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.

1. INTRODUCTION

Discrete Legendre transforms 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, 8], shape analysis of molecular surfaces [14], statistical analysis of directional data [15], and geometric quality assurance [16].

A direct method for computing a discrete orthogonal polynomial transform such as the discrete Legendre transform for N data values requires a matrix-vector multiplication of $O(N^2)$ arithmetic operations, though several authors [2, 22] 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 [12, 13]. 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 reports discussing the theoretical parallelizability of the algorithm have already been written [17, 26]. 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 [17, 26], and present a full implementation of this algorithm. Another contribution is the method used to derive the algorithm. We present a method based on polynomial arithmetic to clarify the properties of

1991 *Mathematics Subject Classification.* 20C15; Secondary 65T10.

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

Parts of this work appeared in preliminary form in the proceedings of the ECMWF Workshop "Towards TeraComputing - The Use of Parallel Processors in Meteorology", Nov. 1998, Reading, UK, World Scientific Publishing Co, 1999.

Inda supported by a doctoral fellowship from CAPES, Brazil. Computer time on the Cray T3E of HPaC, Delft, funded by NCF, The Netherlands.

orthogonal polynomials used by the algorithm, and to remove some unnecessary assumptions made in [12] and [13].

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 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

First, we briefly review some basic concepts from the theory of orthogonal polynomials, that we use in the derivation of 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 usually nonnegative and continuous on $(-1,1)$.

Given an orthogonal polynomial sequence p_i, a positive integer N, and two sequences of numbers $x_0, \ldots, x_{N-1}$ and $w_0, \ldots, w_{N-1}$ called *sample points* and *sample weights*, respectively, we may 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

$$\hat{f}_l = \hat{f}(p_l) = \sum_{j=0}^{N-1} f_j p_l(x_j) w_j. \tag{2.1}$$

This computation may also be formulated as the multiplication of the matrix with elements $p_l(x_j)w_j$ in position (l,j) by the column vector $(f_0, \ldots, f_{N-1})$.

There are at least four distinct transforms that may be associated with an orthogonal polynomial sequence:

1. Given a sequence of function values $f_j = f(x_j)$ of a polynomial f of degree less than N, compute the coefficients of the expansion of f in the basis p_k. This expansion transform can also be viewed as a matrix-vector multiplication.
2. Given the coefficients of a polynomial f in the basis p_k, evaluate f at the points x_j. This is the inverse of 1.
3. The transpose of 1. In matrix terms, this is defined by the multiplication of the transpose matrix of 1 and the input vector.
4. The inverse transpose of 1.

The discrete orthogonal polynomial transform (2.1) is equivalent to transform 4 provided the weights w_j are identically 1.

Example 2.1 (Legendre polynomials)**.** The Legendre polynomials are orthogonal with respect to the uniform weight function 1 on $[-1,1]$, and may be defined recursively by

$$P_{l+1}(x) = \frac{2l+1}{l+1} x \cdot P_l - \frac{l}{l+1} P_{l-1}, \quad P_0(x) = 1, \quad P_1(x) = x. \tag{2.2}$$

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, which will be defined below. In this paper, we call the discrete orthogonal polynomial transform for the Legendre polynomials, with sample weights $\frac{1}{N}$ and with the Chebyshev points as sample points, the *discrete Legendre transform* (DLT).

Example 2.2 (Discrete cosine transform and Chebyshev transform)**.** The Chebyshev polynomials of the first kind are the sequence of orthogonal polynomials defined recursively by

$$T_{k+1}(x) = 2x \cdot T_k(x) - T_{k-1}(x), \quad T_0(x) = 1, \quad T_1(x) = x. \tag{2.3}$$

These are orthogonal with respect to the weight function $\omega(x) = \pi^{-1}(1-x^2)^{-\frac{1}{2}}$.

The *discrete cosine transform* (DCT, or DCT-II in the terminology of [29]) of size N is the discrete orthogonal polynomial transform for the Chebyshev polynomials, with sample weights 1, and sample points

$$x_j^N = \cos\frac{(2j+1)\pi}{2N}, \quad j = 0, \ldots, N-1, \tag{2.4}$$

which are called the *Chebyshev points*, and are the roots of T_N. The DCT is numbered 4 in the list above.

The *Chebyshev transform* is the expansion transform (numbered 1 above) for the Chebyshev polynomials at the Chebyshev points. The Chebyshev transform is the inverse transpose of the DCT defined above, but the relationship between Chebyshev points and Chebyshev polynomials implies that the cosine and Chebyshev transforms are even more closely related. Specifically, the coefficient of T_k in the expansion of a polynomial f with degree less than N and with function values $f_j = f(x_j^N)$, $0 \le j < N$, is $\frac{\epsilon_k}{N}\hat{f}(T_k)$, where

$$\epsilon_k = \begin{cases} 1 & \text{if } k = 0, \\ 2 & \text{if } k > 0. \end{cases} \tag{2.5}$$

Thus, to compute the Chebyshev transform, we can use a DCT and multiply the k-th coefficient by $\frac{\epsilon_k}{N}$. We denote the Chebyshev transform by a tilde. Therefore, we write

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

The inverse Chebyshev transform, numbered 2 above, is

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

A cosine transform can be carried out in $O(N \log N)$ arithmetic operations using an FFT [1, 29], or using the recent algorithm of Steidl and Tasche [27]. 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.

One of the important properties of orthogonal polynomials we will use is:

Lemma 2.3 (Gaussian quadrature). *Let $\{p_k\}$ be an orthogonal polynomial sequence for a nonnegative weight function $\omega(x)$, and $z_0^N, \ldots, z_{N-1}^N$ 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. [9, Theorem 6.1]. □

Example 2.4. 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

$$\frac{1}{\pi}\int_{-1}^{1} \frac{f(x)dx}{\sqrt{1-x^2}} = \frac{1}{N}\sum_{j=0}^{N-1} f(x_j^N), \tag{2.8}$$

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 (2.2) for the Legendre polynomials and (2.3) for the Chebyshev polynomials.

Lemma 2.5 (Three-term recurrence). *Let $\{p_k\}$ be an orthogonal polynomial sequence for a nonnegative weight function. Then $\{p_k\}$ satisfies a three-term recurrence relation*

$$p_{k+1}(x) = (A_k x + B_k)p_k(x) + C_k p_{k-1}(x), \tag{2.9}$$

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

Proof. See e.g. [9, Theorem 4.1]. □

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

Corollary 2.6 (Clebsch-Gordan). *Let $\{p_k\}$ be an orthogonal polynomial sequence with a nonnegative weight function. Then for any polynomial Q of degree m we have*

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

Proof. Rewrite the recurrence (2.9) in the form $x \cdot p_l = A_l^{-1}(p_{l+1} - B_l p_l - C_l p_{l-1})$, 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].

$$\begin{aligned} Q_{l,m}(x) &= (A_{l+m-1}x + B_{l+m-1})Q_{l,m-1}(x) + C_{l+m-1}Q_{l,m-2}(x), \\ Q_{l,0}(x) &= 1, \quad Q_{l,1}(x) = A_l x + B_l, \\ R_{l,m}(x) &= (A_{l+m-1}x + B_{l+m-1})R_{l,m-1}(x) + C_{l+m-1}R_{l,m-2}(x), \\ R_{l,0}(x) &= 0, \quad R_{l,1}(x) = C_l. \end{aligned} \tag{2.10}$$

Lemma 2.7 (Generalized three-term recurrence). *The associated polynomials satisfy* $\deg Q_{l,m} = m$*,* $\deg R_{l,m} \leq m-1$*, and for* $l \geq 1$ *and* $m \geq 0$*,*

$$p_{l+m} = Q_{l,m} \cdot p_l + R_{l,m} \cdot p_{l-1}. \tag{2.11}$$

Proof. Equation (2.11) follows by induction on m, with the case $m = 1$ being the original three-term recurrence (2.9). ☐

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 [12] 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 identically $\frac{1}{N}$. 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.

Our derivation of the Driscoll-Healy algorithm relies on the interpretation of the input data f_j of the transform (2.1) as the function values of a polynomial f of degree less than the problem size N. Thus f is defined to be the unique polynomial of degree less than N such that

$$f(x_j^N) = f_j, \quad j = 0, \ldots, N-1. \tag{2.12}$$

Using this notation and the relation

$$f \cdot p_{l+m} = Q_{l,m} \cdot (f \cdot p_l) + R_{l,m} \cdot (f \cdot p_{l-1}) \tag{2.13}$$

derived from the three-term recurrence (2.11), we may formulate a strategy for computing all the polynomials $f \cdot p_l$, $0 \leq l < N$, in $\log_2 N$ stages:

- At stage 0, compute $f \cdot p_0$ and $f \cdot p_1$.
- At stage 1, use (2.13) with $l = 1$ and $m = \frac{N}{2} - 1$ or $m = \frac{N}{2}$, to compute
  $f \cdot p_{\frac{N}{2}} = Q_{1,\frac{N}{2}-1} \cdot (f \cdot p_1) + R_{1,\frac{N}{2}-1} \cdot (f \cdot p_0)$ and
  $f \cdot p_{\frac{N}{2}+1} = Q_{1,\frac{N}{2}} \cdot (f \cdot p_1) + R_{1,\frac{N}{2}} \cdot (f \cdot p_0)$.
- In general, at each stage $k, 1 \leq k < \log_2 N$, similarly as before use (2.13) with $l = 2q(N/2^k) + 1$, $0 \leq q < 2^{k-1}$, and $m = N/2^k - 1, N/2^k$, 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

$$\mathcal{T}_n f = \sum_{k=0}^{n-1} b_k T_k. \tag{2.14}$$

The important properties of $\mathcal{T}_n$ are given in Lemma 2.8.

Lemma 2.8. *Let f and Q be polynomials. Then, the following holds.*

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* $M \leq K$, *then* $\mathcal{T}_M \mathcal{T}_K = \mathcal{T}_M$.
3. *If* $\deg Q \leq m \leq K$, *then* $\mathcal{T}_{K-m}(f \cdot Q) = \mathcal{T}_{K-m}[(\mathcal{T}_K f) \cdot Q]$.

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 $\deg Q \leq m \leq K$. By Corollary 2.6, $T_k \cdot Q$ is in the linear span of $T_{k-m}, \ldots, T_{k+m}$, so $\mathcal{T}_{K-m}(T_k \cdot Q) = 0$ for $k \geq K$. Therefore

$$\mathcal{T}_{K-m}(f \cdot Q) = \mathcal{T}_{K-m}\left(\sum_{k\geq 0} b_k T_k \cdot Q\right) = \mathcal{T}_{K-m}\left(\sum_{k=0}^{K-1} b_k T_k \cdot Q\right) = \mathcal{T}_{K-m}[(\mathcal{T}_K f) \cdot Q]$$

□

As a corollary of part 1 of Lemma 2.8, we see how we can retrieve the discrete orthogonal polynomial transform from the $f \cdot p_l$'s computed by the strategy above, by a simple truncation.

Corollary 2.9. *Let f be the unique polynomial of degree less than N such that $f(x_j^N) = f_j$, $0 \leq j < N$. 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.

Proof. This follows from the definition of discrete orthogonal polynomial transforms, the Gaussian quadrature rule (2.8) for Chebyshev polynomials applied to the function $f \cdot p_l$, and Lemma 2.8,

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

□

The key property of the truncation operators $\mathcal{T}_n$ is the 'aliasing' property (3), 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 2.8 with $m = l$ and $K = l+1$ to get

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

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

The Driscoll-Healy algorithm follows the strategy described above, but computes truncated polynomials

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

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 (2.13) directly, each stage uses truncated versions. Specifically, (2.13) and part 3 of Lemma 2.8 imply the following recurrences for the Z_l^K:

$$Z_{l+m}^{K-m} = \mathcal{T}_{K-m}[Z_l^K \cdot Q_{l,m} + Z_{l-1}^K \cdot R_{l,m}], \tag{2.15}$$

$$Z_{l+m-1}^{K-m} = \mathcal{T}_{K-m}[Z_l^K \cdot Q_{l,m-1} + Z_{l-1}^K \cdot R_{l,m-1}], \tag{2.16}$$

for $K \ge m$. We will use the special case with $2K$ instead of K and $m = K$,

$$Z_{l+K}^{K} = \mathcal{T}_{K}[Z_l^{2K} \cdot Q_{l,K} + Z_{l-1}^{2K} \cdot R_{l,K}], \tag{2.17}$$

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

The algorithm proceeds in $\log_2 N + 1$ stages, as shown in Algorithm 2.1.

Algorithm 2.1 The Driscoll-Healy algorithm. (Polynomial version.)

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

OUTPUT $(\hat{f}_0, \dots, \hat{f}_{N-1})$: 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 (2.17) and (2.18) to compute new polynomials.
        $Z_{l+K}^K \leftarrow \mathcal{T}_K\left(Z_l^{2K} \cdot Q_{l,K} + Z_{l-1}^{2K} \cdot R_{l,K}\right)$
        $Z_{l+K-1}^K \leftarrow \mathcal{T}_K\left(Z_l^{2K} \cdot Q_{l,K-1} + Z_{l-1}^{2K} \cdot R_{l,K-1}\right)$
      (b) Truncate old polynomials.
        $Z_l^K \leftarrow \mathcal{T}_K Z_l^{2K}$
        $Z_{l-1}^K \leftarrow \mathcal{T}_K Z_{l-1}^{2K}$

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

The organization of the computation is illustrated in Fig. 1. The vertical lines indicate the truncated polynomials Z_l^K and their height indicates the number of Chebyshev coefficients initially appearing. At each stage the polynomials computed are truncated at the height indicated by the grayscales.

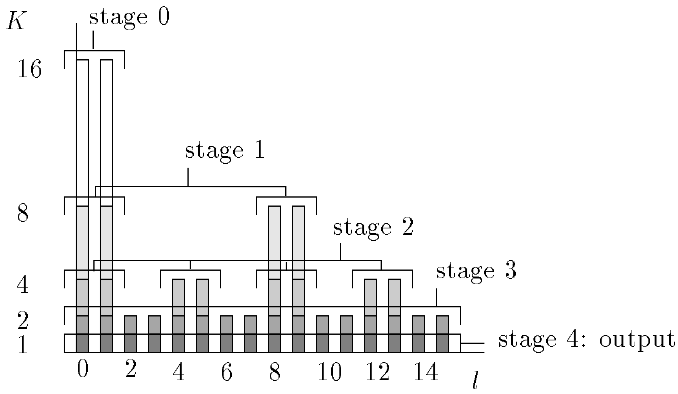

FIGURE 1. The computation of Z_l^K for $N = 16$.

2.3. **Data representation and recurrence procedure.** The description of the Driscoll-Healy algorithm we have given is incomplete. We still need to specify how to represent the polynomials in the algorithm, and 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

$$Z_l^K = \sum_{n=0}^{K-1} z_n^l T_n, \tag{2.19}$$

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 must 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 representation. 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, though $f \cdot p_1$ may have degree N, rather than $N - 1$. This poses no problem, because applying part 4 of Lemma 2.10 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:

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 (2.17) and (2.18) 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 (2.17) and (2.18) 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* $\mathcal{S}_K$, for positive integers K. For any polynomial h, the Lagrange interpolation polynomial $\mathcal{S}_K h$ is the polynomial of degree less than K which agrees with h at the points $x_0^K, \ldots, x_{K-1}^K$. The important properties of $\mathcal{S}_K$ are given in Lemma 2.10.

Lemma 2.10. *Let g and h be polynomials. Then, the following holds.*

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

Proof. Parts 1 and 2 are easy. To prove part 3 assume that $\deg h \le K+m$. By long division, there is a polynomial Q of degree at most m such that $h = \mathcal{S}_K h + T_K \cdot Q$. Applying $\mathcal{T}_{K-m}$, and using part 3 of Lemma 2.8, we obtain

$$\mathcal{T}_{K-m}\mathcal{S}_K h = \mathcal{T}_{K-m} h - \mathcal{T}_{K-m}[T_K \cdot Q] = \mathcal{T}_{K-m} h - \mathcal{T}_{K-m}[(\mathcal{T}_K T_K) \cdot Q] = \mathcal{T}_{K-m} h,$$

since $\mathcal{T}_K T_K = 0$. For part 4 we note that $\deg \mathcal{S}_K h < K$, and use part 3 with $m = 0$ to get $\mathcal{S}_K h = \mathcal{T}_K \mathcal{S}_K h = \mathcal{T}_K h$. □

From the recurrences (2.17) and (2.18) and part 3 of Lemma 2.10 with $2K$ instead of K and $m = K$ it follows that

$$Z_{l+K}^K = \mathcal{T}_K[\mathcal{S}_{2K}(Z_l^{2K} \cdot Q_{l,K}) + \mathcal{S}_{2K}(Z_{l-1}^{2K}, \cdot R_{l,K})] \tag{2.20}$$

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

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

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

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.

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^{2K} and Z_{l-1}^{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}^K and Z_{l+K-1}^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**
   $u_j \leftarrow Q_{l,K}(x_j^{2K})\, f_j + R_{l,K}(x_j^{2K})\, g_j$
   $v_j \leftarrow Q_{l,K-1}(x_j^{2K})\, f_j + R_{l,K-1}(x_j^{2K})\, g_j$
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 $(\tilde{v}_K, \ldots, \tilde{v}_{2K-1})$.

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 take linear combinations of the vectors Z_l^K computed so far to obtain the final result.

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

$$Q_{l,m} = \sum_{n=0}^{m} q_{l,m}^n T_n, \quad R_{l,m} = \sum_{n=0}^{m-1} r_{l,m}^n T_n. \tag{2.22}$$

The problem of finishing the computation at the end of stage $k = \log_2 \frac{N}{M}$, 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 2.11, which follows. The second part of this lemma can be used to halve the number of computations, in the common case where the polynomial recurrence (2.9) has a coefficient $B_k = 0$ for all k.

Lemma 2.11. 1. *If $l \ge 1$ and $0 \le m < M$, then*

$$\hat{f}_{l+m} = \sum_{n=0}^{m} \frac{1}{\epsilon_n}(z_n^l q_{l,m}^n + z_n^{l-1} r_{l,m}^n). \tag{2.23}$$

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. By (2.15) with $K = M$, $\hat{f}_{l+m} = Z_{l+m}^1$ 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 in terms of the Chebyshev coefficients of Z_l^M, Z_{l-1}^M and of $Q_{l,m}, R_{l,m}$, we substitute the expansions (2.19) and (2.22), 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 $R_{l,m}$. $\square$

Assuming that the assumptions of the second part of the lemma are valid, i.e., each term of (2.23) has either $q_{l,m}^n = 0$ or $r_{l,m}^n = 0$, and that the factor $1/\epsilon_n$ is absorbed in the precomputed values $q_{l,m}^n$ and $r_{l,m}^n$, the total number of flops to compute $\hat{f}_{l+m}$ is $2m+1$.

2.5. **Complexity of the algorithm.** Algorithm 2.3 gives the 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

$$
\begin{aligned}
T_{\mathrm{Driscoll-Healy}} = & N[2\alpha(\log_2^2 N - \log_2^2 M) + (4\alpha + 4\beta + 6)\log_2 N - \\
& (2\alpha + 4\beta + 6)\log_2 M + M + 2\beta].
\end{aligned}
\tag{2.24}
$$

Algorithm 2.3 Driscoll-Healy algorithm. (Final version.)

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) $(z_0^0, \ldots, z_{N-1}^0) \leftarrow \mathrm{Chebyshev}(f_0 p_0, \ldots, f_{N-1} p_0)$.
   (b) $(z_0^1, \ldots, z_{N-1}^1) \leftarrow \mathrm{Chebyshev}(f_0 p_1(x_0^N), \ldots, f_{N-1} p_1(x_{N-1}^N))$.

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

$\log_2 \frac{N}{M} + 1$. Compute 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 2.11 depends on α and β and can be obtained theoretically. The derivative of (2.24) according to M equals zero if and only if

$$M \ln^2 2 - 4\alpha \ln M = (2\alpha + 4\beta + 6) \ln 2. \tag{2.25}$$

In our implementation $\alpha = 2.125$ and $\beta = 5$, thus the minimum is $M = 128$. In practice, the optimal choice of M may also depend on the architecture of the machine used.

3. The basic parallel algorithm and its implementation

We designed our parallel algorithm using the BSP model which gives a simple and effective way to produce portable parallel algorithms. It does not depend on a specific computer architecture, and it provides 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 Legendre transform, instead of the more general discrete orthogonal polynomial transform.

3.1. The bulk synchronous parallel model.

In the BSP model [28], a computer consists of a set of p processors, each with its own memory, connected by a communication network that allows processors to access the private memories of other processors. In this 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, in each case followed by a global synchronization. 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;
- 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 [7, 19]. The second reference describes BSPlib, a standard library defined in May 1997 which enables parallel programming in BSP style.

3.2. Data structures and data distributions.

Each processor in the BSP model has its own private memory, so the design of a BSP algorithm requires choosing how to distribute the elements of the data structures used in it over the processors.

At each stage k, $1 \le k \le \log_2 \frac{N}{M}$, 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 Fig. 2, suggests that we distribute all the vectors by blocks, i.e., we 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. Formally, 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.

Since both N and p in Definition 3.1 are powers of two, the block size is $b = N/p$.

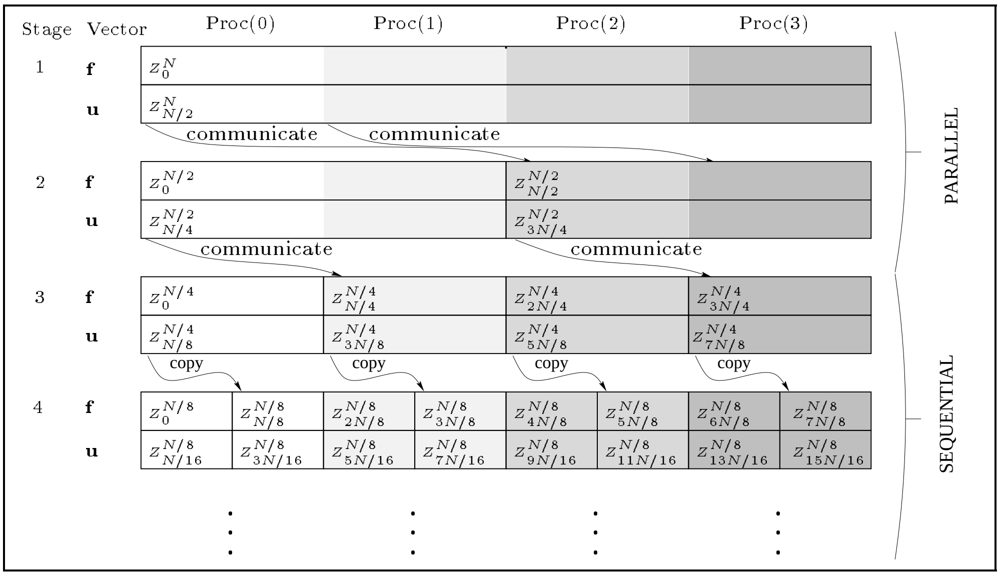

Figure 2. Main data structure and data distribution in the parallel FLT algorithm for $p = 4$. Arrays $\mathbf{f}$ and $\mathbf{g}$ contain the Chebyshev coefficients of the polynomials Z_l^{2K} and Z_{l+1}^{2K}, which are already available at the start of the stage. Arrays $\mathbf{u}$ and $\mathbf{v}$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

3.2.1. *Distribution of the precomputed data.* The precomputed data required to perform the recurrence of stage k are stored in two two-dimensional arrays $\mathbf{Q}$ and

$\mathbf{R}$, each of size $2\log_2 \frac{N}{M} \times N$. Each pair of rows in $\mathbf{Q}$ stores data needed for one stage k, by

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

for $l = 0, 2K, \ldots, N-2K$, $j = 0, 1, \ldots, 2K-1$, where $K = N/2^k$. Thus, polynomials $Q_{l+1,K}$ are stored in row $2k-2$ and polynomials $Q_{l+1,K-1}$ in row $2k-1$. This is shown in Fig. 3. The polynomials $R_{l+1,K}$ and $R_{l+1,K-1}$ are stored in the same way in array $\mathbf{R}$. Note that the indexing of the implementation arrays starts at zero. Each row of $\mathbf{R}$ and $\mathbf{Q}$ is distributed by the block distribution, so that $\mathbf{R}[i,j], \mathbf{Q}[i,j] \in \mathrm{Proc}(j \text{ div } \frac{N}{p})$, and the recurrence is a local operation.

k	$K, K-1$	Proc(0)	Proc(1)	Proc(2)	Proc(3)
1	32	$l = 0$			
	31	$j = 0, \ldots, 63$			
2	16	$l = 0$		$l = 32$	
	15	$j = 0, \ldots, 31$		$j = 0, \ldots, 31$	
3	8	$l = 0$	$l = 16$	$l = 32$	$l = 48$
	7	$j = 0, \ldots, 15$	$j = 0, \ldots, 15$	$j = 0, \ldots, 15$	$j = 0, \ldots, 15$

FIGURE 3. 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 two-dimensional arrays $\mathbf{Q}$ and $\mathbf{R}$. Each pair of rows in an array stores the data needed for one stage k.

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 $\mathbf{T}$ 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 $\mathbf{T}$. 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 similar to that commonly used to store lower triangular matrices. By the second part of Lemma 2.11, either $q_{l,m}^n = 0$ or $r_{l,m}^n = 0$ for each n and m, so we only need to store the value that can be nonzero. Since this depends on whether $n - m$ is even or odd, we obtain an alternating pattern of $q_{l,m}^n$'s and $r_{l,m}^n$'s. Fig. 4 illustrates this data structure.

The termination stage is local if $M \le N/p$, so that the input and output vectors are local. This means that each row of $\mathbf{T}$ must be assigned to one processor, namely to the processor that holds the subvectors for the corresponding value of l. The distribution $\mathbf{T}[i,j] \in \mathrm{Proc}(i \text{ div } \frac{N}{pM})$ achieves this. As a result, the N/M rows of $\mathbf{T}$ are distributed in consecutive blocks of rows.

3.3. **The basic parallel algorithm.** In order to formulate our basic parallel algorithm we introduce the following conventions:

- **Processor identification.** The total number of processors is p. The processor identification number is s, with $0 \le s < p$.
- **Supersteps.** The labels on the left-hand side indicate a superstep and its type: (Cp) computation superstep, (Cm) communication superstep, (CpCm) subroutine containing both computation and communication supersteps. Each communication superstep ends with an explicit synchronization. Supersteps inside loops are executed repeatedly, though they are numbered only once.

	$m=1$	$m=2$	$m=3$	$m=4$	$m=5$	$m=6$	
$l=1$	r^0 q^1	q^0 r^1 q^2	r^0 q^1 r^2 q^3	q^0 r^1 q^2 r^3 q^4	r^0 q^1 r^2 q^3 r^4 q^5	q^0 r^1 q^2 r^3 q^4 r^5 q^6	Proc(0)
$l=9$	r^0 q^1	q^0 r^1 q^2	r^0 q^1 r^2 q^3	q^0 r^1 q^2 r^3 q^4	r^0 q^1 r^2 q^3 r^4 q^5	q^0 r^1 q^2 r^3 q^4 r^5 q^6	
$l=17$	r^0 q^1	q^0 r^1 q^2	r^0 q^1 r^2 q^3	q^0 r^1 q^2 r^3 q^4	r^0 q^1 r^2 q^3 r^4 q^5	q^0 r^1 q^2 r^3 q^4 r^5 q^6	Proc(1)
$l=25$	r^0 q^1	q^0 r^1 q^2	r^0 q^1 r^2 q^3	q^0 r^1 q^2 r^3 q^4	r^0 q^1 r^2 q^3 r^4 q^5	q^0 r^1 q^2 r^3 q^4 r^5 q^6	
$l=33$	r^0 q^1	q^0 r^1 q^2	r^0 q^1 r^2 q^3	q^0 r^1 q^2 r^3 q^4	r^0 q^1 r^2 q^3 r^4 q^5	q^0 r^1 q^2 r^3 q^4 r^5 q^6	Proc(2)
$l=41$	r^0 q^1	q^0 r^1 q^2	r^0 q^1 r^2 q^3	q^0 r^1 q^2 r^3 q^4	r^0 q^1 r^2 q^3 r^4 q^5	q^0 r^1 q^2 r^3 q^4 r^5 q^6	
$l=49$	r^0 q^1	q^0 r^1 q^2	r^0 q^1 r^2 q^3	q^0 r^1 q^2 r^3 q^4	r^0 q^1 r^2 q^3 r^4 q^5	q^0 r^1 q^2 r^3 q^4 r^5 q^6	Proc(3)
$l=57$	r^0 q^1	q^0 r^1 q^2	r^0 q^1 r^2 q^3	q^0 r^1 q^2 r^3 q^4	r^0 q^1 r^2 q^3 r^4 q^5	q^0 r^1 q^2 r^3 q^4 r^5 q^6	

FIGURE 4. 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 $r^n_{l,m}$.

- **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 indicated using

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

  The Put 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

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

  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 2 vectors instead of one. For example

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

  is an abbreviation for

  $$\mathbf{f_i} \leftarrow \mathrm{Copy}(n, \mathbf{u_k})$$
  $$\mathbf{g_j} \leftarrow \mathrm{Copy}(n, \mathbf{v_l})$$

- **Fast Chebyshev transform.** The subroutine

  $$\mathrm{BSP\_FChT}(s0, s1, p1, sign, n, \mathbf{f})$$

  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 $p1$ processors starting from Proc($s0$) work together; $s1$ with $0 \le s1 < p1$ denotes the local processor number within the group. For a group size $p1 = 1$, this subroutine reduces to the sequential fast Chebyshev transform algorithm.

- **Truncation.** The operation

$$\mathbf{f} \leftarrow \text{BSP\_Trunc}(s0, s1, p1, b, K, \mathbf{u})$$

denotes the truncation of all the $N/(2K)$ polynomials stored in $\mathbf{f}$ and $\mathbf{u}$ by copying the first K Chebyshev coefficients of the polynomials stored in $\mathbf{u}$ into the memory space of the last K Chebyshev coefficients of the corresponding polynomials stored in $\mathbf{f}$. A group of $p1$ processors starting from Proc($s0$) work together to truncate one polynomial; $s1$ with $0 \le s1 < p1$ denotes the local processor number within the group. When $p1 = 1$ the block size $b = \frac{N}{p}$ is larger than K, and one processor is in charge of the truncation of one or more polynomials. Algorithm 3.1 gives a description of this operation. In Fig. 2, this operation is depicted by arrows.

Algorithm 3.1 Truncation using the block distribution.

CALL $\mathbf{f} \leftarrow \text{BSP\_Trunc}(s0, s1, p1, b, K, \mathbf{u})$.

DESCRIPTION

$s \leftarrow s0 + s1$
if $p1 = 1$ **then**
  for $l = s \cdot b$ **to** $(s+1)b - 2K$ **step** $2K$ **do**
    $\mathbf{f_{l+K}} \leftarrow \text{Copy}(K, \mathbf{u_l})$
else
  if $s1 < \frac{p1}{2}$ **then**
    $\mathbf{f_{s \cdot b+K}} \leftarrow \text{Put}(s + \frac{p1}{2}, b, \mathbf{u_{s \cdot b}})$

The basic template for the fast Legendre transform is presented as Algorithm 3.2. At each stage $k \le \log_2 \frac{N}{M}$, there are 2^{k-1} independent problems, one for each l. For $k \le \log_2 p$, there are more processors than problems, so that the processors will have to work in groups. Each group of $p1 = p/2^{k-1} > 1$ processors handles one subvector of size $2K$, $K = N/2^k$; each processor handles a block of $2K/p1 = N/p$ vector components. In this case, the l-loop has only one iteration, namely $l = s0 \cdot N/p$, and the j-loop has N/p iterations, starting with $j = s1 \cdot N/p$, so that the indices $l + j$ start with $(s0 + s1)N/p = s \cdot N/p$, and end with $(s0 + s1)N/p + N/p - 1 = (s+1)N/p - 1$. Inter-processor 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 $2K \le N/p$. In that case, $p1 = 1$, $s0 = s$, and $s1 = 0$, and each processor has one or more 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, $sN/p + 2K, \ldots, (s+1)N/p - 2K$, 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 $f_j \cdot x_j^N, 0 \le j < N$, see Subsection 2.3.1. The values $\mathbf{R}[i,j] + \mathbf{Q}[i,j]x_j^N$ for $i = 0, 1$ can be precomputed and stored so that the recurrences only require one multiplication by f_j. 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

Algorithm 3.2 Basic parallel template for the fast Legendre transform.

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 $p \leq N/2$).
N: Transform size (N is a power of 2 with $N \geq 4$).
M: Termination block size (M is a power of 2 with $M \leq \min(N/2, N/p)$).
$\mathbf{f}$: (Input) $\mathbf{f} = (f_0, \ldots, f_{N-1})$: Real vector to be transformed.
(Output) $\mathbf{f} = (\hat{f}_0, \ldots, \hat{f}_{N-1})$: Transformed vector.
Block distributed: $f_j \in \text{Proc}(j \text{ div } \frac{N}{p})$.

STAGE 1:

(1^{Cp}) **for** $j = s\frac{N}{p}$ **to** $(s+1)\frac{N}{p} - 1$ **do**
  $g_j \leftarrow x_j^N f_j$
  $u_j \leftarrow (\mathbf{R}[0, j] + \mathbf{Q}[0, j]x_j^N)f_j$
  $v_j \leftarrow (\mathbf{R}[1, j] + \mathbf{Q}[1, j]x_j^N)f_j$
(2^{CpCm}) BSP_FChT2$(0, s, p, 1, N, \mathbf{f}, \mathbf{g})$
  BSP_FChT2$(0, s, p, 1, N, \mathbf{u}, \mathbf{v})$
(3^{Cm}) $(\mathbf{f}, \mathbf{g}) \leftarrow$ BSP_Trunc2$(0, s, p, \frac{N}{p}, \frac{N}{2}, \mathbf{u}, \mathbf{v})$

STAGE k:

  for $k = 2$ **to** $\log_2 \frac{N}{M}$ **do**
(4^{Cp}) $K \leftarrow \frac{N}{2^k}$
   $p1 \leftarrow \max(\frac{p}{2^{k-1}}, 1)$
   $s0 \leftarrow (s \text{ div } p1)p1$
   $s1 \leftarrow s \bmod p1$
   $(\mathbf{u}_{s\frac{N}{p}}, \mathbf{v}_{s\frac{N}{p}}) \leftarrow \text{Copy2}(\frac{N}{p}, \mathbf{f}_{s\frac{N}{p}}, \mathbf{g}_{s\frac{N}{p}})$
   for $l = s0\frac{N}{p}$ **to** $(s0+1)\frac{N}{p} - \frac{2K}{p1}$ **step** $\frac{2K}{p1}$ **do**
(5^{CpCm}) BSP_FChT2$(s0, s1, p1, -1, 2K, \mathbf{u}_l, \mathbf{v}_l)$
(6^{Cp}) **for** $j = s1\frac{N}{p}$ **to** $s1\frac{N}{p} + \frac{2K}{p1} - 1$ **do**
     $a1 \leftarrow \mathbf{R}[2k-2, l+j]u_{l+j} + \mathbf{Q}[2k-2, l+j]v_{l+j}$
     $a2 \leftarrow \mathbf{R}[2k-1, l+j]u_{l+j} + \mathbf{Q}[2k-1, l+j]v_{l+j}$
     $u_{l+j} \leftarrow a1$
     $v_{l+j} \leftarrow a2$
(7^{CpCm}) BSP_FChT2$(s0, s1, p1, 1, 2K, \mathbf{u}_l, \mathbf{v}_l)$
(8^{Cm}) $(\mathbf{f}, \mathbf{g}) \leftarrow$ BSP_Trunc2$(s0, s1, p1, \frac{N}{p}, K, \mathbf{u}, \mathbf{v})$

STAGE $\log_2 \frac{N}{M} + 1$:

(9^{Cp}) **for** $l = s\frac{N}{p}$ **to** $(s+1)\frac{N}{p} - M$ **step** M **do**
  $\mathbf{f}_l \leftarrow \text{Terminate}(l, M, \mathbf{f}_l, \mathbf{g}_l)$

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 to 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 termination procedure, superstep 10, is a direct implementation of Lemma 2.11 using the data structure $\mathbf{T}$ described in Subsection 3.2.1. Superstep 10 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 will be discussed in Subsection 4.4.

4. Improvements of the parallel algorithm

4.1. **Fast Chebyshev transform 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, 23, 24, 27]. Parallel algorithms or implementations have been less intensively studied, see [25] 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 4.4.6 of Van Loan [29] and the standard algorithm for computing the FFTs of two real input vectors at thematr same time (see e.g. [23]).

The use of an FFT-based algorithm is advantageous because the bulk of the computation is in the FFT and because good FFT implementations are ubiquitous. Since the FFT is separate module, it can easily be replaced, for instance by a new, more efficient, FFT subroutine.

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 part of a complex vector $(\mathbf{x} + i\,\mathbf{y})$. The algorithm is divided in 3 phases. Phase 1, the packing of the input data into an auxiliary complex vector $\mathbf{z}$ of length N, is a simple permutation,

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

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

$$(4.2) \qquad Z_k = \sum_{j=0}^{N-1} z_j e^{\frac{2\pi ijk}{N}}, \qquad 0 \leq k < N.$$

This phase takes $4.25N \log_2 N$ flops if we use a radix-4 algorithm [29]. Finally, in phase 3 we obtain the Chebyshev transform by

$$(4.3) \qquad \begin{cases} \tilde{x}_k = \dfrac{\epsilon_k}{2N} \mathrm{Re}\left(e^{\frac{\pi ik}{2N}}(Z_k + \overline{Z}_{N-k})\right), \\ \tilde{y}_k = \dfrac{\epsilon_k}{2N} \mathrm{Im}\left(e^{\frac{\pi ik}{2N}}(Z_k - \overline{Z}_{N-k})\right), \qquad 0 \leq k < N, \end{cases}$$

where $\frac{\epsilon_k}{N}$ is the normalization factor needed to get the Chebyshev transform from the cosine transform. This phase 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 transforms is best made in two steps. First, we prove that

$$(4.4)\quad \frac{1}{2}(Z_k + \overline{Z}_{N-k}) = \sum_{j=0}^{N-1} \mathrm{Re}(z_j) e^{\frac{2\pi ijk}{N}} = \sum_{j=0}^{N/2-1} \left(x_{2j} e^{\frac{2\pi ijk}{N}} + x_{2j+1} e^{-\frac{2\pi i(j+1)k}{N}} \right),$$

and

(4.5)

$$-\frac{i}{2}(Z_k - \overline{Z}_{N-k}) = \sum_{j=0}^{N-1} \mathrm{Im}(z_j) e^{\frac{2\pi ijk}{N}} = \sum_{j=0}^{N/2-1} \left(y_{2j} e^{\frac{2\pi ijk}{N}} + y_{2j+1} e^{-\frac{2\pi i(j+1)k}{N}} \right).$$

Second, we substitute (4.4) and (4.5) into (4.3) to obtain the desired equality (2.6). Note that (4.3) requires that Z_N be defined, and therefore we extend definition (4.2) to any integer k. Because the extended definition is N-periodic, we can obtain any value Z_k from the computed values $Z_0, \ldots, Z_{N-1}$.

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. Phase 3 is inverted by packing $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{y}}$ into the auxiliary complex vector $\mathbf{Z}$:

$$(4.6)\qquad \begin{cases} Z_0 = N(\tilde{x}_0 + i\,\tilde{y}_0), & \\ Z_k = \dfrac{N}{2} e^{-\frac{\pi ik}{2N}} \left((\tilde{x}_k + i\,\tilde{y}_k) + i(\tilde{x}_{N-k} + i\,\tilde{y}_{N-k}) \right), & 1 \le k < N. \end{cases}$$

To invert phase 2, an inverse complex FFT is computed,

$$(4.7)\qquad z_k = \frac{1}{N} \sum_{j=0}^{N-1} Z_j e^{-\frac{2\pi ijk}{N}}, \quad 0 \le k < N.$$

The inverse of phase 1 is again a permutation,

$$(4.8)\qquad \begin{cases} x_{2j} = \mathrm{Re}(z_j), & y_{2j} = \mathrm{Im}(z_j), \\ x_{2j+1} = \mathrm{Re}(z_{N-j-1}), & y_{2j+1} = \mathrm{Im}(z_{N-j-1}), \quad 0 \le j < N/2. \end{cases}$$

The cost of the inverse FChT algorithm is the same as that of the FChT algorithm, provided the scalings of (4.6) and (4.7) are combined.

An efficient parallelization of this algorithm involves breaking open the parallel FFT inside the FChT2 and merging parts of the FFT with the surrounding computations. In the following subsection we explain the parallelization process.

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,

$$(4.9)\qquad F_N = A_N \cdots A_8 A_4 A_2 P_N,$$

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_K are defined by

$$(4.10)\qquad A_K = I_{N/K} \otimes B_K, \quad K = 2, 4, 8, \ldots, N,$$

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

This matrix decomposition naturally leads to the *radix-2 FFT* algorithm [10, 29]. 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_K. The multiplications are carried out in $\log_2 N$ stages, each with N/K times a butterfly computation. One butterfly computation modifies $K/2$ pairs $(z_j, z_{j+K/2})$ at distance $K/2$ by adding a multiple of $z_{j+K/2}$ to z_j and subtracting the same multiple.

Parallel radix-2 FFTs have already been discussed in the literature, see e.g. [21]. For simplicity, in our exposition we restrict ourselves to FFT algorithms where $p \leq \sqrt{N}$. This class of algorithms uses the block distribution to perform the short distance butterflies with $K \leq N/p$ and the cyclic distribution to perform the long distance butterflies with $K > N/p$. Figure 5a gives an example of the cyclic distribution which is formally 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 $\mathrm{Proc}(j \mod p)$ 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 $K = 2, 4, \ldots, N/p$.
   (c) Permute $\mathbf{z}$ to the cyclic distribution.
   (d) Perform the long distance butterflies of size $K = 2N/p, 4N/p, \ldots, N$.
   (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 (2a), thus saving one complete permutation of BSP cost $2\frac{N}{p}g + l$. The same can be done for (2e) and (3a).

4.2.2. *Increasing the symmetry of the cyclic distribution.* We can eliminate permutation (2e)/(3a) completely by restricting the number of processors slightly further to $p \leq \sqrt{N/2}$, and permuting the vector $\mathbf{z}$ in phase (2c) from block distribution to a slightly modified cyclic distribution, the zig-zag cyclic distribution, shown in Fig. 5b, and formally 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

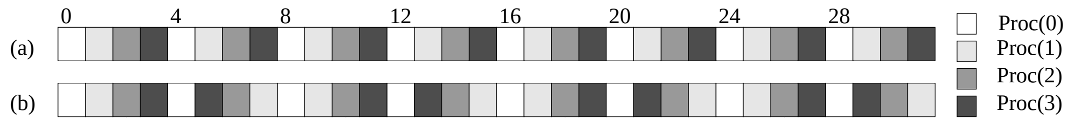

FIGURE 5. (a) Cyclic distribution and (b) zig-zag cyclic distribution for a vector of size 32 distributed over 4 processors.

is stored in $\mathrm{Proc}(j \mod p)$ if $j \mod 2p < p$ and in $\mathrm{Proc}(-j \mod p)$ otherwise, and has local index $j' = j \operatorname{div} p$.

In this distribution, both the components j and $j + K/2$ needed by the butterfly operations with $K > N/p$ and the components j and $N - j$ needed by the extract operation are in the same processor; thus we avoid the permutation (2e)/(3a) 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 $A_{2K}A_K$ together and perform them as one operation. The butterflies have the form $B_{2K}(I_2 \otimes B_K)$, which is a $2K \times 2K$ matrix consisting of 4×4 blocks, each a $K/2 \times K/2$ diagonal submatrix. This matrix is a symmetrically permuted version of the radix-4 butterfly matrix [29]. This approach gives 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. This reduces α from 2.5 to 2.125.

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. **Optimization of the main loop.** Here we show how to reduce the communication even further by giving up the block distribution in the main loop of the FLT algorithm. This discussion 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. Note that in these stages a group of $p1 = p/2^{k-1} > 1$ processors handles only one subproblem of size $2K = 2N/2^k$ corresponding to $l = s0\frac{N}{p}$. Because the operations executed on $\mathbf{f}$ and $\mathbf{u}$ are also executed on vectors $\mathbf{g}$ and $\mathbf{v}$, we omit $\mathbf{g}$ and $\mathbf{v}$ from our discussion.

4.3.1. *Modifying the truncation 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 following sequence of operations will then be replaced by a new procedure.

- permute from zig-zag cyclic to block distribution in stage k;
- truncate at the end of stage k;
- copy at the beginning of stage $k+1$;
- permute from block to zig-zag cyclic distribution in stage $k+1$.

In the new approach, we assume that the last K elements of $\mathbf{f_l}$ and $\mathbf{u_l}$ have already been discarded, so that $\mathbf{f_l}$ and $\mathbf{u_l}$ are in the zig-zag cyclic distribution of K (instead of $2K$) elements over $p1$ processors. Note that for $\mathbf{u_l}$ these elements have not even been computed, see Subsection 4.2.4. The new procedure follows.

1. Keep 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) Vector $\mathbf{f_l}$ receives the first $K/2$ elements of vector $\mathbf{u_l}$ redistributed by the zig-zag cyclic distribution over the first $\frac{p1}{2}$ processors.
   (b) Vector $\mathbf{f_{l+K}}$ receives the first $K/2$ elements of vector $\mathbf{u_{l+K}}$ redistributed by the zig-zag cyclic distribution over the next $\frac{p1}{2}$ processors.

The new procedure is illustrated in Fig. 6. This approach reduces the BSP cost of the truncation/copy operation from $6\frac{N}{p}g + 3l$ to $\frac{N}{p}g$. (The synchronization can be saved by merging the communication superstep with the following redistribution.)

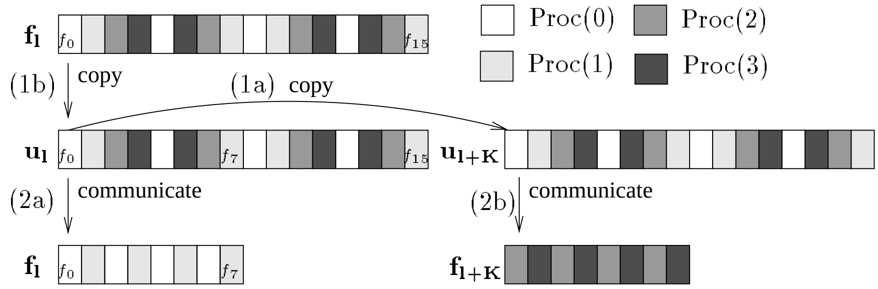

FIGURE 6. Truncation/copy operation of vectors $\mathbf{f_l}$ and $\mathbf{u_l}$ for $K = 16$ and $p1 = 4$. The numbers between brackets denote the phases of the procedure.

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 the data needed at stage $k+2$ (stage $k+1$ will produce the other half). We now show that the operations of stage $k+1$ immediately following the truncation/copy remain local and hence do not require communication. These operations alter $\mathbf{u_l}$ and $\mathbf{u_{l+K}}$ by operation (4.6) of the inverse FChT and the long distance butterflies of the inverse FFT. The restriction $p \leq \sqrt{N/2}$ implies $p1 \leq \sqrt{K}$, so that both the pairs (u_{l+j}, u_{l+K-j}) and (u_{l+K+j}, u_{l+2K-j}), with $j = 1, \ldots, K/2-1$, needed by operation (4.6) of the inverse FChT and the pairs $(u_{l+j}, u_{l+j+H/2})$ and $(u_{l+K+j}, u_{l+K+j+H/2})$, with $K \geq H \geq 4K/p1 = 2K/(p1/2)$ and $j = 0, \ldots, H/2-1$, needed by the long distance butterflies of the inverse FFT are in the same processor. Note that the long distance butterflies are those of stage $k+1$, where $p1$ is halved.

4.3.2. *Moving bit reversal to precomputation.* Another major optimization is to completely avoid the packing/bit reversal permutation (1)/(2a) 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.

After all the optimizations, the total communication and synchronization cost is approximately $\left(5\frac{N}{p}\log_2 p\right) g + (2\log_2 p)\, l$. 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:

$$T_{\mathrm{FLT,par}} \approx 4.25\frac{N}{p}\log_2^2 N + \left(5\frac{N}{p}\log_2 p\right) g + (2\log_2 p)\, l. \tag{4.11}$$

4.4. **Parallel termination.** Sometimes, it is useful to be able to perform the termination procedure of the Driscoll-Healy algorithm in parallel. In particular, this would enable the use of direct methods of $O(N^2)$ complexity, such as the so-called semi-naive method [11], which may be faster for small problem sizes. The termination as expressed by Lemma 2.11 is similar to the multiplication of a dense lower triangular matrix and a vector.

4.4.1. *Lower triangular matrix-vector multiplication.* Let us first consider how to multiply an $n \times n$ lower triangular matrix L by a vector $\mathbf{x}$ of length n on p processors, giving $\mathbf{y} = L\mathbf{x}$. Assume for simplicity that p is square. A parallel algorithm for matrix-vector multiplication was proposed in [6]. This algorithm is based on a two-dimensional distribution of the matrix over the processors, which are numbered $\mathrm{Proc}(s,t)$, $0 \le s < p_0$, $0 \le t < p_1$, where $p = p_0 p_1$. Often, it is best to choose $p_0 = p_1 = \sqrt{p}$. This scheme assigns matrix rows to processor rows $\mathrm{Proc}(s,*)$, and matrix columns to processor columns. Vectors are distributed in the same way as the matrix diagonal.

Since our matrix is lower triangular, we cannot adopt the simplest possible distribution method in this scheme, which is distributing the matrix diagonal, and hence the vectors, by blocks over all the processors. The increase of the row size with the row index would then lead to severe load imbalance in the computation. A better method is to distribute the diagonal cyclically over the processors. Translated into a two-dimensional numbering this means assigning matrix element L_{ij} to $\mathrm{Proc}(i \bmod \sqrt{p}, (j \operatorname{div} \sqrt{p}) \bmod \sqrt{p})$. The rows of the matrix are thus cyclically distributed, and blocks of $\sqrt{p}$ columns are also cyclically distributed. The algorithm first broadcasts input components x_j to $\mathrm{Proc}(*, (j \operatorname{div} \sqrt{p}) \bmod \sqrt{p})$, then computes and accumulates the local contributions $L_{ij}x_j$ and sends the resulting local partial sum to the processor responsible for y_i; this processor then adds the partial sums to compute y_i. The cost of the algorithm is about $\frac{n^2}{p} + 2\frac{n}{\sqrt{p}}g + 2l$.

4.4.2. *Application to termination.* We assume that a suitable truncation has been performed at the end of the main loop of the FLT algorithm. This truncation halves the group size $p1$ and redistributes the data to the input distribution of the termination. We assume, for simplicity of exposition, that $p1$ is square. We adapt the lower triangular matrix-vector multiplication algorithm to the context of the termination, as follows. Let $l \ge 1$ be fixed. We replace n by $M-1$ and p by $p1$, and define L using Lemma 2.11, for instance $L_{ij} = q_{l,i}^{j}/2$ for $i \ge j$, $i-j$ even,

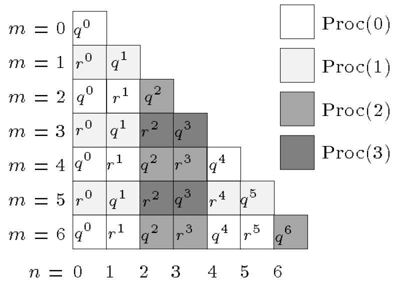

FIGURE 7. Data structure and distribution of the precomputed data needed for parallel termination with $M = 8$. The picture shows the data needed for one value of l, which is handled by $p1 = 4$ processors. The coefficients $q^n = q^n_{l,m}$ and $r^n = r^n_{l,m}$ are stored in a lower triangular matrix fashion.

and $j > 0$. Here, we include the trivial case $i = 0$. The two-dimensional processor numbering is created by the identification $\mathrm{Proc}(s,t) \equiv \mathrm{Proc}(s0 + s + t\sqrt{p1})$, where the offset $s0$ denotes the first processor of the group that handles the termination for l. Figure 7 illustrates the data distribution. In the first superstep, z^l_j is sent by its owner to the processor column that needs it, but only to half the processors, namely those that store $q^j_{l,i}$'s. The value z^{l-1}_j is sent to the other half. There is no need to redistribute the output vector, because it can be accumulated directly in the desired distribution, which is by blocks.

The total time of the parallel termination is about

$$T_{\text{term, par}} \approx \frac{MN}{p} + \frac{2\sqrt{MN}}{\sqrt{p}}g + 2l. \tag{4.12}$$

5. EXPERIMENTAL RESULTS

In this section, we present results on the accuracy and scalability of the implementation of the Legendre transform algorithm for various sizes N. We also investigate the optimal termination block size M.

We implemented the algorithm in ANSI C using the BSPlib communications library [19]. Our programs are completely self-contained, and we did not rely on any system-provided numerical software such as BLAS, FFTs, etc. We tested the accuracy of our implementation on a SUN Ultra 1 workstation which has IEEE 754 floating point arithmetic. The accuracy of double precision (64-bit) arithmetic is 2.2×10^{-16}. The efficiency and scalability test runs were made on a Cray T3E with up to 64 processors, each having a theoretical peak speed of 600 Mflop/s. 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` and measured the elapsed execution times on exclusively dedicated CPUs using the `bsp_time()` function.[1]

[1] We also wrapped our sequential programs as parallel ones. The reason is that our sequential programs compiled on the CRAY T3E with `cc -O3` are four times slower. It seems that the option `-flibrary-level 2` of `bspfront` also improves the execution time of sequential programs on the CRAY T3E.

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 (2.1), using the stable three-term recurrence (2.2) and quadruple precision); $||\cdot||_2$ indicates the L^2-norm.

Table 1 shows the relative errors of the sequential algorithm for various problem sizes using double precision except in the precomputation of the third column, which is carried out in quadruple precision. 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. They also show that our precomputation algorithm is somewhat less accurate for large N. 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. We believe that it is possible to improve the accuracy of the precomputation by exploiting the symmetries of the associated polynomials (that are either odd or even). As an additional advantage the sizes of the arrays $\mathbf{Q}$ and $\mathbf{R}$ can be halved. We will not address this issue here. See [17, 18] for a discussion of other techniques that can be used to get 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.

N	DLT	FLT	FLT-QP
512	7.7×10^{-14}	4.3×10^{-12}	1.5×10^{-14}
1024	3.0×10^{-13}	3.1×10^{-11}	2.3×10^{-13}
8192	1.3×10^{-11}	3.5×10^{-9}	1.3×10^{-11}
65536	2.7×10^{-10}	9.4×10^{-8}	1.6×10^{-10}

TABLE 1. Relative errors for the FLT algorithm. (QP indicates that the precomputation is carried out in quadruple precision.)

5.2. **Efficiency of the sequential implementation.** We measured the efficiency of our FLT algorithm by comparing its execution time with the execution time of the direct DLT algorithm (i.e., a matrix-vector multiplication). Table 2 shows the times obtained by the direct algorithm and the FLT with various termination values: $M = 2$ yields the pure FLT algorithm without early termination; $M = 64$ is the empirically determined value that makes the algorithm perform best (this value is close to the theoretical optimum $M = 128$, see Section 2.4); $M = N/2$ is the maximum termination value that our program can handle, and the resulting algorithm is similar to the semi-naive algorithm [11].

The results indicate that the pure FLT algorithm becomes faster than the DLT algorithm at $N = 128$. Choosing $M = 64$ (or M as large as possible if $N < 128$) further decreases the break-even point.

Though we opened the modules of the FLT algorithm, in principle it is still possible to use highly optimized or even machine specific, assembler coded, FFT

N	DLT	FLT $M=N/2$	FLT $M=64$	FLT $M=2$
16	0.0044	0.0095	–	0.0259
32	0.0167	0.0181	–	0.0572
64	0.0709	0.0391	–	0.1376
128	0.3660	0.0949	0.0949	0.3195
256	1.7249	0.3759	0.3171	0.7528
512	6.8738	1.4613	0.9540	1.8020

TABLE 2. Execution time (in ms) of various Legendre transform algorithms on one processor of a CRAY T3E.

subroutines in both the sequential and the parallel versions. This would yield an even faster program.

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.

Table 3 shows the timing results obtained for the sequential and parallel versions executed on up to 64 processors, with $p \le \sqrt{N/2}$, for $M = 2$ and $M = 64$. These results can best be analyzed in terms of absolute speedups, $S^{abs} = T(seq)/T(p)$, i.e., the time needed to run the sequential program divided by the time needed to run the parallel program on p processors. Our goal is to achieve ratios as close to p as possible. Figure 8 shows the performance ratios obtained for various input sizes with $M = 2$ on up to 64 processors. The speedups for $M = 64$ (not shown) are somewhat lower than for $M = 2$ because early termination does not reduce the parallel overhead of the algorithm; it improves only the computation part.

M	N	seq	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$
2	512	1.80	1.96	1.16	0.72	0.53	0.59	–	–
	1024	4.19	4.50	2.54	1.47	0.95	0.75	–	–
	8192	59.00	59.91	31.00	16.20	8.59	4.93	3.26	3.21
	65536	1210.–	1220.–	623.–	322.–	157.–	69.90	36.30	19.70
64	512	0.95	1.11	0.76	–	–	–	–	–
	1024	2.47	2.71	1.71	1.04	0.71	–	–	–
	8192	43.00	44.70	23.30	12.50	6.76	4.13	2.81	–
	65536	983.–	994.–	515.–	263.–	128.–	55.50	29.60	16.60

TABLE 3. Execution times (in ms) for the FLT on a Cray T3E.

It is clear that for a large problem size ($N = 65536$) the speedup is close to ideal, e.g., $S^{abs} = 61$ on 64 processors with $M = 2$. For smaller problems, reasonable speedups can be obtained using 8 or 16 processors, but beyond that the communication time becomes dominant. The superlinear speedup observed for $N = 65536$ is a well known phenomenon related to cache size.

6. CONCLUSIONS AND FUTURE WORK

As part of this work, we developed and implemented a sequential algorithm for the discrete Legendre transform, based on the Driscoll-Healy algorithm. This

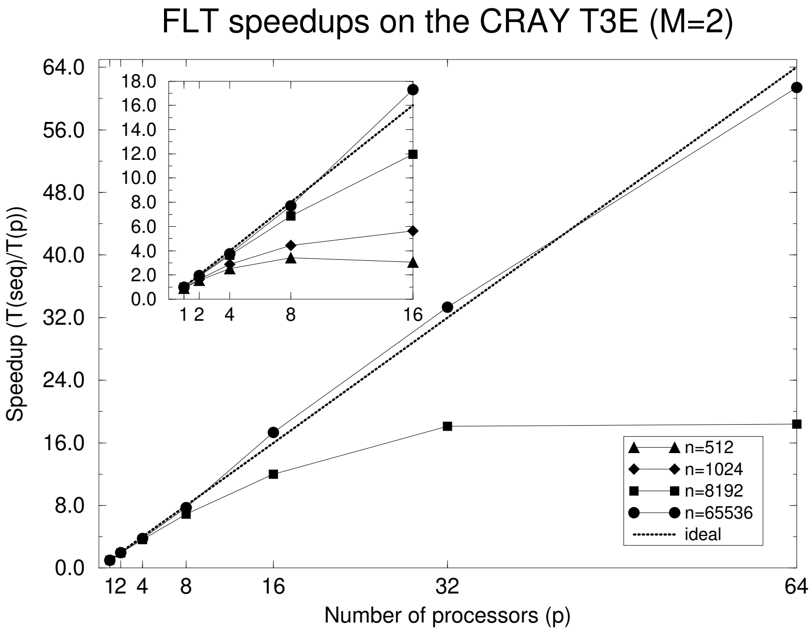

FIGURE 8. Scalability of the FLT on a Cray T3E.

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 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 fast Chebyshev transform also has many applications in its own right, as does the corresponding double fast cosine transform. Our algorithm has the additional benefit of easy parallelization.

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. Driscoll, Healy, and Rockmore [13] have already shown how a variant of the Driscoll-Healy algorithm may be used to compute such transforms at any set of sample points, though the set of points chosen affects the stability of the algorithm.

Acknowledgments

We thank Dennis Healy, Daniel Rockmore, and Peter Kostelec for useful discussions.

Appendix A. Related transforms and algorithms

The derivation of the Driscoll-Healy algorithm given in Section 2 has the feature that it only depends on the properties of truncation operators $\mathcal{T}_K$ given in Lemma 2.8, and on the existence of an efficient algorithm for applying the truncation operators. In particular, Lemma 2.8 and Lemma 2.10 hold as stated when the weight function $\omega(x) = \pi^{-1}(1-x^2)^{\frac{1}{2}}$ is changed, the truncation operators are defined using a polynomial sequence which is orthogonal with respect to the new weight function and starts with the polynomial 1, and the Lagrange interpolation operators are defined using the roots of this 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 $\mathcal{T}_M^K$, for $1 \le M \le K$, such that

1. $\mathcal{T}_M^K$ is a mapping from the space of polynomials of degree less than $2K$ to the space of polynomials of degree less than M.
2. If $M \le L \le K$ then $\mathcal{T}_M^L \mathcal{T}_L^K = \mathcal{T}_M^K$.
3. If $\deg Q \le m \le K \le L$ then $\mathcal{T}_{K-m}^L(f \cdot Q) = \mathcal{T}_{K-m}^K\left[(\mathcal{T}_K^L f) \cdot Q\right]$.

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 $\mathcal{T}_M$ replaced by $\mathcal{T}_M^K$, where $K \le N$ depends on the stage of the algorithm. Using $K = N$ 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

$$
\text{(A.1)} \qquad \begin{aligned} Z_{l+m}^{K-m} &= \mathcal{T}_{K-m}^{K}\left[Z_l^K \cdot Q_{l,m} + Z_{l-1}^K \cdot R_{l,m}\right], \\ Z_{l+m-1}^{K-m} &= \mathcal{T}_{K-m}^{K}\left[Z_l^K \cdot Q_{l,m-1} + Z_{l-1}^K \cdot R_{l,m-1}\right]. \end{aligned}
$$

Cf. (2.15) and (2.16).

This generalization of the approach we have presented may be used to derive the original algorithm of Driscoll and Healy [12], which uses the cosine transforms in the points $\cos \frac{j\pi}{K}$.

Driscoll, Healy, and Rockmore [13] described 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 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 Algorithm 2.2, and for generating the coefficients $q_{l,m}^n, r_{l,m}^n$ used in the termination stage of Algorithm 2.3.

The precomputation of the point values is based on the following recurrences.

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

$$
\text{(B.1)} \qquad \begin{aligned} Q_{l,j+k} &= Q_{l+k,j}Q_{l,k} + R_{l+k,j}Q_{l,k-1}, \\ R_{l,j+k} &= Q_{l+k,j}R_{l,k} + R_{l+k,j}R_{l,k-1}. \end{aligned}
$$

Proof. By induction on j. The proof for $j = 0$ follows immediately from the definition (2.10), 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 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 unequal to zero, 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 (2.10) and (2.3).

The point values needed can be retrieved as follows. Algorithms 2.2 and 2.3 require 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 only need 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

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

Comparing with the cost (2.24) 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. We did not do this yet, because optimizing the one-time precomputation is much less important than optimizing the Driscoll-Healy algorithm itself.

The precomputation of the coefficients $q^n_{l,m}$, $r^n_{l,m}$ required to terminate the Driscoll-Healy algorithm early, as in Lemma 2.11, is based on the following recurrences.

Lemma B.2. *Let $l \geq 1$ and $m \geq 2$. The coefficients $q^n_{l,m}$ satisfy the recurrences*

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

subject to the boundary conditions $q^0_{l,0} = 1, q^0_{l,1} = B_l, q^1_{l,1} = A_l$, and $q^n_{l,m} = 0$ for $n > m$. The $r^n_{l,m}$ satisfy the same recurrences, but with boundary conditions $r^0_{l,1} = C_l$ and $r^n_{l,m} = 0$ for $n \geq m$.

Proof. These recurrences are the shifted three-term recurrences (2.10) 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 $q^n_{l,m}$ and $r^n_{l,m}$ by increasing m, starting with the known values for $m = 0, 1$ and finishing with $m = M - 2$. For each m, we only need to compute the $q^n_{l,m}$ with $n \leq m$, and the $r^n_{l,m}$ with $n < m$. The total number of flops of the precomputation of the Chebyshev coefficients in the general case is

$$T_{\text{precomp, term}} = 7M^2 - 16M - 15. \tag{B.3}$$

When the initial values B_l are identically zero, the coefficients can be packed in alternating fashion into array $\mathbf{T}$, as shown in Fig. 4. 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 can then 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.

References

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

[2] B. Alpert and V. Roklin, *A fast algorithm for the evaluation of Legendre transforms*, SIAM J. Sci. Statist. Comput. **12** no. 1 (1991), 158–179.

[3] S. Barros and T. Kauranne, *Practical aspects and experiences on the parallelization of global weather models*, Parallel Computing **20** (1994), 1335–1356.

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

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

[6] R. H. Bisseling and W. F. McColl, *Scientific computing on bulk synchronous parallel architectures*, in Proc. IFIP 13th World Computer Congress, Vol. I, B. Pehrson and I. Simon, eds., North-Holland, 1994, 509–514.

[7] R. H. Bisseling, *Basic techniques for numerical linear algebra on bulk synchronous parallel computers*, Lecture Notes in Computer Science **1196**, Springer-Verlag, Berlin, 1997, 46–57.

[8] 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), 1058–1075.

[9] T.S. Chihara, *An introduction to orthogonal polynomials*, Gordon and Breach, New York, 1978.

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

[11] G. Dilts, *Computation of spherical harmonic expansion coefficients via FFTs*, J. Comput Phys. **57** No. 3 (1985), 439–453.

[12] J. Driscoll and D. Healy, *Computing Fourier transforms and convolutions on the 2-sphere*, (extended abstract Proc. 34^{th} IEEE FOCS, 1989, 344–349) Adv. in Appl. Math. **15** (1994), 202–250.

[13] J. R. Driscoll, D. Healy, and D. Rockmore. *Fast discrete polynomial transforms with applications to data analysis for distance transitive graphs*, SIAM J. Comput. **26** no. 4 (1997), 1066–1099.

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

[15] D. Healy and P. Kim, *An empirical Bayes approach to directional data and efficient computation on the sphere*, Ann. Stat., (to appear).

[16] ———, *Spherical deconvolution with application to geometric quality assurance*, Technical Report, Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH, 1993.

[17] D. Healy, S. Moore, and D. Rockmore, *Efficiency and stability issues in the numerical computation of Fourier transforms and convolutions on the 2-sphere,* Technical Report, PCS-TR94-222, Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH, 1994.

[18] ———, *FFTs for the 2-sphere - improvements and variations,* Technical Report, PCS-TR96-292, Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH, 1996. To appear in Advances on Applied Mathematics.

[19] 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 Computing **24** no. 14 (1998) 1947–1980.

[20] D. Maslen, *A polynomial approach to orthogonal polynomial transforms*, Preprint MPI/95-9, Max-Planck-Institut für Mathematik, Bonn, 1995.

[21] W.F. McColl, *Scalability, portability and predictability: The BSP approach to parallel programming*, Future Generation Computer Systems **12** (1996), 265–272.

[22] S. Orszag, *Fast eigenfunction transforms*, in Science and Computers, G. Rota, ed., Academic Press, NY, 1986, 23–30.

[23] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical recipes in C: The art of scientific computing*, second edition, Cambridge University Press, Cambridge, UK, 1992.

[24] K. Rao and P. Yip, *Discrete cosine transform : algorithms, advantages, applications*, Academic Press, Boston, MA, 1990.

[25] N. Shalaby, *Parallel discrete cosine transforms: Theory and practice*, Technical report, TR-34-95, Center for Research in Computing Technology, Harvard University, Cambridge, MA, 1995.

[26] N. Shalaby and S.L. Johnsson, *Hierarchical load balancing for parallel fast Legendre transforms*, Proc. 8th SIAM Conf. on Parallel Processing for Scientific Computation, SIAM, Philadelphia, 1997.

[27] G. Steidl and M. Tasche, *A polynomial approach to fast algorithms for discrete Fourier-cosine and Fourier-sine transforms*, Mathematics of Computation **56** no. 193 (1991), 281–296.

[28] L. Valiant, *A bridging model for parallel computation*, Communications of the ACM **33** no. 8 (1990), 103–111.

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

Department of Mathematics, Universiteit Utrecht, PO Box 80010, 3508 TA, Utrecht, The Netherlands
E-mail address: `inda@math.uu.nl`

Department of Mathematics, Universiteit Utrecht, PO Box 80010, 3508 TA, Utrecht, The Netherlands
E-mail address: `Rob.Bisseling@math.uu.nl`

Department of Mathematics, Dartmouth College, Hanover, NH 03755-3551, U.S.A
E-mail address: `maslen@math.dartmouth.edu`