q_{l,k} + R_{l+k,j-2}Q_{l,k-1}\right] \\ &= Q_{l+k+j-1,1}Q_{l,k+j-1} + R_{l+k+j-1,1}Q_{l,k+j-2} \\ &= Q_{l,k+j}, \end{aligned}$$

where we have used the case $j = 1$ to prove the first and last equality and the induction hypothesis for the cases $j - 1, j - 2$ to prove the third equality. In the same way we may show that $Q_{l+k,j}R_{l,k} + R_{l+k,j}R_{l,k-1} = R_{l,k+j}$. □

This lemma is the basis for the computation of the data needed in the recurrences of the Driscoll–Healy algorithm. The basic idea of the Algorithm B.1 is to start with polynomials of degree $0, 1$, given in only one point, and then repeatedly double the number of points by performing a Chebyshev transform, adding zero terms to the Chebyshev expansion, and transforming back, and also double the maximum degree of the polynomials by applying the lemma with $j = K - 1, K$ and $k = K$.

**Algorithm B.1** Precomputation of the point values.

**INPUT** $N$: a power of 2.

**OUTPUT** $Q_{l,m}(x_j^{2^k})$, $R_{l,m}(x_j^{2^k})$ for $1 \le k \le \log_2 N$, $0 \le j < 2^k$, $m = 2^{k-1}, 2^{k-1} - 1$, and $l = 1, 2^{k-1} + 1, \ldots, N - 2^{k-1} + 1$.

**STAGES**

0. **for** $l = 1$ **to** $N$ **do**
   $Q_{l,0}(0) \leftarrow 1, \quad R_{l,0}(0) \leftarrow 0, \quad Q_{l,1}(0) \leftarrow B_l, \quad R_{l,1}(0) \leftarrow C_l$

$k$. **for** $k = 1$ **to** $\log_2 N$ **do**
   $K \leftarrow 2^{k-1}$
   **for** $m = K - 1$ **to** $K$ **do**
      **for** $l = 1$ **to** $N - K + 1$ **step** $K$ **do**
         $(q_{l,m}^0, \ldots, q_{l,m}^{K-1}) \leftarrow \text{Chebyshev}(Q_{l,m}(x_0^K), \ldots, Q_{l,m}(x_{K-1}^K))$
         $(r_{l,m}^0, \ldots, r_{l,m}^{K-1}) \leftarrow \text{Chebyshev}(R_{l,m}(x_0^K), \ldots, R_{l,m}(x_{K-1}^K))$
         $(q_{l,m}^K, \ldots, q_{l,m}^{2K-1}) \leftarrow (0, \ldots, 0)$
         **if** $m = K$ **then** $q_{l,m}^K \leftarrow A_l A_{l+1} \cdots A_{l+m-1}/2^{m-1}$
         $(r_{l,m}^K, \ldots, r_{l,m}^{2K-1}) \leftarrow (0, \ldots, 0)$
         $(Q_{l,m}(x_0^{2K}), \ldots, Q_{l,m}(x_{2K-1}^{2K})) \leftarrow \text{Chebyshev}^{-1}(q_{l,m}^0, \ldots, q_{l,m}^{2K-1})$
         $(R_{l,m}(x_0^{2K}), \ldots, R_{l,m}(x_{2K-1}^{2K})) \leftarrow \text{Chebyshev}^{-1}(r_{l,m}^0, \ldots, r_{l,m}^{2K-1})$
   **for** $l = 1$ **to** $N - 2K + 1$ **step** $2K$ **do**
      **for** $j = 0$ **to** $2K - 1$ **do**
         $Q_{l,2K}(x_j^{2K}) \leftarrow Q_{l+K,K}(x_j^{2K})Q_{l,K}(x_j^{2K}) + R_{l+K,K}(x_j^{2K})Q_{l,K-1}(x_j^{2K})$
         $R_{l,2K}(x_j^{2K}) \leftarrow Q_{l+K,K}(x_j^{2K})R_{l,K}(x_j^{2K}) + R_{l+K,K}(x_j^{2K})R_{l,K-1}(x_j^{2K})$
         $Q_{l,2K-1}(x_j^{2K}) \leftarrow Q_{l+K,K-1}(x_j^{2K})Q_{l,K}(x_j^{2K}) + R_{l+K,K-1}(x_j^{2K})Q_{l,K-1}(x_j^{2K})$
         $R_{l,2K-1}(x_j^{2K}) \leftarrow Q_{l+K,K-1}(x_j^{2K})R_{l,K}(x_j^{2K}) + R_{l+K,K-1}(x_j^{2K})R_{l,K-1}(x_j^{2K})$

Note that $\deg R_{l,m} \le m - 1$, so the Chebyshev coefficients $r_{l,m}^n$ with $n \ge m$ are zero, which means that the polynomial is fully represented by its first $m$ Chebyshev coefficients. In the case of the $Q_{l,m}$, the coefficients are zero for $n > m$. If $n = m$, however, the coefficient is nonzero, and this is a problem if $m = K$. The $K$th coefficient which was set to zero must then be corrected and set to its true value, which can be computed easily by using (2.13) and (2.4).

The FLT algorithm requires the numbers

$$Q_{l,K}(x_j^{2K}), \quad Q_{l,K-1}(x_j^{2K}), \quad R_{l,K}(x_j^{2K}), \quad R_{l,K-1}(x_j^{2K}), \quad 0 \le j < 2K,$$

for $l = r \cdot 2K + 1$, $0 \le r < \frac{N}{2K}$, for all $K$ with $M \le K \le N/2$. After the $m$-loop in stage $k = \log_2 K + 1$ of Algorithm B.1, we have obtained these values for $l = rK + 1$, $0 \le r < N/K$. We need only the values for even $r$, so the others can be discarded. The algorithm must be continued until $K = N/2$, i.e., $k = \log_2 N$.

The total number of flops of the precomputation of the point values is

$$T_{\text{precomp, point}} = 6\alpha N \log_2^2 N + (2\alpha + 12\beta + 12) N \log_2 N. \tag{B.3}$$

Comparing with the cost (2.25) of the Driscoll–Healy algorithm itself and considering only the highest order term, we see that the precomputation costs about three times as much as the Driscoll–Healy algorithm without early termination. This one-time cost, however, can be amortized over many subsequent executions of the algorithm.

Parallelizing the precomputation of the point values can be done most easily by using the block distribution. This is similar to our approach in deriving a basic parallel version of the Driscoll–Healy algorithm. In the early stages of the precomputation, each processor handles a number of independent problems, one for each $l$. At the start of stage $k$, such a problem involves $K$ points. In the later stages, each problem is assigned to one processor group. The polynomials $Q_{l,K}$, $Q_{l,K-1}$, $R_{l,K}$, $R_{l,K-1}$, and $Q_{l+K,K}$, $Q_{l+K,K-1}$, $R_{l+K,K}$, $R_{l+K,K-1}$ are all distributed in the same manner, so that the recurrences are local. The Chebyshev transforms and the addition of zeros may require communication. For the addition of zeros, this is caused by the desire to maintain a block distribution while doubling the number of points. The parallel precomputation algorithm can be optimized following similar ideas as in the optimized main algorithm.

The precomputation of the coefficients $q_{l,m}^n, r_{l,m}^n$ required to terminate the Driscoll–Healy algorithm early, as in Lemma 2.12, is based on the following recurrences.

LEMMA B.2. *Let $l \ge 1$ and $m \ge 2$. The coefficients $q_{l,m}^n$ satisfy the recurrences*

$$\begin{aligned}
q_{l,m}^n &= \frac{1}{2} A_{l+m-1}(q_{l,m-1}^{n+1} + q_{l,m-1}^{n-1}) + B_{l+m-1} q_{l,m-1}^n + C_{l+m-1} q_{l,m-2}^n \text{ for } n \ge 2, \\
q_{l,m}^1 &= A_{l+m-1}(q_{l,m-1}^0 + \frac{1}{2} q_{l,m-1}^2) + B_{l+m-1} q_{l,m-1}^1 + C_{l+m-1} q_{l,m-2}^1, \\
q_{l,m}^0 &= \frac{1}{2} A_{l+m-1} q_{l,m-1}^1 + B_{l+m-1} q_{l,m-1}^0 + C_{l+m-1} q_{l,m-2}^0,
\end{aligned}$$

*subject to the boundary conditions $q_{l,0}^0 = 1, q_{l,1}^0 = B_l, q_{l,1}^1 = A_l$, and $q_{l,m}^n = 0$ for $n > m$. The $r_{l,m}^n$ satisfy the same recurrences but with boundary conditions $r_{l,1}^0 = C_l$ and $r_{l,m}^n = 0$ for $n \ge m$.*

*Proof.* These recurrences are the shifted three-term recurrences (2.13) rewritten in terms of the Chebyshev coefficients of the polynomials by using the equations $x \cdot T_n = (T_{n+1} + T_{n-1})/2$ for $n > 0$ and $x \cdot T_0 = T_1$. ∎

For a fixed $l$, we can compute the $q_{l,m}^n$ and $r_{l,m}^n$ by increasing $m$, starting with the known values for $m = 0, 1$, and finishing with $m = M - 2$. For each $m$, we need to compute only the $q_{l,m}^n$ with $n \le m$ and the $r_{l,m}^n$ with $n < m$. The total number of flops of the precomputation of the Chebyshev coefficients in the general case is

$$T_{\text{precomp, term}} = 7M^2 - 16M - 15. \tag{B.4}$$

When the initial values $B_l$ are identically zero, the coefficients can be packed in alternating fashion into array **T**, as shown in Figure 3.3. In that case the cost is considerably lower, namely, $2.5M^2 - 3.5M - 12$.

The precomputed Chebyshev coefficients can be used to save the early stages in Algorithm B.1. If we continue the precomputation of the Chebyshev coefficients two steps more and finish with $m = M$ instead of $m = M - 2$, we then can switch directly to the precomputation of the point values at stage $K = M$, just after the forward Chebyshev transforms.

Parallelizing the precomputation of the Chebyshev coefficients is straightforward, since the computation for each $l$ is independent. Therefore, if $M \leq N/p$, both the termination and its precomputation are local operations.

**Acknowledgments.** We thank Dennis Healy, Daniel Rockmore, and Peter Kostelec for useful discussions. We thank the anonymous referees for valuable suggestions for improvement.

## REFERENCES

[1] N. Ahmed, T. Natarajan, and K. R. Rao, *Discrete cosine transform*, IEEE Trans. Comput., 23 (1974), pp. 90–93.

[2] B. K. Alpert and V. Rokhlin, *A fast algorithm for the evaluation of Legendre expansions*, SIAM J. Sci. Statist. Comput., 12 (1991), pp. 158–179.

[3] S. R. M. Barros and T. Kauranne, *On the parallelization of global spectral weather models*, Parallel Comput., 20 (1994), pp. 1335–1356.

[4] P. Barrucand and D. Dickinson, *On the associated Legendre polynomials*, in Orthogonal Expansions and Their Continuous Analogues, D. T. Haimo, ed., Southern Illinois University Press, Carbondale, IL, 1968, pp. 43–50.

[5] S. Belmehdi, *On the associated orthogonal polynomials*, J. Comput. Appl. Math., 32 (1990), pp. 311–319.

[6] R. H. Bisseling, *Basic techniques for numerical linear algebra on bulk synchronous parallel computers*, in Numerical Analysis and Its Applications, Lecture Notes in Comput. Sci. 1196, Springer-Verlag, Berlin, 1997, pp. 46–57.

[7] R. H. Bisseling, *BSPEDUpack,* http://www.math.uu.nl/people/bisseling/software.html, 2000.

[8] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping, *The Paderborn University BSP (PUB) library—design, implementation and performance*, in Proceedings of the 13th International Parallel Processing Symposium and the 10th Symposium on Parallel and Distributed Processing, CD-ROM, IEEE Computer Society, Los Alamitos, CA, 1999.

[9] G. L. Browning, J. J. Hack, and P. N. Swarztrauber, *A comparison of three numerical methods for solving differential equations on the sphere*, Monthly Weather Review, 117 (1989), pp. 1058–1075.

[10] T. S. Chihara, *An Introduction to Orthogonal Polynomials*, Gordon and Breach, New York, 1978.

[11] J. W. Cooley and J. W. Tukey, *An algorithm for the machine calculation of complex Fourier series*, Math. Comp., 19 (1965), pp. 297–301.

[12] G. A. Dilts, *Computation of spherical harmonic expansion coefficients via FFTs*, J. Comput. Phys., 57 (1985), pp. 439–453.

[13] J. R. Driscoll and D. M. Healy, Jr., *Computing Fourier transforms and convolutions on the 2-sphere*, Adv. Appl. Math., 15 (1994), pp. 202–250.

[14] J. R. Driscoll and D. M. Healy, Jr., *Computing Fourier transforms and convolutions on the 2-sphere*, extended abstract in Proceedings of the 30th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1989, pp. 344–349.

[15] J. R. Driscoll, D. M. Healy, Jr., and D. N. Rockmore, *Fast discrete polynomial transforms with applications to data analysis for distance transitive graphs*, SIAM J. Comput., 26 (1997), pp. 1066–1099.

[16] B. S. Duncan and A. J. Olson, *Approximation and characterization of molecular surfaces*, Biopolymers, 33 (1993), pp. 219–229.

[17] D. M. Healy, Jr. and P. T. Kim, *Spherical Deconvolution with Application to Geometric Quality Assurance*, Tech. report, Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH, 1993.

[18] D. M. Healy, Jr. and P. T. Kim, *An empirical Bayes approach to directional data and efficient computation on the sphere*, Ann. Statist., 24 (1996), pp. 232–254.

[19] D. M. Healy, Jr., S. S. B. Moore, and D. Rockmore, *Efficiency and Stability Issues in the Numerical Computation of Fourier Transforms and Convolutions on the* 2*-Sphere*, Tech. report PCS-TR94-222, Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH, 1994.

[20] D. M. Healy, Jr., D. Rockmore, P. J. Kostelec, and S. S. B. Moore, *FFTs for the* 2*-Sphere—Improvements and Variations*, Tech. report, Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH, 1998.

[21] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling, *BSPlib: The BSP programming library*, Parallel Comput., 24 (1998), pp. 1947–1980.

[22] J. M. D. Hill, S. R. Donaldson, and A. McEwan, *Installation and User Guide for the Oxford BSP Toolset (v*1.4*) Implementation of BSPlib*, Tech. report, Oxford University Computing Laboratory, Oxford, UK, 1998.

[23] M. A. Inda, *Constructing Parallel Algorithms for Discrete Transforms: From FFTs to Fast Legendre Transforms*, Ph.D. thesis, Department of Mathematics, Utrecht University, Utrecht, The Netherlands, 2000.

[24] D. K. Maslen, *A Polynomial Approach to Orthogonal Polynomial Transforms*, Preprint MPI/95-9, Max-Planck-Institut für Mathematik, Bonn, Germany, 1995.

[25] W. F. McColl, *Scalability, portability and predictability: The BSP approach to parallel programming*, Fut. Gen. Comp. Syst., 12 (1996), pp. 265–272.

[26] V. Lesur and D. Gubbins, *Evaluation of fast spherical transforms for geophysical applications*, Geophys. J. Int., 139 (1999), pp. 547–555.

[27] T. Ooura, *General purpose FFT (fast Fourier/cosine/sine transform) package,* http://momonga.t.u-tokyo.ac.jp/õoura/fft.html, 1998.

[28] S. A. Orszag, *Fast eigenfunction transforms*, in Science and Computers, G.-C. Rota, ed., Academic Press, Orlando, FL, 1986, pp. 23–30.

[29] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed., Cambridge University Press, Cambridge, UK, 1992.

[30] K. R. Rao and P. Yip, *Discrete Cosine Transform: Algorithms, Advantages, and Applications*, Academic Press, San Diego, CA, 1990.

[31] N. Shalaby, *Parallel Discrete Cosine Transforms: Theory and Practice*, Tech. report TR-34-95, Center for Research in Computing Technology, Harvard University, Cambridge, MA, 1995.

[32] N. Shalaby and S. L. Johnsson, *Hierarchical load balancing for parallel fast Legendre transforms*, in Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing, M. Heath et al., eds., SIAM, Philadelphia, 1997.

[33] G. Steidl and M. Tasche, *A polynomial approach to fast algorithms for discrete Fourier-cosine and Fourier-sine transforms*, Math. Comp., 56 (1991), pp. 281–296.

[34] L. G. Valiant, *A bridging model for parallel computation*, Comm. ACM, 33 (1990), pp. 103–111.

[35] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM, Philadelphia, 1992.