PARALLEL FAST LEGENDRE TRANSFORM*

MÁRCIA ALVES DE INDA
Mathematics Department, Utrecht University
Budapestlaan 6, Utrecht, 3584 CD, The Netherlands

ROB H. BISSELING
Mathematics Department, Utrecht University
Budapestlaan 6, Utrecht, 3584 CD, The Netherlands

and

DAVID K. MASLEN
Mathematics Department, Dartmouth College
Hanover, NH 03755, U.S.A.

ABSTRACT

We discuss a parallel implementation of a fast algorithm for the discrete polynomial Legendre transform. We give an introduction to the Driscoll-Healy algorithm using polynomial arithmetic, and present experimental results on the efficiency and accuracy of our implementation. The algorithms were implemented in ANSI C using the BSPlib communications library. Furthermore, we present a new algorithm for computing the Chebyshev transform of two vectors at the same time.

1. Introduction

The discrete polynomial Legendre transform, *DLT*, is a widely used tool in applied science, where it commonly arises in problems associated with spherical geometries. In weather forecasting, the DLT appears inside the discrete spherical harmonic transform used in global spectral weather models.[3,7]

Given 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 polynomial Legendre transform* of a data vector $(f_0, \ldots f_{N-1})$ to be the vector of sums $(\hat{f}_0, \ldots \hat{f}_{N-1})$, given by

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

*To appear in the proceedings of the ECMWF Workshop "Towards TeraComputing - The Use of Parallel Processors in Meteorology", Nov. 1998, Reading, UK, published by World Scientific Publishing Co, 1999.

where P_l is the l^{th} Legendre polynomial defined by the three-term recurrence

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

A direct method for computing a DLT of N data values requires a matrix-vector multiplication of $O(N^2)$ arithmetic operations, though several authors[2,15] have proposed faster algorithms based on approximate methods. In 1989, Driscoll and Healy introduced an exact algorithm that computes a DLT in $O(N(\log N)^2)$ arithmetic operations; they implemented the algorithm and analyzed its stability.[9,10]

In the present article we describe a parallel implementation of the Driscoll-Healy algorithm. Such an algorithm is useful for solving large problem sizes. At least two reports discussing the theoretical parallelizability of the algorithm have already been written.[11,18] We are, however, unaware of any parallel implementations of the Driscoll-Healy algorithm at the time of writing.

The remainder of this paper is organized as follows. In Section 2, we outline a derivation of the Driscoll-Healy algorithm based on polynomial arithmetic. Full proofs are omitted; these will be given in a future expanded article. In Section 3, we introduce the bulk synchronous parallel (BSP) model, and describe our parallel algorithm and its implementation. In Section 4, we present results on the efficiency, accuracy and scalability of the programs. We conclude in Section 5.

2. **The Driscoll-Healy Algorithm**

The Driscoll-Healy algorithm computes the DLT at any set of N sample points, in $O(N(\log N)^2)$ arithmetic operations. The core of this algorithm consists of an algorithm to compute the DLT in the special case where the sample points are the Chebyshev points, i.e., $x_j = x_j^N = \cos\frac{(2j+1)\pi}{2N}$ is the j-th root of the N^{th} Chebyshev polynomial T_N defined recursively by

$$T_{k+1}(x) = 2x \cdot T_k(x) - T_{k-1}(x), \quad T_0(x) = 1, \quad T_1(x) = x, \tag{3}$$

and where 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.

Our derivation of the Driscoll-Healy algorithm relies on the interpretation of the input data f_j of the transform Eq. (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

$$f(x_j^N) = f_j, \quad \text{for } j = 0, \dots N-1. \tag{4}$$

Using this notation one can derive the relation

$$f \cdot P_{l+m} = Q_{l,m} \cdot (f \cdot P_l) + R_{l,m} \cdot (f \cdot P_{l-1}) \tag{5}$$

directly from the generalized three-term recurrence for the Legendre polynomials

$$P_{l+m} = Q_{l,m} \cdot P_l + R_{l,m} \cdot P_{l-1}. \tag{6}$$

Here, $Q_{l,m}, R_{l,m}$ are the *associated polynomials*[†] for the Legendre polynomial sequence,[4,5] defined by the following recurrences on m, which are shifted versions of the recurrence Eq (2) for P_l.

$$\begin{aligned} Q_{l,m}(x) &= (A_{l+m-1}x + B_{l+m-1})Q_{l,m-1}(x) + C_{l+m-1}Q_{l,m-2}(x),\\ Q_{l,1}(x) &= A_l x + B_l, \quad Q_{l,0}(x) = 1,\\ R_{l,m}(x) &= (A_{l+m-1}x + B_{l+m-1})R_{l,m-1}(x) + C_{l+m-1}R_{l,m-2}(x),\\ R_{l,1}(x) &= C_l, \quad R_{l,0}(x) = 0. \end{aligned} \tag{7}$$

The Driscoll-Healy algorithm is a divide and conquer algorithm. Its divide structure is based on the following strategy:

- Start by computing $f \cdot P_0$ and $f \cdot P_1$ at the points x_j^N for $0 \leq j < N$.
- At stage 1, use Eq. (5) 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 Eq. (5) 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 $f \cdot P_{\frac{N}{2^k}},\ f \cdot P_{\frac{N}{2^k}+1};\ f \cdot P_{\frac{3N}{2^k}},\ f \cdot P_{\frac{3N}{2^k}+1};\ \cdots;\ f \cdot P_{\frac{(2^k-1)N}{2^k}},\ f \cdot P_{\frac{(2^k-1)N}{2^k}+1}$.
- At stage $\log_2 N$, compute all the sums $\frac{1}{N}\sum_{j=0}^{N-1} f(x_j^N)P_l(x_j^N)$ using the function values $f(x_j^N)P_l(x_j^N)$ that were computed at the previous stages.

The conquer property of the Driscoll-Healy algorithm is achieved using *truncation operators* which truncate the Chebyshev expansion of the polynomials involved. 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. The truncation operator $\mathcal{T}_n$ applied to f is defined by

$$\mathcal{T}_n f = \sum_{k=0}^{n-1} b_k T_k. \tag{8}$$

Thus, $\mathcal{T}_n f$ is obtained from f by discarding terms of degree n or higher in the expansion of f in terms of Chebyshev polynomials.

It can be proven[13] that the DLT of f of size N is given by

$$\hat{f}_l = \mathcal{T}_1(f \cdot P_l), \quad 0 \leq l < N. \tag{9}$$

The Driscoll-Healy algorithm proceeds in a fashion determined by the basic divide strategy but it 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 $f \cdot P_l$. This is done by using truncated versions of the generalized three-term recurrence Eq. (5) for the

[†]The associated polynomials should not be confused with the associated Legendre functions, which in general are not polynomials.

polynomials Z_l^K:

$$Z_{l+K}^K = \mathcal{T}_K[Z_l^{2K} \cdot Q_{l,K} + Z_{l-1}^{2K} \cdot R_{l,K}] \tag{10}$$

$$Z_{l+K-1}^K = \mathcal{T}_K[Z_l^{2K} \cdot Q_{l,K-1} + Z_{l-1}^{2K} \cdot R_{l,K-1}]. \tag{11}$$

These equations are obtained from Eq. (5) and judicious application of the following property of the truncation operator

Lemma 2.1. *Let f and Q be polynomials. Then*

$$\mathcal{T}_{L-m}(f \cdot Q) = \mathcal{T}_{L-m}[(\mathcal{T}_L f) \cdot Q], \quad \textit{if } \deg Q \leq m \leq L.$$

The Driscoll-Healy algorithm is shown as Algorithms 1 and 2. Its input is the polynomial $f = Z_0^N$, and the output is the sequence of $\hat{f}_l = \mathcal{T}_1(f \cdot P_l) = Z_l^1$. A brief explanation of its main features is given in the following subsections.

Algorithm 1 Driscoll-Healy algorithm.

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

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

STAGES

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

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

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

2.1. *Data Representation and Initialization*

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

$$Z_l^K = \sum_{n=0}^{K-1} z_n^l T_n, \tag{12}$$

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.

We do this using the *Chebyshev transform* of size N,

$$\tilde{f}_k = \frac{\epsilon_k}{N} \sum_{j=0}^{N-1} f_j T_k(x_j^N) = \frac{\epsilon_k}{N} \sum_{j=0}^{N-1} f_j \cos \frac{(2j+1)k\pi}{2N}, \quad 0 \leq k < N. \tag{13}$$

where $\epsilon_0 = 1$, and $\epsilon_k = 2$ if $k > 0$. Its inverse is defined by

$$f_j = \sum_{k=0}^{N-1} \tilde{f}_k T_k(x_j^N) = \sum_{k=0}^{N-1} \tilde{f}_k \cos \frac{(2j+1)k\pi}{2N}, \quad 0 \leq j < N. \tag{14}$$

The Chebyshev transform of size N and its inverse convert a polynomial of degree less than N from point value representation to Chebyshev representation and vice versa. Both transforms can be carried out in $O(N \log_2 N)$ flops using a fast cosine transform, *FCT*, algorithm (see e.g. Ahmed, Natarajan and Rao,[1] Steidl and Tasche,[19] van Loan[21]).

Note that $f \cdot P_0 = f$ is a polynomial of degree less than N but $f \cdot P_1 = f \cdot x$ may have degree N, rather than $N-1$. In the last case, a simple argument shows that a Chebyshev transform of size N (rather than $N+1$) applied in the points $f_j P_1(x_j^N) = f_j x_j^N$ suffices to compute Z_1^N.

2.2. *Intermediate Stages: Carrying on the Recurrence*

To carry on the recurrence in an efficient way we use the procedure described in Algorithm 2. This procedure replaces the polynomial multiplications in the recurrences Eq. (10) and Eq. (11) by a different operation. For example, instead of computing $Z_l^{2K} \cdot Q_{l,K}$ it computes the *Lagrange interpolation polynomial* $\mathcal{S}_{2K}(Z_l^{2K} \cdot Q_{l,K})$, i.e., the polynomial of degree less than $2K$ that agrees with $Z_l^{2K} \cdot Q_{l,K}$ at the points $x_0^{2K}, \ldots, x_{2K-1}^{2K}$. Correctness of the modified procedure can be proven by combining properties of the Lagrange operators $\mathcal{S}$ and the truncation operators $\mathcal{T}$.

2.3. *Terminating the Computation*

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 finished the computation

Algorithm 2 Recurrence algorithm 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 $(\tilde{v}_K, \ldots, \tilde{v}_{2K-1})$.

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

$$Q_{l,m} = \sum_{n=0}^{m} q_{l,m}^n T_n, \quad R_{l,m} = \sum_{n=0}^{m-1} r_{l,m}^n T_n. \tag{15}$$

The problem of finishing the computation at the end of stage $k = \log_2 N/M$, when $K = M$, is equivalent to finding $\hat{f}_l = z_0^l$, for $0 \le l < N$, given the data z_n^l, z_n^{l-1}, $0 \le n < M$, $l = 1, M+1, \ldots, N-M+1$. Our method of finishing the computation uses Lemma 2.2, which follows.

Lemma 2.2. 1. *If $l \ge 1$ and $0 \le m < M$, then*

$$\hat{f}_{l+m} = \frac{1}{2}\sum_{n=1}^{m}(z_n^l q_{l,m}^n + z_n^{l-1} r_{l,m}^n) + (z_0^l q_{l,m}^0 + z_0^{l-1} r_{l,m}^0).$$

2. *$q_{l,m}^n = 0$, if $n - m$ is odd, and $r_{l,m}^n = 0$, if $n - m$ is even.*

3. The Parallel Algorithm and its Implementation

We designed our parallel algorithm using the BSP model. The BSP model 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 first 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. Finally, we refine the basic algorithm by introducing an intermediate data distribution that reduces the communication to a minimum.

3.1. *The Bulk Synchronous Parallel Model*

In the BSP model,[20] 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, both followed by a global synchronization barrier. 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. The time of a computation superstep is simply $w + l$, where w denotes 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. Such a communication superstep is called an h-relation. 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 Bisseling.[6]

BSPlib[12] is a recently defined standard library which enables parallel programming in BSP style. The definition of BSPlib was completed in May 1997. Implementations are available for many different machines, including the Cray T3E, SGI Origin, the IBM SP2, Parsytec Explorer, PCs running the Linux operating system or Windows NT, and also for networks of workstations communicating via Ethernet and TCP/IP or UDP/IP. Programs written in BSPlib can be run on all of these platforms without changing one line of code. BSPlib is available for the languages C, C++, Fortran 77 and Fortran 90. Thus, it is an attractive and efficient alternative to well-known communication libraries such as MPI and PVM. Moreover, BSPlib is easy to learn because it comprises only 20 primitives.

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. The divide and conquer structure of the Driscoll-Healy algorithm suggests both the data structures and data distributions to be used.

At each stage k, $1 \leq k \leq \log_2 N/M$, the number of intermediate polynomial pairs doubles as the number of expansion coefficients halves. At the start of stage 1, we have two polynomials of degree $N-1$; at the end of stage 1, we have four polynomials of degree $N/2-1$, etc. 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 of length N, $\mathbf{u}$ and $\mathbf{v}$.

The data flow of the algorithm, see Fig. 1, suggests to distribute all the vectors by blocks, i.e., to assign one block of consecutive vector elements to each processor. This works well if p is a power of two, as 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 the block size is $b = \lceil N/p \rceil$.

Note that if both N and p are powers of two, the block size is $b = N/p$.

Now we explain how to store and distribute the precomputed data used in the recurrence. To perform the recurrence of stage k, we need to have the values of the polynomials $Q_{l+1,K}$, $Q_{l+1,K-1}$, $R_{l+1,K}$, and $R_{l+1,K-1}$, for $l = 0, 2K, \ldots, N-2K$, at the points $x_j^{2K} = \cos\frac{(2j+1)\pi}{4K}$, $0 \leq j < 2K$. We store these values 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

$$\mathbf{Q}[2k-2, l+j] = Q_{l+1,K}(x_j^{2K}) \quad \text{and} \quad \mathbf{Q}[2k-1, l+j] = Q_{l+1,K-1}(x_j^{2K}), \tag{16}$$

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 depicted in Fig. 2. 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.

To make the recurrence completely local, the values from $\mathbf{R}$ and $\mathbf{Q}$ must be available locally. This can be achieved by distributing each row of these arrays by the block distribution, so that $\mathbf{R}[i,j], \mathbf{Q}[i,j] \in \mathrm{Proc}(j \ \operatorname{div}\ \frac{N}{p})$.

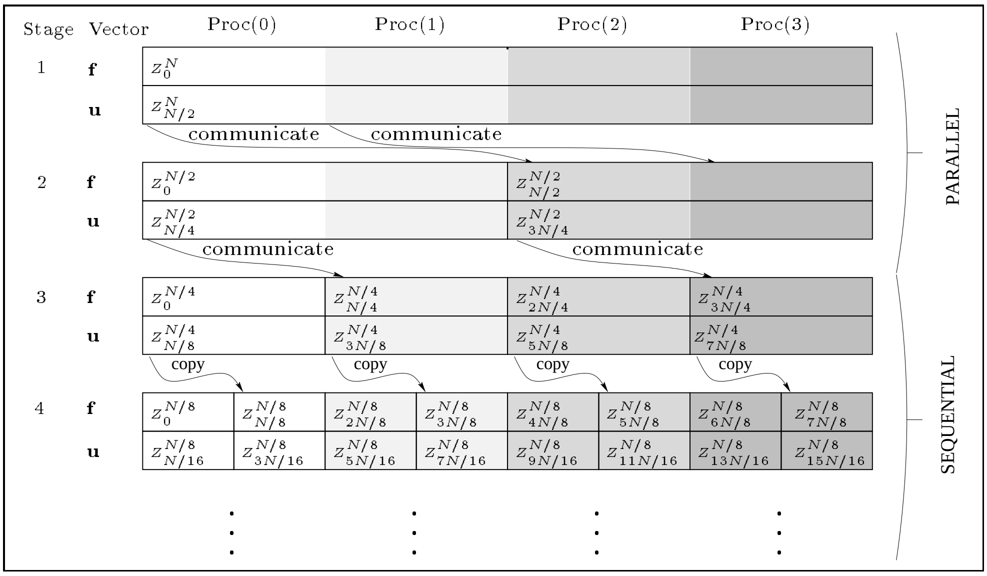

FIGURE 1. Data storage and data distribution in the parallel FLT algorithm for four processors. The Chebyshev coefficients of the intermediate polynomials are stored in four arrays. Array **f** contains the polynomials Z_l^{2K} which are already available at the start of the stage. Array **u** contains the polynomials Z_{l+K}^{2K} which become available at the end of the stage. Similarly, arrays **g** and **v** contain the next higher polynomials Z_{l+1}^{2K} and Z_{l+K+1}^{2K}, respectively; these arrays are not depicted. Each array is divided into four local subarrays by using the block distribution. Each processor has one subarray.

k	$K, K-1$	Proc(0)	Proc(1)	Proc(2)	Proc(3)
1	32	$l = 1$			
	31	$j = 0, \dots, 63$			
2	16	$l = 1$		$l = 33$	
	15	$j = 0, \dots, 31$		$j = 0, \dots, 31$	
3	8	$l = 1$	$l = 17$	$l = 33$	$l = 49$
	7	$j = 0, \dots, 15$	$j = 0, \dots, 15$	$j = 0, \dots, 15$	$j = 0, \dots, 15$

FIGURE 2. Data structure and distribution of the precomputed data needed in the recurrence with $N = 64$, $M = 8$ and $p = 4$. Data are stored in two two-dimensional arrays **Q** and **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 $l = 1, M+1, 2M+1, \dots, N-M+1$, $m = 1, 2, \dots, M-2$ and $n = 0, 1, \dots, m$ are stored in a two-dimensional array **T** of size $N/M \times (M(M-1)/2 - 1)$. The coefficients for one value of l are stored in row $(l-1)/M$ of **T**. Each row has the same internal structure, as follows. The coefficients are stored in increasing order of m. The coefficients with the same m are ordered by increasing n. This format is similar to that commonly used to store lower triangular matrices. For each n and m, either $q_{l,m}^n = 0$ or $r_{l,m}^n = 0$, and hence 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 $q^n_{l,m}$ and $r^n_{l,m}$. Figure 3 illustrates this data structure.

	$m=1$	$m=2$	$m=3$	$m=4$	$m=5$	$m=6$	
$l=1$	$r^0\ q^1$	$q^0\ r^1\ q^2$	$r^0\ q^1\ r^2\ q^3$	$q^0\ r^1\ q^2\ r^3\ q^4$	$r^0\ q^1\ r^2\ q^3\ r^4\ q^5$	$q^0\ r^1\ q^2\ r^3\ q^4\ r^5\ q^6$	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 3. Data structure of the precomputed data needed for termination with $N = 64$, $M = 8$ and $p = 4$. The coefficients $q^n_{l,m}$ and $r^n_{l,m}$, 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 $\mathbf{T}$. In the picture, r^n denotes $r^n_{l,m}$ and q^n denotes $q^n_{l,m}$.

The termination stage becomes local if $M \leq N/p$, so that the input and output vectors are local. The necessary precomputed data must then also be available locally. 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*

Now we formulate our basic parallel algorithm. For this we introduce the following conventions:

- **Processor identification.** The total number of processors is p. The processor identification number is s, with $0 \leq s < p$.
- **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. In principle, each superstep ends with an explicit synchronization (In an actual implementation, synchronizations can sometimes be saved). The supersteps are numbered as textual supersteps. Of course, supersteps inside loops are executed repeatedly, even though they are numbered only once.
- **Indexing.** All the indices are global. This means that array elements have a unique index which is independent of the processor that owns it. This enables us to describe variables and gain access to arrays in an unambiguous manner, even though the array is distributed and each processor has only part of it.

- **Vectors and Subroutine calls.** All the vectors (or one-dimensional arrays) 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

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

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

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

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

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

- **Truncation.** The operation

$$\mathbf{f} \leftarrow \mathrm{BSP\_Trunc}(s, p, s0, s1, p1, N, K, \mathbf{u})$$

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. Note that $s0 + s1 = s$. When $p1 = 1$ one processor is in charge of the truncation of one or more polynomials. Algorithm 3 gives a description of this operation. In Fig. 1, this operation is depicted by arrows.
- **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.

The basic template for the fast Legendre transform is presented as Algorithm 4. At each intermediate stage k, $1 \le k \le \log_2 N/M$, 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

Algorithm 3 Truncation using the block distribution.

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

DESCRIPTION

if $p1 = 1$ **then**
  for $l = s\frac{N}{p}$ **to** $(s+1)\frac{N}{p} - 2K$ **step** $2K$ **do**
    $\mathbf{f_{l+K}} \leftarrow \text{Copy}(K, \mathbf{u_l})$
else
  if $s1 < p1/2$ **then**
    $\mathbf{f}_{\mathbf{s\frac{N}{p}+K}} \leftarrow \text{Put}(s + \frac{p1}{2}, \frac{N}{p}, \mathbf{u}_{\mathbf{s\frac{N}{p}}})$

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 $l = s0N/p$, and the j-loop has N/p iterations, starting with $j = s1N/p$, so that the indices $l + j$ start with $(s0 + s1)N/p = sN/p$, 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). This communication will be discussed separately, in the following subsections.
- At the end of each stage (in supersteps 3, 8).

For $k \geq \log_2 p + 1$, the length of the subvectors involved becomes $2K \leq N/p$. 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 1 are combined into one stage and then performed efficiently, as follows. First, in superstep 1, the polynomials Z_1^N, $Z_{N/2}^N$ and $Z_{N/2+1}^N$ are computed directly from the input vector $\mathbf{f}$. This is possible because the point-value representation of $Z_1^N = \mathcal{T}_N(f \cdot P_1) = \mathcal{T}_N(f \cdot x)$ needed by the recurrences is the vector of $f_j \cdot x_j^N, 0 \leq j < N$, see Subsection 2.1. Note that 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 $Z_0^{N/2} = \mathbf{f}, Z_1^{N/2} = \mathbf{g}, Z_{N/2}^{N/2} = \mathbf{u}$, and $Z_{N/2+1}^{N/2} = \mathbf{v}$ are transformed to Chebyshev representation; and then truncated, in superstep 3, in order 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 $\mathbf{f}$ corresponds to the lower value of l, so

Algorithm 4 Basic parallel template for the fast Legendre transform.

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

ARGUMENTS

s: Processor identification $(0 \le s < p)$.
p: Number of processors (p is a power of 2 with $p \le N/2$).
N: Transform size (N is a power of 2 with $N \ge 4$).
M: Termination block size (M is a power of 2 with $M \le \min(N/2, N/p)$).
$\mathbf{f}$: (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: $f_j \in \mathrm{Proc}(j \text{ div } \frac{N}{p})$.

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$(s, p, 0, s, p, N, N/2, \mathbf{u}, \mathbf{v})$

STAGE k:

for $k = 2$ **to** $\log_2 N/M$ **do**
(4^{Cp}) $K \leftarrow N/2^k$
    $p1 \leftarrow \max(p/2^{k-1}, 1)$
    $s0 \leftarrow (s \text{ div } p1)p1$
    $s1 \leftarrow s \bmod p1$
    $(\mathbf{u}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}, \mathbf{v}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}) \leftarrow \mathrm{Copy2}(\frac{N}{p}, \mathbf{f}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}, \mathbf{g}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}})$
    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$(s, p, s0, s1, p1, N, K, \mathbf{u}, \mathbf{v})$

STAGE $\log_2 N/M + 1$:

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

that in the recurrence the components of $\mathbf{f}$ 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 9) is described separately as Algorithm 5.

Algorithm 5 Termination procedure for the fast Legendre transform.

CALL Terminate$(l, M, \mathbf{f}, \mathbf{g})$

INPUT

l: Block identifier.
M: Termination block size (M is a power of 2; $l \bmod M = 0$).
$\mathbf{f} = (f_0, \ldots, f_{M-1})$: Chebyshev coefficients of polynomial Z_l^M.
$\mathbf{g} = (g_0, \ldots, g_{M-1})$: Chebyshev coefficients of polynomial Z_{l+1}^M.

OUTPUT $\mathbf{h} = (h_0, \ldots, h_{M-1})$: $h_i = \hat{f}_{l+i}, 0 \le i < M$.

STEPS

$h_0 \leftarrow f_0$
$h_1 \leftarrow g_0$
$b \leftarrow 0$
for $m = 1$ **to** $M - 3$ **step** 2 **do**
  $h_{m+1} \leftarrow f_0 \mathbf{T}[l, b] + \frac{1}{2} g_1 \mathbf{T}[l, b+1]$
  $h_{m+2} \leftarrow g_0 \mathbf{T}[l, b+m+1] + \frac{1}{2} f_1 \mathbf{T}[l, b+m+2]$
  for $n = 2$ **to** $m - 1$ **step** 2 **do**
    $h_{m+1} \leftarrow h_{m+1} + \frac{1}{2}\left(f_n \mathbf{T}[l, b+n] + g_{n+1} \mathbf{T}[l, b+n+1]\right)$
    $h_{m+2} \leftarrow h_{m+2} + \frac{1}{2}\left(g_n \mathbf{T}[l, b+n+m+1] + f_{n+1} \mathbf{T}[l, b+n+m+2]\right)$
  $h_{m+2} \leftarrow h_{m+2} + \frac{1}{2} g_{m+1} \mathbf{T}[l, b+n+m+3]$
  $b \leftarrow b + 2m + 3$

3.4. *Fast Chebyshev Transform*

The efficiency of the FLT algorithm strongly depends 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. Press *et al.*[16] and Steidl and Tasche[19]). Parallel algorithms or implementations have been less intensively studied, see Shalaby[17] 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. This algorithm is based on the FCT algorithm 4.4.6 of van Loan[21] and the standard algorithm for computing the FFTs of two real input vectors at the same time (see e.g. Press *et al.*[16]).

The algorithm has the following structure:

1. PACK the two input vectors as one auxiliary complex vector.

2. TRANSFORM the auxiliary vector using an FFT
3. EXTRACT the desired Chebyshev transforms from the transformed auxiliary vector.

The Chebyshev transforms are 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 $(x_j + i\ y_j)$, $0 \le j < N$. Phase 1, the packing of the input data into the auxiliary complex vector $\mathbf{z}$ of lenght N is just a simple permutation,

$$\begin{cases} z_j = (x_{2j} + i\ y_{2j}) \\ z_{N-j-1} = (x_{2j+1} + i\ y_{2j+1}), \end{cases} \quad \text{for} \quad 0 \le j < N/2. \tag{17}$$

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

$$Z_k = \sum_{j=0}^{N-1} z_j e^{\frac{2\pi ijk}{N}}, \quad \text{for } 0 \le k < N. \tag{18}$$

(Note that we define the discrete Fourier transform with a positive sign in the exponent.) Finally, in phase 3 we obtain the Chebyshev transform by

$$\begin{cases} \tilde{x}_k = \dfrac{\epsilon_k}{N}\mathrm{Re}\left(\dfrac{1}{2}e^{\frac{\pi ik}{2N}}(Z_k + \overline{Z}_{N-k})\right) \\ \tilde{y}_k = \dfrac{\epsilon_k}{N}\mathrm{Re}\left(-\dfrac{i}{2}e^{\frac{\pi ik}{2N}}(Z_k - \overline{Z}_{N-k})\right) \end{cases}, \qquad \text{for } 0 \le k < N, \tag{19}$$

where $\frac{\epsilon_k}{N}$ is the normalization factor needed to get the Chebyshev transform from the cosine transform.

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 $\mathbf{Z}$:

$$\begin{cases} Z_0 = N(\tilde{x}_0 + i\ \tilde{y}_0), \\ Z_k = \dfrac{N}{\epsilon_k}e^{-\frac{\pi ik}{2N}}\left((\tilde{x}_k + i\ \tilde{y}_k) + i(\tilde{x}_{N-k} + i\ \tilde{y}_{N-k})\right), \qquad \text{for } 1 \le k < N. \end{cases} \tag{20}$$

In phase 2, an inverse complex FFT is computed,

$$z_k = \frac{1}{N}\sum_{j=0}^{N-1} Z_j e^{-\frac{2\pi ijk}{N}}, \quad \text{for } 0 \le k < N. \tag{21}$$

The desired transforms are stored as the real and imaginary parts of $\mathbf{z}$ respectively, but in a different ordering. The inverse of phase 1 is again a permutation.

$$\begin{cases} x_{2j} = \mathrm{Re}(z_j) \qquad\quad y_{2j} = \mathrm{Im}(z_j) \\ x_{2j+1} = \mathrm{Re}(z_{N-j-1}),\ y_{2j+1} = \mathrm{Im}(z_{N-j-1}), \end{cases} \quad \text{for} \quad 0 \le j < N/2. \tag{22}$$

If we use a radix-4 algorithm[21] to perform the FFT, the flop count for this FChT2 algorithm is $2.125N\log_2 N + 8N - 16$ against $2.125N\log_2 N + 8.25N - 22$ for performing two FChTs one after the other. Theoretically it is only a small improvement although in practice we found the gain to be substantial.

An efficient parallelization of this algorithm within the framework of the FLT algorithm involves breaking open the parallel FFT inside the FChT and merging parts of the FFT with the surrounding computations. In the following subsections we give a brief explanation of the parallelization process.

3.5. *Fast Fourier Transform*

The FFT is a well-known method for computing the discrete Fourier transform Eq. (18) of a complex vector of length N in $O(N\log N)$ operations. It can concisely be written as a decomposition of the Fourier matrix F_N,

$$F_N = A_N \cdots A_4 A_2 P_N, \tag{23}$$

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

$$A_K = I_{N/K} \otimes B_K, \quad \text{for } K = 2, 4, \ldots, N, \tag{24}$$

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 in turn can be written as

$$B_K = \begin{bmatrix} I_{K/2} & \Omega_{K/2} \\ I_{K/2} & -\Omega_{K/2} \end{bmatrix}. \tag{25}$$

Here, the matrix $I_{K/2}$ is the $K/2 \times K/2$ identity matrix and $\Omega_{K/2}$ is the $K/2 \times K/2$ diagonal matrix

$$\Omega_{K/2} = \mathrm{diag}(1, e^{\frac{2\pi i}{N}}, e^{\frac{4\pi i}{N}}, \ldots, e^{\frac{(N-2)\pi i}{N}}). \tag{26}$$

This matrix decomposition naturally leads to an algorithm, which is commonly called the *radix-2 FFT*.[8,21]

Performing a Fourier transform on a vector $\mathbf{z}$ of length N is equivalent to multiplying it with the Fourier matrix F_N. This can best be done by first permuting and then multiplying the vector successively by all the matrices A_K. The multiplications are thus carried out in $\log_2 N$ stages, each with N/K times a butterfly computation. One butterfly computation modifies $K/2$ pairs $(z_j, z_{j+K/2})$ by adding a multiple of $z_{j+K/2}$ to z_j and subtracting the same multiple.

The main choice in developing a parallel FFT is the data distribution for each stage of the computation. It is natural to start with the block distribution, since this renders all butterfly computations local, as long as $K \leq N/p$. In that case, the butterfly matrices are multiplied with a vector block of length K which is completely contained within the local block of the processor, which has length N/p. (Note that blocks are always properly aligned, since the K and N/p are both powers of two.) As a result, the first $\log_2 N - \log_2 p$ stages are local.

To finish the computation, it is convenient to use the cyclic distribution, which is formally defined as follows.

Definition 3.2. (Cyclic distribution). Let $\mathbf{z}$ be a vector of size N. We say that $\mathbf{z}$ is *cyclically distributed* over p processors if for all j, the element z_j is stored in $\mathrm{Proc}(j \bmod p)$ and has local index $j' = j \operatorname{div} p$.

For the cyclic distribution, the butterflies are local provided $K \geq 2p$. In that case, the pair of components to be modified is at distance $K/2 \geq p$ and hence p is a divisor of $K/2$; therefore both components j and $j + K/2$ are on the same processor. As a result, the last $\log_2 N - \log_2 p$ stages are local.

Our approach for the parallel FFT is to start with the block distribution and after $\log_2 N - \log_2 p$ stages switch to the cyclic distribution. (Note that this is equivalent to permuting the vector $\mathbf{z}$.) This can be done if $\log_2 N - \log_2 p \geq \frac{1}{2}\log_2 N$ (i.e., $p \leq \sqrt{N}$). If, however, $p > \sqrt{N}$, the use of the block distribution is exhausted before we can use the cyclic distribution. In that case, other intermediate distributions must be used, see McColl.[14]

We perform the inverse transform by reversing the stages of the algorithm and inverting the butterflies, instead of taking the more common approach of using the same algorithm, but replacing the powers of $e^{\frac{2\pi i}{N}}$ by their conjugates and multiplying by an rescaling factor. This choice enables us to eliminate certain permutations, see the next subsection.

3.6. *Optimization of the Main Loop*

Breaking open the FChT module allows us to radically reduce the amount of communication involved in the parallel FLT algorithm. As a consequence, the amount of local copy operations and computations is also reduced, but to a lesser extent.

The original modular parallel algorithm for the FChT of two vectors $\mathbf{x}$ and $\mathbf{y}$ of size N block distributed over p processors, $p \leq \sqrt{N}$, has the following structure:

1. PACK vectors $\mathbf{x}$ and $\mathbf{y}$ as the auxiliary complex vector $\mathbf{z}$ by permuting them using Eq. (17).
2. TRANSFORM vector $\mathbf{z}$ using an FFT of size N.
   (a) Perform a bit-reversal permutation in $\mathbf{z}$.
   (b) Perform the butterflies of size $2, 4, \ldots N/p$.
   (c) Permute $\mathbf{z}$ to the cyclic distribution.
   (d) Perform the butterflies of size $2N/p, 4N/p, \ldots, N$.
   (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 Eq. (19).
   (c) Permute $\mathbf{z}$ to block distribution and store the result in vectors $\mathbf{x}$ and $\mathbf{y}$.

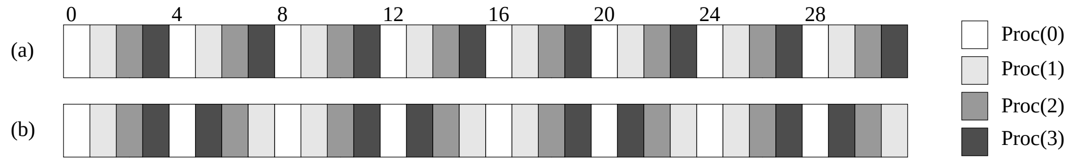

FIGURE 4. (a) Cyclic distribution and (b) zig-zag cyclic distribution for a vector of size 32 distributed over 4 processors.

In our optimized version where modularity is not an issue, we restrict the number of processors slightly further to $p \leq \sqrt{N/2}$ and permute the vector $\mathbf{z}$ from block distribution to a slightly modified cyclic distribution defined as follows.

Definition 3.3. (Zig-zag cyclic distribution). Let $\mathbf{z}$ be a vector of size N. We say that $\mathbf{z}$ is *zig-zag cyclically distributed* over p processors if for all j, the element z_j is stored in Proc($j \mod p$) if $j \mod 2p < p$ and in Proc($-j \mod p$), if $j \mod 2p \geq p$ and has local index $j' = j \operatorname{div} p$.

With this distribution both the components j and $j + K/2$, with $2N/p \leq K \leq N$, needed by the butterfly operations and the components j and $N - j$ needed by the extract operation are in the same processor; thus we can avoid the permutations performed in phases (2e) and (3a) above. The same happens, though in reversed order, in the pack/transform phases of the parallel inverse FChT. Figure 4 illustrates the cyclic and zig-zag cyclic distributions.

By giving up the block distribution in the main loop of the FLT algorithm and instead maintaining the vectors $\mathbf{f_l}, \mathbf{g_l}, \mathbf{u_l}$, and $\mathbf{v_l}$ of size $2K$ in the zig-zag cyclic distribution of $p1$ processors, we can also save the permutations to convert from zig-zag cyclic to block distribution in phase (3c) of the FChT and from block to zig-zag cyclic distribution in the corresponding phase of the inverse FChT. To achieve this we replace the truncation operation, Algorithm 3, by a new truncation operation, namely the redistribution of vectors $\mathbf{f_l}, \mathbf{g_l}, \mathbf{u_l}$, and $\mathbf{v_l}$, now of size K, from the zig-zag cyclic distribution with $p1$ processors to the zig-zag cyclic distribution with $p1/2$ processors, storing the lower halves of vectors $\mathbf{u_l}$ and $\mathbf{v_l}$ in the upper halves of vectors $\mathbf{f_l}, \mathbf{g_l}$. Note that the initialization step must also be modified in order to give the input vectors of stage 2 in the zig-zag cyclic distribution of $p/2$ processors.

Furthermore, the optimized algorithm avoids the packing (1) and bit-reversal (2a) in the FChT just following the recurrence and their corresponding inverses in the inverse FChT preceding the recurrence. This is done by storing the recurrence coefficients permuted by the packing (1) and bit-reversal (2a) permutations. This works because the last two permutations form the inverse of the first two, 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 $(6\frac{N}{p}\log_2 p + 2\frac{N}{p})g + (3\log_2 p + 1)l$. This means that we reduced communications and synchronizations by more than a factor of two. (The basic algorithm has a communication and synchronization cost of $14\frac{N}{p}\log_2 p\, g + 7\log_2 p\, l$.)

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. To make our code more competitive we used a modified radix-2 algorithm. 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.[21]) 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 after any number of stages, and not only after an even number of them.

Supposing N and p are powers of 4, i.e., we can always take pairs of stages together, the total cost of the optimized algorithm is:

$$T_{FLT} = 4.25\frac{N}{p}(\log_2 N)^2 + 26.25\frac{N}{p}\log_2 N - (4.25(\log_2 M)^2 + 26.25\log_2 M + M)\frac{N}{p} + \left(6\frac{N}{p}\log_2 p + 2\frac{N}{p}\right) g + (3\log_2 p + 1)\, l.$$

4. 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 set $M = 2$, i.e., no early termination. We implemented the algorithm in ANSI C using the BSPlib communications library. The test runs were made on a Cray T3E with up to 64 processors, each having a theoretical peak speed of 600 Mflop/s.

We tested the accuracy of our implementation by measuring the error obtained when transforming an arbitrary input vector $\mathbf{f}$ with elements uniformly distributed between 0 and 1. Table 1 shows the relative errors obtained for various problem sizes. The relative errors were computed via the expression

$$\frac{||\hat{\mathbf{f}} - \hat{\mathbf{f}}^*||_{max}}{||\hat{\mathbf{f}}||_{max}},$$

where $\hat{\mathbf{f}}$ is the exact transform (computed by a quadruple precision direct Legendre transform) and $\hat{\mathbf{f}}^*$ the FLT; $||\cdot||_{max}$ indicates the max norm.

Table 1 Estimated relative errors for the FLT algorithm.

N	relative error
1024	7.8×10^{-14}
8192	1.3×10^{-13}
65536	2.6×10^{-12}

We tested the scalability of our parallel implementation using our sequential implementation as basis for comparison. Though we broke open the modules of the algorithm, it is still possible (with a certain amount of work) to replace the FFT subroutine by a highly optimized or even a machine specific, assembler coded, FFT subroutine in both the sequential and the parallel versions. This would yield an even faster program.

Table 2 shows the timing results obtained for the sequential and parallel versions executed on up to 64 processors. It is better to analyze these results 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 5 shows the performance ratios obtained for various input sizes on up to 64 processors.

Table 2 Timing data for BSP_FLT on a Cray T3E. All times are given in milliseconds.

N	seq	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$
512	1.71	1.89	1.23	0.80	0.58	0.61	–	–
1024	3.95	4.36	2.70	1.57	1.08	0.84	–	–
8192	50.60	65.70	33.60	17.40	8.71	5.16	3.38	3.34
65536	1130.–	1250.–	664.–	336.–	162.–	71.10	36.10	20.30

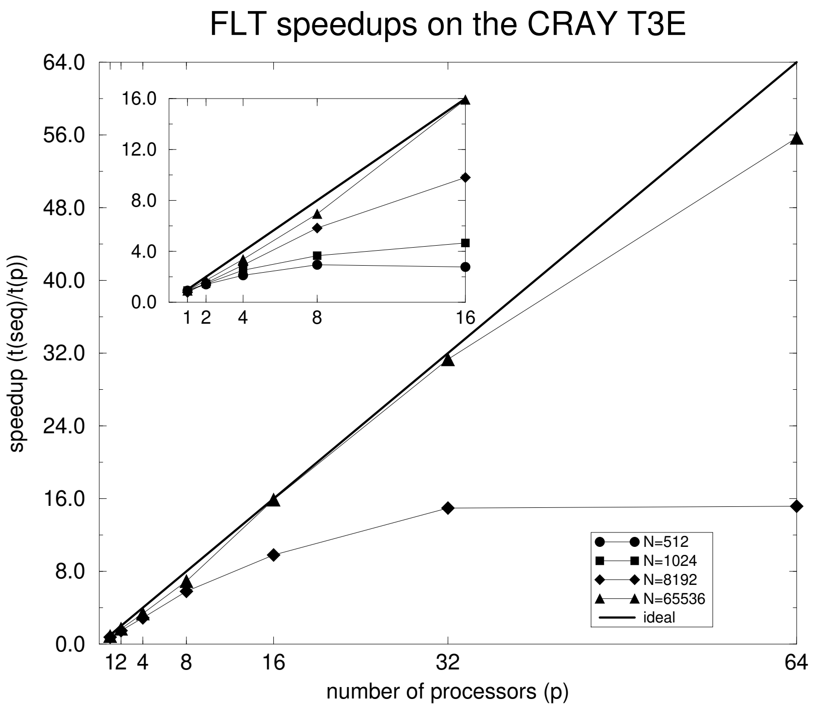

FIGURE 5. Scalability of the program BSP_FLT on a Cray T3E

It is clear that for a large problem size ($N = 65536$) the speedup is close to ideal, e.g., $S^{abs} = 56$ on 64 processors. For smaller problems, reasonable speedups can be obtained using 8 or 16 processors, but beyond that the communication time becomes dominant.

5. 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. We believe this implementation to be quite competitive for large problem sizes. Its complexity $O(N(\log_2 N)^2)$ 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. 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 three redistributions of the data vector in each of the first $\log_2 p$ stages of the algorithm. Two of these redistributions are already required by an FFT and an inverse FFT, indicating that this is close to optimal. Our parallelization approach was first to derive a basic algorithm that uses block and cyclic data distributions, and then to optimize this algorithm by removing permutations and redistributions wherever possible. To facilitate this we proposed a new data distribution, which we call the zig-zag cyclic distribution.

Within the framework of this work, we also developed a new algorithm for the simultaneous computation of two Chebyshev transforms. This is useful in the context of the FLT because the Chebyshev transforms always come in pairs, but such a double fast Chebyshev transform (and the corresponding double fast cosine transform) also has many applications in its own right. 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, and alternative choices of sampling points must be made possible.

6. Acknowledgements

We thank CAPES, Brazil, for supporting Inda with a doctoral fellowship and NCF, The Netherlands, for funding the computer time on the Cray T3E.

7. **References**

1. N. Ahmed, T. Natarajan, and K. Rao, *IEEE Trans. Comput.* **23** (1974) 90.
2. B. Alpert and V. Roklin, *SIAM J. Sci. Statist. Comput.* **12** (1991) 158.
3. S. Barros and T. Kauranne, *Parallel Computing* **20** (1994), 1335.
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, *J. Comput. Appl. Math.* **32** (1990) 311.
6. R. H. Bisseling, in *Lecture Notes in Computer Science* **1196** (Springer-Verlag, Berlin, 1997), p. 46.
7. G. L. Browning, J. J. Hack and P. N. Swarztrauber, *Mon. Wea. Rev.* **117** (1989) 1058.
8. J. W. Cooley and J. W. Tukey, *Math. Comp.* **19** (1965) 297.
9. J. Driscoll and D. Healy, *Adv. in Appl. Math.*, **15** (1994) 202.
10. J. R. Driscoll, D. Healy, and D. Rockmore. *SIAM J. Comput.* **26** (1997) 1066.
11. 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, Dept. of Math. and Com. Sci., Dartmouth College, NH, 1994).
12. 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, *Parallel Computing* **24** (1998) 1947.
13. D. K. Maslen, *A Polynomial Approach to Orthogonal Polynomial Transforms*, (Preprint MPI/95-9, Max-Planck-Institut für Mathematik, Bonn, Germany, 1995).
14. W.F. McColl, *Future Generation Computer Systems*, **12** (1996) 265.
15. S. Orszag, in *Science and Computers*, ed. G. Rota, (Academic Press, NY 1986), p. 23.
16. 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).
17. 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).
18. N. Shalaby and S.L. Johnsson, *Hierarchical Load Balancing for Parallel Fast Legendre Transforms*, (8th SIAM Conference on Parallel Processing for Scientific Computation, 1997).
19. G. Steidl and M. Tasche, *Mathematics of Computation*, **56** (1991) 281.
20. L. Valiant, *Communications of the ACM* **33** (1990) 103.
21. C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, (Society for Industrial and Applied Mathematics, SIAM, Philadelphia 1992).