Constructing Parallel Algorithms for Discrete Transforms:

From FFTs to Fast Legendre Transforms

Constructie van Parallelle Algoritmen voor Discrete Transformaties:
Van FFT's tot Snelle Legendre Transformaties

(met een samenvatting in het Nederlands)

PROEFSCHRIFT
ter verkrijging van de graad van doctor aan de Universiteit Utrecht
op gezag van de Rector Magnificus, Prof. dr. H. O. Voorma, ingevolge het besluit van het College voor Promoties in het openbaar
te verdedigen op woensdag 29 maart 2000 des middags te 4.15 uur

door

Márcia Alves de Inda

geboren op 19 augustus 1969 te Porto Alegre, Brazilië

Promotor: Prof. dr. Henk A. van der Vorst
Co-promotor: Dr. Rob H. Bisseling

Faculteit Wiskunde en Informatica
Universiteit Utrecht

2000 Mathematics Subject Classification: 65T50, 65Y05, 42C15.

Inda, Márcia Alves de
Constructing Parallel Algorithms for Discrete Transforms:
From FFTs to Fast Legendre Transforms
Proefschrift Universiteit Utrecht — Met samenvatting in het Nederlands.

ISBN 90-393-2330-5

The work described in this thesis was carried out at the Mathematics Department of Utrecht University, The Netherlands, with financial support by the Fundação Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES).

Aos carvoeiros[*]

---

[*] In my grandmother's Ega words: a family that is always together … my family.

Preface

The initial target of my doctoral research with parallel discrete transforms was to develop a parallel fast Legendre transform (FLT) algorithm based on the sequential Driscoll-Healy algorithm [**21, 22**]. To do this, I had to study their algorithm in depth. This task was greatly simplified thanks to previous work done by David K. Maslen [**35**], with whom Rob H. Bisseling and I worked together to crack this nut.

After understanding and implementing the sequential FLT algorithm, I aimed at developing a parallel distributed memory version. Using the bulk synchronous parallel (BSP) model [**32, 54**], and assuming that a parallel fast cosine transform (FCT) algorithm was available, it was easy to devise a basic parallel FLT algorithm. Such a basic parallel algorithm was already known [**30, 45**], though, to my knowledge, it had never been implemented. With the basic parallel algorithm at hand, I still had two things to do: develop a parallel FCT algorithm and investigate the possibility of improving the basic parallel FLT algorithm.

Developing a parallel FCT algorithm involved searching for a suitable sequential FCT algorithm, i.e., an algorithm that could be parallelized with a minimum of communication overhead. Since there is substantial knowledge of parallel fast Fourier transforms (FFT), we chose to restrict our search to the class of FCT algorithms that are based on FFTs, i.e., algorithms which (1) pack the input data, (2) transform them using an FFT, and then (3) extract the cosine transform from the transformed data. After going through many of those algorithms I finally decided to implement Narasimha and Peterson's algorithm [**37**], which I had found in van Loan's book [**55**]. The transform phase of this FCT algorithm consists of a real FFT (i.e., an FFT for real input data). In turn, a real FFT can be carried out using a complex FFT.

At this point, I had to deal with four different parallel discrete transform algorithms: FFTs, real FFTs, FCTs, and FLTs. After implementing basic versions of those four transforms, we turned to the problem of improving the basic parallel FLT algorithm. With the introduction of a new data distribution, which we call the zig-zag cyclic distribution, we were able to reduce the communication cost of the parallel real FFT and of the parallel FCT to the same cost as that of a parallel complex FFT of half the size. (Thus, obtaining the data packing and extract phases at no extra communication cost.) Breaking open the FCT module inside the FLT algorithm, further reduced the communication cost of the basic parallel FLT algorithm by a factor of three. This last optimization step was greatly facilitated by the introduction of the FCT2 algorithm, an algorithm that computes two discrete cosine transforms simultaneously.

A project that started with the aim of developing a parallel fast Legendre transform, ended up creating a collection of useful parallel transforms that can be used throughout in computational sciences. From solving numerical differential equations to signal processing, FFTs are among the most used numerical tools in computational sciences. As "real versions" of the FFT, the RFFT and the FCT are also very much in use. The FLT is still young and has to conquer its space between its better known cousins. The discrete Legendre transform, however, is widely used as part of two-dimensional Legendre transforms or (three-dimensional) spherical harmonic transforms. My parallel FLT is only a first step in developing parallel two-dimensional FLTs and parallel fast spherical harmonic transforms.

Together with this thesis, I intend to release a version of the Bulk Synchronous Parallel Fast Transform package (BSPFTpack) that I developed with the hope to bring parallel transforms within easy reach of anyone interested in parallel computing. This package was written in ANSI C and uses BSPlib as communication library, which is freely available. (An alternative would be to adapt the program to use another communication library such as MPI.) Together with the parallel package I also intend to release its sequential version.

My thesis has the following structure. In Chapter 1, I describe the BSP model and discuss relevant aspects of parallel computing. In Chapter 2, I derive a parallel FFT algorithm which serves as the basis for the rest of my thesis. In Chapter 3, I derive parallel algorithms for the RFFT, the FCT, and the FCT2. In Chapter 4, I introduce the Driscoll-Healy algorithm and derive a parallel FLT algorithm. Appendix A describes the BSP parameters for the Cray T3E, which is the parallel computer used for the numerical experiments. Appendix B presents the sine/cosine table used in my programs, and Appendices C and D contain material supplementary to the FLT chapter.

Contents

Preface v

List of Algorithms xi

1. *Basic Concepts in Parallel Computation* 1
- 1.1. Terminology 1
- 1.2. Bulk synchronous parallel model 2
- 1.3. Measuring the performance of a parallel algorithm: the BSP cost function 4
- 1.4. Data distributions and permutations 7
- 1.5. BSP algorithms 9
- 1.6. Implementation issues 11

2. *Fast Fourier Transform* 13
- 2.1. Introduction 13
  - 2.1.1. Background 14
  - 2.1.2. Brief introduction to parallel radix-2 FFTs 18
- 2.2. The parallel algorithm 20
  - 2.2.1. Basic idea 20
  - 2.2.2. Group-cyclic distribution family 21
  - 2.2.3. Fourier matrix decomposition 23
- 2.3. Implementation of the parallel algorithm 25
  - 2.3.1. Generalized butterflies 27
  - 2.3.2. Parallel bit reversal 29
  - 2.3.3. Permutations involving the group-cyclic family 32
  - 2.3.4. Permutation from block to cyclic distribution 33
  - 2.3.5. BSP cost 35
- 2.4. Variants of the algorithm 37
  - 2.4.1. Parallel FFT using other data distributions 37

2.4.2. Generalized butterfly phase with adjustable size 38
2.4.3. Cache-friendly parallel FFT 39
2.5. Experimental results and discussion 40
2.5.1. Accuracy 40
2.5.2. Performance of the sequential implementation: the need for cache-friendly algorithms 41
2.5.3. Scalability of the parallel implementation 42
2.6. Alternative algorithms 46
2.6.1. Six-pass algorithms and transpose algorithms 48
2.6.2. Parallel mixed-radix FFT: a generalized 6-pass algorithm 50
2.7. Conclusions and future work 54

3. *Real Fast Fourier Transform and Fast Cosine Transform* 57
3.1. Introduction 57
3.2. Parallel CFFT-based algorithms and the zig-zag cyclic distribution 59
3.2.1. Selection of a suitable sequential algorithm 59
3.2.2. Zig-zag cyclic distribution 61
3.2.3. Permutation from block to zig-zag cyclic distribution 63
3.2.4. Pairwise operations using the zig-zag cyclic distribution 66
3.3. Parallel algorithms and their inverses 72
3.3.1. Parallel real fast Fourier transform algorithm 72
3.3.2. Parallel fast cosine transform algorithm 73
3.3.3. Inverse transforms 75
3.4. Simultaneous fast cosine transform of two vectors 77
3.4.1. Derivation 78
3.4.2. Parallel forward transform 80
3.4.3. Parallel backward transform 83
3.5. Results and discussion 84
3.5.1. Accuracy 85
3.5.2. Performance of the real FFT 86
3.5.3. Performance of the FCT and FCT2 88
3.6. Conclusions 92

4. *Fast Legendre Transform* 93
4.1. Introduction 93
4.2. Driscoll-Healy algorithm 94
4.2.1. Orthogonal polynomials 94
4.2.2. Derivation of the Driscoll-Healy algorithm 98

4.2.3. Data representation and recurrence procedure 101
4.2.4. Early termination 104
4.2.5. Complexity of the algorithm 105
4.3. Basic parallel algorithm and its implementation 106
4.3.1. Data structures and data distributions 106
4.3.2. Basic parallel template 109
4.4. Improvements of the parallel algorithm 114
4.4.1. Optimization of the main loop 114
4.4.2. Optimized template 123
4.4.3. Parallel termination 125
4.5. Experimental results and discussion 127
4.5.1. Accuracy 127
4.5.2. Efficiency of the sequential implementation 128
4.5.3. Scalability of the parallel implementation 129
4.6. Conclusions and future work 131

Final Remarks and Future Work 133

Appendix A. *BSP Parameters on the Cray T3E* 135

Appendix B. *Lookup Table* 137

Appendix C. *Related Polynomial Transforms and Algorithms* 141

Appendix D. *Precomputation Algorithm for the FLT* 143

Bibliography 147

Summary 151

Samenvatting 155

Curriculum Vitae 159

List of Publications 161

Acknowledgments 163

List of Algorithms

2.1 Sequential radix-2 FFT algorithm. 16

2.2 Template for the parallel fast Fourier transform. 27

2.3 Template for the sequential generalized butterfly operations. 28

2.4 Template for the parallel bit reversal. 31

2.5 Template for the parallel permutation from r_1-cyclic to r_2-cyclic distribution. 33

2.6 Template for the parallel permutation from cyclic to block distribution. 35

2.7 Template for the parallel permutation from block to cyclic distribution. 36

2.8 Template for the parallel mixed-radix FFT. 53

3.1 Template for the parallel permutation from block to zig-zag cyclic distribution. 65

3.2 Template for the sequential generalized zig-zag butterfly operations. 69

3.3 Template for the sequential extract/pack phase of the RFFT. 70

3.4 Template for the sequential extract phase of the FCT. 72

3.5 Template for the parallel forward real fast Fourier transform. 73

3.6 Template for the parallel forward fast cosine transform. 75

3.7 Template for the parallel backward real fast Fourier transform. 77

3.8 Template for the sequential transposed conjugate generalized zig-zag butterfly operations. 78

3.9 Template for the parallel forward fast cosine transform of two vectors. 81

3.10 Template for the parallel ϱ permutation. 82

3.11 Template for the sequential extract phase of the FCT2. 82

3.12 Template for the parallel backward fast cosine transform of two vectors. 83

3.13 Template for the sequential pack phase of the backward FCT2. 84

4.1 Polynomial version of the Driscoll-Healy algorithm. 101
4.2 Recurrence procedure using the Chebyshev transform. 104
4.3 Driscoll-Healy algorithm. 106
4.4 Truncation procedure for the FLT. 110
4.5 Basic template for the parallel fast Legendre transform. 112
4.6 Termination procedure for the FLT. 113
4.7 Template for the optimized redistribution operation of the FLT. 119
4.8 Template for the (parallel) modified recurrence. 121
4.9 Template for the modified parallel forward FChT2. 122
4.10 Template for the optimized parallel fast Legendre transform. 124
4.11 Template for the (sequential) modified recurrence. 125
D.1 Precomputation of the point values needed by the FLT. 144

1

Basic Concepts in Parallel Computation

In this chapter, we establish a basic notation for presenting and discussing algorithms. We pay special attention to the case of parallel algorithms where the need to specify what each processor does at each moment poses an extra difficulty.

1.1. *Terminology*

The computational methods presented in this thesis can be divided into two classes:

1. Low level of detail algorithms: close to the mathematical notation; do not specify data structures.
2. High level of detail algorithms or *templates*: close to a programming language notation; make explicit use of data structures.

Low level of detail algorithms (or simply algorithms) will be used to explain ideas and methods; templates will be used to expose implementation aspects. The templates given are not necessarily completely optimized; the main objective is to present them in a clear, easy to understand, way.

Unless stated otherwise, we estimate the theoretical (time) complexity of an algorithm by counting the number of floating point operations (flops) it performs. Regardless of its type: addition, multiplication, or division, all flops have the same weight (equal to one). The flop count is summarized into a cost function that depends on the size of the input and, if applicable, on the number of processors. We designate the cost functions by a capital C with a subscript that indicates the algorithm. For example,

$$C_{\text{FFT-2}} = C_{\text{FFT-2}}(N) = 5N \log_2 N$$

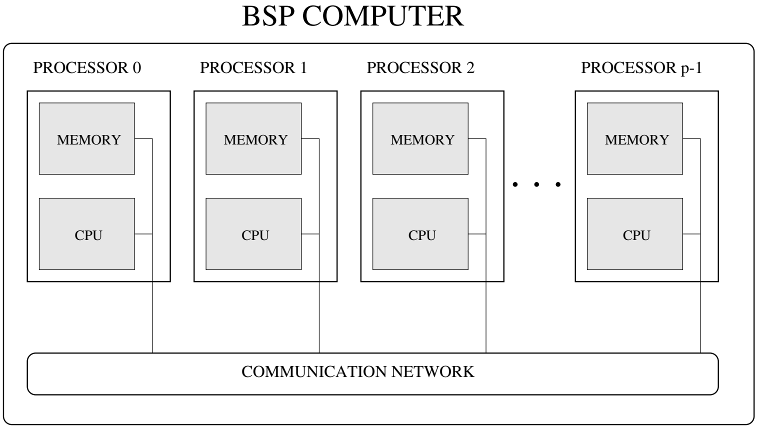


FIGURE 1.1. Schematic representation of a BSP computer.

is the cost of a radix-2 FFT algorithm for an input vector of N complex elements. Measuring the complexity of an algorithm in flops is convenient because this measure does not depend on the actual implementation or on the computer being used. Furthermore, a flop can be seen as a time unit (defined as the time needed to perform one flop). To estimate the execution time (in seconds) of an algorithm on a computer with speed v flop/s we just divide its cost function by v.

1.2. *Bulk synchronous parallel model*

The bulk synchronous parallel (BSP) model [**54**] is a parallel programming 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 BSP model, 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 (see Figure 1.1). Accessing local memory (the processor's own memory) is faster than accessing remote memory (memory owned by other processors), but access time is considered to be independent from the computer architecture. In this model, algorithms consist of a sequence of *supersteps* and *synchronization barriers*. The use of supersteps and synchronization barriers imposes a sequential structure on parallel algorithms, and this greatly simplifies the design process.

The variant of the BSP model that we use is a *single program multiple data* (SPMD) model, i.e., each one of the p processors executes a copy of the same program, though each processor has its own data. The program distinguishes between the processors through a parameter s (the processor identification number). Special cases are treated using "if" statements. In our model, a superstep is either a *computation superstep*, or a *communication superstep*. A computation superstep is a sequence of local computations carried out on data already available locally before the start of the superstep. A communication superstep consists of communication of data between processors. To ensure the correct execution of the algorithm, global synchronization barriers (i.e., places of the algorithm where all processors must synchronize with each other) precede and/or follow a communication superstep.

Communication between processors is carried out by two types of one sided communication primitives: *put* and *get*. We assume that both procedures are fully buffered [**32**], which means that all the data to be copied are read into a buffer before any write operation is performed:

- In a buffered put procedure, a processor copies data from an area of its local memory into a buffer, and then, after all copy-to-buffer operations have been performed, remotely writes the data into a memory area of another processor.
- In a buffered get procedure, a processor causes remote copying of data from a memory area of another processor into a buffer, and then, after all copy-to-buffer operations have been performed, writes the data into an area of its own local memory.

The following fragment (to be executed by processors $s = 0, 1, 2$) exemplifies a communication superstep containing both gets and puts. Figure 1.2 illustrates the sequence of events.

```
1. Synchronize
2. Put $\mathbf{a}$ in $\mathbf{a}$ of processor $(s+1) \bmod 3$.
   if $s = 0$ then
       for $j = 0$ to $2$ do
           Get $\mathbf{b}$ from processor $j$ and write it into $\mathbf{c}_j$.
3. Synchronize
```

The following rules are basic guidelines on where to use synchronization barriers.

1. A communication superstep containing put procedures must be followed by a synchronization barrier, so that the transferred data can be used in the next superstep.
2. A communication superstep containing get procedures must be preceded and followed by a synchronization barrier. The extra synchronization, before the superstep, is needed to ensure that the data being fetched is in place.

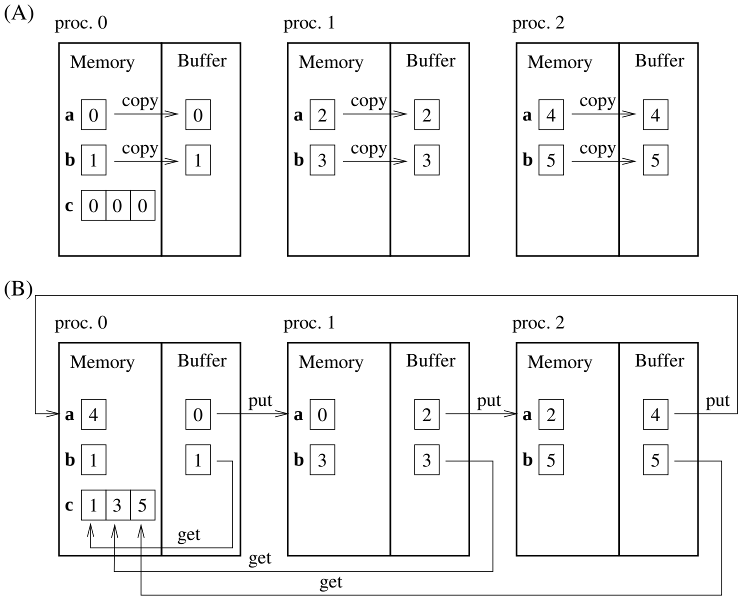


FIGURE 1.2. Schematic representation of a communication superstep. (A) First the data is copied into a buffer and then (B) it is written into the destination.

Using these two rules and ensuring that no two different sources write data into the same destination memory area ensures a well behaved BSP algorithm.

For further details and some basic techniques, see [**9, 32**]. The second reference describes BSPlib, a standard library defined in May 1997 which enables parallel programming in BSP style. The Paderborn University BSP (PUB) library [**12**] is another library that permits programming in BSP style; it provides the extra feature of subset synchronization.

1.3. *Measuring the performance of a parallel algorithm: the BSP cost function*

A BSP computer is characterized by four global parameters which depend on the specific machine being used:

- p, the number of processors;
- v, the computing velocity 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 v is used to estimate the total execution time after the cost function is computed. The flop count of a computation superstep is simply the maximum amount of work (in flops) of any processor. The flop count of a communication superstep is $hg + l$ or $hg + 2l$ (depending on the number of global synchronizations). Here h is the maximum number of data elements sent or received by any processor in that superstep. The cost function of an algorithm is obtained by adding the flops of the separate supersteps. This yields an expression of the form $a + bg + cl$. This cost function can be used to predict the execution time of a BSP algorithm in different parallel computers and to compare different parallel algorithms.

Scalability is an important concept when analyzing the performance of parallel algorithms. The scalability of a parallel algorithm refers to the performance of that algorithm as a function of the number of processors p. There are basically two approaches to the problem:

1. keep the problem size constant and increase the number of processors;
2. increase the problem size as a function of the number of processors.

The first approach deals with practical questions such as: “How many processors should I use to solve a problem of a certain size as fast as possible?” The second approach, also referred to as isoscalability analysis, deals with the “asymptotic” behavior of the algorithm.

A deep analysis of this issue is beyond the scope of this thesis. However, we need a minimal background to be able to evaluate the quality of our algorithms. To study the scalability of a parallel algorithm we compare its execution time (or estimated execution time) with the execution time of a good sequential algorithm that does the same job. For a fair comparison, one must choose a sequential algorithm that is practical and performs well on a sequential computer, but is also similar to the parallel algorithm being tested. For this reason, in this thesis we always compare optimized sequential and parallel versions of the same basic algorithm.

The *absolute speedup* of a parallel algorithm is defined by

$$S^{abs}(N,p) = \frac{\mathrm{Time}(N, seq)}{\mathrm{Time}(N,p)}, \tag{1.1}$$

where $\mathrm{Time}(N, seq)$ is the execution time of the sequential algorithm for a certain input size N and $\mathrm{Time}(N,p)$ is the execution time of the parallel algorithm for the same size on p processors. When estimating the scalability using the BSP cost function, $\mathrm{Time}(N, seq)$ may be replaced by the cost of the sequential algorithm $C_{\mathrm{seq}}(N)$ and $\mathrm{Time}(N,p)$ by the cost of the parallel algorithm $C_{\mathrm{par}}(N,p)$. The *absolute efficiency* of a parallel algorithm is defined by

$$E^{abs}(N,p) = \frac{S^{abs}(N,p)}{p}. \tag{1.2}$$

Ideally, $\mathrm{Time}(N,p) = \mathrm{Time}(N, seq)/p$, giving $S^{abs}(N,p) = p$ and $E^{abs}(N,p) = 1$. Though in theory superlinear speedups, i.e., $S^{abs}(N,p) > p$, are impossible, in practice the cache effect (or other causes) can lead to superlinear speedups, see Section 2.5 for a definition of cache effect and a discussion of superlinear speedups.

Information about the practical scalability of an algorithm can be obtained by considering its absolute speedup (or absolute efficiency) as a function of p for a fixed problem size. In general the absolute speedups will increase with p up to a certain number of processors p^* and then they will start to decrease. Our goal when designing a parallel algorithm is to achieve absolute speedups as close to p as possible (or absolute efficiencies as close to 1 as possible) for p as large as possible. An ideal p to run a certain problem size would be, for example, the largest p for which $E^{abs}(N,p) \geq a$, with $0 < a < 1$. In the overall picture, a parallel algorithm which scales well is an algorithm which can maintain high levels of efficiency[1] for various combinations of the problem size and the number of processors.

Of course small problems will never scale well on a large number of processors. For this reason, the notion of isoefficiency (or isoscalability) can be used to predict the overall (and asymptotic) behavior of a parallel algorithm. The idea is to predict how the problem size N depends on p if the efficiency level is maintained constant, i.e., in BSP terms,

$$E^{abs}(N,p) = \frac{C_{\mathrm{seq}}(N)}{p \cdot C_{\mathrm{par}}(N,p)} = a, \quad \text{with } 0 < a < 1. \tag{1.3}$$

Since the parallel algorithms presented in this thesis are perfectly load balanced with respect to computation and do not contain redundant computations, the computation cost of our algorithms is $C_{\mathrm{par},\mathtt{Comp}}(N,p) = C_{\mathrm{seq}}(N)/p$, so that (1.3) can be rewritten as

$$E^{abs}(N,p) = (1 + \frac{p \cdot C_{\mathrm{par},\mathtt{Comm}}(N,p)}{C_{\mathrm{seq}}(N)})^{-1} = a, \tag{1.4}$$

where $C_{\mathrm{par},\mathtt{Comm}}(N,p)$ is the total of both communication and synchronization costs. Defining $W = C_{\mathrm{seq}}(N)$ to be the total amount of work done by the algorithm and rearranging (1.4) gives

$$W = b \cdot p \cdot C_{\mathrm{par},\mathtt{Comm}}(N,p), \tag{1.5}$$

---

[1] The notion of what are high levels of efficiency can vary according to the difficulty of the problem being solved. But, in general, one could define it to be $E^{abs}(N,p) \geq 0.5$.

where $b = a/(1-a)$. Relation (1.5) implicitly defines the *isoefficiency function* $W = f(p)$ [**34**]. In simple cases, this function can be explicitly computed. Otherwise, we have to be contented with an asymptotic analysis. With this function, it is possible to compare the level of scalability of two algorithms independently of their purpose. We do this by classifying them by their asymptotic behavior. An ideal isoscalable parallel algorithm has $W = O(p)$, which means that a certain efficiency level a can be maintained if W grows (at least) linearly with p. We use the term $f(p)$ *isoscalable* to designate a parallel algorithm for which $W = O(f(p))$.

Note that in theory the BSP parameters g and l are constant, but in practice they may vary with p. This means that the theoretical isoefficiency function is only an indicator of the practical isoefficiency function, which will depend on the specific computer used. In Chapter 2, we give an example by deriving the asymptotic isoefficiency function for our parallel FFT algorithm (Section 2.3.5) and comparing theoretical predictions with experimental data (Section 2.5).

1.4. *Data distributions and permutations*

In the description of parallel algorithms it is important to specify how to distribute the elements of the data structures used in the algorithm over the processors. When distributing a vector of size N over p processors, it is often desirable that all processors receive the same number of elements. Obviously, this only happens if p divides N. Otherwise, our goal is to minimize the maximum number of elements in a processor. (This lowest maximum is achieved if the maximum number of elements in a processor is $\lceil N/p \rceil$.) Two common data distributions used to achieve this goal are the following. Let $\mathbf{f}$ be a vector of size N.

Definition 1.1 (Block distribution, $\mathrm{B}(p, N)$). We say that $\mathbf{f}$ is *block distributed* over p processors if, for all j, the element f_j is stored in processor $s = j \operatorname{div} b$ and has local index $j' = j \bmod b$, where $b = \lceil N/p \rceil$ is the block size.

Definition 1.2 (Cyclic distribution, $\mathrm{C}(p, N)$). We say that $\mathbf{f}$ is *cyclically distributed* over p processors if, for all j, the element f_j is stored in processor $s = j \bmod p$ and has local index $j' = j \operatorname{div} p$.

The operators "div" and "mod" are the quotient operator and remainder operator, respectively. To the extent used in this thesis, these two operators are defined as follows.

Definition 1.3 (Quotient and remainder operators). Let j be an integer and m be a positive integer. Then

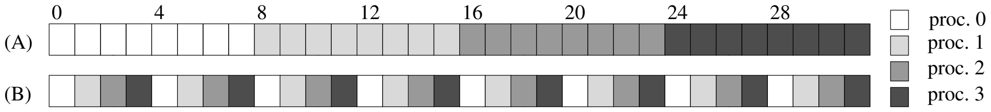


FIGURE 1.3. (A) Block distribution and (B) cyclic distribution for a vector of size 32 distributed over 4 processors. The block size b is 8.

1. the quotient operator, div, is defined by $j \operatorname{div} m = \lfloor \frac{j}{m} \rfloor$, and
2. the remainder operator, mod, is defined by $j \bmod m = j - \lfloor \frac{j}{m} \rfloor \cdot m$.

Though we often use brackets for clarity, in our notation we assume the following precedence order: (1) unary negation; (2) multiplication, division, quotient, and remainder; (3) addition and subtraction. Within the same category, evaluation is from left to right. This precedence rules are the same as used in the programming language ANSI C.[2]

Figure 1.3 gives an example of both the block and the cyclic distribution. Other, less trivial distributions can, and should, be used when the usual ones are not suitable. This is done, for instance, in Chapter 3 where we introduce the zig-zag cyclic distribution in connection with the computation of parallel fast cosine transforms.

There are basically two ways of looking at the same data distribution: the *logical view*, and the *storage view*. The logical view emphasizes the logical sequence of the elements in the vector while the storage view emphasizes the way the elements are actually stored. For the block distribution, both views are the same. Figure 1.4 illustrates the difference between the two views for the case of the cyclic distribution.

DEFINITION 1.4. Let u and N be integers such that u divides N. Then the *(perfect) shuffle* permutation $\sigma_{u,N}$ is defined by

$$\begin{aligned} \sigma_{u,N} : \{0,\ldots,N-1\} &\to \{0,\ldots,N-1\} \\ j &\mapsto k = (j \bmod u) \cdot \frac{N}{u} + j \operatorname{div} u. \end{aligned} \tag{1.6}$$

---

[2] The precedence rules imply the following:

1. $-a \bmod b = (-a) \bmod b$ (note that $-a \bmod b = [b - (a \bmod b)] \bmod b = (bd - a) \bmod b$, for any integer d);
2. $-a \operatorname{div} b = (-a) \operatorname{div} b$, which differs from$-(a \operatorname{div} b)$;
3. $a \cdot b \operatorname{div} c = (a \cdot b) \operatorname{div} c$, and $a \operatorname{div} b \cdot c = (a \operatorname{div} b) \cdot c$;
4. $a \operatorname{div} b \bmod c = (a \operatorname{div} b) \bmod c$, and $a \bmod b \operatorname{div} c = (a \bmod b) \operatorname{div} c$;
5. $a \cdot b \bmod c = (a \cdot b) \bmod c$, and $a \bmod b \cdot c = (a \bmod b) \cdot c$;
6. $a + b \bmod c = a + (b \bmod c)$, $a - b \bmod c = a - (b \bmod c)$, and $a \bmod b + c = (a \bmod b) + c$.

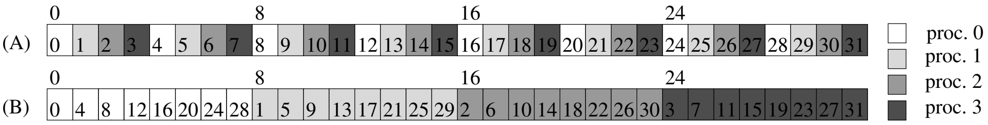


FIGURE 1.4. (A) Logical view and (B) storage view for a vector of size 32 cyclically distributed over 4 processors.

The name 'shuffle' comes from card games.[3] The inverse of $\sigma_{u,N}$ is $\sigma_{\frac{N}{u},N}$.

There exists an equivalence between redistributing a vector and permuting it. For example, if a vector $\mathbf{f}$ is permuted by moving each component j into the new position $\sigma_{p,N}(j)$, then the resulting distributed array can be seen as storing the permuted vector in the block distribution or, alternatively, storing the original vector in the cyclic distribution.

In general, it is easier to develop and understand the ideas behind a parallel algorithm by using the terminology of distributions and visualizing the vectors and other data structures involved in the logical view. But the terminology of distributions is too vague to be used when writing down a parallel template, since it does not directly specify where the element with global index j is stored. For this reason, in a template it is better to use the terminology of permutations (thus, maintaining the block distribution throughout the template) and visualizing the data structures in the storage view.

The distribution of other types of data structures such as matrices is done in a similar way. For example, we could distribute the rows (or columns) of a matrix using the block (or cyclic) distribution, or we could distribute both rows and columns using a two dimensional *Cartesian* distribution as illustrated in Figure 1.5.

1.5. *BSP algorithms*

Besides specifying the data distribution, the description of a BSP algorithm requires identifying the processors by a unique name and exposing the superstep structure. In the list that follows we introduce the terminology used in our parallel algorithms.

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

---

[3] Suppose that you want to permute a vector by $\sigma_{u,N}$. A nice mental picture for remembering this permutation is to imagine that each element of the vector to be permuted is a card, and that you need to cyclically redistribute those cards between u players. You reserve a fraction of $\frac{N}{u}$ consecutive spaces of the vector for each player, and then redistribute the elements as if cyclically dealing the cards between the u players.

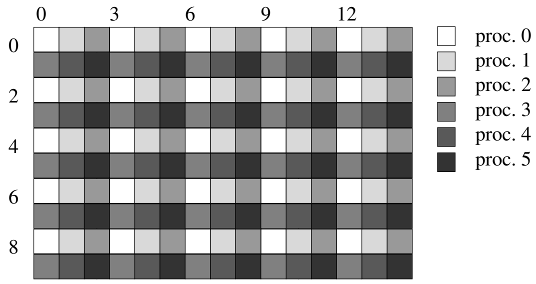


FIGURE 1.5. Two-dimensional Cartesian distribution over 2×3 processors (logical view). The rows of the matrix are cyclically distributed over 2 processor groups. In each row, the elements are cyclically distributed over the 3 processors of its group.

- **Supersteps.** Each superstep is numbered textually and labeled according to its type: (`Comp`) computation superstep, (`Comm`) communication superstep, (`CpCm`) subroutine containing both computation and communication supersteps. Global synchronizations are explicitly indicated by the keyword *Synchronize*. Supersteps inside loops are executed repeatedly, though they are numbered only once.
- **Indexing.** All the indices of vectors or array structures are global. This means that array elements have a unique index which is independent of the processor that owns it. This property 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. (In an actual implementation, it is more convenient to convert the indexing scheme to a local one.)
- **Data distributions.** When explaining ideas or describing parallel algorithms, we use the terminology of distributions. When describing parallel templates, we use the terminology of permutations so that the global indices of the data structures always refer to a block distribution.
- **Communication.** Communication between processors is indicated using

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

and

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

The first operation puts n elements of array $\mathbf{f}$, starting from element i, into processor pid and stores them there in array $\mathbf{g}$ starting from element j. The second operation gets n elements of array $\mathbf{g}$ starting from element j from processor *pid* and stores them locally in vector **f** starting from element i. Subscripts are not needed when the first element of the array is 0 or when communicating scalars. When communicating more than one element, we use boldface to emphasize that we are dealing with a vector and not with a scalar.

1.6. *Implementation issues*

In the BSP model, as with any other model, the transition from theory to practice can bring up additional issues. When implementing a BSP algorithm, it is good practice to exploit the possibilities of sending data in large packets and of using unbuffered communication instead of buffered communication.

In the BSP model, the cost of a communication superstep is determined by the maximum amount of data sent or received by a processor. This definition implies that a code that communicates data in small packets, e.g.,

```
for $j = 0$ to $n - k$ step $k$ do
    $\mathbf{g_{i+j}} \leftarrow \mathrm{Put}(pid, k, \mathbf{f_j})$
```

has the same cost as a code that communicates larger packets,

$$\mathbf{g_i} \leftarrow \mathrm{Put}(pid, n, \mathbf{f})$$

In an actual implementation, this assumption may not be true. Depending on the BSP implementation and on the computer being used, the overhead of sending the corresponding address information together with the actual data is significant for small k. Figure 1.6 shows an example of the influence of packet size using the Oxford BSPlib library[4] on a Cray T3E. The communication time per word drops from ~ 1.70 μs/word to ~ 0.19 μs/word when k increases from 1 to 10, and continues dropping, though less intensively (to ~ 0.08 μs/word for $p = 64$, or to ~ 0.03 μs/word for $p = 4$), for k up to 500. For this reason, implementations that communicate data using large packets are, in general, much faster than implementations that communicate single elements. In cases where the data to be sent is non-contiguous, but still regular, the extra effort of packing the data before sending them, and unpacking the data after receiving them generally pays off. In the irregular case, where address information must be sent togheter with the data, this is not true.

The use of buffered communication is not always necessary to guarantee the correctness of an algorithm. For example, no buffer is needed if a processor wants to read data from a memory area and it knows that this area will not be modified

---

[4] Freely available at `http://www.bsp-worldwide/implements/oxtool`

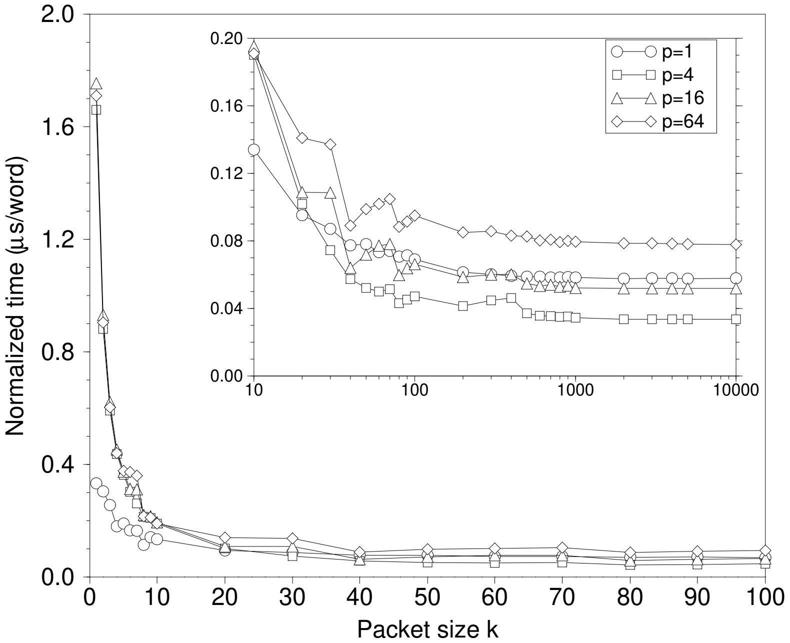


FIGURE 1.6. Influence of packet size when using the Oxford BSPlib library on a Cray T3E. The total message size is maintained at 10000 while the packet size is increased from $k = 1$ to 5000. Times measured using `bsp_hpputs`.

by any other processor in the same superstep. In such a case, the use of unbuffered communication is always cheaper, since buffered communication implies an intrinsic overhead: extra memory space is needed and extra time is required to copy the data to the buffer.

Unbuffered communication is naturally connected to the use of packing (or unpacking) operations. Generally speaking, packing (or unpacking) an array cannot be done in place, thus it requires an auxiliary array. The same auxiliary array can be used in writing a program that uses unbuffered communication.

Since the two aspects discussed above can significantly affect an implementation, whenever appropriate, we discuss methods of sending data in packets. We also write our templates in a style that permits unbuffered communication, whenever this does not compromise the clarity of the resulting template.

2

Fast Fourier Transform

2.1. *Introduction*

The *discrete Fourier transform* (DFT) plays an important role in computational science. DFT applications ranges from solving numerical differential equations to signal processing. (For an introduction to DFT applications see e.g. [**13, 19**].) The widespread use of DFTs in computational science is mainly due to the existence of fast algorithms, known by the general name of fast Fourier transform (FFT), which compute the DFT of an input vector of size N in $O(N \log N)$ operations instead of the $O(N^2)$ operations needed by a direct approach, i.e., by a matrix-vector multiplication.

In 1965, Cooley and Tukey [**16**] published a paper describing the FFT idea (giving special attention to the so called *Radix-2 FFT*). Since then, many variants of the algorithm have appeared. For an extensive discussion of the family of FFT algorithms, see Van Loan [**55**]. In recent years, after the dawn of parallel computing, the originally sequential FFT algorithms have been modified and adapted to the needs of parallel computation (see e.g. [**4, 5, 15, 23, 25, 26, 31, 34, 36, 50**]).

The lack of a unified parallel computing model and the existence of many different parallel architectures has made it rather difficult to develop efficient and portable parallel FFTs. Recently, however, as the parallel programming environments have become less machine dependent, examples of such algorithms have appeared. Typical examples are the 6-pass (or 6-step) approach (see, e.g., [**4, 26, 31**]) and the transpose approach [**25, 34**]. Those algorithms regard the input vector of size $N = N_0 N_1$ as an $N_0 \times N_1$ matrix, and carry out the computations in a similar way as done for two-dimensional FFTs. Since those algorithms require the number of processors p to be a divisor of N_0 and N_1, they can only be used if $p \leq \sqrt{N}$.

As the number of available processors grows and the communication speed increases, it is important to develop parallel algorithms that can handle more than $\sqrt{N}$ processors. Though generalized algorithms have already been proposed, they only work for very specific combinations of N and p such as $p = N^{(m-1)/m}/k$, where $1 \leq k \leq \sqrt{N}$ [**34**, Chap. 10.3] or $N = (N/p)^k$ [**36**], and both N and p are powers of two. Furthermore, to our knowledge none of those algorithms were implemented.

Our main aim in this chapter is to present a new parallel FFT algorithm and its implementation. Our parallel algorithm works for any $p < N$ as long as both are powers of two (which is required because of the radix-2 framework). We dedicate the remainder of this section to giving a brief introduction to the basic framework of radix-2 and radix-4 FFTs. In Section 2.2, we derive our parallel FFT algorithm by inserting suitable permutation matrices into the basic radix-2 decomposition of the Fourier matrix. This approach leads to a simple and easy to implement distributed memory parallel FFT algorithm. In Section 2.3, we present a set of templates that are used in the implementation of the algorithm. In Section 2.4, we present variants of our FFT algorithm. We show how to modify the algorithm to accept vectors that are not in the block distribution. We also show how to obtain a *cache-friendly* version of our algorithm, that is, an algorithm that takes advantage of the cache memory of a computer (i.e. a small but very fast memory) by breaking up the computations into small sections in such a way that the data stored in the cache is completely used before new data is brought in. In Section 2.5, we present results regarding the performance of our implementation and discuss aspects such as the *cache effect*. In Section 2.6, we discuss the 6-pass approach and the transpose approach as alternative approaches for the case that N is not a power of two. We also introduce the *parallel mixed-radix FFT algorithm*, which can be seen as a generalized 6-pass algorithm. We believe that our algorithm, which is based on the work of Agarwal and Cooley [**1**] for vector processors, is most promising, since it can compute FFTs of any combination of p and $N = N_0 N_1 \ldots N_{H-1}$ as long p divides each N/N_l and sequential FFTs of size N_l are available. We conclude the chapter with Section 2.7.

2.1.1. Background. The DFT of a complex vector $\mathbf{z}$ of size N is defined as the complex vector $\mathbf{Z}$, also of size N, with components

$$Z_k = \sum_{j=0}^{N-1} z_j e^{\frac{2\pi ijk}{N}}, \qquad 0 \leq k < N. \tag{2.1}$$

The inverse DFT, which transforms the complex vector $\mathbf{Z}$ back into the vector $\mathbf{z}$, is then defined by

$$z_j = \frac{1}{N} \sum_{k=0}^{N-1} Z_k e^{-\frac{2\pi i j k}{N}}, \qquad 0 \leq k < N. \tag{2.2}$$

Alternatively, the DFT can be seen as a matrix-vector multiplication:

$$\mathbf{Z} = F_N \cdot \mathbf{z}. \tag{2.3}$$

The complex matrix F_N is known as the $N \times N$ Fourier matrix; it has elements $(F_N)_{jk} = w_k^j$, where

$$w_k = e^{\frac{2\pi i}{k}}. \tag{2.4}$$

Note that

$$F_N^{-1} = \frac{1}{N} \bar{F}_N. \tag{2.5}$$

Though it is possible to develop FFT algorithms that compute the DFT of a vector of arbitrary size, the radix-2 FFT algorithm only works for N's that are powers of two. For simplicity, we will restrict our discussion to such values of N. The sequential iterative radix-2 FFT algorithm starts with the so-called *bit reversal* permutation of the input vector (see Section 2.3.2), and proceeds in $\log_2 N$ butterfly stages, numbered $K = 2, 4, \ldots, N$, which modify the vector. Each butterfly stage consists of N/K times a butterfly computation:

$$\begin{pmatrix} z_{t+j} \\ z_{t+j+K/2} \end{pmatrix} \leftarrow \begin{pmatrix} z_{t+j} + w_K^j \cdot z_{t+j+K/2} \\ z_{t+j} - w_K^j \cdot z_{t+j+K/2} \end{pmatrix}, \quad \text{for } j = 0, 1, \ldots, K/2 - 1, \tag{2.6}$$

where $t = 0, K, \ldots, N - K$ indicates the beginning of a butterfly block in the vector. These operations cost one complex multiplication and two additions, or 10 real flops, per pair. The total flop count of the radix-2 FFT is therefore

$$C_{\text{FFT-2}}(N) = 10 \cdot \frac{K}{2} \cdot \frac{N}{K} \cdot \log_2 N = 5N \log_2 N.$$

Algorithm 2.1 is an in-place version of this algorithm.

Following van Loan's matrix approach [**55**], Algorithm 2.1 can be described as a sequence of sparse matrix-vector multiplications which correspond to the following decomposition of the Fourier matrix[1]

$$F_N = A_{N,N} \cdots A_{8,N} A_{4,N} A_{2,N} P_N, \tag{2.7}$$

---

[1] Actually, the matrix decomposition corresponding to the algorithm of Cooley and Tukey [**16**] is $F_N = P_N \tilde{A}_{N,N} \cdots \tilde{A}_{8,N} \tilde{A}_{4,N} \tilde{A}_{2,N}$, where $\tilde{A}_{K,N} = P_N^{-1} A_{K,N} P_N$.

Algorithm 2.1 Sequential radix-2 FFT algorithm.

CALL Seq_CFFT$(N, \mathbf{y})$.

ARGUMENTS

N: Vector size; N is a power of 2 with $N \geq 2$.
$\mathbf{y} = (y_0^{in}, \dots, y_{N-1}^{in})$: Complex vector of size N.

OUTPUT $\mathbf{y} \leftarrow (y_0^{out}, \dots, y_{N-1}^{out})$, where $y_k^{out} = \sum_{j=0}^{N-1} y_j^{in} \exp(2\pi ikj/N)$.

DESCRIPTION

```
1. Perform a bit reversal on $\mathbf{y}$.
2. Perform $\log_2 N$ butterfly stages $A_{K,N}$ on $\mathbf{y}$.
   $K \leftarrow 2$
   while $K \leq N$ do
       for $t = 0$ to $N - K$ step $K$ do
           for $j = 0$ to $K/2 - 1$ do
               $a \leftarrow w_K^j \cdot y_{t+j+K/2}$
               $y_{t+j+K/2} \leftarrow y_{t+j} - a$
               $y_{t+j} \leftarrow y_{t+j} + a$
       $K \leftarrow 2 \cdot K$
```

where P_N is an $N \times N$ permutation matrix corresponding to the bit reversal permutation (step 1 of Algorithm 2.1), and the $N \times N$ matrices $A_{K,N}$ correspond to the butterfly stages (step 2 of Algorithm 2.1). The block structure of the butterfly stages leads to block-diagonal matrices of the form

$$A_{K,N} = I_{N/K} \otimes B_K, \tag{2.8}$$

which is shorthand for a block-diagonal matrix $\text{diag}(B_K, \dots, B_K)$ with N/K copies of the $K \times K$ matrix B_K on the diagonal. The symbol $\otimes$ represents the direct (or Kronecker) product of two matrices, which is formally defined at the end of this subsection. The matrix B_K is known as the $K \times K$ *2-butterfly matrix* which corresponds to the butterfly computation (2.6). This matrix can be written as

$$B_K = \begin{bmatrix} I_{K/2} & \Omega_{K/2} \\ I_{K/2} & -\Omega_{K/2} \end{bmatrix}. \tag{2.9}$$

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

$$\Omega_{K/2} = \text{diag}(w_K^0, w_K^1, \dots, w_K^{K/2-1}). \tag{2.10}$$

Later on we will also need generalized versions of $A_{K,N}$:

$$A_{K,N}^{\alpha} = I_{N/K} \otimes B_K^{\alpha}, \tag{2.11}$$

where B_K^α is the *generalized 2-butterfly matrix* [**5, 11, 18**]

$$B_K^\alpha = \begin{bmatrix} I_{K/2} & \Omega_{K/2}^\alpha \\ I_{K/2} & -\Omega_{K/2}^\alpha \end{bmatrix}, \tag{2.12}$$

which has the same form as the original B_k, but the weights w_k^j in (2.10) are replaced by $w_k^{j+\alpha}$, where α can be any real number.

In practice, often a radix-4 FFT is used. A radix-4 algorithm can be derived completely analogously to the radix-2 algorithm, yielding a similar matrix decomposition. The algorithm starts with a reversal of pairs of bits instead of a reversal of single bits, and proceeds in $\log_4 N$ *4-butterflies* stages which involve quadruples of vector components instead of pairs. Since 34 flops are performed per quadruple, this brings the flop count down to

$$C_{\text{FFT-4}}(N) = 34 \cdot \frac{K}{4} \cdot \frac{N}{K} \cdot \log_4 N = 4.25 N \log_2 N.$$

The resulting algorithm has the disadvantage that either it must be assumed that N is a power of four, or special precautions must be taken which complicate the algorithm.

We take a slightly different approach: wherever possible we take pairs of stages $A_{K,N} A_{K/2,N}$ together and perform them as one operation. Our $K \times K$ *4-butterfly matrix* has the form

$$D_K = B_K(I_2 \otimes B_{K/2}) = \begin{bmatrix} I_{K/4} & \Lambda_{K/4}^2 & \Lambda_{K/4} & \Lambda_{K/4}^3 \\ I_{K/4} & -\Lambda_{K/4}^2 & i\Lambda_{K/4} & -i\Lambda_{K/4}^3 \\ I_{K/4} & \Lambda_{K/4}^2 & -\Lambda_{K/4} & -\Lambda_{K/4}^3 \\ I_{K/4} & -\Lambda_{K/4}^2 & -i\Lambda_{K/4} & i\Lambda_{K/4}^3 \end{bmatrix}, \tag{2.13}$$

where $\Lambda_{K/4}$ is the $K/4 \times K/4$ diagonal matrix

$$\Lambda_{K/4} = \operatorname{diag}(w_K^0, w_K^1, \ldots, w_K^{K/4-1}). \tag{2.14}$$

This matrix is a symmetrically permuted version of the radix-4 butterfly matrix [**55**].[2] This approach gives the efficiency of a radix-4 FFT algorithm, and the flexibility of treating a parallel FFT within the radix-2 framework. For example, if we wish to permute the data sometime during the computation, for reasons of data locality, this can happen after any stage, and not only after an even number of stages.

The property (2.5) is often used to obtain an algorithm for the inverse FFT:

$$F_N^{-1} = \frac{1}{N} \bar{F}_N = \bar{A}_{N,N} \cdots \bar{A}_{4,N} \bar{A}_{2,N} P_N. \tag{2.15}$$

---

[2] In verifying this, note that van Loan defines the weights to be $w_K = \exp(-\frac{2\pi i}{K})$.

The backward algorithm is basically the same as the forward one, the only difference being that the powers of w_K are replaced by their conjugates and that the final result must be rescaled.

Since $F_N^T = F_N$, it is also possible to compute the FFT and its inverse using a transposed algorithm:

$$F_N = P_N A_{2,N}^T A_{4,N}^T \cdots A_{N,N}^T, \tag{2.16}$$

and

$$F_N^{-1} = \frac{1}{N}\bar{F}_N^T = \frac{1}{N} P_N \bar{A}_{2,N}^T \bar{A}_{4,N}^T \cdots \bar{A}_{N,N}^T. \tag{2.17}$$

FFT algorithms obtained using (2.16) and (2.17) are commonly classified as decimation in frequency (DIF) FFTs, whereas algorithms obtained using (2.7) and (2.15) are classified as decimation in time (DIT) FFTs. These names come from applications in signal processing. See e.g. [**55**, Chap 1.9.4] for more details.

To finish this subsection we define the direct product of two matrices and give some properties that will be used in the course of the chapter.

DEFINITION 2.1 (Direct product). Let A be a $q \times r$ matrix and B be an $m \times n$ matrix. Then the direct product (or Kronecker product) of A and B is the $qm \times rn$ matrix defined by

$$A \otimes B = \begin{bmatrix} a_{0,0}B & \cdots & a_{0,r-1}B \\ \vdots & \ddots & \vdots \\ a_{q-1,0}B & \cdots & a_{q-1,r-1}B \end{bmatrix}.$$

As one would expect, the direct product is associative, but it is not commutative. Lemma 2.2 summarizes some direct product properties that follow directly from the definition. (See [**43, 55**] for other useful properties).

LEMMA 2.2 (Properties of the direct product). *The following holds.*

1. $(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$, *as long as the products are defined.*
2. $(I_m \otimes I_n) = I_{mn}$.
3. *If* A *and* B *are square matrices of order* m *and* n, *respectively, then* $(A \otimes I_n)(I_m \otimes B) = (A \otimes B) = (I_m \otimes B)(A \otimes I_n)$.
4. *If* A *and* B *are square matrices of order* n *such that* $AB = BA$, *then* $(I_m \otimes A)(I_m \otimes B) = (I_m \otimes B)(I_m \otimes A)$.

2.1.2. Brief introduction to parallel radix-2 FFTs. Since the introduction of parallel computers, and even before that, methods for parallelizing FFT algorithms have been proposed. The earliest methods produced parallel algorithms that, using p processors, carry out an FFT of size N in $O(\log p)$ computation supersteps, which are interleaved by $O(\log p)$ communication supersteps that need to communicate $O(\frac{N}{p})$ data elements (see e.g. [**15, 23, 50**], [**34**, Chap. 10.2]). Such methods appeared as a direct consequence of the divide and conquer structure of the radix-2 FFT algorithm. The paper [**15**] by Chu and George discusses several existing parallel algorithms of this type including three variants of their own. Restricting the discussion to the DIF FFT class and vector sizes that are powers of two, they present a common framework in which all the algorithms they discuss are reorderings from one another in the following sense.

Each butterfly stage K of an FFT of size N, performs pairwise operations that combine elements j and $j+K/2$ from the vector being transformed using the weight $w_K^{j \bmod K}$. Writing j in its binary representation $j = (j_{m-1}, \dots, j_0)_2$, where $m = \log_2 N$, we observe that elements j and $j+K/2$ differ only in bit $\log_2 K - 1$ and that $w_K^{j \bmod K} = w_K^{(j_{\log_2 K-1}, \dots, j_0)_2}$. If the ordering of the vector is changed, so that original element j is stored as element l, the butterfly stages must be modified to carry out the same operations. Since we can represent the new ordering using a permutation of the original bits, it is easy to know which elements to combine and what weights to use. For example, if $N = 16$ a possible reordering of the input vector could be $l = (j_0, j_2, j_1, j_3)_2$, where $j = (j_3, j_2, j_1, j_0)_2$. The butterfly stage corresponding to $K = 16$ should then combine elements $l = (j_0, j_2, j_1, 0)_2$ with $l+1 = (j_0, j_2, j_1, 1)_2$ using weights $w_{16}^{(j_2, j_1, j_0)_2}$.

In the parallel scenario, any group of $\log_2 p$ bits can be used to represent the processor number, while the remaining $\log_2(N/p)$ bits are used to represent the local index. If the bit corresponding to the next butterfly stage is one of the $\log_2(N/p)$ bits that represent the local index, then the next butterfly stage is local, otherwise communication is needed.

Swarztrauber [**50**] carries out a similar discussion. He starts with a more general formulation of the problem, where N is not restricted to powers of two and both DIF and DIT FFTs are discussed, but when discussing the distributed memory framework, he only considers FFTs on a hypercube, restricting both p and N to powers of two. The problem of the algorithms discussed in [**15, 23, 50**] and [**34**, Chap. 10.2] is that reorderings are carried out by means of exchanging one bit at a time. Since there are $\log_2 p$ bits in the processor part, $\log_2 p$ communication supersteps of size $O(\frac{N}{p}g + l)$ are needed. A less expensive solution to the problem is to exchange all the processor bits with a group of local bits corresponding to butterfly stages that were already performed. Since the communication cost of the permutation that exchanges many bits is the same as for exchanging one bit, the reduction in the communication cost is huge.

The basic idea for such algorithms already appears in the original Cooley and Tukey paper [**16**]. In their derivation of the FFT algorithm, they start by considering the case where N can be decomposed as $N = N_0 N_1$, and rewrite (2.1) as

$$Z_{k_1,k_0} = \sum_{j_1=0}^{N_1-1} \left(\sum_{j_0=0}^{N_0-1} z_{j_0,j_1} w_N^{j_0 k_0 N_1} \right) w_N^{j_1(k_1 N_0 + k_0)}, \quad 0 \le k_1 < N_1, 0 \le k_0 < N_0, \tag{2.18}$$

where $Z_{k_1,k_0} = Z_{k_1 N_0 + k_0} = Z_k$, and $z_{j_0,j_1} = z_{j_0 N_1 + j_1} = z_j$. Since $w_N^{j_0 k_0 N_1} = w_{N_0}^{j_0 k_0}$, the inner sum of (2.18) corresponds to a DFT of size N_0, which can be computed by the same process as before if N_0 is not prime. They remark that this process can be applied to any possible factorization of N, $N = N_0 \ldots N_{H-1}$ and that, if N is composite enough, real gains (over the $O(N^2)$ direct approach) can be achieved. Afterwards they derive the radix-2 algorithm by choosing N to be a power of two. If instead of decomposing N into its prime factors, we stop at a higher level, we obtain a decomposition for the FFT into a sequence of shorter FFTs that, in the parallel case, can be spread out over the processors. This is what happens in our FFT algorithm presented in the next section and in the algorithms discussed in Section 2.6.

2.2. *The parallel algorithm*

2.2.1. Basic idea. Since our parallel FFT algorithm is based on the radix-2 decomposition (2.7) of the Fourier matrix, N must be a power of two. For practical reasons N/p must be integer and therefore p must also be power of two.

Suppose, for simplicity, that $p \le \sqrt{N}$ (in other words, $p \le N/p$). In this case, our parallel FFT algorithm has two phases: in phase 0, it uses the block distribution to perform the first $\log_2(N/p)$ stages, i.e., those involving butterflies with $K \le N/p$ (which we call short distance butterflies), and in phase 1, it uses the cyclic distribution to perform the remaining $\log_2 p$ stages, i.e., those involving butterflies with $K > N/p$ (the long distance butterflies).

The general case, where p can be larger than $\sqrt{N}$, is a simple extension of the previous case. A total of $H = \lceil \log_2(N)/\log_2(N/p) \rceil = \lceil \log_{\frac{N}{p}} N \rceil$ phases is performed. (The number of phases H is the largest integer for which $(N/p)^{H-1} \le N$.) In phase 0, the short distance butterflies are performed in the block distribution. Afterwards, in each intermediate phase $1 \le J < H-1$ a group of $\log_2(N/p)$ butterfly stages (the medium distance butterflies) is performed in the cyclic distribution restricted to a subgroup of processors of size $(N/p)^J$. Note that, if $p \le N/p$, then $\log_{\frac{N}{p}} H \le 2$, which means that no intermediate phase is performed. Finally, in phase $H-1$, the remaining long distance butterflies are performed in the cyclic distribution over p processors. The distributions mentioned above are defined in the following subsection.

Our algorithm has only $O(\log N/\log p)$ communication supersteps of size $O(\frac{N}{p}g+l)$ which is a significant improvement over the $O(\log p)$ communication supersteps also of size $O(\frac{N}{p}g+l)$ of the algorithms discussed in the previous section. McColl [**36**] outlined a parallel FFT algorithm which is a special case of the FFT algorithm we present here. His algorithm only works for $N=(N/p)^H$. Furthermore, his algorithm sends the indices corresponding to the weights together with the data vector, increasing the communication costs unnecessarily.

2.2.2. Group-cyclic distribution family.

DEFINITION 2.3 (Cyclic distribution in r groups, $\mathrm{C}^r(p,N)$). Let r, p, and N be integers with $1\le r\le p\le N$, such that r divides both p and N. Let $\mathbf{f}$ be a vector of size N to be distributed over p processors organized in r groups. Define $M=N/r$ to be the size of the subvector of a group and $u=p/r$ to be the number of processors in a group. We say that $\mathbf{f}$ is *cyclically distributed in* r *groups* (or r*-cyclically distributed*) over p processors if, for all j, the element f_j has local index $j'=(j \bmod M)\operatorname{div} u$, and is stored in processor s_0+s_1, where $s_0=(j \operatorname{div} M)\cdot u$ is the number of the first processor in the group (i.e., the processor offset) and $s_1=(j \bmod M)\bmod u$ is the processor identification within the group.

We use the name *group-cyclic distribution family* to designate all the r-cyclic distributions generated by the same N and p. This family includes both the cyclic and the block distribution as extreme cases, $\mathrm{C}(p,N)=\mathrm{C}^1(p,N)$ and $\mathrm{B}(p,N)=\mathrm{C}^p(p,N)$. Figure 2.1 illustrates the use of the group-cyclic distribution family in a parallel FFT.

Let u, v, and N be integers such that u divides v and v divides N. We define the following permutation:

$$\begin{aligned}&\gamma_{u,v,N}:\{0,\ldots,N-1\}\to\{0,\ldots,N-1\}\\ &j=j_0\cdot M+j_1\cdot u+j_2\mapsto l=j_0\cdot M+j_2\cdot\frac{N}{v}+j_1,\end{aligned}\tag{2.19}$$

where $M=\frac{N}{v}u$, $j_0=j\operatorname{div}M$, $j_1=(j\bmod M)\operatorname{div}u$, and $j_2=j\bmod u$. Note that $\gamma^{-1}_{u,v,N}=\gamma_{\frac{N}{v},\frac{N}{u},N}$ and that $\gamma_{1,v,N}=\gamma_{N,N,N}$ is the identity permutation. Permutations γ and σ are closely related, cf. (1.6). Permuting a vector of size N by $\gamma_{u,v,N}$ can be achieved by dividing the vector into $r=v/u$ subvectors of size $M=N/r$, then performing a shuffle permutation $\sigma_{u,M}$ on each of the subvectors. This relation is expressed by

$$\gamma_{u,v,N}(j)=j\operatorname{div}M\cdot M+\sigma_{u,M}(j\bmod M),\tag{2.20}$$

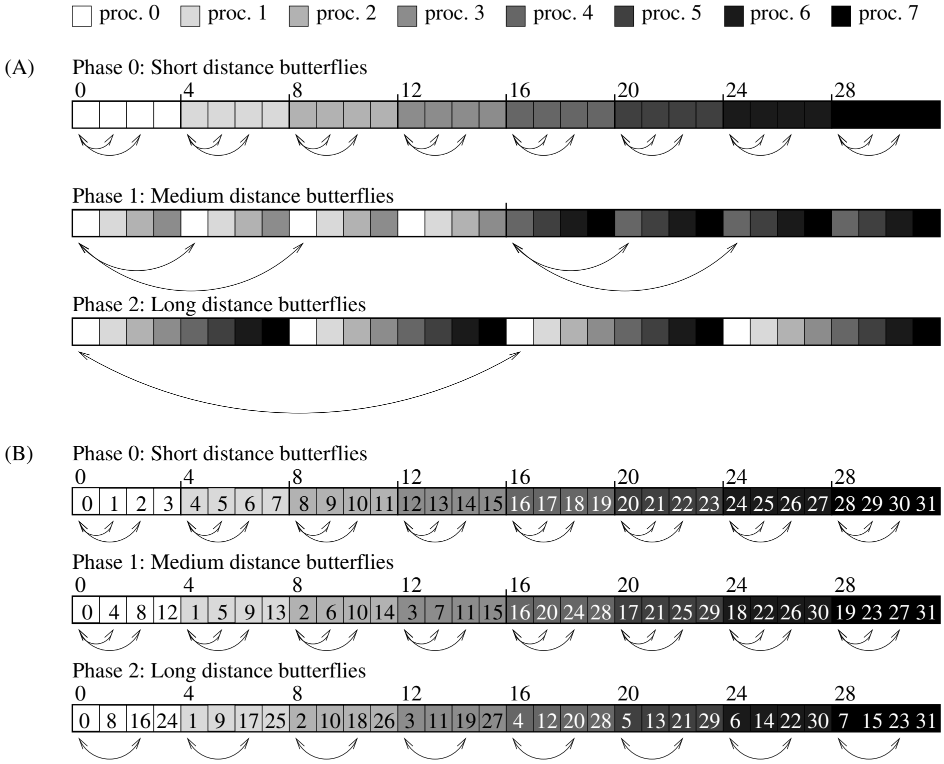


FIGURE 2.1. Butterfly operations using the group-cyclic distribution family $\mathrm{C}^r(p, N) = \mathrm{C}^r(8, 32)$. (A) Logical view, (B) storage view. The short distance butterflies ($A_{2,32}$ and $A_{4,32}$) are performed using the $\mathrm{C}^8(8, 32)$ distribution (block distribution). The medium distance butterflies ($A_{8,32}$ and $A_{16,32}$) are performed using the $\mathrm{C}^2(8, 32)$ distribution. The long distance butterflies ($A_{32,32}$) are performed using the $\mathrm{C}^1(8, 32)$ distribution (cyclic distribution). For clarity, not all butterfly pairs are shown.

which implies that $\gamma_{u,u,N} = \sigma_{u,N}$.

In the case that p divides N, redistributing a vector of size N from block distribution to r-cyclic distribution over p processors is equivalent to permuting it by $\gamma_{u,p,N}$, where $u = p/r$. Using matrix notation, this permutation is expressed by the $N \times N$ permutation matrix:

$$(\Gamma_{u,p,N})_{lj} = \begin{cases} 1, & \text{if } l = \gamma_{u,p,N}(j), \\ 0, & \text{otherwise.} \end{cases} \tag{2.21}$$

Multiplying a vector $\mathbf{y}$ by $\Gamma_{u,p,N}$ results in a vector with components $(\Gamma_{u,p,N}\mathbf{y})_l = y_{\gamma^{-1}_{u,p,N}(l)}$, for all l; in other words, this multiplication corresponds to redistributing the vector from block distribution to cyclic distribution in $r = p/u$ groups. The matrix corresponding to the inverse permutation $\gamma^{-1}_{u,p,N}$ is $\Gamma^{-1}_{u,p,N} = \Gamma^{T}_{u,p,N} = \Gamma_{\frac{N}{p},\frac{N}{u},N}$. From now on, we use the abbreviations Γ_u and γ_u to denote $\Gamma_{u,p,N}$ and $\gamma_{u,p,N}$, respectively. We restrict the use of subscripts p and N to cases where they are not obvious from the context. We also use $S_{u,M}$ to denote $\Gamma_{u,u,M}$. Note that $\Gamma_{u,p,N} = (I_r \otimes S_{u,\frac{N}{p}u})$, cf. (2.20).

2.2.3. Fourier matrix decomposition. To obtain the parallel FFT template, we modify the original radix-2 decomposition of the Fourier matrix (2.7) by inserting identity permutation matrices $I_N = \Gamma^{-1}_{u,p,N}\Gamma_{u,p,N} = \Gamma^{-1}_u\Gamma_u$ corresponding to the changes of distribution, and regrouping the matrices in the resulting decomposition. In the case that $p < \sqrt{N}$ this is done as follows:

$$\begin{aligned} F_N &= \Gamma_p^{-1}\Gamma_p \cdot A_{N,N} \cdot \Gamma_p^{-1}\Gamma_p \cdots \Gamma_p^{-1}\Gamma_p \cdot A_{2\frac{N}{p},N} \cdot \Gamma_p^{-1}\Gamma_p \cdot A_{\frac{N}{p},N} \ldots A_{2,N}P_N \\ &= \Gamma_p^{-1} \cdot \Gamma_p A_{N,N}\Gamma_p^{-1} \cdot \Gamma_p \cdots \Gamma_p^{-1} \cdot \Gamma_p A_{2\frac{N}{p},N}\Gamma_p^{-1} \cdot \Gamma_p \cdot A_{\frac{N}{p},N} \ldots A_{2,N}P_N. \end{aligned} \tag{2.22}$$

Since Γ_1 is the identity permutation, by defining

$$\hat{A}_{k,u,p,N} = \Gamma_{u,p,N}A_{ku,N}\Gamma^{-1}_{u,p,N}, \tag{2.23}$$

we can rewrite (2.22) as

$$F_N = \Gamma_p^{-1} \cdot \underbrace{\hat{A}_{\frac{N}{p},p,p,N} \cdots \hat{A}_{2\frac{N}{p^2},p,p,N}}_{\text{phase 1}} \cdot \Gamma_p\Gamma_1^{-1} \cdot \underbrace{\hat{A}_{\frac{N}{p},1,p,N} \cdots \hat{A}_{2,1,p,N}}_{\text{phase 0}} \cdot \Gamma_1 P_N. \tag{2.24}$$

As we did with the permutations γ, from now on we denote $\hat{A}_{k,u,p,N}$ by $\hat{A}_{k,u}$, reserving the indices p and N for when they are not obvious from the context. Following the same procedure as above, in the general case, we arrive to the following decomposition of the Fourier matrix:

$$\begin{aligned} F_N = \Gamma_p^{-1} &\underbrace{\hat{A}_{\frac{N}{p},p} \ldots \hat{A}_{2\frac{(N/p)^{H-1}}{p},p}}_{\text{phase } H-1} \Gamma_p \cdot \Gamma^{-1}_{(\frac{N}{p})^{H-2}} \underbrace{\hat{A}_{\frac{N}{p},(\frac{N}{p})^{H-2}} \ldots \hat{A}_{2,(\frac{N}{p})^{H-2}}}_{\text{phase } H-2} \Gamma_{(\frac{N}{p})^{H-2}} \cdot \ldots \\ &\ldots \cdot \Gamma^{-1}_{\frac{N}{p}} \underbrace{\hat{A}_{\frac{N}{p},\frac{N}{p}} \ldots \hat{A}_{2,\frac{N}{p}}}_{\text{phase 1}} \Gamma_{\frac{N}{p}} \cdot \Gamma_1^{-1} \underbrace{\hat{A}_{\frac{N}{p},1} \ldots \hat{A}_{2,1}}_{\text{phase 0}} \Gamma_1 \cdot P_N. \end{aligned} \tag{2.25}$$

Matrices $\hat{A}_{k,u}$ are block diagonal matrices with block size $\frac{N}{p}$:

$$\hat{A}_{k,u,p,N} = I_r \otimes \mathrm{diag}(A^{0/u}_{k,n}, A^{1/u}_{k,n}, \ldots, A^{(u-1)/u}_{k,n}), \tag{2.26}$$

where $r = p/u$, $n = N/p$, and $A^{\alpha}_{k,n}$ was defined previously, cf. (2.11). We shall formally state this as Corollary 2.6 which follows from Theorem 2.5. The proof of Theorem 2.5 uses the following lemma.

LEMMA 2.4. *Let u, M, and k be powers of two such that $u \le M$ and $2 \le k \le M/u$. Define $K = ku$. Let j be an index, $0 \le j < M$. Then*

1. *If $j \bmod K < K/2$, then $\sigma_{u,M}(j) \bmod k < k/2$.*
2. *If $j + K/2 < M$, then $\sigma_{u,M}(j + K/2) = \sigma_{u,M}(j) + k/2$.*
3. *If $j_1 = j \bmod \frac{M}{u}$, and $j_0 = j \operatorname{div} \frac{M}{u}$, then*

$$\frac{\sigma^{-1}_{u,M}(j) \bmod K}{K} = \frac{j_1 \bmod k + j_0/u}{k}.$$

PROOF. Part 1: $\sigma_{u,M}(j) \bmod k = (j \bmod u \cdot \frac{M}{u} + j \operatorname{div} u) \bmod k = (j \operatorname{div} u) \bmod k$. Now, $j \operatorname{div} u = (j \operatorname{div} K \cdot K + j \bmod K) \operatorname{div} u = j \operatorname{div} K \cdot k + (j \bmod K) \operatorname{div} u$. As a consequence $\sigma_{u,M}(j) \bmod k = (j \bmod K) \operatorname{div} u < (K/2) \operatorname{div} u = k/2$.

Part 2: $\sigma_{u,M}(j + K/2) = (j + K/2) \bmod u \cdot \frac{M}{u} + (j + K/2) \operatorname{div} u = j \bmod u \cdot \frac{M}{u} + j \operatorname{div} u + k/2 = \sigma_{u,M}(j) + k/2$.

Part 3: $\sigma^{-1}_{u,M}(j) \bmod K = (j \bmod \frac{M}{u} \cdot u + j \operatorname{div} \frac{M}{u}) \bmod K = (j_1 \cdot u + j_0) \bmod K = (j_1 \operatorname{div} k \cdot K + j_1 \bmod k \cdot u + j_0) \bmod K = j_1 \bmod k \cdot u + j_0$, which gives $(\sigma^{-1}_{u,M}(j) \bmod K)/K = (j_1 \bmod k \cdot u + j_0)/K = (j_1 \bmod k + j_0/u)/k$.

□

THEOREM 2.5. *Let u, M, and k be powers of two such that $u \le M$ and $2 \le k \le M/u$. Define $K = ku$ and $n = M/u$. Then*

$$S_{u,M} A_{K,M} S^{-1}_{u,M} = \operatorname{diag}(A^{0/u}_{k,n}, A^{1/u}_{k,n}, \ldots, A^{(u-1)/u}_{k,n}).$$

PROOF. To prove the theorem, it is sufficient to prove that

$$S_{u,M} A_{K,M} S^{-1}_{u,M} \mathbf{y} = \operatorname{diag}(A^{0/u}_{k,n}, \ldots, A^{(u-1)/u}_{k,n})\mathbf{y}, \quad \text{for all } \mathbf{y}.$$

First note that the vector $A_{K,M}\mathbf{x}$ can be described by

$$\begin{cases} (A_{K,M}\mathbf{x})_j = x_j + w_K^{j \bmod K} x_{j+K/2}, \\ (A_{K,M}\mathbf{x})_{j+K/2} = x_j - w_K^{j \bmod K} x_{j+K/2}, \quad 0 \le j \bmod K < K/2. \end{cases} \tag{2.27}$$

Let $\mathbf{x} = S^{-1}_{u,M}\mathbf{y}$ and $S_{u,M}(A_{K,M}\mathbf{x}) = \mathbf{z}$, and substitute $x_j = y_{\sigma_{u,M}(j)}$ and $z_{\sigma_{u,M}(j)} = (A_{K,M}\mathbf{x})_j$ in (2.27). This gives

$$\begin{cases} z_{\sigma_{u,M}(j)} = y_{\sigma_{u,M}(j)} + w_K^{j \bmod K} y_{\sigma_{u,M}(j+K/2)}, \\ z_{\sigma_{u,M}(j+K/2)} = y_{\sigma_{u,M}(j)} - w_K^{j \bmod K} y_{\sigma_{u,M}(j+K/2)}, \quad 0 \le j \bmod K < K/2. \end{cases} \tag{2.28}$$

Defining $l = \sigma_{u,M}(j)$ and applying Lemma 2.4 to j gives the following. Part 1 of Lemma 2.4 says that, if $j \bmod K < K/2$ then $l \bmod k < k/2$. Furthermore, by Part 2, $\sigma_{u,M}(j+K/2) = l+k/2$. Finally, applying Part 3 to l gives $w_K^{j \bmod K} = w_K^{\sigma^{-1}_{u,M}(l) \bmod K} = w_K^{u(l' \bmod k)+s_1} = w_k^{l' \bmod k+s_1/u}$, where $l' = l \bmod \frac{M}{u}$ and $s_1 = l \operatorname{div} \frac{M}{u}$.

Substituting the above results in (2.28) gives the following description of vector $\mathbf{z} = S_{u,M} A_{K,M} S^{-1}_{u,M} \mathbf{y}$:

$$\begin{cases} z_l = y_l + w_k^{l' \bmod k+s_1/u} y_{l+k/2}, & \\ z_{l+k/2} = y_l - w_k^{l' \bmod k+s_1/u} y_{l+k/2}, & 0 \leq l \bmod k < k/2. \end{cases} \tag{2.29}$$

Writing the index $l = s_1 \cdot n + (l' \operatorname{div} k) \cdot k + l' \bmod k$, it is easy to see that $z_l = (\operatorname{diag}(A^{0/u}_{k,n}, A^{1/u}_{k,n}, \ldots, A^{(u-1)/u}_{k,n}) \cdot \mathbf{y})_l$, proving the theorem. (See Figure 2.2B.) □

Corollary 2.6. *Let r, p, and N be powers of two with $1 \leq r \leq p < N$. Define $u = p/r$ and $n = N/p$. Let k be a power of two with $2 \leq k \leq n$. Then*

$$\hat{A}_{k,u,p,N} = I_r \otimes \operatorname{diag}(A^{0/u}_{k,n}, A^{1/u}_{k,n}, \ldots, A^{(u-1)/u}_{k,n}).$$

Proof. Define $K = ku$ and $M = N/r$. Then

$$\begin{aligned} \hat{A}_{k,u,p,N} &= \Gamma_{u,p,N} A_{K,N} \Gamma^{-1}_{u,p,N} = (I_r \otimes S_{u,M})(I_r \otimes A_{K,M})(I_r \otimes S^{-1}_{u,M}) \\ &= I_r \otimes (S_{u,M} A_{K,M} S^{-1}_{u,M}) = I_r \otimes \operatorname{diag}(A^{0/u}_{k,n}, A^{1/u}_{k,n}, \ldots, A^{(u-1)/u}_{k,n}). \end{aligned}$$

□

Starting from the Fourier matrix decomposition (2.25), it is easy to develop a parallel (BSP) FFT algorithm. Since all the matrices $\hat{A}_{k,u}$ are block diagonal matrices with block size equal to N/p, any multiplication $\hat{A}_{k,u} \cdot \mathbf{y}$ can be handled locally, provided that the vector $\mathbf{y}$ is in the block distribution. This property guarantees that matrix decomposition (2.25) detaches communication and computation completely. On the one hand, each generalized butterfly phase $(\hat{A}_{\frac{N}{p},u} \cdots \hat{A}_{2,u}) \cdot \mathbf{y}$, is a strict computation superstep. On the other hand, each permutation Γ_u is a strict communication superstep. In the next section we give a complete description of the resulting parallel algorithm.

2.3. *Implementation of the parallel algorithm*

Algorithm 2.2 is a direct implementation of the matrix decomposition (2.25). The input vector $\mathbf{y}$ is transformed in place. Superstep 1 permutes the input vector by a bit reversal: $\mathbf{y} \leftarrow P_N \cdot \mathbf{y}$. Superstep 2 carries out the short distance butterflies: $\mathbf{y} \leftarrow (A_{\frac{N}{p},N} \cdots A_{2,N}) \cdot \mathbf{y}$. (Since γ_1 is the identity permutation, $\hat{A}_{k,1} = A_{k,N}$, for $k = 2, \ldots, N/p$.) Superstep 3 permutes $\mathbf{y}$ to the r-cyclic distribution, with $r =$

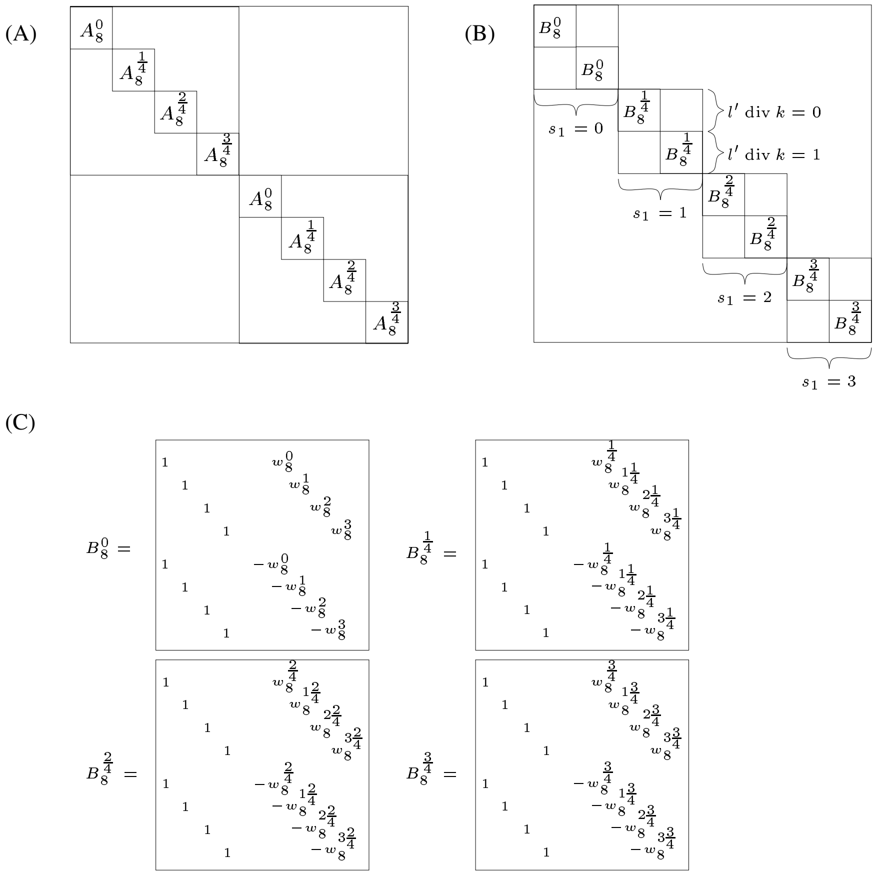


FIGURE 2.2. (A) Structure of the $N \times N$ matrix $\hat{A}_{k,u} = I_r \otimes \mathrm{diag}(A_{k,n}^{0/u}, A_{k,n}^{1/u}, \ldots, A_{k,n}^{(u-1)/u})$. Example with $N = 128$, $p = 8$, $r = 2$, $k = 8$, and hence $u = 4$, $K = 32$, and $n = 16$. (For clarity, the $A^{\alpha}_{k,n}$ are depicted as A^{α}_{k}) (B) Matrix $\mathrm{diag}(A_{k,n}^{0/u}, A_{k,n}^{1/u}, \ldots, A_{k,n}^{(u-1)/u})$. Note that s_1 is constant in each block $A_{k,n}^{s_1/u}$ and l' div k is constant in each block $B_k^{s_1/u}$. (C) Matrices $B_k^{s_1/u}$, with $s_1 = 0, \ldots, u-1$. The values s_1 and $l' \bmod k$ determine the exponent of the weights $w_k^{l' \bmod k + s_1/u}$.

$\max(1, p/(N/p))$: $\mathbf{y} \leftarrow \Gamma_{p/r} \cdot \mathbf{y}$. Each time superstep 4 is executed, it computes a group of medium distance butterflies: $\mathbf{y} \leftarrow (\hat{A}_{\frac{N}{p},(\frac{N}{p})^J} \ldots \hat{A}_{2,(\frac{N}{p})^J}) \cdot \mathbf{y}, 1 \leq J \leq H-2$. Superstep 5 prepares the vector for the next butterfly phase by permuting it to the r-cyclic distribution, with $r = \max(1, p/(N/p)^{J+1})$: $\mathbf{y} \leftarrow (\Gamma_{\frac{p}{r}} \Gamma^{-1}_{(\frac{N}{p})^J}) \cdot \mathbf{y}$. Superstep 6

Algorithm 2.2 Template for the parallel fast Fourier transform, using the group-cyclic distribution family.

```
CALL  BSP_CFFT($s, p, sign, N, \mathbf{y}$).
ARGUMENTS
    $s$: Processor identification; $0 \leq s < p$.
    $p$: Number of processors; $p$ is a power of 2 with $p < N$.
    $sign$: Transform direction; $+1$ for forward, $-1$ for backward.
    $N$: Transform size; $N$ is a power of 2 with $N \geq 2$.
    $\mathbf{y} = (y_0^{in}, \dots, y_{N-1}^{in})$: Complex vector of size $N$ (block distributed).
OUTPUT  $\mathbf{y} \leftarrow (y_0^{out}, \dots, y_{N-1}^{out})$, where $y_k^{out} = \sum_{j=0}^{N-1} y_j^{in} \exp(sign \cdot 2\pi ikj/N)$.
DESCRIPTION
$1^{\mathtt{CpCm}}$    Parallel bit reversal permutation.
        BSP_BitRev($s, p, N, \mathbf{y}$)
$2^{\mathtt{Comp}}$    Short distance butterflies.
        BTFLY($0, sign, \frac{N}{p}, 4, \mathbf{y}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{P}}}$)
$3^{\mathtt{CpCm}}$    Permutation to $\mathrm{C}^{\max(1,p/(N/p))}(p, N)$ distribution.
        BSP_BlockToCyclic($s - s \bmod \frac{N}{p}, s \bmod \frac{N}{p}, \min(p, \frac{N}{p}), \frac{N}{p}, \mathbf{y}_{(\mathbf{s}-\mathbf{s} \bmod \frac{\mathbf{N}}{\mathbf{P}})\frac{\mathbf{N}}{\mathbf{P}}}$)
        $H \leftarrow \lceil \log_{\frac{N}{p}} N \rceil$
        for $J = 1$ to $H - 2$ do
$4^{\mathtt{Comp}}$        Medium distance butterflies.
            BTFLY($\frac{s \bmod (N/p)^J}{(N/p)^J}, sign, \frac{N}{p}, 4, \mathbf{y}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{P}}}$)
$5^{\mathtt{Comm}}$        Permutation to $\mathrm{C}^{\max(1,p/(N/p)^{J+1})}(p, N)$ distribution.
            BSP_CyclicToCyclic($s, p, N, (N/p)^J, \min(p, (N/p)^{J+1}), \mathbf{y}$)
$6^{\mathtt{Comp}}$    Long distance butterflies.
        BTFLY($\frac{s}{p}, sign, \frac{N}{p}, 4\frac{(N/p)^{H-1}}{p}, \mathbf{y}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{P}}}$)
$7^{\mathtt{CpCm}}$    Permutation to $\mathrm{B}(p, N)$ distribution.
        BSP_CyclicToBlock($0, s, p, \frac{N}{p}, \mathbf{y}$)
```

carries out the long distance butterflies: $\mathbf{y} \leftarrow (\hat{A}_{\frac{N}{p},p} \dots \hat{A}_{2\frac{(N/p)^{H-1}}{p},p}) \cdot \mathbf{y}$. Finally, superstep 7 permutes the vector back to the block distribution: $\mathbf{y} \leftarrow \Gamma_p^{-1} \cdot \mathbf{y}$. Note that, to obtain the normalized inverse transform, the output vector must be divided by N. The subroutines used in the FFT template are described in the following subsections.

2.3.1. Generalized butterflies. The subroutine BTFLY (Algorithm 2.3) is a sequential subroutine that multiplies the input vector by $A_{n,n}^{\alpha} \dots A_{k_0,n}^{\alpha} A_{k_0/2,n}^{\alpha}$. Step 1 performs pairs of generalized butterfly operations. The $k - th$ iteration of the outermost while-loop performs the pair of generalized butterflies stages $A_{k,n}^{\alpha} A_{k/2,n}^{\alpha}$, the intermediate for-loop corresponds to the t-th repetition of the *generalized 4-butterfly*

$D_k^\alpha = B_k^\alpha(I_2 \otimes B_{k/2}^\alpha)$, cf. (2.13), which is computed by the innermost for-loop. Step 2 is only executed if the number of butterfly stages is odd. It computes the last generalized 2-butterfly $A_{n,n}^\alpha$. The FFT algorithm computes the desired (short, medium, or long distance) butterfly stages corresponding to phase J, $0 \leq J < H$, by defining the input parameter $\alpha = (s \bmod u)/u$, where $u = \min(p, (N/p)^J)$, and performing the generalized butterfly stages on the local part of the (permuted) vector $\mathbf{y}$ (i.e., the subvector $\mathbf{y}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}$ of size N/p that starts at element sN/p).

Algorithm 2.3 Template for the sequential generalized butterfly operations.

CALL BTFLY$(\alpha, sign, n, k_0, \mathbf{y})$.

ARGUMENTS

α: Butterfly parameter, used to compute the correct weights; $0 \leq \alpha < 1$.
$sign$: Transform direction; $+1$ for forward, -1 for backward.
n: Vector size; n is a power of 2 with $n \geq 2$.
k_0: Smaller 4-butterfly size; k_0 is a power of 2 with $4 \leq k_0 \leq 2n$.
$\mathbf{y} = (y_0, \ldots, y_{n-1})$: Complex vector of size n.

OUTPUT $\mathbf{y} \leftarrow A_{n,n}^\alpha \ldots A_{k_0,n}^\alpha A_{k_0/2,n}^\alpha \mathbf{y}$.

DESCRIPTION

```
1. Perform pairs of butterfly stages $A_{k,n}^\alpha A_{k/2,n}^\alpha$.
   $k \leftarrow k_0$
   while $k \leq n$ do
       for $t = 0$ to $n - k$ step $k$ do
           for $j = 0$ to $k/4 - 1$ do
               $yw1 \leftarrow w_k^{sign\cdot(j+\alpha)} \cdot y_{t+j+k/2}$
               $yw2 \leftarrow w_k^{sign\cdot 2(j+\alpha)} \cdot y_{t+j+k/4}$
               $yw3 \leftarrow w_k^{sign\cdot 3(j+\alpha)} \cdot y_{t+j+3k/4}$
               $a \leftarrow y_{t+j} + yw2$
               $b \leftarrow y_{t+j} - yw2$
               $c \leftarrow yw1 + yw3$
               $d \leftarrow yw1 - yw3$
               $y_{t+j} \leftarrow a + c$
               $y_{t+j+k/4} \leftarrow b + sign \cdot id$
               $y_{t+j+k/2} \leftarrow a - c$
               $y_{t+j+3k/4} \leftarrow b - sign \cdot id$
       $k \leftarrow 4 \cdot k$
2. Perform the last butterfly stage $A_{n,n}^\alpha$.
   if $k = 2n$ then
       for $j = 0$ to $n/2 - 1$ do
           $a \leftarrow w_n^{sign\cdot(j+\alpha)} \cdot y_{j+n/2}$
           $y_{j+n/2} \leftarrow y_j - a$
           $y_j \leftarrow y_j + a$
```

Using a lookup table (see Appendix B), the cost of an $A_{k,n}^\alpha A_{k/2,n}^\alpha$ butterfly operation is $34 \cdot \frac{k}{4} \cdot \frac{n}{k} = \frac{17}{2}n$. Summing over all pairs $A_{k,n}^\alpha A_{k/2,n}^\alpha$ and adding $5n$ flops for the last butterfly (if necessary) gives a cost of

$$\begin{aligned}
&C_{\mathrm{BTFLY}}(n,k_0) = \\
&\quad = \frac{17}{2}n\cdot[(\log_2 n-\log_2 k_0+2)\operatorname{div}2]+5n\cdot[(\log_2 n-\log_2 k_0+2)\bmod 2] \\
&\quad = \frac{17}{4}n\cdot[\log_2 n-\log_2 k_0+2]+\frac{3}{4}n\cdot[(\log_2 n-\log_2 k_0+2)\bmod 2],
\end{aligned} \tag{2.30}$$

for the generalized butterfly algorithm (Algorithm 2.3).

The total computation cost of our parallel FFT (Algorithm 2.2) is obtained by adding the costs of the butterfly phases:

$$C_{\mathrm{BTFLY}}(\frac{N}{p},4)=\frac{17}{4}\frac{N}{p}\log_2\frac{N}{p}+\frac{3}{4}\frac{N}{p}(\log_2\frac{N}{p}\bmod 2),$$

for phases $J=0$ to $H-2$, where $H=\lceil\log_{\frac{N}{p}}N\rceil$, and for the last phase $H-1$ if $(N/p)^{H-1}=p$, i.e., $H=\log_{\frac{N}{p}}N$. Otherwise, the cost for the last phase is

$$\begin{aligned}
C_{\mathrm{BTFLY}}(\frac{N}{p},4\frac{(N/p)^{H-1}}{p}) =& \frac{17}{4}\frac{N}{p}(\log_2 N\bmod\log_2\frac{N}{p}) \\
&+\frac{3}{4}\frac{N}{p}[(\log_2 N\bmod\log_2\frac{N}{p})\bmod 2].
\end{aligned}$$

This gives

$$\begin{aligned}
C_{\mathrm{FFT,par,\mathtt{Comp}}}(N,p)=\frac{17}{4}\frac{N}{p}\log_2 N+\frac{3}{4}\frac{N}{p}[&(\log_2\frac{N}{p}\bmod 2)(\log_2 N\operatorname{div}\log_2\frac{N}{p}) \\
&+(\log_2 N\bmod\log_2\frac{N}{p})\bmod 2],
\end{aligned} \tag{2.31}$$

where the second term corresponds to the extra cost we have to pay for performing 2-butterflies. The communication and synchronization costs of our parallel FFT are discussed in Section 2.3.5 after we discuss the parallel permutation subroutines.

2.3.2. Parallel bit reversal. The bit reversal matrix P_N is defined by

$$(P_N)_{jk}=\begin{cases}1, & \text{if } j=\mathrm{rev}_N(k),\\ 0, & \text{otherwise.}\end{cases} \tag{2.32}$$

Here, rev_N is the bit reversal permutation

$$\begin{aligned}
\mathrm{rev}_N:\{0,\ldots,N-1\}&\to\{0,\ldots,N-1\} \\
j=\sum_{l=0}^{m-1}b_l2^l&\mapsto k=\sum_{l=0}^{m-1}b_{m-l-1}2^l,
\end{aligned} \tag{2.33}$$

where $m = \log_2 N$ and $(b_{m-1} \dots b_0)_2$ is the binary representation of j. Note that $\mathrm{rev}_N^{-1} = \mathrm{rev}_N$, which means that $P_N^{-1} = P_N$.

The bit reversal permutation has the following very useful property.

LEMMA 2.7. *Let $u = 2^q$, $N = 2^m$, with $q \leq m$, and define $v = N/u$. Then*

$$\mathrm{rev}_N(j) = \mathrm{rev}_v(j \operatorname{div} u) + v \cdot \mathrm{rev}_u(j \bmod u), \quad 0 \leq j < N.$$

PROOF. Let $d = m - q$, so that $v = 2^d$. If the binary representation of j is $(b_{m-1} \dots b_0)_2$, then

$$j = j \bmod u + u \cdot (j \operatorname{div} u) = \sum_{l=0}^{q-1} b_l 2^l + 2^q \sum_{l=0}^{d-1} b_{l+q} 2^l.$$

Now,

$$\mathrm{rev}_N(j) = \sum_{l=0}^{m-1} b_{m-l-1} 2^l = \sum_{l=0}^{d-1} b_{m-l-1} 2^l + 2^d \cdot \sum_{l=0}^{q-1} b_{q-l-1} 2^l = a + v \cdot b.$$

If we show that $a = \mathrm{rev}_v(j \operatorname{div} u)$ and $b = \mathrm{rev}_u(j \bmod u)$, we are done. In fact,

$$\begin{aligned} a &= \sum_{l=0}^{d-1} b_{m-l-1} 2^l = \sum_{l=0}^{d-1} b_{m-q-l-1+q} 2^l = \quad \text{(substituting } c_l = b_{l+q}\text{)} \\ &= \sum_{l=0}^{d-1} c_{d-l-1} 2^l = \mathrm{rev}_v \left(\sum_{l=0}^{d-1} c_l 2^l \right) = \mathrm{rev}_v \left(\sum_{l=0}^{d-1} b_{l+q} 2^l \right) = \mathrm{rev}_v(j \operatorname{div} u) \end{aligned}$$

and

$$b = \sum_{l=0}^{q-1} b_{q-l-1} 2^l = \mathrm{rev}_u \left(\sum_{l=0}^{q-1} b_l 2^l \right) = \mathrm{rev}_u(j \bmod u).$$

□

COROLLARY 2.8. *Let $u \leq N$ be powers of two. Define $v = N/u$. Then*

$$P_N = (I_u \otimes P_v)(P_u \otimes I_v) S_{u,N}$$

PROOF. The matrix $(I_u \otimes P_v)(P_u \otimes I_v) S_{u,N}$ corresponds to a sequence of three permutations:

1. $j \to l = \sigma_{u,N}(j) = j \bmod u \cdot v + j \operatorname{div} u$
2. $l \to t = \mathrm{rev}_u(l \operatorname{div} v) \cdot v + l \bmod v = \mathrm{rev}_u(j \bmod u) \cdot v + j \operatorname{div} u$
3. $t \to k = t \operatorname{div} v \cdot v + \mathrm{rev}_v(t \bmod v) = \mathrm{rev}_u(j \bmod u) \cdot v + \mathrm{rev}_v(j \operatorname{div} u) = \mathrm{rev}_N(j)$

□

Let $\mathbf{y}$ be a vector of size $N = 2^m$ block distributed over $p = 2^q$ processors. Suppose that we want to permute it by a bit reversal permutation, i.e., perform $\mathbf{y} \leftarrow P_N \cdot \mathbf{y}$. Applying Corollary 2.8 with $u = p$, it is possible to split the parallel bit reversal permutation in two parts as follows.

1. $\mathbf{y} \leftarrow (P_p \otimes I_{\frac{N}{p}})S_{p,N} \cdot \mathbf{y}$, which is a global permutation that sends the elements to the correct processors, but with local indices still in the original order:

$$j \to t = \underbrace{\mathrm{rev}_p(j \bmod p)}_{\mathrm{Proc}(t)} \cdot \frac{N}{p} + \underbrace{j \operatorname{div} p}_{t'}.$$

   Having as basis the block distribution, from now on, we use $\mathrm{Proc}(k) = k \operatorname{div} \frac{N}{p}$ to denote the processor in which element k is stored, and $k' = k \bmod \frac{N}{p}$ to denote the local index of the element.

2. $\mathbf{y} \leftarrow (I_p \otimes P_{\frac{N}{p}}) \cdot \mathbf{y}$, which is a local bit reversal permutation in the local index t':

$$t' \to k' = \mathrm{rev}_{\frac{N}{p}}(t')$$

Algorithm 2.4 carries out a parallel bit reversal using this idea. If we combine the local bit reversal (superstep 2 from Algorithm 2.4) with the short distance butterfly phase (superstep 2 of Algorithm 2.2) we have a complete local sequential FFT. This means that we can easily replace the two supersteps by any optimized FFT subroutine we can lay our hands on.

Algorithm 2.4 Template for the parallel bit reversal.

CALL BSP_BitRev($s, p, n, \mathbf{y}$).

ARGUMENTS

s: Processor identification; $0 \le s < p$.
p: Number of processors; p is a power of 2 with $p < N$.
N: Vector size; N is a power of 2 with $N \ge 2$.
$\mathbf{y} = (y_0, \ldots, y_{N-1})$: Complex vector of size N (block distributed).

OUTPUT $\mathbf{y} \leftarrow P_N\mathbf{y}$.

DESCRIPTION

```
1^Comm  Global permutation.
        for $j = s\frac{N}{p}$ to $s\frac{N}{p} + \frac{N}{p} - 1$ do
            $dest \leftarrow \mathrm{rev}_p(j \bmod p)$
            $x_{dest\cdot\frac{N}{p} + j \operatorname{div} p} \leftarrow \mathrm{Put}(dest, 1, y_j)$
        Synchronize
2^Comp  Local bit reversal.
        for $t' = 0$ to $\frac{N}{p} - 1$ do
            $y_{s\frac{N}{p} + \mathrm{rev}_{\frac{N}{p}}(t')} \leftarrow x_{s\frac{N}{p} + t'}$
```

If $p < N/p$, it is possible to optimize communication superstep 1 of Algorithm 2.4 by sending packets of data. This is done in a similar way as when permuting from block to cyclic distribution (see Section 2.3.4); the only difference is in the destination processor, which is $\mathrm{rev}_p(j \bmod p)$ instead of $j \bmod p$.

2.3.3. Permutations involving the group-cyclic family. Permuting a vector from the $\mathrm{C}^{r_1}(p,N)$ distribution to the $\mathrm{C}^{r_2}(p,N)$ distribution, where $r_1 = p/u_1$ and $r_2 = p/u_2$ may be any possible group size, can be done as follows: first, use $\gamma_{u_1}^{-1}$ to permute the vector to the block distribution, and then use γ_{u_2} to permute it to the $\mathrm{C}^{r_2}(p,N)$ distribution. This operation is expensive if performed in parallel, because all the data have to be moved twice around the processors. The best approach is to combine both permutations into one:

$$\begin{aligned} \gamma_{u_1}^{u_2} : \{0,\ldots,N-1\} &\to \{0,\ldots,N-1\} \\ j &\mapsto l = \gamma_{u_2}(\gamma_{u_1}^{-1}(j)). \end{aligned} \tag{2.34}$$

(Note that $(\gamma_{u_1}^{u_2})^{-1} = \gamma_{u_2}^{u_1}$, and that $\gamma_{u_1}^{u_2}$ is an abbreviation for $\gamma_{u_1,p,N}^{u_2}$.) In the general case, there is no simple formula for computing the destination index l. Some combinations of the parameters r_1, r_2, p, and N, however, lead to simpler expressions for the destination index. The simplest case is when r_1 or r_2 is equal to p, i.e., one of the distributions involved is the block distribution. This situation occurs in supersteps 3 and 7 of the FFT algorithm (Algorithm 2.2) and is discussed in Section 2.3.4. Here, we discuss the special case $r_2 < r_1 < p$, with $\frac{r_1}{r_2} \le \frac{N}{p} < p$, which occurs in superstep 5 of the FFT algorithm. We also assume that N and p (and as a consequence $M_1 = N/r_1, M_2 = N/r_2, u_1$, and u_2) are powers of two.

Since $p > r_1 > r_2$, it follows that $\frac{N}{p} < M_1 < M_2$ and the original index j can be decomposed in the following way:

$$j = j_0 \cdot M_2 + j_1 \cdot M_1 + j_2 \cdot \frac{N}{p} + j_3,$$

where $j_0 = j \operatorname{div} M_2, j_1 = (j \bmod M_2) \operatorname{div} M_1, j_2 = (j \bmod M_1) \operatorname{div} \frac{N}{p}$, and $j_3 = j \bmod \frac{N}{p}$. The intermediate index $k = \gamma_{u_1}^{-1}(j)$ is then

$$k = j_0 \cdot M_2 + j_1 \cdot M_1 + j_3 \cdot u_1 + j_2.$$

In turn, k can be decomposed as

$$k = k_0 \cdot M_2 + k_1 \cdot u_2 + k_2,$$

where $k_0 = k \operatorname{div} M_2 = j_0$, $k_1 = (k \bmod M_2) \operatorname{div} u_2 = j_1 \cdot \frac{M_1}{u_2} + (j_3 \cdot u_1) \operatorname{div} u_2 = j_1 \cdot \frac{M_1}{u_2} + j_3 \operatorname{div} \frac{u_2}{u_1}$, and $k_2 = k \bmod u_2 = (j_3 \cdot u_1) \bmod u_2 + j_2 = (j_3 \bmod \frac{u_2}{u_1}) \cdot u_1 + j_2$.

The destination index l is then

$$l = k_0 \cdot M_2 + k_2 \cdot \frac{N}{p} + k_1 = j_0 \cdot M_2 + \left(j_3 \bmod \frac{u_2}{u_1} \cdot u_1 + j_2\right) \cdot \frac{N}{p} + j_1 \cdot \frac{M_1}{u_2} + j_3 \operatorname{div} \frac{u_2}{u_1}$$

$$= \underbrace{\left(j_0 \cdot u_2 + j_3 \bmod \frac{u_2}{u_1} \cdot u_1 + j_2\right)}_{\mathrm{Proc}(l)} \cdot \frac{N}{p} + \underbrace{j_1 \cdot \frac{M_1}{u_2} + j_3 \operatorname{div} \frac{u_2}{u_1}}_{l'} .$$

Algorithm 2.5 describes this permutation.

Algorithm 2.5 Template for the parallel permutation from r_1-cyclic to r_2-cyclic distribution.

CALL BSP_CyclicToCyclic$(s, p, N, u_1, u_2, \mathbf{y})$.

ARGUMENTS

s: Processor identification; $0 \le s < p$.
p: Number of processors; p is a power of 2 with $p < N$.
N: Vector size; N is a power of 2 with $N \ge 2$.
u_1: Number of processors in the old group; $u_1 = p/r_1$.
u_2: Number of processors in the new group; $u_2 = p/r_2$.
Here r_1 and r_2 are powers of two with $1 \le r_2 < r_1 < p$ and $r_1/r_2 \le N/p < p$.
$\mathbf{y} = (y_0, \ldots, y_{N-1})$: Complex vector of size N (block distributed).

OUTPUT $\mathbf{y} \leftarrow \Gamma_{u_2}\Gamma_{u_1}^{-1} \cdot \mathbf{y}$.

DESCRIPTION

```
1^Comm  Global permutation $\gamma_{u_1}^{u_2}$.
        $M_1 \leftarrow u_1 \cdot \frac{N}{p}$
        $M_2 \leftarrow u_2 \cdot \frac{N}{p}$
        $j_0 \leftarrow s\frac{N}{p} \operatorname{div} M_2$
        $j_1 \leftarrow (s\frac{N}{p} \bmod M_2) \operatorname{div} M_1$
        $j_2 \leftarrow (s\frac{N}{p} \bmod M_1) \operatorname{div} \frac{N}{p}$
        for $j = s\frac{N}{p}$ to $s\frac{N}{p} + \frac{N}{p} - 1$ do
            $j_3 \leftarrow j \bmod \frac{N}{p}$
            $dest \leftarrow j_0 \cdot u_2 + j_3 \bmod \frac{u_2}{u_1} \cdot u_1 + j_2$
            $y_{dest \cdot \frac{N}{p} + j_1 \cdot \frac{M_1}{u_2} + j_3 \operatorname{div} \frac{u_2}{u_1}} \leftarrow \mathrm{Put}(dest, 1, y_j)$
        Synchronize
```

2.3.4. Permutation from block to cyclic distribution. The permutations $\sigma_{p,N}$ and $\sigma_{p,N}^{-1}$ are the permutations that convert a vector from block to cyclic distribution and vice versa. In the case that $p < b = N/p$, both $\sigma_{p,N}$ and $\sigma_{p,N}^{-1}$ can be optimized by sending packets of size b/p (here we assume that p divides b).

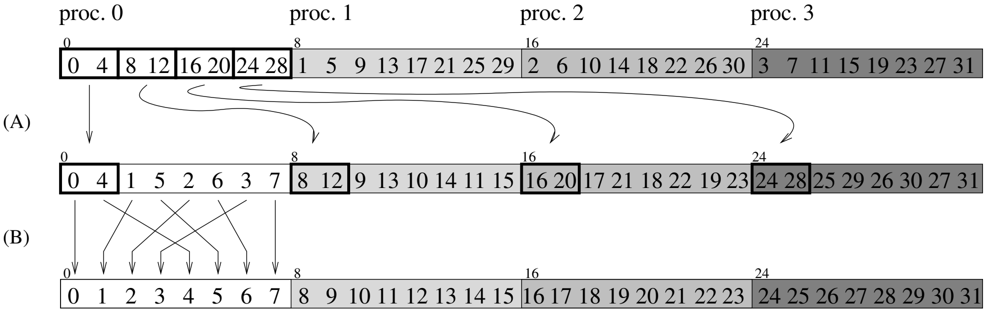


FIGURE 2.3. Schematic representation of a two stage permutation from cyclic to block distribution (storage view). Example with $N = 32$ and $p = 4$. (A) Global cyclic permutation of packets of size 2. (B) Local permutation from virtual $\mathrm{C}(4, 8)$ distribution to virtual $\mathrm{B}(4, 8)$ distribution.

For $\sigma^{-1}_{p,N}$, this is done as follows. First perform a global *cyclic permutation of packets* on the global index j.

$$j \to t = \underbrace{j_1}_{\mathrm{Proc}(t)} \cdot b + \underbrace{j_0 \cdot \frac{b}{p} + j_2}_{t'}, \tag{2.35}$$

where $j_0 = j \operatorname{div} b, j_1 = (j \bmod b) \operatorname{div} \frac{b}{p}$, and $j_2 = j \bmod \frac{b}{p}$. Then perform a local permutation $\sigma^{-1}_{p,b}$ on the local index t'

$$t' \to k' = t' \bmod \frac{b}{p} \cdot p + t' \operatorname{div} \frac{b}{p}.$$

Assuming that each local subvector of size b is distributed over p virtual processors, this last permutation can be seen as permuting the subvector from a virtual $\mathrm{C}(p, b)$ distribution to a virtual $\mathrm{B}(p, b)$ distribution, see Figure 2.3. The resulting global index k is then

$$\begin{aligned} k &= j_1 \cdot b + k' = j_1 \cdot b + t' \bmod \frac{b}{p} \cdot p + t' \operatorname{div} \frac{b}{p} \\ &= j_1 \cdot b + j_2 \cdot p + j_0 = (j_1 \cdot \frac{b}{p} + j_2) \cdot p + j_0 \\ &= [(j \bmod b) \operatorname{div} \frac{b}{p} \cdot \frac{b}{p} + (j \bmod b) \bmod \frac{b}{p}] \cdot p + j \operatorname{div} b = j \bmod b \cdot p + j \operatorname{div} b, \end{aligned}$$

which is indeed $\sigma^{-1}_{p,N}(j)$. For $\sigma_{p,N}$, a similar result is achieved by first performing a local permutation $\sigma_{p,b}$, and then a global cyclic permutation of packets.

Algorithms 2.6 and 2.7 are templates describing the implementation of permutations $\sigma_{p,N}^{-1}$ and $\sigma_{p,N}$, respectively. Both templates perform the permutations on a vector $\mathbf{y}$ of size $N = pb$ which is block distributed over p processors. The parameter $s_1 = s$ is the processor identification number. The extra parameter $s_0 = 0$ is the processor offset; it is introduced for later use.

Algorithm 2.7 can also be used to carry out the permutation $\gamma_{p,rp,rN}$. This is possible because permuting a vector of size rN by $\gamma_{p,rp,rN}$ is the same as dividing it into r subgroups of vectors of size N, then performing a shuffle permutation $\sigma_{u,N}$, on each of the subvectors, cf. (2.20). In this case $s_1 = s \bmod p$, $s_0 = s - s_1$, and $\mathbf{y}$ is a subvector starting at element $s_0 b$ of the original vector. For the same reason, Algorithm 2.6 can also be used to carry out permutation $\gamma_{p,rp,rN}^{-1}$.

Algorithm 2.6 Template for the parallel permutation from cyclic to block distribution.

CALL BSP_CyclicToBlock($s_0, s_1, p, b, \mathbf{y}$).

ARGUMENTS

s_0, s_1: Processor offset and processor identification within group; $0 \le s_1 < p$.
p: Number of processors in group.
b: Block size; p divides b, if $b > p$.
$\mathbf{y} = (y_0, \ldots, y_{pb-1})$: Complex vector of size $p \cdot b$ (block distributed within group).

OUTPUT $\mathbf{y} \leftarrow S_{p,pb}^{-1}\mathbf{y}$.

DESCRIPTION

```
                if $p \ge b$ then
$1^{\text{Comm}}$          Global $\sigma_{p,pb}^{-1}$ permutation.
                    for $j = s_1 \cdot b$ to $(s_1 + 1) \cdot b - 1$ do
                        $k \leftarrow j \bmod b \cdot p + j \operatorname{div} b$
                        $y_k \leftarrow \mathrm{Put}(s_0 + k \operatorname{div} b, 1, y_j)$
                    Synchronize
                else
$2^{\text{Comm}}$          Global cyclic permutation of packets.
                    for $proc = 0$ to $p - 1$ do
                        $x_{proc \cdot b + s_1 \cdot \frac{b}{p}} \leftarrow \mathrm{Put}(s_0 + proc, \frac{b}{p}, y_{s_1 \cdot b + proc \cdot \frac{b}{p}})$
                    Synchronize
$3^{\text{Comp}}$          Local $\sigma_{p,b}^{-1}$ permutation.
                    for $j' = 0$ to $b - 1$ do
                        $k' \leftarrow j' \bmod \frac{b}{p} \cdot p + j' \operatorname{div} \frac{b}{p}$
                        $y_{s_1 \cdot b + k'} \leftarrow x_{s_1 \cdot b + j'}$
```

2.3.5. BSP cost. To compute the total cost of our parallel FFT algorithm (Algorithm 2.2) we need to sum the computation, communication, and synchronization costs. The computation costs were already computed in Section 2.3.1. To simplify

Algorithm 2.7 Template for the parallel permutation from block to cyclic distribution.

CALL BSP_BlockToCyclic($s_0, s_1, p, b, \mathbf{y}$).

ARGUMENTS

s_0, s_1: Processor offset and processor identification within group; $0 \le s_1 < p$.
p: Number of processors in group.
b: Block size; p divides b, if $b > p$.
$\mathbf{y} = (y_0, \ldots, y_{pb-1})$: Complex vector of size pb (block distributed within group).

OUTPUT $\mathbf{y} \leftarrow S_{p,pb}\mathbf{y}$.

DESCRIPTION

```
          if $p \ge b$ then
1^Comm        Global $\sigma_{p,pb}$ permutation.
              for $j = s_1 \cdot b$ to $(s_1 + 1) \cdot b - 1$ do
                  $dest \leftarrow j \bmod p$
                  $y_{dest\cdot b+j \,\mathrm{div}\, p} \leftarrow \mathrm{Put}(s_0 + dest, 1, y_j)$
              Synchronize
          else
2^Comp        Local $\sigma_{p,b}$ permutation.
              for $j' = 0$ to $b - 1$ do
                  $k' \leftarrow j' \bmod p \cdot \frac{b}{p} + j' \,\mathrm{div}\, p$
                  $x_{s_1\cdot b+k'} \leftarrow y_{s_1\cdot b+j'}$
3^Comm        Global cyclic permutation of packets.
              for $proc = 0$ to $p - 1$ do
                  $y_{proc\cdot b+s_1\cdot\frac{b}{p}} \leftarrow \mathrm{Put}(s_0 + proc, \frac{b}{p}, x_{s_1\cdot b+proc\cdot\frac{b}{p}})$
              Synchronize
```

the final result we only include the higher order term of the total computation cost (2.31),

$$C_{\mathrm{FFT,par,Comp}}(N,p) = \frac{17}{4}\frac{N}{p}\log_2 N, \tag{2.36}$$

which is exact when only 4-butterflies are performed.

The communication and synchronization costs are the costs involved in performing the bit reversal and the permutations related to the group-cyclic distribution family. The maximum amount of data sent or received during a permutation involving complex numbers is equal to N/p complex values (or $2N/p$ real values). If the permutation is performed with puts, the number of synchronizations is 1, giving a total cost of

$$C_{\mathrm{permut}}(N,p) = 2\frac{N}{p} \cdot g + l \tag{2.37}$$

for each of the $\lceil \log_{\frac{N}{p}} N \rceil + 1$ permutations performed in the FFT algorithm. The total cost of the FFT algorithm is

$$C_{\mathrm{FFT,\ par}}(N,p) = \frac{17}{4}\frac{N}{p}\log_2 N + 2\frac{N}{p}(\lceil \log_{\frac{N}{p}} N \rceil + 1)\cdot g + (\lceil \log_{\frac{N}{p}} N \rceil + 1)\cdot l. \quad (2.38)$$

In Section 2.5, we discuss the validity of cost function (2.38) as a fair estimator of the true cost of the FFT algorithm.

The asymptotic isoefficiency function (see Section 1.3) of the parallel FFT can be computed as follows. Suppose that $N \geq lp/(2g)$ and that $p > 1$. Then the total communication cost of the algorithm is

$$\begin{aligned} C_{\mathrm{FFT,par,}\mathtt{Comm}}(N,p) &= 2\frac{N}{p}(\lceil \log_{\frac{N}{p}} N \rceil + 1)\cdot g + (\lceil \log_{\frac{N}{p}} N \rceil + 1)\cdot l \\ &\leq 4\frac{N}{p}(\lceil \log_{\frac{N}{p}} N \rceil + 1)\cdot g < 8\frac{N}{p}\lceil \log_{\frac{N}{p}} N \rceil \cdot g \\ &< 16\frac{N}{p}\frac{\log_2 N}{\log_2 N - \log_2 p}\cdot g. \end{aligned}$$

The total amount of work of the parallel FFT algorithm is $W = (17/4)N\log_2 N$. From (1.5) it is clear that the efficiency level of the algorithm can be maintained above a certain value a if

$$W \geq 16\cdot b\cdot p\frac{N}{p}\frac{\log_2 N}{\log_2 N - \log_2 p}\cdot g, \quad (2.39)$$

where $b = a/(1-a)$, which is equivalent to

$$N \geq 2^{\frac{64}{17}bg}p. \quad (2.40)$$

This linear dependence of N on p implies that W should grow at least as fast as $W = O(p\log p)$, which means that our FFT parallel algorithm is $O(p\log p)$ isoscalable. In other words, isoefficiency can be maintained if N/p is large enough.

2.4. *Variants of the algorithm*

2.4.1. Parallel FFT using other data distributions. Up to now, we discussed an FFT algorithm where the input and output (I/O) vector must be block distributed. There exist many applications of the FFT where the it would be better if the I/O vector would be distributed by other distributions (see e.g. Chapters 3 and 4). Here, we discuss how to modify our parallel FFT algorithm to accept I/O vectors that are not distributed by the block distribution.

The first and the last supersteps of Algorithm 2.2 are permutations. Because of this, the algorithm can be modified to accept any I/O data distribution without any extra communication cost, or even at a smaller communication cost depending on the desired distributions. If the input vector is not in the block distribution, this modification is done by combining the redistribution to block distribution with the bit reversal permutation. If the output vector is expected to be in a distribution other than the block distribution, this is done by substituting the permutation from cyclic to block distribution by a permutation from the cyclic to the desired distribution.

If the desired distribution for the output vector is the cyclic distribution, the last communication superstep can be completely skipped. The first permutation can also be skipped if the input vector is stored by the distribution associated with the bit reversal permutation. Applications where the input vector is bit reversed and the output vector is cyclically distributed are advantageous, because, in such cases, two complete permutations can be skipped.[3] This saves two thirds of the total communication cost in the common case that $p \leq N/p$.

While the cyclic distribution is simple and widely used, the distribution associated with the bit reversal permutation is awkward. Fortunately, it is possible to modify Algorithm 2.2 so that the natural input distribution, i.e., the distribution that does not involve any communication as the first superstep, is the cyclic distribution. The first three supersteps of Algorithm 2.2 are described by the following matrix decomposition.

$$\Gamma_u \cdot A_{\frac{N}{p},N} \dots A_{2,N} \cdot P_N, \tag{2.41}$$

where $u = \min(p, N/p)$. Knowing that $A_{K,N} = I_p \otimes A_{K,\frac{N}{p}}$, and that the bit reversal matrix can be decomposed as $P_N = (I_p \otimes P_{\frac{N}{p}}) \cdot (P_p \otimes I_{\frac{N}{p}}) \cdot S_{p,N}$ (cf. Corollary 2.8), we rewrite matrix (2.41) as

$$\begin{aligned} &\Gamma_u \cdot (I_p \otimes A_{\frac{N}{p},\frac{N}{p}}) \dots (I_p \otimes A_{2,\frac{N}{p}}) \cdot (I_p \otimes P_{\frac{N}{p}}) \cdot (P_p \otimes I_{\frac{N}{p}}) \cdot S_{p,N} \\ &\quad = \Gamma_u \cdot (I_p \otimes F_{\frac{N}{p}}) \cdot (P_p \otimes I_{\frac{N}{p}}) \cdot S_{p,N} = \Gamma_u \cdot (P_p \otimes I_{\frac{N}{p}}) \cdot (I_p \otimes F_{\frac{N}{p}}) \cdot S_{p,N}. \end{aligned} \tag{2.42}$$

Here we used Lemma 2.2. The first three supersteps of the parallel FFT algorithm derived from this new decomposition are: ($1^{\texttt{Comm}}$) permutation from block to cyclic distribution, ($2^{\texttt{Comp}}$) local FFT, ($3^{\texttt{Comm}}$) permutation defined by $\Gamma_{\min(p,N/p)} \cdot (P_p \otimes I_{\frac{N}{p}})$. In the case that the input vector is already cyclically distributed, the first superstep can be skipped.

2.4.2. Generalized butterfly phase with adjustable size. In our original algorithm, we chose to insert the permutation matrices Γ_u in the leftmost possible position. This procedure corresponds to factoring N as $N = \frac{N}{(N/p)^{H-1}}(N/p)^{H-1}$, and

---

[3] The idea of skipping permutations to save communication time or to reduce the overhead caused by local permutations is known. Cooley and Tukey [**16**] already suggested this in order to save local bit reversals. Other authors (e.g., [**26, 33, 56**], see also Chapters 3 and 4) give examples where skipping permutations saves communication time.

gives an algorithm with a minimum of permutations. However, if $p \neq (N/p)^{H-1}$, it is possible to insert the permutation matrices at an earlier position without increasing the number of permutations. The resulting algorithm would correspond to a different factorization of N.

We can use this flexibility to reduce the computation cost of some combinations of p and N by inserting the permutations so that a maximal number of generalized butterfly stages are paired off. Another reason to permute the vector at an earlier stage is that the sizes of the butterfly phases can be better balanced (i.e., choose all factors N_L having approximately the same size). This would enhance the performance on a cache-sensitive computer (see the discussion in Section 2.5). However, a more effective way of enhancing the performance on a cache-sensitive computer is to reduce the butterfly sizes so that they always fit completely into the cache. We suggest a method in the following subsection.

2.4.3. Cache-friendly parallel FFT. Each computation superstep of our parallel FFT algorithm performs a butterfly phase which consists of a sequence of generalized butterfly stages represented by the operation $\mathbf{y} \leftarrow R^{\alpha}_{l,n}\mathbf{y}$, where l and n are powers of two with $2 \leq l \leq n$, and

$$R^{\alpha}_{l,n} = A^{\alpha}_{n,n} \cdots A^{\alpha}_{2l,n} A^{\alpha}_{l,n}, \tag{2.43}$$

is an $n \times n$ matrix. Suppose that the cache memory of a computer is such that the data needed by a butterfly phase of size n/v, where $v < n$ is a power of two, fits totally in the computer cache. We can view v as the number of virtual processors available in each processor. If we decompose (2.43) into a sequence of smaller butterfly phases of size less or equal to n/v which can be carried out independently from each other, we can get the most out of the cache of the computer.

Define $h = \lceil \log_{\frac{n}{v}} n \rceil$ and $j = \lceil \log_{\frac{n}{v}} l \rceil - 1$, so that $l > (\frac{n}{v})^j$. Similarly to (2.25), if we denote $\Gamma_{u,v,n}$ by Γ_u, we can write

$$\begin{aligned} R^{\alpha}_{l,n} = {} & \Gamma_v^{-1} \underbrace{\hat{A}^{\alpha}_{\frac{n}{v},v} \cdots \hat{A}^{\alpha}_{2\frac{(n/v)^{h-1}}{v},v}}_{\text{phase } h-j-1} \Gamma_v \cdot \Gamma^{-1}_{(\frac{n}{v})^{h-2}} \underbrace{\hat{A}^{\alpha}_{\frac{n}{v},(\frac{n}{v})^{h-2}} \cdots \hat{A}^{\alpha}_{2,(\frac{n}{v})^{h-2}}}_{\text{phase } h-j-2} \Gamma_{(\frac{n}{v})^{h-2}} \cdot \ldots \\ & \ldots \cdot \Gamma^{-1}_{(\frac{n}{v})^{j+1}} \underbrace{\hat{A}^{\alpha}_{\frac{n}{v},(\frac{n}{v})^{j+1}} \cdots \hat{A}^{\alpha}_{2,(\frac{n}{v})^{j+1}}}_{\text{phase } 1} \Gamma_{(\frac{n}{v})^{j+1}} \cdot \Gamma^{-1}_{(\frac{n}{v})^{j}} \underbrace{\hat{A}^{\alpha}_{\frac{n}{v},(\frac{n}{v})^{j}} \cdots \hat{A}^{\alpha}_{\frac{l}{(n/v)^j},(\frac{n}{v})^{j}}}_{\text{phase } 0} \Gamma_{(\frac{n}{v})^{j}}, \end{aligned} \tag{2.44}$$

where the $n \times n$ matrix $\hat{A}^{\alpha}_{\frac{k}{u},u}$ is an abbreviation for $\hat{A}^{\alpha}_{\frac{k}{u},u,v,n} = \Gamma_{u,v,n} A^{\alpha}_{k,n} \Gamma^{-1}_{u,v,n}$. Generalized versions of Theorem 2.5 and Corollary 2.6 can be used to proof that

$$\hat{A}^{\alpha}_{\frac{k}{u},u,v,n} = I_{\frac{v}{u}} \otimes \operatorname{diag}(A^{\alpha/u}_{\frac{k}{u},\frac{n}{v}}, A^{(\alpha+1)/u}_{\frac{k}{u},\frac{n}{v}}, \ldots, A^{(\alpha+u-1)/u}_{\frac{k}{u},\frac{n}{v}}). \tag{2.45}$$

The matrix decomposition (2.44) can be used to construct an alternative algorithm for the computation of the generalized butterfly phases. One way of implementing this alternative sequential algorithm is by replacing the permutations γ by operations that gather the elements corresponding to a generalized butterfly phase in an auxiliary vector of size n/v, perform the generalized butterfly phase, and then store the elements back in their original place. Note that if $\alpha = 0$ then the resulting algorithm can be used to construct a cache-friendly FFT algorithm.

2.5. *Experimental results and discussion*

In this section, we present results on the performance of our implementation of the FFT. We implemented the FFT algorithm in ANSI C using the BSPlib communications library [**32**]. Our programs are completely self-contained, and we did not rely on any system-provided numerical software such as BLAS, FFTs, etc.

We tested our implementation on a Cray T3E with up to 64 processors, each having a theoretical peak speed of 600 Mflop/s. The accuracy of double precision (64-bit) arithmetic is 1.0×10^{-15}. We also give accuracy results from calculations on a SUN workstation using IEEE 754 floating point arithmetic, which has a double precision accuracy of 2.2×10^{-16}, and which is the standard used in many computers. To make a consistent comparison of the results, we compiled all test programs using the `bspfront` driver with options `-O3 -flibrary-level 2 -fcombine-puts` and measured the elapsed execution times on exclusively dedicated CPUs using the system clock. The times given correspond to an average of the execution times of a forward FFT and (normalized) backward FFT.

2.5.1. Accuracy. We tested the overall accuracy of our implementation by measuring the error obtained when transforming a random complex vector $\mathbf{f}$ with values $\mathrm{Re}(f_j)$ and $\mathrm{Im}(f_j)$ uniformly distributed between 0 and 1. The relative error is defined as

$$\frac{||\mathbf{F}^* - \mathbf{F}||_2}{||\mathbf{F}||_2}, \tag{2.46}$$

where $\mathbf{F}^*$ is the vector obtained by transforming the original vector $\mathbf{f}$ by a forward (or backward) FFT, and $\mathbf{F}$ is the exact transform, which we computed using the same algorithm but using quadruple precision. $||\cdot||_2$ indicates the L^2-norm.

Table 2.1 shows the relative errors of the sequential algorithm for various problem sizes. Since the error for the forward and backward FFT are approximately the same, we present only the results for the forward transform. The errors of the parallel implementation are of the same order as in the sequential case. In fact, the error of the parallel implementation only differs from the error of the sequential one if the butterfly stages are not paired in the same way.

TABLE 2.1. Relative errors for the sequential FFT algorithm.

N	CRAY T3E	IEEE 754
512	2.4×10^{-16}	1.9×10^{-16}
1024	5.2×10^{-16}	1.6×10^{-16}
2048	8.4×10^{-16}	1.8×10^{-16}
4096	2.1×10^{-15}	1.9×10^{-16}
8192	3.2×10^{-15}	2.0×10^{-16}
16384	6.5×10^{-15}	2.2×10^{-16}
32768	2.3×10^{-14}	2.3×10^{-16}
65536	3.4×10^{-14}	2.3×10^{-16}

2.5.2. Performance of the sequential implementation: the need for cache-friendly algorithms. Our sequential FFT algorithm was implemented using Algorithm 2.3 with $\alpha = 0$. Its performance can be analyzed by looking at its execution times or its *(FFT) flop rates*:

$$\mathrm{FFT}^{\mathrm{rate}}(N) = \frac{5N \log_2 N}{\mathrm{Time}(N)}, \tag{2.47}$$

where Time(N) is the execution time. Analyzing the performance of an FFT algorithm by using the number of flops of the radix-2 FFT as basis is a standard and useful procedure. By doing so, it is possible to compare different algorithms with different cost functions and also evaluate the overall performance of the algorithm as a function of N.

Table 2.2 gives timing results and FFT flop rates for various problem sizes. The flop rates show that the performance of the algorithm increases until $N = 4096$, when it suddenly drops. This sudden decrease in performance happens because the data

Table 2.2. Timing results (in ms) and FFT flop rates (in Mflop/s) of the sequential FFT on the Cray T3E.

N	Time	$\mathrm{FFT}^{\mathrm{rate}}$
32	0.02	33.4
64	0.03	56.9
128	0.10	47.3
256	0.15	69.6
512	0.41	56.5
1024	0.66	77.5
2048	1.82	62.1
4096	2.94	83.5
8192	19.95	26.7
16384	58.91	19.5
32768	149.79	16.4
65536	318.28	16.5

space allocated by the program becomes too large to fit completely in the cache memory of the CRAY T3E,[4] which means that the computation becomes more expensive, because more accesses to the main memory are needed.

Degradation of performance is characteristic of any computer that uses a cache. Since this type of architecture is common, it is important to develop sequential algorithms that can take advantage of the cache of such processors. Such algorithms should divide the computation load into blocks small enough to fit in the cache memory of the computer. For more details see the discussion on Section 2.4.3.

2.5.3. Scalability of the parallel implementation. The timing results obtained by our parallel algorithm are summarized in Table 2.3. We also present the theoretical predictions using the cost function (2.38) and the values of the BSP parameters v, g, and l obtained by a benchmark program (see Table A.1 in Appendix A for their values).

Except for the out-of-cache computations (boldface entries in the table), the timings show that the BSP cost function predicts well the behavior of the parallel implementation. The discrepancy between experimental and theoretical results for out-of-cache computations was expected, since the computation speed, which we assumed to be constant, suddenly drops when the computations cannot be done completely

---

[4] The cache size of the CRAY T3E is 96 Kbytes, which means that a sequential FFT of size up to $N = 4096$ fits completely in the cache (64 Kbytes for the data vector + 8 Kbytes for the weights table).

TABLE 2.3. Predicted and obtained execution times (in ms) for the FFT on a Cray T3E. Boldface entries indicate out-of-cache computations.

p	512		1024		2048		4096		8192		16384		32768		65536	
	pred	exp	pred	exp	pred	exp	pred	exp	pred	exp	pred	exp	pred	exp	pred	exp
seq	0.56	0.41	1.25	0.66	2.74	1.82	5.99	2.94	12.97	**19.95**	27.93	**58.91**	59.9	**149.8**	127.7	**318.3**
1	0.58	0.40	1.28	0.66	2.81	1.81	6.12	2.95	13.23	**19.95**	28.46	**58.98**	60.9	**149.8**	129.8	**315.8**
2	0.37	0.42	0.77	0.90	1.61	1.66	3.44	3.90	7.33	8.93	15.61	**33.65**	33.2	**87.1**	70.3	**207.3**
4	0.25	0.28	0.45	0.47	0.89	0.99	1.83	1.81	3.83	4.40	8.09	9.72	17.1	**39.7**	36.1	**101.1**
8	0.21	0.21	0.32	0.33	0.56	0.69	1.06	1.22	2.12	2.33	4.36	5.28	9.1	12.5	19.1	**46.7**
16	0.20	0.22	0.25	0.26	0.37	0.39	0.63	0.56	1.16	1.20	2.30	2.26	4.7	5.4	9.8	12.7
32	0.26	0.29	0.23	0.33	0.29	0.37	0.42	0.53	0.70	0.75	1.29	1.43	2.5	2.9	5.1	7.1
64	0.46	0.31	0.48	0.38	0.51	0.55	0.46	0.63	0.61	0.76	0.91	0.98	1.5	1.7	2.9	3.2

in-cache. These results show that the BSP model is a valid tool for analyzing and predicting parallel performance.

Figure 2.4 shows the absolute speedups obtained for various input sizes on up to 64 processors. The figure shows moderate speedups for small problem sizes ($N \leq 4096$), if $p \leq 8$. For $N \geq 8192$, speedups of up to 1.5 times the ideal speedup are achieved. Such amazing speedups are possible because of the so called *cache effect*: when $N \geq 8192$ the total amount of memory needed by the FFT is too large to fit in the cache memory of one processor, but, if the problem is executed using a sufficiently large number of processors, the memory required by each processor becomes small enough to fit in the cache. This effect is welcome, but it masks the real scalability of the algorithm.

Another way of analyzing the scalability of our parallel implementation is to look at the FFT flop rate per processor, as done in Figure 2.5. Note that there is a sudden rise in the flop rate when the local problem size becomes small enough to fit in the

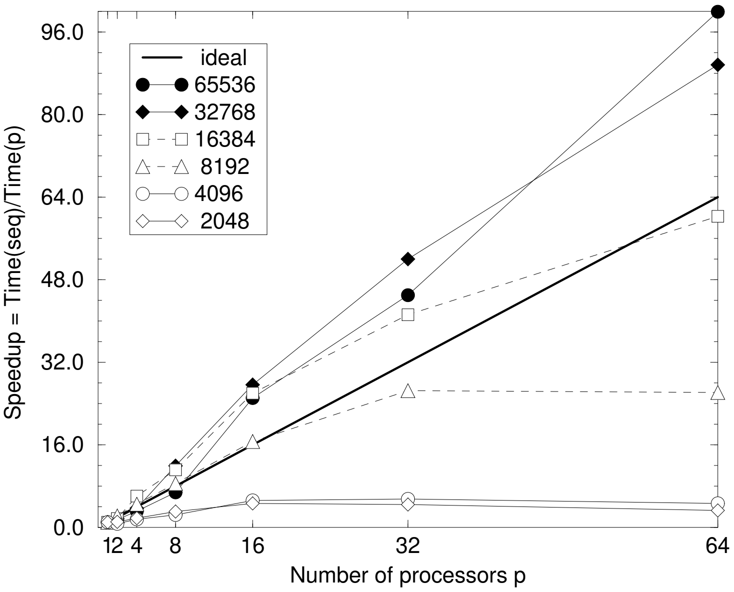


FIGURE 2.4. Scalability of the FFT on a Cray T3E measured as speedup.

cache. In this way the cache effect can be easily spotted and the scalability of the algorithm better judged. FFT sizes that fit completely in the cache ($N \leq 4096$) have a completely different behavior than larger problems. For small sizes ($N \leq 4096$) the flop rate decreases suddenly in going from one to two processors, then it is somewhat constant up to 8–16 processors and after that it decreases steadily. For large sizes ($N > 8192$) the flop rate is nearly constant, both before and after the transition out-of-cache/in-cache, indicating a good scalability. The case $N = 8192$ is an intermediate case where there is an increase in the flop rate in going from one to two processors, but a deterioration of performance when the number of processors becomes too large.

We can also examine the scalability of our parallel algorithm using its cost function. Here we have two conflicting goals: on the one hand, we want to use the cost function of our algorithm to analyze its theoretical complexity (making sure no machine specific characteristics, such as the cache effect, masks the results); on the other hand, we want our cost model to be as close to reality as possible. In other words, we want to use a theoretical cost model that can, for example, compare two algorithms and decide which is better independently of the specific implementation or the specific

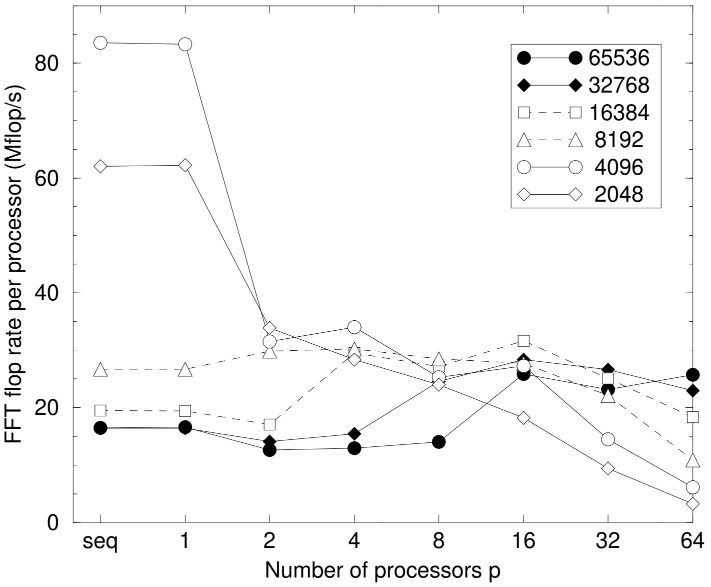


FIGURE 2.5. Scalability of the FFT on a Cray T3E measured as FFT flop rate per processor.

machine, but we also want to be able to predict the performance of our algorithm on a specific machine. A good cost model should capture the essence of these two distinct goals.

As already pointed out, the BSP cost model proves to be reliable when there is no cache effect involved. With this assurance, we can analyze the speedups achieved by our algorithm without worrying about the cache effect.[5] The theoretical speedups are shown in Figure 2.6. Comparing them with the speedups of Figure 2.4, we obtain the same kind of results for $N < 4096$. For larger N, the speedups are smaller, as expected, but they remain at high levels. We expect that a cache-friendly implementation, as proposed in Section 2.4.3, would yield similar speedup results. Note that the curves are smooth, because the theoretical cost function assumes the computation cost to be $4.25\frac{N}{p}\log_2 N$ regardless of whether exclusively 4-butterflies are executed. This is in contrast with the experiments, where 2-butterflies occur occasionally for

---

[5] If we want to predict the cache effect, we need to refine our cost model, by allowing different computation speeds according to the local problem size.

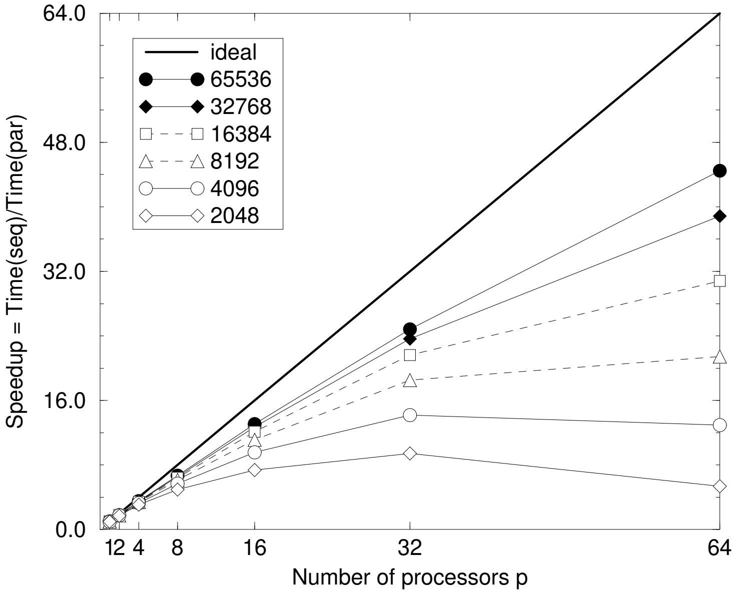


FIGURE 2.6. Predicted scalability of the FFT on a Cray T3E measured as speedup.

certain values of N and p, increasing the computation time, and causing oscillations in the curves.

In Section 2.3.5 we carried out an asymptotic analysis of the isoefficiency function of our FFT algorithm. We concluded that, for N/p large enough, isoefficiency is achieved. Figure 2.7 shows the predicted and obtained efficiencies as a function of p for various values of N/p. As expected, the predicted values converge to a horizontal line as N/p increases. The experimental results must be analyzed keeping in mind the cache effect, which causes the sudden increase in the efficiency. It is clear that efficiency can be maintained at reasonable levels for N/p as small as 256, and at very good levels for $N/p = 4096$ which indicates that isoefficiency is already achieved for N/p considerably smaller than the lower bound $N/p \geq \max(l/(2g), 2^{\frac{64}{17}bg})$ guaranteeing isoefficiency that was computed in Section 2.3.5.

2.6. *Alternative algorithms*

Up to now, we studied the problem of parallelizing an FFT based on the radix-2 framework. This restriction on N may be undesired in some practical cases, mainly

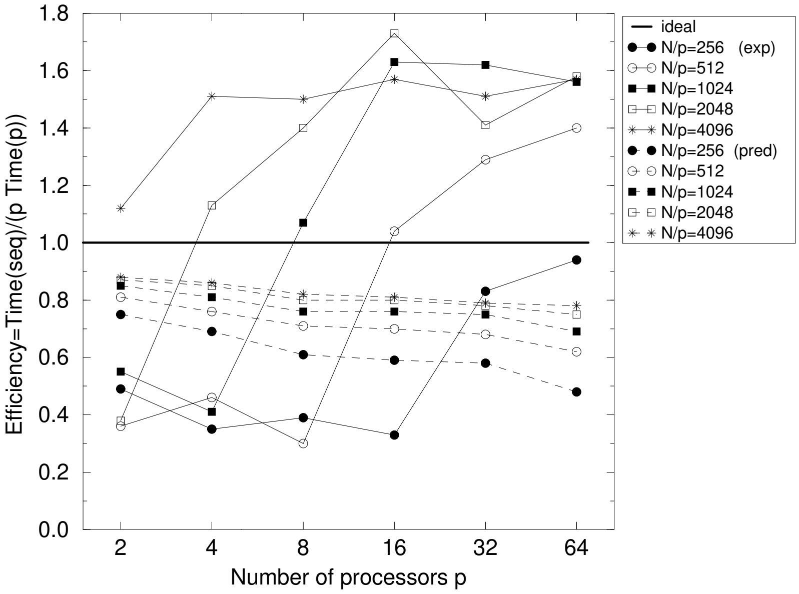


FIGURE 2.7. Efficiency as a function of p for a constant problem size N/p on a Cray T3E. Dashed lines: predicted values. Solid lines: obtained values.

when computing multidimensional transforms. This happens because the small number of choices of N can lead to prohibitively large problems. Suppose, for example, that we need to compute a two-dimensional FFT and that both dimensions need a minimum of 10^4 points. Since the first power of two larger than 10^4 is 2^{14}, instead of a total of 10^8 points we will have to use 2^{28} points, which means approximately 2.6 times more work and storage space. Since DFTs of arbitrary size cannot always be carried out using fast algorithms, a compromise solution is to allow sizes of the form such as $N = 2^{m_1} \cdot 3^{m_2} \cdot 5^{m_3}$ [**46, 52, 53**].

We dedicate this section to deriving a parallel mixed radix FFT algorithm that works for any problem size $N = N_0 N_1 \dots N_{H-1}$ and number of processors p as long as p divides each N_l, $0 \leq l < H$, and sequential FFTs of size N_l are available. Before deriving our algorithm, which is based on the work of Agarwal and Cooley [**1**], we describe the 6-pass approach (see e.g. [**4, 26, 31**]) and the transpose approach [**25**], [**34**, Chap. 10.3] for computing parallel FFTs.

2.6.1. Six-pass algorithms and transpose algorithms. A 6-pass (or 6-step) FFT algorithm can be obtained by rewriting (2.18) as

$$Z_{k_1,k_0} = \sum_{j_1=0}^{N_1-1} \left(\sum_{j_0=0}^{N_0-1} z_{j_0,j_1} w_{N_0}^{j_0 k_0} \right) w_N^{j_1 k_0} w_{N_1}^{j_1 k_1}. \tag{2.48}$$

As the double indices suggest, the 6-pass framework is generally presented by viewing the vectors **z** and **Z** as matrices. Here, we propose a different approach. It can be shown that (2.48) corresponds to the following decomposition of the Fourier matrix:

$$F_N = S_{N_1,N}(I_{N_0} \otimes F_{N_1}) C_{N_1,N} S_{N_0,N} (I_{N_1} \otimes F_{N_0}) S_{N_1,N}. \tag{2.49}$$

Here, $C_{N_1,N}$ is the $N \times N$ diagonal matrix $\mathrm{diag}(c_0, \ldots, c_{N-1})$, with

$$c_j = w_N^{(j \bmod N_1)(j \operatorname{div} N_1)} = w_{N_1}^{(j \bmod N_1)(j \operatorname{div} N_1)/N_0}. \tag{2.50}$$

Suppose that p is a divisor of N_0 and of N_1 and that the input vector **z** is block distributed over p processors. An in-place 6-pass FFT algorithm that delivers the output in natural order, also in the block distribution, is the following:

$1^{\mathtt{Comm}}$ Permutation $\sigma_{N_1,N}$: $\mathbf{z} \leftarrow S_{N_1,N} \cdot \mathbf{z}$
$2^{\mathtt{Comp}}$ N_1 FFTs of size N_0: $\mathbf{z} \leftarrow (I_{N_1} \otimes F_{N_0}) \cdot \mathbf{z}$
$3^{\mathtt{Comm}}$ Permutation $\sigma_{N_0,N}$: $\mathbf{z} \leftarrow S_{N_0,N} \cdot \mathbf{z}$
$4^{\mathtt{Comp}}$ Multiplication by twiddle factors: $\mathbf{z} \leftarrow C_{N_1,N} \cdot \mathbf{z}$
$5^{\mathtt{Comp}}$ N_0 FFTs of size N_1: $\mathbf{z} \leftarrow (I_{N_0} \otimes F_{N_1}) \cdot \mathbf{z}$
$6^{\mathtt{Comm}}$ Permutation $\sigma_{N_1,N}$: $\mathbf{z} \leftarrow S_{N_1,N} \cdot \mathbf{z}$

In superstep 1, permutation $\sigma_{N_1,N}$ brings together the elements z_{j_0,j_1}, $0 \le j_0 < N_0$, which are needed to perform the j_1-th FFT of superstep 2. (If the vector **z** is viewed as an $N_0 \times N_1$ matrix, this permutation can be viewed as a matrix transposition.) In superstep 3, vector **z** is brought back to its original order, so that the FFTs of superstep 5 can be computed locally. But first, in superstep 4, vector **z** is multiplied by a set of twiddle factors. The permutation of superstep 6 is needed to bring the vector back to block distribution in its natural ordering. A 6-pass algorithm has a BSP cost of $O(\frac{N}{p} \log N) + 3\frac{N}{p} g + 3l$, which is of the same order as the cost of Algorithm 2.2 for the case $H = 2$. Note that, if we do not require that the input/output vector be block distributed, permutation supersteps 1 and 6 can be dropped. In those cases, the 6-pass approach is known as the 4-step approach (see e.g. [**31**]).

We use the name *transpose FFT algorithm* to designate parallel FFT algorithms, similar to the one above, where supersteps 4 and 5 are replaced by a single calculation which has the same cost as the FFT (superstep 5). Gupta and Kumar [**25**] developed a transpose algorithm for an unordered radix-2 FFT[6] which works in the specific case that $N = M^2$ and p divides M. In their book [**34**, Chap. 10.3], Kumar and collaborators proposed a generalized transpose (radix-2) FFT algorithm for the case that $N = M^H$ and p divides M.

Because the multiplication by twiddle factors costs $O(N)$ flops, the transpose FFT approach is less expensive than the 6-pass approach. Nevertheless, a parallel FFT implementation that uses the 6-pass approach can easily be optimized by replacing the sequential FFT subroutine used by a faster one.

A 6-pass FFT algorithm can be transformed into a transpose FFT algorithm by replacing the multiplication by twiddle factors and the sequence of FFTs, given as supersteps 4 and 5 in the description above, by a sequence of *generalized fast Fourier transforms* (GFFTs), which are fast algorithms for computing the *generalized discrete Fourier transform* (GDFT) [**5, 11, 18**]:

$$\mathbf{Z} = F_N^{\alpha}\mathbf{z}, \tag{2.51}$$

where F_N^{α} is the *generalized Fourier matrix*, defined by $(F_N^{\alpha})_{jk} = w_N^{(j+\alpha)k}$. A GFFT algorithm is derived in a similar way as the FFT algorithm. For example, if N is a power of two, F_N^{α} can be decomposed as

$$F_N^{\alpha} = A_{N,N}^{\alpha} \cdots A_{8,N}^{\alpha} A_{4,N}^{\alpha} A_{2,N}^{\alpha} P_N, \tag{2.52}$$

and, therefore, Algorithms 2.3 and 2.4 can be used to compute the GDFT of a vector of size N in $4.25N\log_2 N$ flops.

To see the connection between the 6-pass approach and the GFFT approach, define

$$E_N^{\alpha} = \mathrm{diag}(w_N^0, w_N^{\alpha}, \ldots, w_N^{(N-1)\alpha}), \tag{2.53}$$

so that $C_{N_1,N} = \mathrm{diag}(E_{N_1}^0, E_{N_1}^{1/N_0}, \ldots, E_{N_1}^{(N_0-1)/N_0})$, where $N_0 = N/N_1$. Since

$$F_N^{\alpha} = F_N E_N^{\alpha}, \tag{2.54}$$

this gives

$$\begin{aligned}(I_{N_0} \otimes F_{N_1})C_{N_1,N} &= \mathrm{diag}(F_{N_1}E_{N_1}^0, F_{N_1}E_{N_1}^{1/N_0}, \ldots, F_{N_1}E_{N_1}^{(N_0-1)/N_0}) \\ &= \mathrm{diag}(F_{N_1}^0, F_{N_1}^{1/N_0}, \ldots, F_{N_1}^{(N_0-1)/N_0}).\end{aligned} \tag{2.55}$$

For the case that $N = N_0 N_1 \ldots N_{H-1}$, successive substitution of (2.48) (or (2.49)) into itself leads to generalized 6-pass algorithms. Though it is easy to guess what to do to derive a generalized 6-pass algorithm, getting the details right can be laborious.

---

[6] An unordered radix-2 FFT is an FFT that takes either the input or the output vector in bit reversed order.

Agarwal and Cooley [**1**] introduced a generalized 6-pass algorithm to be used on a vector computer. Though their algorithm works for a general N, they only report implementing it for powers of two. Based on [**1**], Averbuch and collaborators [**5**] derived the corresponding transpose FFT algorithm. Here again, the algorithm works for a general N, but the implementation only works for $N = M^H$ that are powers of two. Their parallel implementation, which was designed for a shared memory machine, can handle any number of processors by assigning a group of GFFTs to each available processor. If N/p is not a multiple of M, then load imbalance is created because some processors receive more GFFTs than others. Cormen and Nicol [**17**] also implemented a radix-2 version of this algorithm to be used in the computation of out-of-core FFTs.

2.6.2. Parallel mixed-radix FFT: a generalized 6-pass algorithm. The algorithm by Agarwal and Cooley [**1**] can be seen as a generalized 6-pass algorithm. Their algorithm is a mixed-radix FFT which was originally developed with a vector processor architecture in mind. In this subsection, we propose a parallel distributed memory version of their algorithm.

Agarwal and Cooley's idea was to develop a method that could make use of long vectors throughout the computation, so that the pipelines of the vector processor could be used efficiently. To achieve this, they used the following generalization of (2.48).

Let $N = N_0 N_1 \dots N_{H-1}$. Define $N_{-1} = N_H = 1$, $L_l = N_{-1} N_0 \dots N_l$, and $M_l = N_l \dots N_{H-1} N_H$. Let $(k_{H-1}, \dots, k_1, k_0)$, where $k_l = (k \bmod L_l) \operatorname{div} L_{l-1}$, be the mixed-radix representation of an index $k = k_{H-1} \cdot L_{H-2} + \cdots + k_1 \cdot L_0 + k_0$ with the ordered sequence of radices $N_0, N_1, \dots, N_{H-1}$, and let $(j_0, j_1, \dots j_{H-1})$, where $j_l = (j \bmod M_l) \operatorname{div} M_{l+1}$, be the mixed-radix representation of an index $j = j_0 \cdot M_1 + j_1 \cdot M_2 + \cdots + j_{H-1}$ with the ordered sequence of radices $N_{H-1}, \dots, N_1, N_0$. The discrete Fourier transform of a vector of size $N = N_0 N_1 \dots N_{H-1}$ can be written as

$$Z_{(k_{H-1},\dots,k_0)} = \\ \sum_{j_{H-1}=0}^{N_{H-1}-1} \{ \dots [\sum_{j_1=0}^{N_1-1} (\sum_{j_0=0}^{N_0-1} z_{(j_0,\dots,j_{H-1})} w_{N_0}^{j_0 k_0}) w_{L_1}^{j_1 k_0^*} w_{N_1}^{j_1 k_1}] \dots \} w_{L_{H-1}}^{j_{H-1} k^*_{H-2}} w_{N_{H-1}}^{j_{H-1} k_{H-1}}, \tag{2.56}$$

for $0 \le k_l < N_l, 0 \le l < H$, where $k_l^* = k_l \cdot L_{l-1} + \cdots + k_1 \cdot L_0 + k_0 = k \bmod L_l$.

In (2.56) each sum over j_l, for $0 \le l < H$, corresponds to a multiplication by twiddle factors ($w_{L_l}^{j_l k^*_{l-1}} = w_{N_l}^{j_l k^*_{l-1}/L_{l-1}}$) followed by an FFT of size N_l. Since the same operation is repeated N/N_l times (though with different twiddle factors), each summing phase can be carried out using long vectors.

In the parallel case, instead of long vectors, we want the computation to have independent parts which can be computed in parallel. We choose the number of processors p so that all the N_l's divide N/p, which is equivalent to the requirement that p divides all the N/N_l. In each phase we have N/N_l FFT computations which are independent from each other, hence they can be distributed over the processors. The requirement that all the N_l's divide N/p ensures that each processor gets the same number of FFTs, achieving perfect load balance. For example, if $N = 2^{m_0}3^{m_1}5^{m_2}$, then p must be of the form $2^{q_0}3^{q_1}5^{q_2}$, with $q_j < m_j$, to satisfy the requirement.

Now we have to tackle the problem of how to redistribute the working vector before each computation phase, so that the data needed are locally available. Suppose that the input vector is in the block distribution. The first phase involves computing FFTs over the index j_0. This means that all elements for which the global index j differs only in the j_0-th N_0-*ary digit* (or, simply, *digit*), must be in the same processor. Since the vector is block distributed, the processor to which element j belongs is

$$\mathrm{Proc}(j) = j \,\mathrm{div}\, \frac{N}{p} = (j_0 \cdot M_1 + j_1 \cdot M_2 + \cdots + j_{H-2} \cdot M_{H-1}) \,\mathrm{div}\, \frac{N}{p}.$$

Unless $p = 1$, j_0 helps determining the processor number, and, therefore, different j_0 can imply different processors.

To guarantee that all elements for which the global index j differs only in the j_0-th digit are in the same processor we have to submit the vector to a permutation $j \to t = j_{\pi(0)} \cdot N_{\pi(1)}N_{\pi(2)} \ldots N_{\pi(H-1)} + j_{\pi(1)} \cdot N_{\pi(2)}N_{\pi(3)} \ldots N_{\pi(H-1)} + \cdots + j_{\pi(H-1)}$, where $\pi : \{0, 1, \ldots, H-1\} \to \{0, 1, \ldots, H-1\}$ is a permutation over the radix digits, with $\pi(H-1) = 0$. Note that t is well defined, because $0 \le j_{\pi(l)} < N_{\pi(l)}$. To see that all the needed elements are in the same processor note that

$$\mathrm{Proc}(t) = t \,\mathrm{div}\, \frac{N}{p} = (t \,\mathrm{div}\, N_0 \cdot N_0 + j_0) \,\mathrm{div}\, \frac{N}{p} = (t \,\mathrm{div}\, N_0 \cdot N_0) \,\mathrm{div}\, \frac{N}{p},$$

i.e., j_0 does not influence the processor number. (Here we have used the fact that N_0 divides N/p.) Note that if $(t_0, t_1, \ldots, t_{H-1})$ is the mixed-radix representation of index $t = t_0 \cdot N_{\pi(1)}N_{\pi(2)} \ldots N_{\pi(H-1)} + t_1 \cdot N_{\pi(2)}N_{\pi(3)} \ldots N_{\pi(H-1)} + \cdots + t_{H-1}$ with the ordered sequence of radices $N_{\pi(H-1)}, \ldots, N_{\pi(1)}, N_{\pi(0)}$, then

$$t_l = [t \bmod (N_{\pi(l)}N_{\pi(l+1)} \ldots N_{\pi(H-1)})] \,\mathrm{div} (N_{\pi(l+1)}N_{\pi(l+2)} \ldots N_{\pi(H)}) = j_{\pi(l)}.$$

Here, we defined $\pi(H) = H$ so that $N_{\pi(H)} = 1$.

The same reasoning applies to the other phases of the algorithm: at each phase l, the vector must be permuted so that the original index j_l is the least significant. This leaves much freedom in the choice of permutations to use. Each possible choice of permutations leads to a slightly different algorithm, in which the ordering of the twiddle factors is changed.

Our template, which is given as Algorithm 2.8 uses the sequence of permutations suggested by [**1**]. That is,

- Start with an *index reversal permutation*

  $$\begin{aligned} \mathrm{irev}_{N_0,\ldots,N_{H-1}} &: \{0,\ldots,N-1\} \to \{0,\ldots,N-1\} \\ j = j_0 \cdot M_1 + j_1 \cdot M_2 + \cdots + j_{H-1} &\mapsto j_{H-1} \cdot L_{H-2} + \cdots + j_1 \cdot L_0 + j_0, \end{aligned} \tag{2.57}$$

  where $j_l = (j \bmod M_l) \operatorname{div} M_{l+1}$. (The index reversal permutation is a generalization of the bit reversal permutation.)
- After each computation phase l, permute the vector by $\sigma_{N_l,N}$.

This sequence of permutations takes a vector which is block distributed and transforms it so that the result is also block distributed (in natural order). It is possible to show that this sequence of permutations corresponds to the following decomposition of the Fourier matrix:

$$\begin{aligned} F_N = S_{N_{H-1},N}(I_{N/N_{H-1}} \otimes F_{N_{H-1}})W_{N_{H-1},N} \cdots \\ S_{N_1,N}(I_{N/N_1} \otimes F_{N_1})W_{N_1,N}S_{N_0,N}(I_{N/N_0} \otimes F_{N_0})P_{N_0,\ldots,N_{H-1},N}, \end{aligned} \tag{2.58}$$

where

$$W_{N_l,N} = \mathrm{diag}(I_{M_{l+1}} \otimes E^0_{N_l}, I_{M_{l+1}} \otimes E^{1/L_{l-1}}_{N_l}, \ldots, I_{M_{l+1}} \otimes E^{(L_{l-1}-1)/L_{l-1}}_{N_l}), \tag{2.59}$$

and $P_{N_0,\ldots,N_{H-1},N}$ is the $N \times N$ permutation matrix:

$$(P_{N_0,\ldots,N_{H-1},N})_{lj} = \begin{cases} 1, & \text{if } l = \mathrm{irev}_{N_0,\ldots,N_{H-1}}(j), \\ 0, & \text{otherwise,} \end{cases} \tag{2.60}$$

which corresponds to the index reversal.

In Algorithm 2.8, the vector $\mathbf{z}$ of size N, which must be in the block distribution, is transformed in place. The factors of N should have been chosen previously. Note that when $l = 0$, the multiplication by twiddle factors can be skipped, since all the weights are equal to 1. The algorithm uses subroutine $\mathrm{FFT}(sign, M, \mathbf{y})$, which computes the forward FFT of a vector $\mathbf{y}$ of size M, if $sign = 1$, or the backward FFT, otherwise. This means that sequential FFT algorithms for vector sizes $N_0, N_1, \ldots, N_{H-1}$ must be available. As an alternative, the multiplication by twiddle factors and FFTs can be replaced by suitable GFFTs. Cache-friendly versions of Algorithm 2.8 can readily be obtained by further substitution of (2.58) in each of the F_{N_l}. If the I/O vector is needed in a different distribution than the block distribution, then the first and the last permutations can be replaced without increasing the cost of the algorithm.

As important as designing the algorithm itself is choosing the factorization of N to be used. To achieve the best results, with the fewest extra flops caused by

Algorithm 2.8 Template for the parallel mixed-radix FFT.

```
CALL   BSP_CFFT_Mix($s, p, sign, N, H, N_0, \ldots, N_{H-1}, \mathbf{y}$).
INPUT
         $s$: Processor identification number; $0 \leq s < p$.
         $p$: Number of processors; $p$ divides all $N/N_l$.
         $sign$: Transform direction; $+1$ for forward, $-1$ for backward.
         $N$: Transform size.
         $H, N_0, \ldots, N_{H-1}$: $H$-dimensional factorization of $N$; $1 < N_l \leq N/p$.
         $\mathbf{z} = (z_0^{in}, \ldots, z_{N-1}^{in})$: Complex vector of size $N$ (block distributed).
OUTPUT   $\mathbf{z} \leftarrow (z_0^{out}, \ldots, z_{N-1}^{out})$, where $z_k^{out} = \sum_{j=0}^{N-1} z_j^{in} \exp(sign \frac{2\pi ikj}{N})$.
DESCRIPTION
         $N_{-1} \leftarrow 1$
         $N_H \leftarrow 1$
$1^{\mathtt{Comm}}$    Perform an index reversal permutation $\mathrm{irev}_{N_0,\ldots,N_{H-1}}$ on $\mathbf{z}$.
         for $j = s\frac{N}{p}$ to $s\frac{N}{p} + \frac{N}{p} - 1$ do
             $k \leftarrow \mathrm{irev}_{N_0,\ldots,N_{H-1}}(j)$
             $z_k \leftarrow \mathrm{Put}(k \,\mathrm{div}\, \frac{N}{p}, 1, z_j)$
         Synchronize

         for $l = 0$ to $H - 1$ do
             $L_{l-1} \leftarrow N_{-1}N_0 \ldots N_{l-1}$
             $M_{l+1} \leftarrow N_{l+1} \ldots N_H$
$2^{\mathtt{Comp}}$        Multiply by twiddle factors and perform local FFTs of size $N_l$.
             for $m_1 = 0$ to $L_{l-1} - 1$
                 $\alpha \leftarrow m_1/L_{l-1}$
                 for $m_0 = 0$ to $M_{l+1} - 1$
                     $k \leftarrow (m_1 M_{l+1} + m_0)N_l$
                     if $k \,\mathrm{div}\, \frac{N}{p} = s$ then
                         for $j = 0$ to $N_l - 1$ do
                             $z_{k+j} \leftarrow z_{k+j} \cdot w_{N_l}^{sign(j\alpha)}$
                         FFT($sign, N_l, \mathbf{z_k}$)
$3^{\mathtt{Comm}}$        Perform a $\sigma_{N_l,N}$ permutation on $\mathbf{z}$.
             for $j = s\frac{N}{p}$ to $s\frac{N}{p} + \frac{N}{p} - 1$ do
                 $k \leftarrow \sigma_{N_l,N}(j)$
                 $z_k \leftarrow \mathrm{Put}(k \,\mathrm{div}\, \frac{N}{p}, 1, z_j)$
             Synchronize
```

twidle factor multiplications and the smallest number of communication supersteps, we want to factor N in as few factors as possible. In the case that N is a power of two this is easy: since all the N_l's and p must be powers of two, we choose $N = (N/p)^{H-1}\frac{N}{(N/p)^{H-1}}$, where $H = \lceil \log_{\frac{N}{p}} N \rceil$. In the general case, matters become more complicated. Suppose that $N = 2^{m_0}3^{m_1}5^{m_2}$, and $p = 2^{q_0}3^{q_1}5^{q_2}$, where $q_j < m_j$. Then $N/p = 2^{d_0}3^{d_1}5^{d_2}$, where $d_j = m_j - q_j$. Now, we use $c_{l,0}, c_{l,1}$, and $c_{l,2}$ to express

M_l by $M_l = 2^{c_{l,0}} 3^{c_{l,1}} 5^{c_{l,2}}$. The following method can be used to decompose N in suitable factors.

- Start by choosing $N_0 = N/p$, so that $M_1 = p$.
- Define $H = 1$.
- While $M_H > 1$
    - Choose $N_H = 2^{\min(d_0, c_{H,0})} 3^{\min(d_1, c_{H,1})} 5^{\min(d_2, c_{H,2})}$.
    - $H \leftarrow H + 1$.

The same method can also be used for N's that have larger primes in their prime factorization. If, however, the factorization of N has many different prime factors with small multiplicity, the choices of p are limited. Furthermore, the number of permutations may become large.

2.7. *Conclusions and future work*

In Section 2.2, we presented a new parallel FFT algorithm, Algorithm 2.2. This algorithm is a mixed radix-2 and radix-4 FFT. It was derived based on the matrix decomposition corresponding to the radix-2 algorithm by inserting suitable permutation matrices corresponding to the group-cyclic distribution family. The use of the group-cyclic distribution family gives a parallel algorithm which is simple to understand and easy to implement.

The use of matrix notation proved to be a powerful tool for deriving and adapting the parallel FFT algorithms according to one's needs. With the help of matrix notation, we showed how to modify our original algorithm to accept I/O vectors that are not in block distribution, without incurring extra communication cost. Indeed, if the vector is cyclically distributed, we showed how to eliminate the first and the last permutation altogether, reducing the communication cost to one third of the original cost. Since the cyclic distribution is simple and widely used, this property can be exploited to obtain faster applications. We also used matrices to derive a cache-friendly variant of our parallel algorithm, and to give a simple proof of the equivalence between algorithms that use twiddle factors and algorithms that use generalized DFTs.

We presented results concerning the performance of our implementation of Algorithm 2.2. The tests were carried out on a Cray T3E with up to 64 processors. Our implementation proved to scale reasonably well for small problem sizes ($N \leq 4096$) with up to 8 processors, and to scale very well for larger problem sizes ($N \geq 16384$). In part, the very favorable results obtained for larger N are due to the cache effect. Because of this effect, we analyzed our results in terms of FFT flop rate per processor, and also using the theoretical cost function. Both analyses confirmed the previous results.

Because the cache-based architecture of the Cray T3E influences our results so much, and there are many other computers with a similar architecture, we proposed the use of cache-friendly FFT algorithms. A cache-friendly sequential algorithm can be derived from our parallel algorithm by substituting the processors by virtual processors. It is also possible to derive a cache-friendly parallel FFT algorithm by writing each generalized butterfly phase as a sequence of smaller generalized butterfly phases. We expect the scalability of such an algorithm to be similar to the theoretical scalability of our algorithm. Though cache-friendly algorithms could also use the twiddle factor approach, the amount of extra computation required by the extra multiplication phases can increase considerably, depending on the ratio between cache and problem size.

We also analyzed the asymptotic behavior of our algorithm using its isoefficiency function. We concluded that our FFT parallel algorithm is $O(p \log p)$ isoscalable. This is to say that the efficiency level is maintained as long as N/p is large enough. A study of the experimental data obtained on the Cray T3E indicates that reasonable efficiency levels ($E(p,N) \approx 0.5$) are already maintained for N/p as small as 256, and good efficiency levels are maintained for $N/p > 4096$.

We also presented the parallel mixed-radix FFT algorithm, Algorithm 2.8. This algorithm is based on [**1**], and can be seen as a generalization of the 6-pass FFT approach. This algorithm is valid for any vector size $N = N_0 \dots N_{H-1}$ and processor number p as long as p divides all the N/N_l's, and sequential FFTs for the sizes $N_0, \dots, N_{H-1}$ are available. (Sequential versions of Algorithm 2.8 can be used for this purpose.) Though the algorithm was presented using twiddle factors (the 6-pass approach), it can also be performed using generalized FFTs (the transpose approach). The reverse is valid for Algorithm 2.2 in which the generalized butterflies can be replaced by a multiplication by twiddle factors followed by an unordered FFT. While the generalized butterfly approach is cheaper than the twiddle factor approach, the latter is based on FFT routines and hence can easily be optimized by plugging in faster sequential FFT implementations.

For an efficient parallel implementations of Algorithm 2.8 it is necessary to factor N in as few factors as possible, so that a minimum number of communication supersteps is needed. We proposed a method to do so in the case that $N = 2^{m_0}3^{m_1}5^{m_2}$, and $p = 2^{q_0}3^{q_1}5^{q_2}$, where $q_j < m_j$. This method can easily be extended for N which have larger primes in its prime factorization.

In order to efficiently implement the mixed FFT algorithm (Algorithm 2.8), it is necessary to implement the permutations involved taking care that large packets of data are being sent. Another important issue is the development of scalable lookup tables.

3

Real Fast Fourier Transform and Fast Cosine Transform

3.1. *Introduction*

In this chapter we discuss fast algorithms for discrete Fourier-like transforms of real vectors. More specifically, we derive new parallel algorithms for computing the DFT of a real vector and for computing the *discrete cosine transform* (DCT, or DCT-II in the terminology of [**55**]). As with the complex DFT, real DFTs and DCTs have many practical applications, ranging from solving numerical differential equations to signal processing. An introduction to applications of the RFFT can be found in [**13, 19**]. The book [**42**] is an excellent survey of the DCT; it treats definition, algorithms, and applications.

The DFT of a real vector $\mathbf{y}$ of size N is the complex vector $\mathbf{Y}$, also of size N, defined by (2.1). Vector $\mathbf{Y}$ is *conjugate even*, which means that it has the following property:

$$Y_{N-k} = \bar{Y}_k. \tag{3.1}$$

This property allows the vector to be packed as a real vector of size N in the following way

$$\begin{cases} f_0 = \mathrm{Re}(Y_0), & f_1 = \mathrm{Re}(Y_{N/2}), \\ f_{2k} = \mathrm{Re}(Y_k), & f_{2k+1} = \mathrm{Im}(Y_k), \qquad 1 \le k < N/2, \end{cases} \tag{3.2}$$

where **f** is the storage vector. Note that the real part of element $Y_{N/2}$ is stored in the place that would, in first instance, be reserved for the imaginary part of element Y_0. Since $Y_0 = \bar{Y}_N = \bar{Y}_0$ and $Y_{N/2} = \bar{Y}_{N/2}$ imply that both $\mathrm{Im}(Y_0)$ and $\mathrm{Im}(Y_{N/2})$ are zero, no information is lost.[1]

Any complex vector can be packed as a real vector of double the size where the real parts of the elements of the complex vector are the even elements of the real vector and the imaginary parts are the odd elements of the real vector. From now on we assume this packing for all complex vectors. This assumption will be useful for the design of the templates of this chapter, because it allows two ways of accessing vectors, by reals or by complex numbers. To avoid confusion, we always indicate that the complex vector is packed as real when we explicitly make use of this packing.

The DCT of a real vector $\mathbf{x}$ of size N is the real vector $\tilde{\mathbf{x}}$, also of size N, defined by

$$\tilde{x}_k = \sum_{j=0}^{N-1} x_j \cos \frac{(2j+1)k\pi}{2N}, \quad k = 0, \ldots, N-1. \tag{3.3}$$

Its inverse is

$$x_j = \frac{1}{N} \sum_{k=0}^{N-1} \epsilon_k \tilde{x}_k \cos \frac{(2j+1)k\pi}{2N}, \quad j = 0, \ldots, N-1, \tag{3.4}$$

where

$$\epsilon_k = \begin{cases} 1 & \text{if } k = 0, \\ 2 & \text{if } k > 0. \end{cases} \tag{3.5}$$

Both the DFT of a real vector and the DCT can be carried out using approximately half the number of flops needed by an FFT of a complex vector of the same size. The corresponding algorithms are generally called *real FFT* (RFFT) algorithms in the case of the DFT of a real vector, and *fast cosine transform* (FCT) algorithms in the case of the DCT. There exists a substantial amount of literature on the RFFT and many implementations of sequential RFFTs are available, see e.g. [**41, 49, 51, 55**]. The same holds for the FCT, see e.g. [**2, 41, 42, 47, 49, 55**]. In both cases, parallel algorithms or implementations have been less intensively studied, see [**31, 40**] for a recent discussion on RFFTs and [**33, 44**] for a recent discussion on FCTs.

---

[1] The equality $\bar{Y}_N = \bar{Y}_0$ is obtained by extending definition (2.1) to any integer k. Because the extended definition is N-periodic, it is possible to obtain any value Y_k from the computed values $Y_0, \ldots, Y_{N-1}$.

The sequential algorithms on which we base our parallel RFFT and FCT algorithms use complex FFTs as their kernel. Such algorithms have a *pack-transform-extract* structure:

1. PACK the input vector as an auxiliary complex vector of half the size.
2. TRANSFORM the auxiliary vector using a complex FFT.
3. EXTRACT the desired transform from the transformed auxiliary vector.

The pack and extract phase of such algorithms are either a permutation or a simple $O(N)$ operation that computes the new vector by combining the elements of the old vector in a pairwise fashion. This means that the bulk of the computation is in the complex FFT (CFFT). Though the cost of CFFT-based algorithms tends to be a slightly higher than non-CFFT-based algorithms [**44, 47, 49, 51, 55**], the use of CFFT-based algorithms is advantageous, because as a separate module, the CFFT can easily be replaced, for instance by a new, more efficient, CFFT subroutine. Even in the cases that the parallel CFFT module needs to be modified to reduce the communication cost, we can still make use of the techniques developed for the parallel CFFT and reuse parts of the code.

Our parallel algorithms are optimal in the sense that no extra communication (besides the communication already needed for the transform phase) is needed. Such optimality is achieved because of the introduction of a new data distribution scheme that we call the zig-zag cyclic distribution.

This chapter has the following structure. In Section 3.2, we describe the characteristics that a CFFT-based algorithm should have in order to be suitable for parallelization. We also introduce the zig-zag cyclic distribution and discuss the properties that lead to communication-optimal algorithms. In Section 3.3, we present parallel templates for the RFFT and the FCT of a single vector and discuss how to invert such algorithms. In Section 3.4, we derive a new algorithm for the simultaneous computation of two FCTs, and present parallel templates for the forward and backward transforms. In Section 3.5, we present results concerning the accuracy, efficiency, and scalability of both algorithms. We conclude with Section 3.6.

3.2. *Parallel CFFT-based algorithms and the zig-zag cyclic distribution*

3.2.1. Selection of a suitable sequential algorithm. The first step in parallelizing a CFFT-based algorithm is to choose a sequential algorithm that is suitable for parallelization. The pack-transform-extract structure of such an algorithm suggests that a suitable sequential algorithm would be one where both the pack and extract phases can be performed locally in the block distribution. If that were the case, the parallelization would be straightforward: perform the pack phase locally in the block distribution, use Algorithm 2.2 to perform the transform phase, and perform the extract phase locally in the block distribution. Unfortunately, however, this is not the case. In all currently known CFFT-based algorithms either the pack phase or the extract phase (or both) cannot be performed locally in the block distribution.

Because of this, an efficient parallel CFFT-based algorithm will require using a modified version of Algorithm 2.2 to perform the transform phase. If, for example, the pack phase is a mere permutation, it can be performed for free by combining it with the bit reversal permutation at the beginning of the CFFT algorithm. A similar combination can be done if the extract phase is a permutation. Furthermore, if the extract phase can be performed locally in the cyclic distribution, the permutation back to block distribution can be postponed until after the extract phase so that no extra communication is needed. Another possibility is to modify the distribution in which the long distance butterfly phase of the CFFT algorithm is performed in such a way that both the long distance butterflies and the extract phase can be performed locally.

The sequential RFFT algorithm we selected as the basis for our parallel RFFT algorithm has the following structure.

1. PACK the input vector **y** as an auxiliary complex vector **z** of half the size:

$$z_j = (y_{2j} + i\, y_{2j+1}), \qquad 0 \le j < N/2. \tag{3.6}$$

2. TRANSFORM the auxiliary complex vector using a CFFT of half the size:

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

3. EXTRACT the desired RFFT from the transformed auxiliary vector:

$$Y_k = Z_k - \frac{1}{2}(1 + ie^{\frac{2\pi ik}{N}})(Z_k - \bar{Z}_{N/2-k}), \qquad 0 \le k \le N/2. \tag{3.7}$$

(To compute Y_0 and $Y_{N/2}$ we use $Z_{N/2} = Z_0$.)

The algorithm described above is a standard RFFT algorithm (see e.g. [**41**, Chap. 12.3]) except for a small modification in the extract phase which we adopted from Ooura's FFT package [**38**]. It has the advantage that the pack phase can be skipped, since the complex vector **z** is already packed as a real vector of double the size. The extract phase, however, cannot be performed locally in the block distribution or in the cyclic distribution.[2] This means that we need to modify the distribution of the long

---

[2] Actually, all the CFFT-based algorithms for computing real DFTs and DCTs use some form of reflection symmetry in the pack or extract phase.

distance butterfly stages of the CFFT algorithm in order to get an efficient parallel algorithm.

The sequential FCT algorithm we selected as the basis for our parallel FCT algorithm has the following structure.

1. PACK the real input vector $\mathbf{x}$ as an auxiliary real vector $\mathbf{y}$:

$$\begin{cases} y_j = x_{2j}, & \\ y_{N-j-1} = x_{2j+1}, & 0 \le j < N/2. \end{cases} \tag{3.8}$$

2. TRANSFORM the auxiliary vector using an RFFT algorithm:

$$Y_k = \sum_{j=0}^{N-1} y_j e^{\frac{2\pi ijk}{N}}, \qquad 0 \le k \le N/2.$$

3. EXTRACT the desired cosine transform from the transformed auxiliary complex vector:

$$\begin{cases} \tilde{x}_k = \mathrm{Re}(e^{\frac{\pi ik}{2N}} \cdot Y_k), & \\ \tilde{x}_{N-k} = \mathrm{Im}(e^{\frac{\pi ik}{2N}} \cdot Y_k), & 0 \le k \le N/2. \end{cases} \tag{3.9}$$

(For simplicity, we extended definition (3.3) to all integers, so that element $\tilde{x}_N = 0$ is defined.)

This algorithm was introduced by Narasimha and Peterson [**37**]. The pack phase of the FCT is a permutation that can be combined with the trivial packing of the RFFT and the bit reversal of the CFFT inside the RFFT. The extract phase is composed of two steps: first extract the RFFT and then extract the FCT. Since the right hand side of (3.9) only depends on one element, any distribution that allows the extract phase of the RFFT to be performed locally will also allow the extract phase of the FCT to be performed locally.

As a consequence, both algorithms can be efficiently parallelized if we find a data distribution where both the long distance butterflies of a CFFT of size $N/2$ and the extract operation of the RFFT can be handled locally. In the following subsection we introduce such a distribution.

3.2.2. Zig-zag cyclic distribution.

DEFINITION 3.1 (Zig-zag cyclic distribution, $\mathrm{Z}(p, N)$). Let $\mathbf{y}$ be a vector of size N. We say that $\mathbf{y}$ is *zig-zag cyclically distributed* over p processors if, for all j, the element y_j is stored in processor $j \bmod p$ if $j \operatorname{div} p$ is even and in processor $-j \bmod p$ otherwise, and if it has local index $j \operatorname{div} p$.

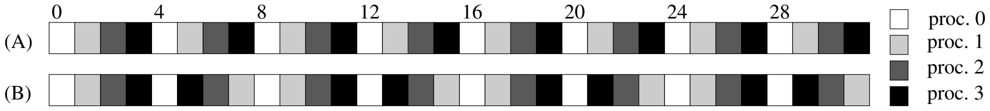


FIGURE 3.1. (A) Cyclic distribution and (B) zig-zag cyclic distribution for a vector of size 32 distributed over 4 processors (logical view).

The zig-zag cyclic distribution is a variant of the cyclic distribution. Both distributions arrange the elements of a vector in a cyclic fashion, but with different periods. While the cyclic distribution has period p, the zig-zag cyclic distribution has period $2p$. Figure 3.1 illustrates the difference between the two distributions.

LEMMA 3.2. *Let n and p be powers of two. Suppose that vector $\mathbf{y}$ of size n is zig-zag cyclically distributed over p processors. Then, the following holds.*

1. *If $p < n$ and $l < n$ is a multiple of $2p$, then elements with indices j and $j + l$, $0 \le j < n - l$ are in the same processor.*
2. *If $p < m$, where $m \le n$ is a power of two, then elements with indices k and $m - k$, $0 \le k \le m/2$, are in the same processor. (In the case $m = n$, element $m - 0 = n$ does not exist, and the range of k should be read as $0 < k \le m/2$.)*

PROOF. For any $j \in \mathbb{Z}$, $(j + l) \operatorname{div} p = j \operatorname{div} p + l/p$. Since l/p is even, $(j + l) \operatorname{div} p$ is even if and only if $j \operatorname{div} p$ is even. Furthermore, $(j + l) \bmod p = j \bmod p$ and $-(j+l) \bmod p = -j \bmod p$. As a consequence, if a vector is zig-zag cyclic distributed, elements with indices j and $j + l$, $0 \le j < n - l$, are in the same processor, proving Part 1 of the lemma.

To prove Part 2 of the lemma first suppose that $k \bmod p = 0$ so that also $-k \bmod p = 0$. In this case $(m - k) \bmod p = -k \bmod p = 0$, which implies that elements k and $m - k$ are in the same processor, namely processor 0. On the other hand, if $k \bmod p \neq 0$, then $(m - k) \operatorname{div} p = \frac{m}{p} - (k \operatorname{div} p) - 1$, which implies that $(m - k) \operatorname{div} p$ is odd if and only if $k \operatorname{div} p$ is even. This observation and the equalities $(m - k) \bmod p = -k \bmod p$ and $-(m - k) \bmod p = k \bmod p$, imply that elements k and $m - k$ are in the same processor. $\square$

We use Lemma 3.2 to verify that the zig-zag cyclic distribution is capable of handling the long distance butterfly stages of a CFFT of size $N/2$ and the extract phase of an RFFT of size N for almost any combination of p and N. Part 1 of Lemma 3.2 says that a vector of size $N/2$ that is zig-zag cyclically distributed has elements j and $j + K/2$ on the same processor as long as K is a power of two with $K \ge 4p$ (and, of course, $0 \le j < N/2 - K/2$). This means that any butterfly stage with $K \geq 4p$ can be performed locally. The long distance butterfly stages of a CFFT of size $N/2$ start from $K = 2(\frac{N}{2p})^{H-1}$, where $H = \lceil \log_{\frac{N}{2p}} \frac{N}{2} \rceil$. Since $2(\frac{N}{2p})^{H-1} < 4p$ if and only if $p = (\frac{N}{2p})^{H-1}$, the long distance butterfly phase can always be carried out locally unless $p = (\frac{N}{2p})^{H-1}$. In this exceptional case, extra communication cannot be avoided. Part 2 of Lemma 3.2 says that elements k and $N/2-k$ of the vector are on the same processor, which guarantees that the extract phase of the RFFT can also be performed locally provided that $p < N/2$.

From now on, we assume that $p < \sqrt{N/2}$ (i.e., $p < \frac{N}{2p}$). This assumption simplifies the discussion that follows, because in this case $H = 1$ or 2. If $H = 1$, there is no communication involved. If $H = 2$, the modified CFFT algorithm of the transform phase of the RFFT consists only of the short distance butterfly phase and the long distance butterfly phase, and the latter can be performed locally in the zig-zag cyclic distribution. Furthermore, restricting the number of processors by $p < \frac{N}{2p}$ reduces the communication cost to a minimum of $3(\frac{N}{p}g + l)$, corresponding to the three permutations of the resulting parallel algorithms, which have the following structure.

$1^{\mathtt{Comm}}$ Combined pack/bit reversal permutation.
$2^{\mathtt{Comp}}$ Short distance butterflies.
$3^{\mathtt{Comm}}$ Permutation to zig-zag cyclic distribution.
$4^{\mathtt{Comp}}$ Long distance butterflies.
$5^{\mathtt{Comp}}$ Extraction.
$6^{\mathtt{Comm}}$ Permutation to block distribution.

Note, however, that extending the algorithm to the general case is straightforward. Since the medium distance butterfly phases were already discussed in connection with the CFFT, the only missing part is the permutation from the last $\mathrm{C}^r(N/2,p)$ distribution to the zig-zag cyclic distribution. In the following subsections we discuss practical aspects related to the zig-zag cyclic distribution.

3.2.3. Permutation from block to zig-zag cyclic distribution. The permutation that redistributes a vector from block to zig-zag cyclic distribution is defined by

$$\begin{aligned} \zeta_{p,N} : \{0,\dots,N-1\} &\to \{0,\dots,N-1\} \\ j = j_0 \cdot p + j_1 &\mapsto k = \begin{cases} j_1 \cdot \frac{N}{p} + j_0, & \text{if } j_0 \text{ is even,} \\ -j_1 \bmod p \cdot \frac{N}{p} + j_0, & \text{otherwise,} \end{cases} \end{aligned} \tag{3.10}$$

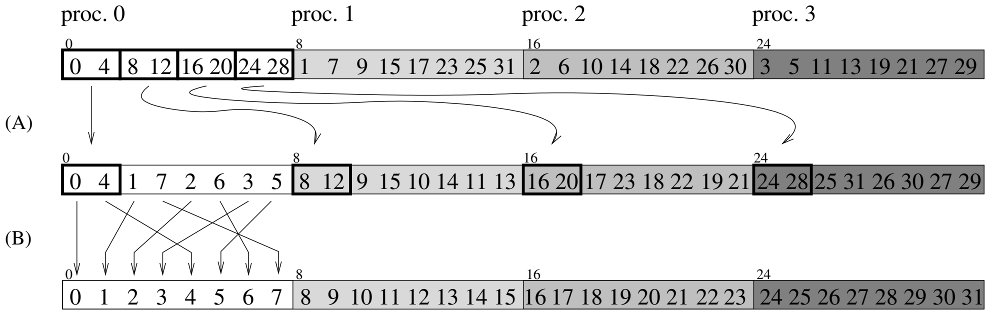


FIGURE 3.2. Schematic representation of a two stage permutation from zig-zag cyclic to block distribution (storage view). Example with $N = 32$ and $p = 4$. (A) Global cyclic permutation of packets of size 2. (B) Local permutation from virtual $\mathrm{Z}(4, 8)$ distribution to virtual $\mathrm{B}(4, 8)$ distribution. (Cf. Figure 2.3.)

where $j_0 = j \operatorname{div} p$, $j_1 = j \bmod p$. Note that $-j_1 \bmod p = -j \bmod p$. The inverse of $\zeta_{p,N}$ is

$$\begin{aligned} \zeta_{p,N}^{-1} &: \{0, \ldots, N-1\} \to \{0, \ldots, N-1\} \\ l = l_0 \cdot \frac{N}{p} + l_1 &\mapsto j = \begin{cases} l_1 \cdot p + l_0, & \text{if } l_1 \text{ is even}, \\ l_1 \cdot p + (-l_0 \bmod p), & \text{otherwise}, \end{cases} \end{aligned} \tag{3.11}$$

where $l_0 = l \operatorname{div} \frac{N}{p}$, $l_1 = l \bmod \frac{N}{p}$. (Note that l_1 is even if and only if l is even, provided that $p < N$.) These two permutations are closely related to the corresponding cyclic distribution versions, the shuffle permutations $\sigma_{p,N}$ and $\sigma_{p,N}^{-1}$, cf. (1.6).

In the permutation $\zeta_{p,N}$, the destination local index $l' = j_0$ is the same as for the permutation $\sigma_{p,N}$, but, if j_0 is odd, the destination processor is $\mathrm{Proc}(l) = -j_1 \bmod p$, instead of $\mathrm{Proc}(l) = j_1$. Conversely, in the permutation $\zeta_{p,N}^{-1}$ the destination processor $\mathrm{Proc}(j) = l_1 \cdot p \operatorname{div} \frac{N}{p}$ is the same as for $\sigma_{p,N}^{-1}$, but, if l_1 is odd, the destination local index is $j' = [l_1 \cdot p + (-l_0) \bmod p] \bmod \frac{N}{p}$, instead of $j' = (l_1 \cdot p + l_0) \bmod \frac{N}{p}$.

This similarity permits us to treat $\zeta_{p,N}$ and $\zeta_{p,N}^{-1}$ in the same way as we treated $\sigma_{p,N}$ and $\sigma_{p,N}^{-1}$ (see Section 2.3.4). Since it is possible to send packets if $p < N/p$ and we assumed that $p < N/(2p)$, we only consider sending packets. In the case of the permutation $\zeta_{p,N}$, a local permutation $\zeta_{p,\frac{N}{p}}$ is followed by a global cyclic permutation of packets of size b/p, where $b = N/p$ (here we assume that p divides b). In the case of the permutation $\zeta_{p,N}^{-1}$, a global cyclic permutation of packets of size b/p is followed by a local permutation $\zeta_{p,\frac{N}{p}}^{-1}$ (see Figure 3.2).

Algorithm 3.1 is a template for permuting a vector from block to zig-zag cyclic distribution and vice versa. We included the parameter *type* to indicate the type of the input vector, which can be real or complex.

Algorithm 3.1 Template for the parallel permutation from block to zig-zag cyclic distribution.

CALL BSP_BlockToZig($s, p, sign, b, type, \mathbf{y}$).

ARGUMENTS

```
$s$: Processor identification; $0 \le s < p$.
$p$: Number of processors.
$sign$: Permutation direction; +1 for $\zeta$, −1 for $\zeta^{-1}$.
$b$: Block size; $p$ divides $b$.
$type$: Vector type; real or complex.
$\mathbf{y} = (y_0, \ldots, y_{p\cdot b-1})$: Vector of size $p \cdot b$ (block distributed).
```

OUTPUT $y_k \leftarrow y_j$, $0 \le j < p \cdot b$, $k = \zeta_{p,p\cdot b}(j)$ if $sign = 1$, or $k = \zeta^{-1}_{p,p\cdot b}(j)$ if $sign = -1$.

DESCRIPTION

```
        if $sign = 1$ then
1^Comp      Local $\zeta_{p,b}$ permutation.
            for $j' = 0$ to $b - 1$ do
                $j_0 \leftarrow j' \operatorname{div} p$
                if $j_0$ is even then
                    $k' \leftarrow j' \bmod p \cdot \frac{b}{p} + j_0$
                else
                    $k' \leftarrow -j' \bmod p \cdot \frac{b}{p} + j_0$
                $x_{s\cdot b+k'} \leftarrow y_{s\cdot b+j'}$
2^Comm      Global cyclic permutation of packets.
            for $proc = 0$ to $p - 1$ do
                $y_{proc\cdot b+s\cdot\frac{b}{p}} \leftarrow \mathrm{Put}(proc, \frac{b}{p}, x_{s\cdot b+proc\cdot\frac{b}{p}})$
        else
3^Comm      Global cyclic permutation of packets.
            for $proc = 0$ to $p - 1$ do
                $x_{proc\cdot b+s\cdot\frac{b}{p}} \leftarrow \mathrm{Put}(proc, \frac{b}{p}, y_{s\cdot b+proc\cdot\frac{b}{p}})$
            Synchronize
4^Comp      Local $\zeta^{-1}_{p,b}$ permutation.
            for $j' = 0$ to $b - 1$ do
                $j_0 \leftarrow j' \operatorname{div} \frac{b}{p}$
                $j_1 \leftarrow j' \bmod \frac{b}{p}$
                if $j_1$ is even then
                    $k' \leftarrow j_1 \cdot p + j_0$
                else
                    $k' \leftarrow j_1 \cdot p + (-j_0 \bmod p)$
                $y_{s\cdot b+k'} \leftarrow x_{s\cdot b+j'}$
```

3.2.4. Pairwise operations using the zig-zag cyclic distribution. The lemmas that follow prove relations that are useful for the construction of the subroutines that carry out the long distance butterflies of the CFFT and the extract phases of the RFFT and of the FCT.

LEMMA 3.3. *Let* p, K, *and* N *be powers of two such that* $4p \leq K \leq N$. *Define* $k = K/p$. *Let* j *be an index,* $0 \leq j < N$.

1. *If* $j \bmod K < K/2$, *then* $\zeta_{p,N}(j) \bmod k < k/2$.
2. *If* $j < N - K/2$, *then* $\zeta_{p,N}(j + K/2) = \zeta_{p,N}(j) + k/2$.

PROOF. Part 1: Since k divides $\frac{N}{p}$ it follows that

$$\begin{aligned}\zeta_{p,N}(j) \bmod k &= (j \operatorname{div} p) \bmod k = [(j \operatorname{div} K \cdot K + j \bmod K) \operatorname{div} p] \bmod k \\ &= [j \operatorname{div} K \cdot \frac{K}{p} + (j \bmod K) \operatorname{div} p] \bmod k = \underbrace{(j \bmod K)}_{<K/2} \operatorname{div} p < k/2.\end{aligned}$$

Part 2: $(j + K/2) \bmod p = j \bmod p$, and $(j + K/2) \operatorname{div} p = j \operatorname{div} p + k/2$, which implies that $\zeta_{p,N}(j + K/2) = \zeta_{p,N}(j) + k/2$. ☐

LEMMA 3.4. *Let* p, m, *and* N *be powers of two such that* $p < m \leq N$. *Let* j *be an index,* $0 \leq j \leq m/2$. *Define* $s = \zeta_{p,N}(j) \operatorname{div} \frac{N}{p}$, *and* $k' = \zeta_{p,N}(j) \bmod \frac{N}{p}$.
Then

$$\zeta_{p,N}(m - j) = \begin{cases} s \cdot \frac{N}{p} + \frac{m}{p} - k', & \textit{if } s = 0, \\ s \cdot \frac{N}{p} + \frac{m}{p} - k' - 1, & \textit{otherwise.} \end{cases}$$

PROOF. First note that $k' = j \operatorname{div} p$ and that $s = 0$ if and only if $j \bmod p = 0$. If $j \bmod p = 0$, then $s = 0 = (m - j) \bmod p = -(m - j) \bmod p$, and $\zeta_{p,N}(m - j) = (m - j) \operatorname{div} p = \frac{m}{p} - \frac{j}{p} = s \cdot \frac{N}{p} + \frac{m}{p} - k'$. On the other hand, if $j \bmod p \neq 0$, s can assume two values: $s = j \bmod p = -(m - j) \bmod p$, if $j \operatorname{div} p$ is even, or $s = -j \bmod p = (m - j) \bmod p$, otherwise. Furthermore, if $j \bmod p \neq 0$, then $(m - j) \operatorname{div} p$ is odd if and only if $j \operatorname{div} p$ is even. It follows that in both cases

$$\zeta_{p,N}(m - j) = s \cdot \frac{N}{p} + (m - j) \operatorname{div} p = s \cdot \frac{N}{p} + \frac{m}{p} - j \operatorname{div} p - 1.$$

☐

LEMMA 3.5. *Let* p, K, *and* N *be powers of two such that* $4p \leq K \leq N$. *Define* $k = K/p$. *Let* l *be an index,* $0 \leq l < N$. *Define* $s = l \operatorname{div} \frac{N}{p}$ *and* $l' = l \bmod \frac{N}{p}$. *Then*

$$\frac{\zeta^{-1}_{p,N}(l) \bmod K}{K} = \begin{cases} \frac{l' \bmod k + s/p}{k}, & \textit{if } l' \textit{ is even or } s = 0, \\ \frac{l' \bmod k + 1 - s/p}{k}, & \textit{otherwise.} \end{cases}$$

Proof. If l' is even or $s = 0$, then $\zeta_{p,N}^{-1}(l) \bmod K = (l' \cdot p + s) \bmod K = l' \bmod k \cdot p + s$, giving

$$\frac{\zeta_{p,N}^{-1}(l) \bmod K}{K} = \frac{l' \bmod k \cdot p + s}{k\,p} = \frac{l' \bmod k + s/p}{k}.$$

If l' is odd and $s > 0$, then $\zeta_{p,N}^{-1}(l) \bmod K = [l' \cdot p + (-s \bmod p)] \bmod K = l' \bmod k \cdot p + (-s \bmod p)$, giving

$$\begin{aligned}\frac{\zeta_{p,N}^{-1}(l) \bmod K}{K} &= \frac{l' \bmod k \cdot p + (-s \bmod p)}{k\,p}\\ &= \frac{l' \bmod k \cdot p + p - s}{k\,p} = \frac{l' \bmod k + 1 - s/p}{k}.\end{aligned}$$

□

In the following subsections, we discuss how to use the lemmas above to construct templates for the generalized butterflies of the CFFT algorithm used inside the RFFT, and for the extract phases of the RFFT and the FCT algorithms.

3.2.4.1. *Generalized butterflies for the zig-zag cyclic distribution.* With the help of Lemma 3.3 and Lemma 3.5 it is easy to construct a local generalized butterfly $\tilde{B}_k^\alpha$ to be used in the new long distance butterfly stages. Lemma 3.3 states that a pair originally stored as elements j and $j + K/2$ corresponds to the new pair l and $l + k/2$, where $l = \zeta_{p,\frac{N}{2}}(j)$, and Lemma 3.5 enables us to compute the corresponding weight $w_K^{j \bmod K}$, based only on the local index l' and the parameter $\alpha = s/p$. The resulting generalized butterfly $\tilde{B}_k^\alpha$ is similar to the old B_k^α; the only difference being that the weights $w_k^{j+\alpha}$ are replaced by $w_k^{j+1-\alpha}$ if j is odd and $\alpha \neq 0$. Figure 3.3 shows why we constructed the matrix $\tilde{B}_k^\alpha$ in this way.[3]

Subroutine BTFLY_ZIG (Algorithm 3.2) carries out the butterfly operations in the zig-zag cyclic distribution. It multiplies the input vector by $\tilde{A}_{n,n}^\alpha \cdots \tilde{A}_{k_0,n}^\alpha \tilde{A}_{k_0/2,n}^\alpha$, where $\tilde{A}_{k,n}^\alpha = I_{n/k} \otimes \tilde{B}_k^\alpha$ is a generalized zig-zag butterfly stage. The new parallel CFFT algorithm carries out the long distance butterfly stages of a CFFT of size $N/2$ by first permuting the vector being transformed to the zig-zag cyclic distribution $\mathrm{Z}(p, N/2)$, and then executing subroutine BTFLY_ZIG using $\alpha = s/p$, $n = N/(2p)$, and the local part of the vector being transformed as arguments. (Since the subroutine is only used for the forward transform, argument *sign* is omitted.) Note that

---

[3] In fact, it is possible to generalize the zig-zag cyclic distribution by constructing a *group zig-zag cyclic family* of distributions in the same way we did with the cyclic distribution. Using this family we could factor the Fourier matrix F_N in a similar way as we did in (2.25) (because of Lemma 3.2, the maximum number of butterfly stages between permutations would be smaller), and prove a theorem and a corollary similar to Theorem 2.5 and Corollary 2.6.

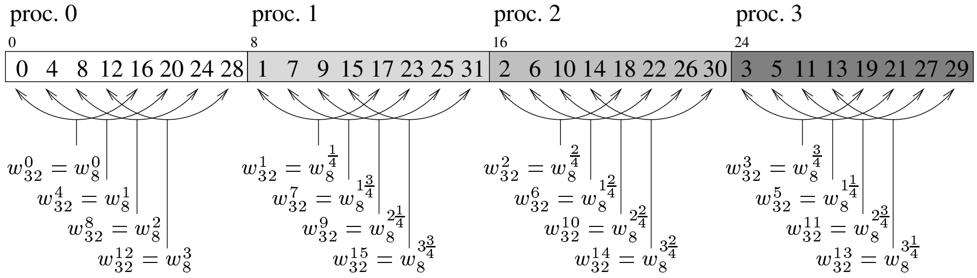

proc. 0
proc. 1
proc. 2
proc. 3
0
8
16
24
0 4 8 12 16 20 24 28
1 7 9 15 17 23 25 31
2 6 10 14 18 22 26 30
3 5 11 13 19 21 27 29
$w_{32}^{0} = w_8^{0}$
$w_{32}^{4} = w_8^{1}$
$w_{32}^{8} = w_8^{2}$
$w_{32}^{12} = w_8^{3}$
$w_{32}^{1} = w_8^{\frac{1}{4}}$
$w_{32}^{7} = w_8^{1\frac{3}{4}}$
$w_{32}^{9} = w_8^{2\frac{1}{4}}$
$w_{32}^{15} = w_8^{3\frac{3}{4}}$
$w_{32}^{2} = w_8^{\frac{2}{4}}$
$w_{32}^{6} = w_8^{1\frac{2}{4}}$
$w_{32}^{10} = w_8^{2\frac{2}{4}}$
$w_{32}^{14} = w_8^{3\frac{2}{4}}$
$w_{32}^{3} = w_8^{\frac{3}{4}}$
$w_{32}^{5} = w_8^{1\frac{1}{4}}$
$w_{32}^{11} = w_8^{2\frac{3}{4}}$
$w_{32}^{13} = w_8^{3\frac{1}{4}}$

FIGURE 3.3. Schematic representation of butterfly stage $K = kp$ with the input vector of size N zig-zag cyclically distributed over p processors (storage view). Example with $N = K = 32, p = 4$, and $k = 8$. The arrows indicate the pairs to be combined and the corresponding weights.

subroutine BTFLY_ZIG can also be used for computing the short distance butterfly stages in the block distribution by defining $\alpha = 0$. Algorithm 3.2 has the same cost function as Algorithm 2.3, which is $C_{\mathrm{BTFLY}}(n, k_0)$, cf. (2.30).

3.2.4.2. *Extract phase of the RFFT.* The extract phase of the RFFT algorithm is a pairwise operation involving complex elements originally stored as j and $N/2 - j$, and weights $\exp(2\pi i j/N)$. Lemma 3.4 with $m = N/2$ states that the original pair j and $N/2 - j$ is transformed into the new pair k and $s \cdot \frac{N}{2p} + \frac{N}{2p} - k' - \lceil \frac{s}{p} \rceil$, where $k = \zeta_{p,\frac{N}{2}}(j)$, $s = k \operatorname{div} \frac{N}{2p}$ is the processor where element k is stored, and $k' = k \bmod \frac{N}{2p}$ is the local index of k, whereas Lemma 3.5 gives a formula to compute the weight $\exp(2\pi i j/N)$, based only on the local index k' and the parameter $\alpha = s/p$.

Consider the local operation on a complex vector of size n:

$$\begin{cases} y_{k'} \leftarrow y_{k'} - \frac{1}{2}(1 + ie^{\frac{2\pi i k'}{2n}})(y_{k'} - \bar{y}_{n-k'}), & 0 \le k' \le n, \qquad \text{if } \alpha = 0, \\ y_{k'} \leftarrow y_{k'} - \frac{1}{2}(1 + ie^{\frac{2\pi i (k'+\alpha_1)}{2n}})(y_{k'} - \bar{y}_{n-k'-1}), & 0 \le k' < n, \quad \text{otherwise,} \end{cases} \tag{3.12}$$

where

$$\alpha_1 = \begin{cases} \alpha, & \text{if } k' \text{ is even or } \alpha = 0, \\ 1 - \alpha, & \text{otherwise.} \end{cases} \tag{3.13}$$

If $n = N/(2p)$, $\alpha = s/p$, and $\mathbf{y}$ is the local part of the complex vector being transformed, (3.12) corresponds to the local part of the extract phase (3.7) of the RFFT algorithm. Subroutine RFFT_EXTRACT (Algorithm 3.3) is a sequential subroutine that performs (3.12), provided that $sign = 1$. (We introduced the argument $sign$ because the same subroutine will be used in the inverse RFFT, but with $sign = -1$,

Algorithm 3.2 Template for the sequential generalized zig-zag butterfly operations.

```
CALL BTFLY_ZIG($\alpha, n, k_0, \mathbf{y}$).
ARGUMENTS
    $\alpha$: Butterfly parameter, used to compute the correct weights; $0 \leq \alpha < 1$.
    $n$: Vector size; $n$ is a power of 2 with $n \geq 2$.
    $k_0$: Smaller 4-butterfly size; $k_0$ is a power of 2 with $4 \leq k_0 \leq 2n$.
    $\mathbf{y} = (y_0, \ldots, y_{n-1})$: Complex vector of size $n$.
OUTPUT $\mathbf{y} \leftarrow \tilde{A}^{\alpha}_{n,n} \cdots \tilde{A}^{\alpha}_{k_0,n} \tilde{A}^{\alpha}_{k_0/2,n} \mathbf{y}$.
DESCRIPTION
    1. Perform pairs of butterfly stages $\tilde{A}^{\alpha}_{k,n} \tilde{A}^{\alpha}_{k/2,n}$.
       $k \leftarrow k_0$
       while $k \leq n$ do
           for $t = 0$ to $n - k$ step $k$ do
               for $j = 0$ to $k/4 - 1$ do
                   if $j$ is even or $\alpha = 0$ then $\alpha_1 \leftarrow \alpha$ else $\alpha_1 \leftarrow 1 - \alpha$
                   $yw1 \leftarrow w_k^{j+\alpha_1} \cdot y_{t+j+k/2}$
                   $yw2 \leftarrow w_k^{2(j+\alpha_1)} \cdot y_{t+j+k/4}$
                   $yw3 \leftarrow w_k^{3(j+\alpha_1)} \cdot y_{t+j+3k/4}$
                   $a \leftarrow y_{t+j} + yw2$
                   $b \leftarrow y_{t+j} - yw2$
                   $c \leftarrow yw1 + yw3$
                   $d \leftarrow yw1 - yw3$
                   $y_{t+j} \leftarrow a + c$
                   $y_{t+j+k/4} \leftarrow b + id$
                   $y_{t+j+k/2} \leftarrow a - c$
                   $y_{t+j+3k/4} \leftarrow b - id$
           $k \leftarrow 4 \cdot k$
    2. Perform the last butterfly stage $\tilde{A}^{\alpha}_{n,n}$.
       if $k = 2n$ then
           for $j = 0$ to $n/2 - 1$ do
               if $j$ is even or $\alpha = 0$ then $\alpha_1 \leftarrow \alpha$ else $\alpha_1 \leftarrow 1 - \alpha$
               $a \leftarrow w_n^{j+\alpha_1} \cdot y_{j+n/2}$
               $y_{j+n/2} \leftarrow y_j - a$
               $y_j \leftarrow y_j + a$
```

see Section 3.3.3.) Subroutine RFFT_EXTRACT uses the complex-packed-as-real notation. Note that the special packing of the global complex output vector $\mathbf{y}$, given by (3.2), requires that elements y_0 and $y_{N/2}$ of the global vector be stored at processor 0 as elements f_0 and f_1 of the local vector. The algorithm exploits the symmetries of expression (3.12) to perform the extract operation with cost

$$C_{\text{RFFT\_EXTRACT}}(n) = \frac{15}{2} n, \tag{3.14}$$

provided that the weights $\sin(\pi(k'+\alpha_1)/n)$ and $\cos(\pi(k'+\alpha_1)/n)$ are stored in a table.

Algorithm 3.3 Template for the sequential extract/pack phase of the RFFT.

CALL RFFT_EXTRACT($\alpha, sign, n, \mathbf{f}$).

ARGUMENTS

α: Parameter used to compute the correct weights; $0 \le \alpha < 1$.
$sign$: Transform direction; $+1$ for forward, -1 for backward.
n: Vector size; n is a power of 2 with $n \ge 2$.
$\mathbf{f} = (f_0, \ldots, f_{2n-1})$: Complex vector of size n packed as real.

OUTPUT $\mathbf{f}$, computed using (3.12).

DESCRIPTION

```
1. Special case $\alpha = 0$.
   if $\alpha = 0$ then
       if $sign = 1$ then
           a. Forward transform.
              $aux \leftarrow f_0 - f_1$
              $f_0 \leftarrow f_0 + f_1$
              $f_1 \leftarrow aux$
       else
           b. Inverse transform.
              $f_0 \leftarrow \frac{1}{2}(f_0 + f_1)$
              $f_1 \leftarrow f_0 - f_1$
       $k_0 \leftarrow 1$
   else $k_0 \leftarrow 0$
2. Perform extract operation.
   for $k' = k_0$ to $n/2 - 1$ do
       if $k'$ is even or $\alpha = 0$ then $\alpha_1 \leftarrow \alpha$ else $\alpha_1 \leftarrow 1 - \alpha$
       $rdiff \leftarrow f_{2k'} - f_{2(n-k'-\lceil\alpha\rceil)}$
       $isum \leftarrow f_{2k'+1} + f_{2(n-k'-\lceil\alpha\rceil)+1}$
       $raux \leftarrow \frac{1}{2}((1 - \sin(\frac{2\pi(k'+\alpha_1)}{2n})) \cdot rdiff - sign\ \cos(\frac{2\pi(k'+\alpha_1)}{2n}) \cdot isum)$
       $iaux \leftarrow \frac{1}{2}((1 - \sin(\frac{2\pi(k'+\alpha_1)}{2n})) \cdot isum + sign\ \cos(\frac{2\pi(k'+\alpha_1)}{2n}) \cdot rdiff)$
       $f_{2k'} \leftarrow f_{2k'} - raux$
       $f_{2k'+1} \leftarrow f_{2k'+1} - iaux$
       $f_{2(n-k'-\lceil\alpha\rceil)} \leftarrow f_{2(n-k'-\lceil\alpha\rceil)} + raux$
       $f_{2(n-k'-\lceil\alpha\rceil)+1} \leftarrow f_{2(n-k'-\lceil\alpha\rceil)+1} - iaux$
```

Note 1: If $\alpha = 0$, $(f_n + if_{n+1}) \leftarrow (f_n + if_{n+1})$ does not need to be computed.
Note 2: For $\alpha > 0$, k' is even if and only if $n - k' - \lceil\alpha\rceil$ is odd, so that the value $\alpha_1 = \alpha_1(k')$ used in the algorithm is correct both for k' and $n - k' - \lceil\alpha\rceil$.

3.2.4.3. *Extract phase of the FCT.* It is no problem to perform the extract phase (3.9) of the FCT locally in the zig-zag cyclic distribution, since only one element $Y_j, 0 \le j \le N/2$, of the complex vector computed in the extract phase of the RFFT is used to compute the real elements $\tilde{x}_j$ and $\tilde{x}_{N-j}$ of the output vector $\tilde{\mathbf{x}}$. The

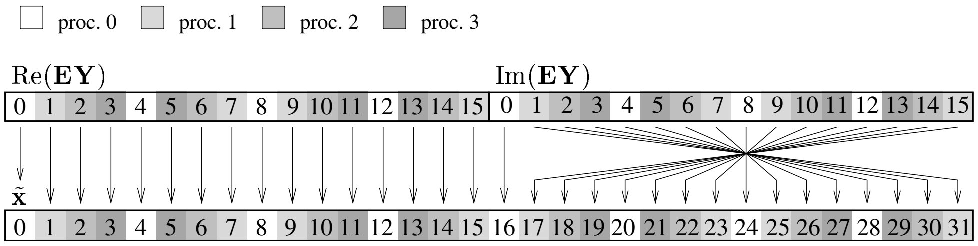


FIGURE 3.4. Extract phase of the FCT (logical view). Example with $N = 32$ and $p = 4$. The real part of $\mathbf{EY}$ is used to compute the first half of $\tilde{\mathbf{x}}$ and the imaginary part of $\mathbf{EY}$ is used to compute the second half of $\tilde{\mathbf{x}}$. Here $\mathbf{EY}$ is the vector with elements $e^{\frac{\pi i k}{2N}} \cdot Y_k$.

correct weights $\exp(\pi i j/(2N))$ are also easily found using Lemma 3.5. (Note that j corresponds to the original index, before the permutation to the zig-zag cyclic distribution.) There is only one point which still needs our attention: vector $\mathbf{Y}$ is a complex vector of size $N/2$ and vector $\tilde{\mathbf{x}}$ is a real vector of size N. Therefore, it is not immediately clear how to store the output vector $\tilde{\mathbf{x}}$. The elements $\tilde{x}_j$ and $\tilde{x}_{N-j}$ become available on the same processor, namely the processor of Y_j. Because $\mathbf{Y}$ is distributed in the zig-zag cyclic distribution of length $N/2$, it is possible to distribute $\tilde{\mathbf{x}}$ without communication by the zig-zag cyclic distribution of length N, see Figure 3.4.

This is done by defining $\alpha = s/p$ and $n = N/p$, and computing the local part of the output vector $\tilde{\mathbf{x}}$ from the local part of the transformed auxiliary vector $\mathbf{Y}$ using the local extract operation

$$\left\{\begin{array}{rll} \begin{array}{r} \tilde{x}_{k'} \leftarrow \mathrm{Re}(e^{\frac{\pi i k'}{2n}} \cdot Y_{k'}) \\ \tilde{x}_{n-k'} \leftarrow \mathrm{Im}(e^{\frac{\pi i k'}{2n}} \cdot Y_{k'}) \end{array}, & 0 \le k' \le n/2, & \text{if } \alpha = 0, \\ \begin{array}{r} \tilde{x}_{k'} \leftarrow \mathrm{Re}(e^{\frac{\pi i (k'+\alpha_1)}{2n}} \cdot Y_{k'}) \\ \tilde{x}_{n-k'-1} \leftarrow \mathrm{Im}(e^{\frac{\pi i (k'+\alpha_1)}{2n}} \cdot Y_{k'}) \end{array}, & 0 \le k' < n/2, & \text{otherwise,} \end{array}\right. \tag{3.15}$$

where α_1 is defined by (3.13). (For simplicity of notation in the case $\alpha = 0$, we included global element $\tilde{x}_N = 0$.)

Subroutine FCT_EXTRACT (Algorithm 3.4) carries out the local extract phase. The local part of the complex vector $\mathbf{Y}$ is packed as real using the vector $\mathbf{f}$. Once again, if $\alpha = 0$, elements f_0 and f_1 must be treated as exceptions. The cost of the algorithm is

$$C_{\mathrm{FCT\_EXTRACT}}(n) = 3n, \tag{3.16}$$

provided that the sines and cosines are stored in a table.

Algorithm 3.4 Template for the sequential extract phase of the FCT.

CALL FCT_EXTRACT($\alpha, n, \mathbf{f}, \tilde{\mathbf{x}}$).

ARGUMENTS

α: Parameter used to compute the correct weights; $0 \leq \alpha < 1$.
n: Vector size; n is a power of 2 with $n \geq 2$.
$\mathbf{f} = (f_0, \ldots, f_{n-1})$: Complex vector of size $n/2$ packed as real.
$\tilde{\mathbf{x}} = (\tilde{x}_0, \ldots, \tilde{x}_{n-1})$: Real vector of size n.

OUTPUT $\tilde{\mathbf{x}}$, computed using (3.15).

DESCRIPTION

```
if $\alpha = 0$ then
    $\tilde{x}_0 \leftarrow f_0$
    $\tilde{x}_{n/2} \leftarrow \cos(\frac{\pi}{4}) \cdot f_1$
    $k_0 \leftarrow 1$
else $k_0 \leftarrow 0$
for $k' = k_0$ to $n/2 - 1$ do
    if $k'$ is even or $\alpha = 0$ then $\alpha_1 \leftarrow \alpha$ else $\alpha_1 \leftarrow 1 - \alpha$
    $\tilde{x}_{k'} \leftarrow \cos(\frac{\pi(k'+\alpha_1)}{2n}) \cdot f_{2k'} - \sin(\frac{\pi(k'+\alpha_1)}{2n}) \cdot f_{2k'+1}$
    $\tilde{x}_{n-k'-\lceil\alpha\rceil} \leftarrow \sin(\frac{\pi(k'+\alpha_1)}{2n}) \cdot f_{2k'} + \cos(\frac{\pi(k'+\alpha_1)}{2n}) \cdot f_{2k'+1}$
```

3.3. *Parallel algorithms and their inverses*

In the previous section we discussed almost all the parts needed to construct the parallel templates for the RFFT and FCT algorithms. In the case of the RFFT, the work of writing the actual template is now just a simple assembling job. In the case of the FCT, we still need to explain how to compose the pack permutation (3.8) with the bit reversal.

3.3.1. Parallel real fast Fourier transform algorithm. The RFFT template (Algorithm 3.5) works as a mere coordinator that hands over the work to be executed by its subroutines. The parallel bit reversal and the short distance butterflies are carried out by subroutines BSP_BitRev and BTFLY (Algorithms 2.4 and 2.3). Subroutine BSP_BlockToZig (Algorithm 3.1) permutes the vector from block to zig-zag cyclic distribution and vice versa. Subroutine BTFLY_ZIG (Algorithm 3.2) computes the long distance butterflies corresponding to a vector that is zig-zag cyclically distributed, and subroutine RFFT_EXTRACT (Algorithm 3.3) performs the extract phase. As already said, the template only works when $p < \sqrt{N/2}$. Its cost can be computed by summing the costs of each subroutine, or even simpler by adding the cost $C_{\mathrm{FFT,par}}(\frac{N}{2}, p) = \frac{17}{4}\frac{N}{2p}\log_2\frac{N}{2} + 6\frac{N}{2p} \cdot g + 3 \cdot l$ of a complex FFT (given by

(2.38)) to the cost $C_{\mathrm{RFFT\_EXTRACT}}(\frac{N}{2p}) = \frac{15}{2}\frac{N}{2p}$ of the local extract phase (3.12),

$$C_{\mathrm{RFFT,par}}(N,p) = \frac{17}{8}\frac{N}{p}\log_2 N + \frac{13}{8}\frac{N}{p} + 3\frac{N}{p}\cdot g + 3\cdot l. \tag{3.17}$$

Algorithm 3.5 Template for the parallel forward real fast Fourier transform.

CALL BSP_RFFT$(s, p, sign = 1, N, \mathbf{y})$.

ARGUMENTS

s: Processor identification; $0 \le s < p$.
p: Number of processors; p is a power of 2 with $p < \sqrt{N/2}$.
$sign = 1$: Transform direction is forward.
N: Transform size; N is a power of 2 with $N \ge 2$.
$\mathbf{y} = (y_0, \ldots, y_{N-1})$: Real vector of size N (block distributed).

OUTPUT $\mathbf{y} \leftarrow (\mathrm{Re}(Y_0), \mathrm{Re}(Y_{N/2}), \mathrm{Re}(Y_1), \mathrm{Im}(Y_1), \ldots, \mathrm{Re}(Y_{N/2-1}), \mathrm{Im}(Y_{N/2-1}))$, where $Y_k = \sum_{j=0}^{N-1} y_j \exp(2\pi ijk/N)$, $0 \le k \le N/2$ ($Y_{N-k} = \bar{Y}_k$ is not explicitly computed).

DESCRIPTION

$1^{\mathtt{CpCm}}$ Parallel complex bit reversal permutation.
BSP_BitRev$(s, p, N/2, \mathbf{y})$

$2^{\mathtt{Comp}}$ Short distance butterflies.
BTFLY$(0, 1, \frac{N}{2p}, 4, \mathbf{y}_{\mathbf{s\frac{N}{p}}})$

$3^{\mathtt{CpCm}}$ Permutation from block to zig-zag cyclic distribution.
BSP_BlockToZig$(s, p, 1, \frac{N}{2p}, \mathrm{complex}, \mathbf{y})$

$4^{\mathtt{Comp}}$ Long distance butterflies.
BTFLY_ZIG$(\frac{s}{p}, \frac{N}{2p}, 4\frac{N}{2p^2}, \mathbf{y}_{\mathbf{s\frac{N}{p}}})$

$5^{\mathtt{Comp}}$ RFFT extract phase.
RFFT_EXTRACT$(\frac{s}{p}, 1, \frac{N}{2p}, \mathbf{y}_{\mathbf{s\frac{N}{p}}})$

$6^{\mathtt{CpCm}}$ Permutation from zig-zag cyclic to block distribution.
BSP_BlockToZig$(s, p, -1, \frac{N}{2p}, \mathrm{complex}, \mathbf{y})$

Note : In the main routine, vector $\mathbf{y}$ is always addressed as real. Hence we use $\mathbf{y}_{\mathbf{s\frac{N}{p}}}$ and not $\mathbf{y}_{\mathbf{s\frac{N}{2p}}}$ to denote the beginning of the local vector.

3.3.2. Parallel fast cosine transform algorithm. The bit reversal permutation of a complex vector of size $N/2$ packed as real can be formulated as the following real permutation

$$\begin{aligned} \mathrm{Rrev}_{N/2} : \{0, \ldots, N-1\} &\to \{0, \ldots, N-1\} \\ j &\mapsto l = \mathrm{rev}_N(\sigma_{2,N}(j)). \end{aligned} \tag{3.18}$$

Lemma 3.6. *Let N be a power of two, and j be an index $0 \leq j < N$. Then*

1. $\sigma_{2,N}(N-j-1) = N - \sigma_{2,N}(j) - 1$.
2. $\mathrm{rev}_N(N-j-1) = N - \mathrm{rev}_N(j) - 1$.

Proof. Straightforward from the definitions. ☐

With the help of (3.18) and of Lemma 3.6 we obtain the following description for the composition of the pack phase (3.8) and the complex bit reversal of size $N/2$.

$$\rho_N : \{0,\dots,N-1\} \to \{0,\dots,N-1\}$$

$$j \mapsto l = \begin{cases} \mathrm{rev}_N(\sigma_{2,N}(j \operatorname{div} 2)), & \text{if } j \text{ is even,} \\ N - \mathrm{rev}_N(\sigma_{2,N}(j \operatorname{div} 2)) - 1, & \text{otherwise.} \end{cases} \tag{3.19}$$

Using Lemma 2.7 we obtain the corresponding destination processor and the destination index:

$$l = \mathrm{Proc}(l) \cdot \frac{N}{p} + l', \tag{3.20}$$

where

$$\mathrm{Proc}(l) = \begin{cases} \mathrm{rev}_p(\sigma_{2,N}(j \operatorname{div} 2) \bmod p), & \text{if } j \text{ is even,} \\ (p - \mathrm{rev}_p(\sigma_{2,N}(j \operatorname{div} 2) \bmod p) - 1), & \text{otherwise.} \end{cases} \tag{3.21}$$

and

$$l' = \begin{cases} \mathrm{rev}_{\frac{N}{p}}(\sigma_{2,N}(j \operatorname{div} 2) \operatorname{div} p), & \text{if } j \text{ is even,} \\ \frac{N}{p} - \mathrm{rev}_{\frac{N}{p}}(\sigma_{2,N}(j \operatorname{div} 2) \operatorname{div} p) - 1, & \text{otherwise.} \end{cases} \tag{3.22}$$

Starting with (3.20), we derived a parallel version of this permutation based on packets, which practically halved the execution time of the implementation. The resulting pack, send packet, and unpack phases are quite complicated, and we will omit the details here. (The final template of the permutation would be similar to Algorithm 3.10 and is not presented here.)

The template for the parallel FCT algorithm is presented as Algorithm 3.6. Except for the permutation ρ, all the phases are carried out by previously discussed subroutines; these are BTFLY, BSP_BlockToZig, BTFLY_ZIG, RFFT_EXTRACT, and FCT_EXTRACT (Algorithms 2.3, 3.1, 3.2, 3.3, and 3.4, respectively). The cost of the algorithm is

$$C_{\mathrm{FCT,par}}(N,p) = \frac{17}{8}\frac{N}{p}\log_2 N + \frac{37}{8}\frac{N}{p} + 3\frac{N}{p} \cdot g + 3 \cdot l, \tag{3.23}$$

which is obtained by adding the cost $C_{\mathrm{RFFT,par}}(N,p)$ of a parallel RFFT to the cost $C_{\mathrm{FCT\_EXTRACT}}(N/p) = 3N/p$ of the local extract phase (3.15).

Algorithm 3.6 Template for the parallel forward fast cosine transform.

CALL BSP_FCT$(s, p, sign = 1, N, \mathbf{x})$.

ARGUMENTS

s: Processor identification; $0 \leq s < p$.
p: Number of processors; p is a power of 2 with $p < \sqrt{N/2}$.
$sign = 1$: Transform direction is forward.
N: Transform size; N is a power of 2 with $N \geq 2$.
$\mathbf{x} = (x_0, \ldots, x_{N-1})$: Real vector of size N (block distributed).

OUTPUT $\mathbf{x} \leftarrow (\tilde{x}_0, \ldots, \tilde{x}_{N-1})$, where $\tilde{x}_k = \sum_{j=0}^{N-1} x_j \cos(\pi(j + \frac{1}{2})k/N)$.

DESCRIPTION

```
$1^{\mathtt{Comm}}$   Parallel $\rho$ permutation.
       for $j = s\frac{N}{p}$ to $(s+1)\frac{N}{p} - 1$ step 2 do
           $l' \leftarrow \mathrm{rev}_{\frac{N}{p}}(\sigma_{2,N}(j \,\mathrm{div}\, 2) \,\mathrm{div}\, p)$
           $proc \leftarrow \mathrm{rev}_p(\sigma_{2,N}(j \,\mathrm{div}\, 2) \bmod p)$
           $y_{proc\frac{N}{p}+l'} \leftarrow \mathrm{Put}(proc, 1, x_j)$
           $y_{(p-proc-1)\frac{N}{p}+\frac{N}{p}-l'-1} \leftarrow \mathrm{Put}(p - proc - 1, 1, x_{j+1})$
       Synchronize
$2^{\mathtt{Comp}}$   Short distance butterflies.
       BTFLY$(0, 1, \frac{N}{2p}, 4, \mathbf{y}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}})$
$3^{\mathtt{CpCm}}$   Complex permutation from block to zig-zag cyclic distribution.
       BSP_BlockToZig$(s, p, 1, \frac{N}{2p}, \mathrm{complex}, \mathbf{y})$
$4^{\mathtt{Comp}}$   Long distance butterflies.
       BTFLY_ZIG$(\frac{s}{p}, \frac{N}{2p}, 4\frac{N}{2p^2}, \mathbf{y}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}})$
$5^{\mathtt{Comp}}$   RFFT extract phase.
       RFFT_EXTRACT$(\frac{s}{p}, 1, \frac{N}{2p}, \mathbf{y}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}})$
$6^{\mathtt{Comp}}$   FCT extract phase.
       FCT_EXTRACT$(\frac{s}{p}, \frac{N}{p}, \mathbf{y}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}, \mathbf{x}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}})$
$7^{\mathtt{CpCm}}$   Real permutation from zig-zag cyclic to block distribution.
       BSP_BlockToZig$(s, p, -1, \frac{N}{p}, \mathrm{real}, \mathbf{x})$
```

Note : In the main routine, vector $\mathbf{y}$ is always addressed as real. Hence we use $\mathbf{y}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}$ and not $\mathbf{y}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{2p}}}$ to denote the beginning of the local vector.

3.3.3. Inverse transforms.

The inverse RFFT, i.e., the inverse FFT of a conjugate even complex vector, is obtained by inverting the phases of the forward RFFT and executing them in reversed order. The inverse RFFT algorithm has the following phases.

1. PACK the complex input vector **Y** as an auxiliary complex vector **Z** of half the size:
$$Z_k = Y_k - \frac{1}{2}(1 - ie^{-\frac{2\pi ik}{N}})(Y_k - \bar{Y}_{N/2-k}), \qquad 0 \le k < N/2. \tag{3.24}$$
(To compute Z_0 use the special packing of **Y**.)
2. TRANSFORM the auxiliary vector using an inverse CFFT of half the size:
$$z_j = \frac{1}{N/2}\sum_{k=0}^{N/2-1} Z_k e^{-\frac{2\pi ijk}{N/2}}, \qquad 0 \le j < N/2.$$
3. EXTRACT the desired inverse RFFT from the transformed auxiliary vector:
$$y_{2j} = \mathrm{Re}(z_j), \quad y_{2j+1} = \mathrm{Im}(z_j), \qquad 0 \le j < N/2. \tag{3.25}$$

The pack phase (3.24), which is similar to the extract phase (3.7) of the RFFT, is carried out in two phases: phase one permutes the input vector to the zig-zag cyclic distribution; phase two executes the local part of (3.24) by calling subroutine RFFT_EXTRACT with $sign = -1$, $\alpha = s/p$, $n = N/(2p)$, and the local part of the (ζ permuted) complex vector **y** as parameters. To reduce the communication to a minimum, the transform phase is carried out using a transposed version of the inverse CFFT algorithm (see below). The extract phase can be skipped since the complex vector was packed as real.

The transposed version of the inverse CFFT algorithm is obtained by the following decomposition of the inverse Fourier matrix F_N^{-1}:
$$NF_N^{-1} = \bar{F}_N^T = P_N \bar{A}_{2,N}^T \bar{A}_{4,N}^T \bar{A}_{8,N}^T \cdots \bar{A}_{N,N}^T. \tag{3.26}$$
Note that $\bar{A}_{K,N}^T = I_{N/K} \otimes \bar{B}_K^T$ and
$$\bar{B}_K^T = \begin{bmatrix} I_{K/2} & I_{K/2} \\ \bar{\Omega}_{K/2} & -\bar{\Omega}_{K/2} \end{bmatrix}. \tag{3.27}$$
($\bar{B}_K^T$ is also known as Gentleman-Sande butterfly.) Computing the inverse CFFT using (3.26) amounts to reversing the order of the stages of the CFFT and transposing each stage. This means that no permutation back to block distribution is needed after the pack phase, because the long distance butterflies can be locally performed on a vector that is already zig-zag cyclically distributed. (If we had used the same CFFT algorithm as in the forward transform, but with different sign, i.e., conjugated weights, an extra permutation would have been needed.)

Algorithm 3.7 is a template for the inverse RFFT. As in the forward version, the template only hands over the work to its subroutines. Note that the resulting vector still needs to be scaled by dividing it by $N/2$. To complete the description of the inverse RFFT we give a template for the transposed generalized butterflies, subroutine TBTFLY_ZIG, as Algorithm 3.8.

Similarly to the inverse RFFT algorithm, the inverse FCT algorithm is obtained by inverting the various phases and performing them in reversed order. We do not give a template.

Algorithm 3.7 Template for the parallel backward real fast Fourier transform.

CALL BSP_RFFT($s, p, sign = -1, N, \mathbf{y}$).

ARGUMENTS

s: Processor identification; $0 \leq s < p$.
p: Number of processors; p is a power of 2 with $p < \sqrt{N/2}$.
$sign = -1$: Transform direction is backward.
N: Transform size; N is a power of 2 with $N \geq 2$.
$\mathbf{y} = (\mathrm{Re}(Y_0), \mathrm{Re}(Y_{N/2}), \mathrm{Re}(Y_1), \mathrm{Im}(Y_1), \ldots, \mathrm{Re}(Y_{N/2-1}), \mathrm{Im}(Y_{N/2-1}))$: Complex vector of size N, such that $Y_{N-k} = \bar{Y}_k$, packed as real using (3.2) (block distributed).

OUTPUT $\mathbf{y} \leftarrow (y_0, y_1, \ldots, y_{N-1})$, where $y_j = \sum_{k=0}^{N-1} Y_k \exp(-2\pi i jk/N)$, $0 \leq j \leq N-1$, $(y_j \in \mathbb{R})$.

DESCRIPTION

1^{CpCm} Permutation to zig-zag cyclic distribution.
BSP_BlockToZig($s, p, 1, \frac{N}{2p}$, complex, $\mathbf{y}$)

2^{Comp} Inverse RFFT pack phase.
RFFT_EXTRACT($\frac{s}{p}, -1, \frac{N}{2p}, \mathbf{y}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}$)

3^{Comp} Long distance transposed butterflies.
TBTFLY_ZIG($\frac{s}{p}, \frac{N}{2p}, 4\frac{N}{2p^2}, \mathbf{y}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}$)

4^{CpCm} Permutation to block distribution.
BSP_BlockToZig($s, p, -1, \frac{N}{2p}$, complex, $\mathbf{y}$)

5^{Comp} Short distance transposed butterflies.
TBTFLY_ZIG($0, \frac{N}{2p}, 4, \mathbf{y}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}$)

6^{CpCm} Parallel complex bit reversal permutation.
BSP_BitRev($s, p, N/2, \mathbf{y}$)

Note : In the main routine, vector $\mathbf{y}$ is always addressed as real. Hence we use $\mathbf{y}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}$ and not $\mathbf{y}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{2p}}}$ to denote the beginning of the local vector.

3.4. *Simultaneous fast cosine transform of two vectors*

Our main motivation in developing an algorithm that computes the FCT of two vectors simultaneously (FCT2) was to apply it in the fast Legendre transform algorithm discussed in the following chapter. However, such an algorithm can be used in other applications that need to compute the FCT of more than one vector at a time,

Algorithm 3.8 Template for the sequential transposed conjugate generalized zig-zag butterfly operations.

CALL TBTFLY_ZIG($\alpha, n, k_0, \mathbf{y}$).

ARGUMENTS

α: Butterfly parameter, used to compute the correct weights; $0 \leq \alpha < 1$.
n: Vector size; n is a power of 2 with $n \geq 2$.
k_0: Smallest 4-butterfly size; k_0 is a power of 2 with $4 \leq k_0 \leq 2n$.
$\mathbf{y} = (y_0, \ldots, y_{n-1})$: Complex vector of size n.

OUTPUT $\mathbf{y} \leftarrow (\overline{\tilde{A}^{\alpha}_{k_0/2,n}})^T (\overline{\tilde{A}^{\alpha}_{k_0,n}})^T \ldots (\overline{\tilde{A}^{\alpha}_{n,n}})^T \mathbf{y}$.

DESCRIPTION

```
1. Perform pairs of butterfly stages $(\overline{\tilde{A}^{\alpha}_{k/2,n}})^T (\overline{\tilde{A}^{\alpha}_{k,n}})^T$.
   $k \leftarrow n$
   while $k \geq k_0$ do
       for $t = 0$ to $n - k$ step $k$ do
           for $j = 0$ to $k/4 - 1$ do
               if $j$ is even or $\alpha = 0$ then $\alpha_1 \leftarrow \alpha$ else $\alpha_1 \leftarrow 1 - \alpha$
               $a \leftarrow y_{t+j} + y_{t+j+k/2}$
               $b \leftarrow y_{t+j} - y_{t+j+k/2}$
               $c \leftarrow y_{t+j+k/4} + y_{t+j+3k/4}$
               $d \leftarrow y_{t+j+k/4} - y_{t+j+3k/4}$
               $y_{t+j} \leftarrow a + c$
               $y_{t+j+k/4} \leftarrow \bar{w}_k^{2(j+\alpha_1)}(a - c)$
               $y_{t+j+k/2} \leftarrow \bar{w}_k^{j+\alpha_1}(b - id)$
               $y_{t+j+3k/4} \leftarrow \bar{w}_k^{3(j+\alpha_1)}(b + id)$
       $k \leftarrow k/4$
2. Perform the last butterfly stage $(\overline{\tilde{A}^{\alpha}_{k_0/2,n}})^T$.
   if $k = k_0/2$ then
       for $t = 0$ to $n - k$ step $k$ do
           for $j = 0$ to $k/2 - 1$ do
               if $j$ is even or $\alpha = 0$ then $\alpha_1 \leftarrow \alpha$ else $\alpha_1 \leftarrow 1 - \alpha$
               $a \leftarrow y_{t+j} - y_{t+j+k/2}$
               $y_{t+j} \leftarrow y_{t+j} + y_{t+j+k/2}$
               $y_{t+j+k/2} \leftarrow \bar{w}_k^{j+\alpha_1} \cdot a$
```

for instance a two-dimensional FCT. The new algorithm is based on the same FCT algorithm we used as the basis for our parallel FCT algorithm [**37**], and on standard techniques for computing the FFTs of two real input vectors at the same time (see e.g. [**41**]).

3.4.1. Derivation. The FCT2 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 pack-transform-extract phases are described below.

1. PACK the input data into an auxiliary complex vector $\mathbf{z}$ of length N:

$$\begin{cases} z_j = (x_{2j} + i\, y_{2j}), & \\ z_{N-j-1} = (x_{2j+1} + i\, y_{2j+1}), & 0 \le j < N/2. \end{cases} \tag{3.28}$$

2. TRANSFORM the auxiliary complex vector using a CFFT of size N:

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

3. EXTRACT the two FCTs from the transformed vector:

$$(\tilde{x}_k + i\tilde{y}_k) = \frac{1}{2}(e^{\frac{\pi i k}{2N}} Z_k + e^{-\frac{\pi i k}{2N}} Z_{N-k}), \qquad 0 \le k < N. \tag{3.29}$$

(To compute $\tilde{x}_0 + i\tilde{y}_0$ remember that $Z_N = Z_0$.)

To verify that the procedure above indeed produces the desired FCTs, we first compute

$$e^{\frac{\pi i k}{2N}} Z_k = \sum_{j=0}^{N/2-1} [(x_{2j} + iy_{2j}) e^{\frac{\pi i k(4j+1)}{2N}} + (x_{2j+1} + iy_{2j+1}) e^{-\frac{\pi i k(4j+3)}{2N}}], \tag{3.30}$$

and

$$e^{-\frac{\pi i k}{2N}} Z_{N-k} = \sum_{j=0}^{N/2-1} [(x_{2j} + iy_{2j}) e^{-\frac{\pi i k(4j+1)}{2N}} + (x_{2j+1} + iy_{2j+1}) e^{\frac{\pi i k(4j+3)}{2N}}]. \tag{3.31}$$

Now, by adding (3.30) to (3.31), and dividing the result by 2 we obtain (3.29):

$$\begin{aligned}
&\frac{1}{2}(e^{\frac{\pi i k}{2N}} Z_k + e^{-\frac{\pi i k}{2N}} Z_{N-k}) = \\
&\quad = \frac{1}{2} \sum_{j=0}^{N/2-1} [(x_{2j} + iy_{2j})(e^{\frac{\pi i k(4j+1)}{2N}} + e^{-\frac{\pi i k(4j+1)}{2N}}) \\
&\qquad\qquad + (x_{2j+1} + iy_{2j+1})(e^{-\frac{\pi i k(4j+3)}{2N}} + e^{\frac{\pi i k(4j+3)}{2N}})] \\
&\quad = \sum_{j=0}^{N/2-1} [(x_{2j} + iy_{2j}) \cos \frac{2 \cdot 2j + 1}{2N} k\pi + (x_{2j+1} + iy_{2j+1}) \cos \frac{2 \cdot (2j+1) + 1}{2N} k\pi] \\
&\quad = \sum_{j=0}^{N-1} (x_j + iy_j) \cos \frac{2j+1}{2N} k\pi = \tilde{x}_k + i\tilde{y}_k.
\end{aligned}$$

The inverse FCT2 transform is obtained by inverting the procedure described above. The inverted phases are performed in the reverse order, which gives the following result.

1. PACK the input data into the auxiliary complex vector **Z**:

$$Z_k = e^{-\frac{\pi i k}{2N}}[(\tilde{x}_k + i\,\tilde{y}_k) + i(\tilde{x}_{N-k} + i\,\tilde{y}_{N-k})], \qquad 0 \le k < N. \tag{3.32}$$

(Z_0 is obtained using $\tilde{x}_N = \tilde{y}_N = 0$.)

2. TRANSFORM the auxiliary vector using an inverse CFFT of size N:

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

3. EXTRACT the two inverse FCTs from the transformed vector:

$$\begin{cases} (x_{2j} + iy_{2j}) = z_j, & \\ (x_{2j+1} + iy_{2j+1}) = z_{N-j-1}, & 0 \le j < N/2. \end{cases} \tag{3.33}$$

3.4.2. Parallel forward transform. The template for the forward FCT2 is shown as Algorithm 3.9. Throughout the algorithm, the real vectors **x** and **y** (to be transformed) are viewed as the complex vector $\mathbf{z} = \mathbf{x} + i\mathbf{y}$ of size N. The algorithm is given in terms of complex numbers, because using the complex-packed-as-real notation could lead to confusion. Note that formulating the algorithm using the two real vectors **x** and **y** as input and output does not imply any extra overhead as long as the two permutations at the beginning and at the end of the algorithm are modified to serve this purpose.

The pack phase combined with the bit reversal gives the permutation

$$\begin{aligned} \varrho : \{0,\ldots,N-1\} &\to \{0,\ldots,N-1\} \\ j \mapsto l &= \begin{cases} \mathrm{rev}_N(j \operatorname{div} 2), & \text{if } j \text{ is even,} \\ N - \mathrm{rev}_N(j \operatorname{div} 2) - 1, & \text{otherwise.} \end{cases} \end{aligned} \tag{3.34}$$

Note that $\varrho^{-1} = \varrho$. This is easily proven using the binary representation of j. The property $\varrho^{-1} = \varrho$ implies that the permutation ϱ can be performed by a sequence of independent swaps of pairs $(j, \varrho(j))$. This implies that, in the sequential case, this permutation can easily be performed in place, and, therefore, the sequential FCT2 algorithm can also be easily performed in place. Subroutine BSP_Rho (Algorithm 3.10) is a parallel version of permutation ϱ. It consists of a local modified $\sigma_{2,\frac{N}{p}}$ permutation, followed by a global permutation of packets and a local bit reversal.

The short and long distance butterfly phases of the CFFT are performed by the subroutine BTFLY_ZIG (Algorithm 3.2). Note that, since the transform phase is a CFFT of size N, the restriction on p is relaxed to $p < \sqrt{N}$.

Algorithm 3.9 Template for the parallel forward fast cosine transform of two vectors.

```
CALL  BSP_FCT2($s, p, sign = 1, N, \mathbf{z}$).
ARGUMENTS
    $s$: Processor identification; $0 \leq s < p$.
    $p$: Number of processors; $p$ is a power of 2 with $p < \sqrt{N}$.
    $sign = 1$: Transform direction is forward.
    $N$: Transform size; $N$ is a power of 2 with $N \geq 2$.
    $\mathbf{z} = (x_0 + iy_0, \ldots, x_{N-1} + iy_{N-1})$: Two real vectors of size $N$ packed as complex (block distributed).
OUTPUT  $\mathbf{z} \leftarrow (\tilde{x}_0 + i\tilde{y}_0, \ldots, \tilde{x}_{N-1} + i\tilde{y}_{N-1})$, where $\tilde{x}_k = \sum_{j=0}^{N-1} x_j \cos(\pi(j+\frac{1}{2})k/N)$, and $\tilde{y}_k = \sum_{j=0}^{N-1} y_j \cos(\pi(j+\frac{1}{2})k/N)$.
DESCRIPTION
$1^{\texttt{Comm}}$   Parallel $\varrho$ permutation.
        BSP_Rho($s, p, \frac{N}{p}, \mathbf{z}$)
$2^{\texttt{Comp}}$   Short distance butterflies.
        BTFLY_ZIG($0, \frac{N}{p}, 4, \mathbf{z}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}$)
$3^{\texttt{CpCm}}$   Complex permutation to zig-zag cyclic distribution.
        BSP_BlockToZig($s, p, 1, \frac{N}{p}$, complex, $\mathbf{z}$)
$4^{\texttt{Comp}}$   Long distance butterflies.
        BTFLY_ZIG($\frac{s}{p}, \frac{N}{p}, 4\frac{N}{p^2}, \mathbf{z}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}$)
$5^{\texttt{Comp}}$   FCT2 extract phase.
        FCT2_EXTRACT($\frac{s}{p}, \frac{N}{p}, \mathbf{z}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}$)
$6^{\texttt{CpCm}}$   Complex permutation to block distribution.
        BSP_BlockToZig($s, p, -1, \frac{N}{p}$, complex, $\mathbf{z}$)
```

The extract phase is performed by subroutine FCT2_EXTRACT (Algorithm 3.11) which executes the following local extract operation

$$\tilde{z}_{k'} \leftarrow \frac{1}{2}(e^{\frac{\pi i(k'+\alpha_1)}{2n}} Z_{k'} + e^{-\frac{\pi i(k'+\alpha_1)}{2n}} Z_{n-k'-\lceil\alpha\rceil}), \qquad 0 \leq k' < n, \tag{3.35}$$

where $n = N/p$, $\alpha = s/p$, and α_1 is defined by (3.13). If the global vectors $\mathbf{z}$ and $\mathbf{Z}$ used in the extract phase (3.29) are both permuted by $\zeta_{p,N}$, it is possible to prove that (3.35) corresponds to the local part of the extract phase (3.29). This is done with the help of Lemmas 3.4 and 3.5. Exploiting the symmetries in (3.35), subroutine FCT2_EXTRACT has cost

$$C_{\text{FCT2\_EXTRACT}}(n) = 10n, \tag{3.36}$$

provided that the sines and cosines are stored in a table. The total cost of the FCT2 algorithm is the cost of a CFFT of size N ($C_{\text{FFT,par}}(N,p)$, cf. (2.38)) plus the cost

Algorithm 3.10 Template for the parallel ϱ permutation.

CALL BSP_Rho($s, p, b, \mathbf{y}$).

ARGUMENTS

s: Processor identification; $0 \leq s < p$.
p: Number of processors.
b: Block size; $2p$ divides b.
$\mathbf{y} = (y_0, \ldots, y_{p \cdot b-1})$: Vector of size $p \cdot b$ (block distributed).

OUTPUT $y_{\varrho(j)} \leftarrow y_j$, $0 \leq j < p \cdot b$.

DESCRIPTION

```
$1^{\text{Comp}}$  Local permutation.
    for $j' = 0$ to $b-1$ do
        if $j'$ is even then $j_0 \leftarrow j' \operatorname{div}(2p)$ else $j_0 \leftarrow \frac{b}{2p} - j' \operatorname{div}(2p) - 1$
        $x_{s \cdot b + j' \bmod (2p) \cdot \frac{b}{2p} + j_0} \leftarrow y_{s \cdot b + j'}$
$2^{\text{Comm}}$  Global permutation of packets.
    for $j' = 0$ to $b-1$ step $\frac{b}{p}$ do
        $proc = \operatorname{rev}_p(j' \operatorname{div} \frac{b}{p})$
        $y_{proc \cdot b + s \cdot \frac{b}{2p}} \leftarrow \operatorname{Put}(proc, \frac{b}{2p}, x_{s \cdot b + j'})$
        $y_{(p-proc-1) \cdot b + (2p-s-1) \cdot \frac{b}{2p}} \leftarrow \operatorname{Put}(p - proc - 1, \frac{b}{2p}, x_{s \cdot b + j' + \frac{b}{2p}})$
    Synchronize
$3^{\text{Comp}}$  Local bit reversal permutation.
    for $j' = 0$ to $b-1$ do
        if $j' < \operatorname{rev}_b(j')$ then $y_{s \cdot b + \operatorname{rev}_b(j')} \leftrightarrow y_{s \cdot b + j'}$
```

Algorithm 3.11 Template for the sequential extract phase of the FCT2.

CALL FCT2_EXTRACT($\alpha, n, \mathbf{z}$).

ARGUMENTS

α: Parameter used to compute the correct weights; $0 \leq \alpha < 1$.
n: Vector size; n is a power of 2 with $n \geq 2$.
$\mathbf{z} = (Z_0, \ldots, Z_{n-1})$: Complex vector of size n.

OUTPUT $\mathbf{z} \leftarrow (\tilde{z}_0, \ldots, \tilde{z}_{n-1})$, computed using (3.35).

DESCRIPTION

```
if $\alpha = 0$ then
    $z_{n/2} \leftarrow \cos(\frac{\pi}{4}) \cdot z_{n/2}$
    $k_0 \leftarrow 1$
else $k_0 \leftarrow 0$
for $k' = k_0$ to $n/2 - 1$ do
    if $k'$ is even or $\alpha = 0$ then $\alpha_1 \leftarrow \alpha$ else $\alpha_1 \leftarrow 1 - \alpha$
    $sum \leftarrow \frac{1}{2}(z_{k'} + z_{n-k'-\lceil\alpha\rceil})$
    $diff \leftarrow \frac{1}{2}(z_{k'} - z_{n-k'-\lceil\alpha\rceil})$
    $z_{k'} \leftarrow \cos(\frac{\pi(k'+\alpha_1)}{2n}) \cdot sum + i \sin(\frac{\pi(k'+\alpha_1)}{2n}) \cdot diff$
    $z_{n-k'-\lceil\alpha\rceil} \leftarrow \sin(\frac{\pi(k'+\alpha_1)}{2n}) \cdot sum - i \cos(\frac{\pi(k'+\alpha_1)}{2n}) \cdot diff$
```

of the extract phase ($C_{\text{FCT2\_EXTRACT}}(N/p) = 10N/p$):

$$C_{\text{FCT2,par}}(N,p) = \frac{17}{4}\frac{N}{p}\log_2 N + 10\frac{N}{p} + 6\frac{N}{p}\cdot g + 3\cdot l. \qquad (3.37)$$

3.4.3. Parallel backward transform. The template for the backward FCT2 is shown as Algorithm 3.12. It works as follows. Before executing the pack phase, the input vector is permuted to the zig-zag cyclic distribution. The pack phase is carried out by the subroutine FCT2_PACK (Algorithm 3.13) which performs the local operation:

$$Z_k \leftarrow e^{-\frac{\pi i(k'+\alpha_1)}{2n}}(\tilde{z}_{k'} + i\tilde{z}_{n-k'-\lceil\alpha\rceil}), \qquad 0 \le k' < n, \qquad (3.38)$$

Algorithm 3.12 Template for the parallel backward fast cosine transform of two vectors.

CALL BSP_FCT2($s, p, sign = -1, N, \mathbf{z}$).

ARGUMENTS

- s: Processor identification; $0 \le s < p$.
- p: Number of processors; p is a power of 2 with $p < \sqrt{N}$.
- $sign = -1$: Transform direction is backward.
- N: Transform size; N is a power of 2 with $N \ge 2$.
- $\mathbf{z} = (\tilde{x}_0 + i\tilde{y}_0, \ldots, \tilde{x}_{N-1} + i\tilde{y}_{N-1})$: Two real vectors of size N packed as complex (block distributed).

OUTPUT $\mathbf{z} \leftarrow (x_0 + iy_0, \ldots, x_{N-1} + iy_{N-1})$, where $x_j = \frac{1}{N}\sum_{k=0}^{N-1}\epsilon_k\tilde{x}_k\cos(\pi(j+\frac{1}{2})k/N)$, and $y_j = \frac{1}{N}\sum_{k=0}^{N-1}\epsilon_k\tilde{y}_k\cos(\pi(j+\frac{1}{2})k/N)$.

DESCRIPTION

```
$1^{\text{CpCm}}$  Complex permutation from block to zig-zag cyclic distribution.
      BSP_BlockToZig($s, p, 1, \frac{N}{p}$, complex, $\mathbf{z}$)
$2^{\text{Comp}}$  Backward FCT2 pack phase.
      FCT2_PACK($\frac{s}{p}, \frac{N}{p}, \mathbf{z}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}$)
$3^{\text{Comp}}$  Long distance butterflies.
      TBTFLY_ZIG($\frac{s}{p}, \frac{N}{p}, 4\frac{N}{p^2}, \mathbf{z}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}$)
$4^{\text{CpCm}}$  Complex permutation from zig-zag cyclic to block distribution.
      BSP_BlockToZig($s, p, -1, \frac{N}{p}$, complex, $\mathbf{z}$)
$5^{\text{Comp}}$  Short distance butterflies.
      TBTFLY_ZIG($0, \frac{N}{p}, 4, \mathbf{z}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}$)
$6^{\text{Comp}}$  Normalization of the CFFT.
      for $j = s\frac{N}{p}$ to $(s+1)\frac{N}{p}-1$ do
          $z_j \leftarrow \frac{1}{N}\cdot z_j$
$7^{\text{Comm}}$  Parallel $\varrho^{-1}$ permutation.
      BSP_Rho($s, p, \frac{N}{p}, \mathbf{z}$)
```

Algorithm 3.13 Template for the sequential pack phase of the backward FCT2.

CALL FCT2_PACK($\alpha, n, \mathbf{z}$).

ARGUMENTS

α: Parameter used to compute the correct weights; $0 \leq \alpha < 1$.
n: Vector size; n is a power of 2 with $n \geq 2$.
$\mathbf{z} = (\tilde{z}_0, \ldots, \tilde{z}_{n-1})$: Complex vector of size n.

OUTPUT $\mathbf{z} \leftarrow (Z_0, \ldots, Z_{n-1})$, computed using (3.38).

DESCRIPTION

```
if $\alpha = 0$ then
    $z_{n/2} \leftarrow 2 \cdot \cos(\frac{\pi}{4}) \cdot z_{n/2}$
    $k_0 \leftarrow 1$
else $k_0 \leftarrow 0$
for $k' = k_0$ to $n/2 - 1$ do
    if $k'$ is even or $\alpha = 0$ then $\alpha_1 \leftarrow \alpha$ else $\alpha_1 \leftarrow 1 - \alpha$
    $sum \leftarrow z_{k'} + iz_{n-k'-\lceil\alpha\rceil}$
    $diff \leftarrow z_{k'} - iz_{n-k'-\lceil\alpha\rceil}$
    $z_{k'} \leftarrow \exp(-\frac{\pi i(k'+\alpha_1)}{2n}) \cdot sum$
    $z_{n-k'-\lceil\alpha\rceil} \leftarrow \exp(\frac{\pi i(k'+\alpha_1)}{2n}) \cdot diff$
```

where α_1 is defined by (3.13). (Here we used $\tilde{z}_n = \tilde{x}_n + i\tilde{y}_n = 0$ if $\alpha = 0$.) With the help of Lemmas 3.4 and 3.5 it is possible to prove that (3.38) corresponds to the local part of the pack operation (3.32), provided that $\alpha = s/p$ and $n = N/p$. Subroutine FCT2_PACK has cost

$$C_{\text{FCT2\_PACK}}(n) = 8n, \tag{3.39}$$

provided that the sines and cosines used are stored in a table.

The transform phase is executed using a transposed CFFT algorithm. The extract phase is combined with the bit reversal giving the permutation ϱ^{-1} which is the same as ϱ. The total cost of the inverse FCT2 is the same as for the forward algorithm.

3.5. *Results and discussion*

In this section, we present results concerning the performance of our implementations of the RFFT, FCT, and FCT2 algorithms. Our implementations follow the same conventions as described in Section 2.5, and were tested on the same machine (i.e., a Cray T3E with up to 64 processors, with double precision (64-bit) accuracy of 1.0×10^{-15}). Accuracy results are also given using the more commonly used IEEE 754 floating point arithmetic for which the double precision accuracy is 2.2×10^{-16}. The execution times represent the time to perform the forward transform followed by the (normalized) backward transform divided by two.

3.5.1. Accuracy. We tested the overall accuracy of our implementations by measuring the relative error (2.46) obtained when transforming a random real vector $\mathbf{f}$ with values uniformly distributed between 0 and 1. In the case of the inverse RFFT, the same input vector is used, but it is viewed as a conjugate even vector packed using (3.2). In the case of the FCT2, two different random vectors are transformed, but we present only the error in the first one, which is the same vector as the one transformed by the FCT.

Table 3.1. Relative errors for the sequential forward RFFT, forward FCT, and forward FCT2 implementations using a Cray T3E.

N	Forward RFFT	Forward FCT	Forward FCT2
512	2.3×10^{-16}	1.2×10^{-15}	1.2×10^{-15}
1024	5.5×10^{-16}	3.0×10^{-15}	2.9×10^{-15}
2048	8.5×10^{-16}	4.1×10^{-15}	4.1×10^{-15}
4096	2.2×10^{-15}	9.0×10^{-15}	8.7×10^{-15}
8192	3.2×10^{-15}	3.0×10^{-14}	3.0×10^{-14}
16384	6.8×10^{-15}	4.8×10^{-14}	4.6×10^{-14}
32768	2.3×10^{-14}	6.8×10^{-14}	6.8×10^{-14}
65536	3.6×10^{-14}	1.6×10^{-13}	1.5×10^{-13}

Table 3.2. Relative errors for the sequential forward RFFT, forward FCT, and forward FCT2 implementations using IEEE 754 floating point arithmetic.

N	Forward RFFT	Forward FCT	Forward FCT2
512	1.5×10^{-16}	1.9×10^{-16}	1.9×10^{-16}
1024	1.8×10^{-16}	3.1×10^{-16}	3.2×10^{-16}
2048	1.8×10^{-16}	2.3×10^{-16}	2.3×10^{-16}
4096	1.8×10^{-16}	2.6×10^{-16}	2.6×10^{-16}
8192	2.0×10^{-16}	2.6×10^{-16}	2.6×10^{-16}
16384	2.1×10^{-16}	2.7×10^{-16}	2.7×10^{-16}
32768	2.2×10^{-16}	3.0×10^{-16}	3.0×10^{-16}
65536	2.2×10^{-16}	3.0×10^{-16}	3.1×10^{-16}

Tables 3.1 and 3.2 show the relative errors of the sequential implementations for various problem sizes. Only the errors corresponding to the forward transforms are shown. The errors corresponding to the backward transforms and to the parallel (forward and backward) transforms are of the same order. Note that the errors

TABLE 3.3. Timing results (in ms) obtained for the sequential RFFT and the sequential CFFT on the Cray T3E.

N	Time RFFT	Time CFFT/2
32	0.011	0.012
64	0.027	0.017
128	0.039	0.047
256	0.106	0.074
512	0.173	0.204
1024	0.454	0.330
2048	0.756	0.908
4096	2.017	1.471
8192	3.428	9.975
16384	21.947	29.457
32768	63.308	74.895
65536	159.740	159.140
131072	338.350	370.990

corresponding to the sequential and parallel implementations of a transform of a certain size N are the same, as long as the butterfly stages are paired in the same way. The results show that all the FFT and FFT-like algorithms have about the same accuracy. Furthermore, the IEEE 754 arithmetic is clearly superior.

3.5.2. Performance of the real FFT. We examine the performance of our sequential RFFT implementation by comparing it with our sequential CFFT implementation (see Section 2.5). Table 3.3 confirms the theoretical prediction that the execution time of an RFFT should be about half the execution time of a CFFT of the same length. It is also noticeable that the RFFT performs better when the problem size N is an odd power of two. In that case, the RFFT is computed using a CFFT of size $N/2$, where all butterfly stages are paired. This is also the situation where the CFFT of size N performs worse, since its last butterfly stage is not paired.

The timing results obtained by our parallel RFFT implementation are summarized in Table 3.4. As in the sequential case, the execution times are roughly half the execution times of the corresponding parallel CFFT (cf. Table 2.3), and the RFFT implementation performs better when all the butterfly stages are paired. Note that the execution times of the parallel RFFT implementation for one processor are larger than those of the sequential implementation. This happens because, in the parallel implementation, the permutations from block to zig-zag cyclic distribution and vice

TABLE 3.4. Timing results (in ms) obtained for the parallel RFFT on a Cray T3E.

N	*seq*	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$
512	0.17	0.19	0.26	0.16	0.18	−−	−−	−−
1024	0.45	0.44	0.45	0.29	0.23	0.24	−−	−−
2048	0.76	0.84	0.91	0.50	0.35	0.28	−−	−−
4096	2.02	2.05	1.78	1.02	0.60	0.41	0.42	−−
8192	3.43	4.83	4.06	2.00	1.22	0.61	0.53	−−
16384	21.95	25.23	9.88	4.66	2.45	1.27	0.81	0.81
32768	63.31	72.03	37.51	11.86	5.74	2.66	1.50	1.07
65536	159.74	173.99	97.63	44.16	15.32	6.80	3.43	1.92
131072	338.35	376.66	227.13	111.80	52.53	15.61	7.93	3.72

versa, which correspond to the identity permutation when $p = 1$, are not treated as special cases, causing a certain overhead.

Comparing the cost functions of the parallel RFFT algorithm (3.17) and of the parallel CFFT algorithm (2.38) we notice that the computation cost and the communication cost of the RFFT are approximately half those of the CFFT, but the synchronization costs of both algorithms are the same. From this, we would expect smaller speedups for the RFFT implementation than for the CFFT implementation, in particular for small N, which can be verified by comparing Figure 2.4 with Figure 3.5.

The superlinear speedups in Figure 3.5 indicate that the cache effect plays an important role in the scalability results. In the previous chapter, we used FFT flop rates as a way to filter out the cache effect. Since FFT flop rates are specific to one algorithm, we do not use them in this chapter, but we use absolute efficiencies instead. The absolute efficiency measure behaves in a similar way as the measure of flop rate per processor. As with the flop rate per processor, the ideal efficiency is a constant (in this case, equal to one). Theoretically, the efficiency is a number between zero and one. In practice, the cache effect (or other causes) can lead to efficiencies larger than one.

Figure 3.6 shows efficiency data for the RFFT. For $N \leq 8192$, the problem fits completely in the cache. Since there is no cache effect, the efficiency is always less than one and decreases with p. For $p \leq 8$, an efficiency of approximately 50% is achieved, indicating reasonable scalability. For $N \geq 65536$, the cache effect can clearly be identified by the sudden increase in the efficiency values. After identifying the critical number of processors for the cache effect we can analyze the efficiency below and above the critical value. Below this point, the efficiency is nearly constant

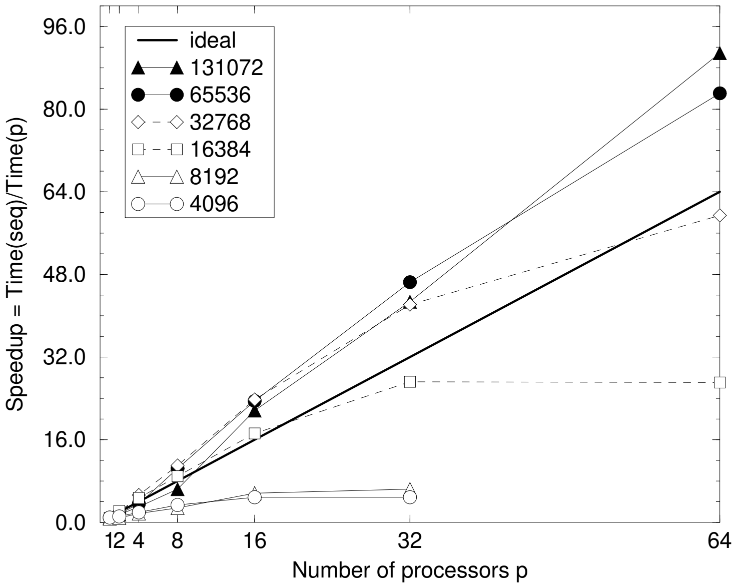


FIGURE 3.5. Scalability of the RFFT on a Cray T3E measured as speedup.

at a value of about 80% indicating a very good scalability. Well above this point, the efficiency also remains nearly constant, again suggesting a very good scalability. The dashed lines indicate intermediate cases where the efficiency initially remains constant, indicating a very good scalability, but then starts dropping for larger p.

3.5.3. Performance of the FCT and FCT2. The timing results obtained by our FCT and FCT2 implementations are summarized in Tables 3.5 and 3.6, respectively. The execution times given for the FCT2 implementation are normalized, i.e., divided by two, so that they can directly be compared with the execution times of the FCT implementation. For convenience, we also express the difference between the two transforms as a percentage, $(\text{Time}_{\text{FCT}}(N) - \text{Time}_{\text{FCT2}}(N))/\text{Time}_{\text{FCT}}(N) \times 100$. The results are presented as Table 3.7. Negative values indicate that the FCT2 implementation performs worse than the FCT implementation.

From Table 3.7 it is clear that in general the FCT2 performs better than the FCT, with two exceptions. The first exception occurs when N and p are such that the FCT fits totally (or almost totally) into the cache memory, while the FCT2 does

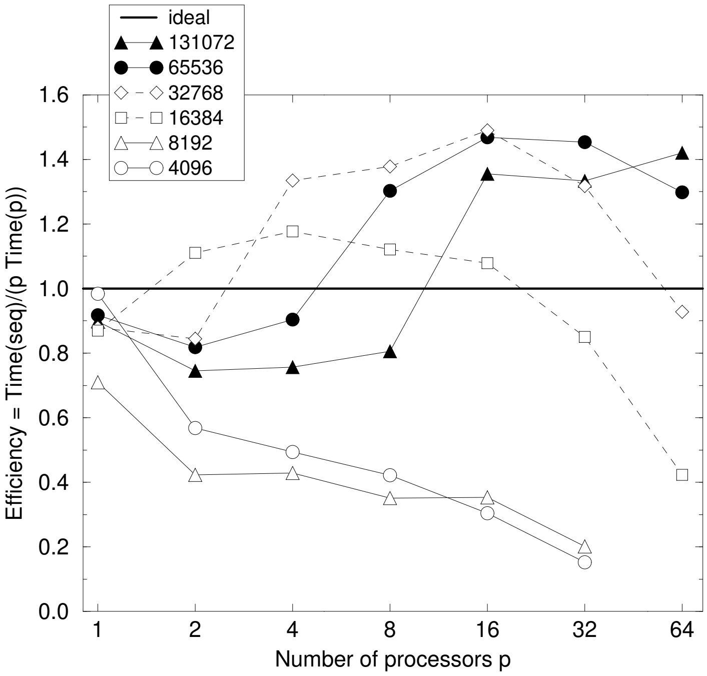


FIGURE 3.6. Scalability of the FFT on a Cray T3E measured as efficiency.

not. This means that the FCT implementation already benefits from the cache effect, but the FCT2 does not (these cases are marked by the small boxes of the table). The second exception occurs in the sequential case, and in the parallel case when $p = 1$. In these cases, the FCT implementation tends to be better than the FCT2 when N is an odd power of two. This happens because all the butterflies of the FCT algorithm are paired, but the last butterfly of the FCT2 algorithm is not, which means that the real computational cost of the FCT2 algorithm is larger by $3N/4$ (or $3N/8$ if the cost function is normalized), than the cost (3.37), derived for the ideal, all-paired situation. The reverse is valid when N is an even power of two. In that case the real computational cost of the FCT algorithm is larger than the cost (3.23) by $3/8N$.

In the truly parallel case, with $p > 1$, the theoretical difference between the cost of the FCT algorithm (3.23) and half the cost of the FCT2 algorithm (3.37) is:

$$T_{\mathrm{FCT}}(N,p) - \frac{1}{2}T_{\mathrm{FCT2}}(N,p) = \begin{cases} \frac{1}{2}\frac{N}{p} + \frac{3}{2}l, & \text{if } \frac{N}{p} \text{ is an even power of two,} \\ -\frac{1}{4}\frac{N}{p} + \frac{3}{2}l, & \text{otherwise.} \end{cases}$$

TABLE 3.5. Execution times (in ms) for the FCT on a Cray T3E. The line indicates the threshold between in-cache computations and out-of-cache computations.

N	*seq*	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$
512	0.20	0.23	0.29	0.19	0.24	−−	−−	−−
1024	0.48	0.54	0.49	0.33	0.28	0.31	−−	−−
2048	0.91	1.07	1.01	0.56	0.43	0.42	−−	−−
4096	2.28	2.71	2.03	1.19	0.71	0.56	0.60	−−
8192	6.63	8.99	5.24	2.54	1.55	0.85	0.82	−−
16384	28.53	39.34	14.39	6.97	3.69	1.96	1.30	1.35
32768	77.59	104.49	50.88	16.81	8.69	4.05	2.41	1.87
65536	183.54	237.29	129.96	57.77	19.38	8.91	4.81	3.01
131072	393.52	500.65	287.75	145.39	64.59	19.21	10.08	5.16

TABLE 3.6. Normalized execution times (in ms) for the FCT2 on a Cray T3E.

N	*seq*	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$
512	0.21	0.23	0.22	0.14	0.12	0.15	−−	−−
1024	0.40	0.44	0.45	0.25	0.18	0.15	−−	−−
2048	0.99	1.11	0.89	0.51	0.30	0.22	0.23	−−
4096	1.88	3.05	2.09	1.01	0.64	0.32	0.30	−−
8192	11.42	14.60	5.60	2.50	1.34	0.67	0.44	0.48
16384	33.61	40.03	19.74	6.51	3.40	1.53	0.88	0.64
32768	82.82	95.29	50.64	22.67	7.96	3.62	1.89	1.11
65536	180.71	204.93	115.52	56.21	26.17	7.86	4.08	1.98
131072	406.39	454.20	244.40	126.32	67.62	26.19	8.63	4.13

TABLE 3.7. Comparison between the FCT2 and the FCT implementations. The entries (in percentage) indicate how much faster (positive values) or slower (negative values) the FCT2 is compared to the FCT.

N	*seq*	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$
512	−5	2	23	27	52	−−	−−	−−
1024	16	19	8	26	34	51	−−	−−
2048	−8	−4	11	9	30	48	−−	−−
4096	18	−13	−3	16	10	42	50	−−
8192	−72	−62	−7	1	14	21	46	−−
16384	−18	−2	−37	7	8	22	32	53
32768	−7	9	0	−35	8	10	21	41
65536	2	14	11	3	−35	12	15	34
131072	−3	9	15	13	−5	−36	14	20

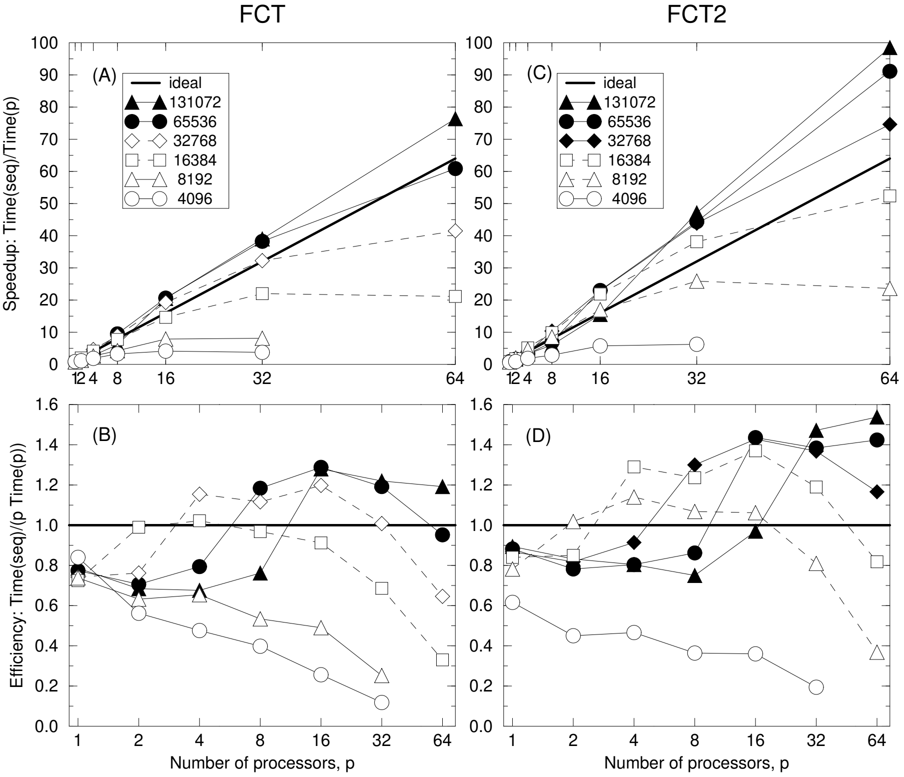


FIGURE 3.7. Scalability of the FCT and FCT2 on a Cray T3E.

Note that we added $\frac{1}{2}\frac{N}{p}$ to (3.23) to account for the normalization of the inverse transform. Since l is often larger than N/p the FCT2 algorithm is, in most cases, cheaper than the FCT algorithm. Furthermore, the FCT2 algorithm should scale better than the FCT algorithm. Figure 3.7 presents the absolute speedups and absolute efficiencies obtained for the FCT and FCT2 implementations. As with the RFFT and CFFT implementations, large problem sizes scale very well with an increase in the, both before and after the transition from out-of-cache computations to in-cache computations. Intermediate problem sizes (dashed lines) scale very well before the transition but poorly for larger p. Small problem sizes scale relatively well for small p ($p \leq 8, 16$).

As an alternative to the FCT2 algorithm we could use a hybrid FCT2 algorithm which would use the FCT2 algorithm when N/p is an even power of two and the FCT algorithm when N/p is an odd power of two. The FCT algorithm would have to be modified to handle the two transforms at the same time, so that no extra synchronizations would be needed.

3.6. *Conclusions*

In this chapter we showed how to construct efficient CFFT-based parallel algorithms for the discrete Fourier transform of a real vector and for the discrete cosine transform. By introducing the zig-zag cyclic distribution we were able to develop parallel algorithms for these transforms that are optimal in the sense that no extra communication (besides the communication already needed by the CFFT phase) is needed. The zig-zag cyclic distribution can also be used to develop CFFT-based parallel algorithms for other discrete transforms, such as the discrete sine transform (see e.g. [**55**]), or to parallelize a similar problem that needs to combine elements which are at a fixed distance (e.g., j and $j + K$) and, also, mirror image pairs of elements such as j and $N - j$.

Though we only presented algorithms that work for $p < \sqrt{N/2}$ (or $p < \sqrt{N}$ for the FCT2), it is easy to extend them to any $p < N$ that is a power of two, by computing the medium distance butterfly stages in the same way as we did in the CFFT algorithm of Chapter 2. However, the resulting algorithms are not always optimal as they do need extra communication when $p = (n/p)^{H-1}$, where $H = \lceil \log_{n/p} n \rceil$, and $n = N/2$ for the RFFT and for the FCT, or $n = N$ for the FCT2.

To develop efficient inverse algorithms, we replaced the CFFT algorithm used in the forward transform by its transposed version. In this way no extra communication is needed.

We introduced the FCT2 algorithm which computes the DCT of two vectors simultaneously. This algorithm is simpler and easier to parallelize than the corresponding FCT algorithm. This was particularly useful in the development of an efficient parallel FLT algorithm (see Chapter 4). Though the performance of the sequential FCT2 algorithm is practically the same as the sequential FCT2 algorithm, the parallel FCT2 algorithm is in most cases superior to the parallel FCT algorithm.

We presented results concerning the accuracy, efficiency, and scalability of our implementations. In general, the implementations of all three algorithms and their inverses perform very well on the Cray T3E with up to 64 processors. Due to the cache effect, absolute speedups of up to $1.5p$ were observed. To filter out the cache effect, we also examined the scalability in terms of absolute efficiency. We determined that problems of size $N \geq 32768$ (or $N \geq 16384$ for the FCT2) scale very well, and we also determined that smaller problems with size $N \leq 8192$ (or $N \leq 4096$ for the FCT2) scale reasonably well up to 8 processors.

4

Fast Legendre Transform

4.1. *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 [**6, 14**]), shape analysis of molecular surfaces [**24**], statistical analysis of directional data [**28**], and geometric quality assurance [**27**].

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 [**3, 39**] 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 [**21, 22**]. 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; at least two reports discussing the theoretical parallelizability of the algorithm have already been written [**30, 45**]. We are, however, unaware of any parallel implementation of the Driscoll-Healy algorithm at the time of writing.

In this chapter, we derive a new parallel algorithm that has a lower theoretical time complexity than those of [**30, 45**], 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 orthogonal polynomials used by the algorithm, and to remove some unnecessary assumptions made in [**21**] and [**22**].

The remainder of this chapter is organized as follows. In Section 4.2, we describe some important properties of orthogonal polynomials and orthogonal polynomial transforms, and present a derivation of the Driscoll-Healy algorithm. In Section 4.3, we describe a basic parallel algorithm and its implementation. In Section 4.4, we refine the basic algorithm by introducing an intermediate data distribution that reduces the communication to a minimum. In Section 4.5, we present results on the accuracy, efficiency, and scalability of our implementation. We conclude with Section 4.6. Appendices C and D describe a generalization of the algorithm and the precomputation of the data needed by the algorithm.

This chapter is a result of joint work with David K. Maslen and Rob H. Bisseling. A preliminary version was published in [**33**], and the full paper has been submitted to SIAM Journal of Scientific Computing.

4.2. *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.

4.2.1. Orthogonal polynomials. A sequence of polynomials $p_0, p_1, p_2, \dots$ 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

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

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 $x_0, \dots, x_{N-1}$ and $w_0, \dots, w_{N-1}$ called *sample points* and *sample weights*, respectively, we may define the *discrete orthogonal polynomial transform* of a data vector $(f_0, \dots, f_{N-1})$ to be the vector of sums $(\hat{f}_0, \dots, \hat{f}_{N-1})$, where

$$\hat{f}_l = \hat{f}(p_l) = \sum_{j=0}^{N-1} f_j p_l(x_j) w_j. \tag{4.1}$$

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 $(f_0, \dots, f_{N-1})$.

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 (4.1) is equivalent to transform 4 provided the weights w_j are identically 1.

EXAMPLE 4.1 (Legendre polynomials). The Legendre polynomials are orthogonal with respect to the uniform weight function 1 on $[-1,1]$, and may be defined recursively by

$$P_{l+1}(x) = \frac{2l+1}{l+1} x \cdot P_l - \frac{l}{l+1} P_{l-1}, \quad P_0(x) = 1, \quad P_1(x) = x. \tag{4.2}$$

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 work, we call the discrete orthogonal polynomial transform for the Legendre polynomials, with sample weights $1/N$ and with the Chebyshev points as sample points, the *discrete Legendre transform* (DLT).

EXAMPLE 4.2 (Discrete cosine transform and Chebyshev transform). The Chebyshev polynomials of the first kind are the sequence of orthogonal polynomials defined recursively by

$$T_{k+1}(x) = 2x \cdot T_k(x) - T_{k-1}(x), \quad T_0(x) = 1, \quad T_1(x) = x. \tag{4.3}$$

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

The DCT (discrete cosine transform, cf. (3.3)) of size N is the discrete orthogonal polynomial transform for the Chebyshev polynomials, with sample weights 1, and sample points

$$x_j^N = \cos\frac{(2j+1)\pi}{2N}, \quad j = 0, \ldots, N-1, \tag{4.4}$$

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, 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 $f_j = f(x_j^N)$, $0 \leq j < N$, is $\epsilon_k \hat{f}(T_k)/N$, where ϵ_k is defined by (3.5). Thus, to compute the Chebyshev transform, we can use a DCT and multiply the k-th coefficient by ϵ_k/N. Therefore, the Chebyshev transform is defined by

$$\tilde{f}_k = \frac{\epsilon_k}{N} \sum_{j=0}^{N-1} f_j T_k(x_j^N) = \frac{\epsilon_k}{N} \sum_{j=0}^{N-1} f_j \cos \frac{(2j+1)k\pi}{2N}, \quad k = 0, \ldots, N-1. \tag{4.5}$$

(In this chapter we use the tilde to denote the Chebyshev transform instead of the DCT.) The inverse Chebyshev transform, numbered 2 above, is

$$f_j = \sum_{k=0}^{N-1} \tilde{f}_k T_k(x_j^N) = \sum_{k=0}^{N-1} \tilde{f}_k \cos \frac{(2j+1)k\pi}{2N}, \quad j = 0, \ldots, N-1. \tag{4.6}$$

Compare the definition of the Chebyshev transform (4.5) and its inverse (4.6) with the definition of the DCT (3.3) and its inverse (3.4). A DCT can be carried out in $O(N \log N)$ flops by using an FCT algorithm (for an FFT-based algorithm see Chapter 3 or [**2, 55**], for a non-FFT-based algorithm see [**47**]). This also provides us with a fast Chebyshev Transform (FChT) algorithm. In the discussion that follows we use a cost function of the form

$$C_{\mathrm{FChT}}(N) = \alpha N \log_2 N + \beta N \tag{4.7}$$

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 4.3 (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 (which are all real). 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. [**48**, Theorem 3.6.12]. □

EXAMPLE 4.4. 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

$$\frac{1}{\pi}\int_{-1}^{1}\frac{f(x)dx}{\sqrt{1-x^2}} = \frac{1}{N}\sum_{j=0}^{N-1} f(x_j^N), \tag{4.8}$$

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 (4.2) for the Legendre polynomials and (4.3) for the Chebyshev polynomials.

LEMMA 4.5 (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*

$$p_{k+1}(x) = (A_k x + B_k)p_k(x) + C_k p_{k-1}(x), \tag{4.9}$$

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

PROOF. See e.g. [**48**, Theorem 3.6.3]. □

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

COROLLARY 4.6 (Clebsch-Gordan). *Let* $\{p_k\}$ *be an orthogonal polynomial sequence for 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 (4.9) 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. [**7, 8**].

$$\begin{aligned}
&Q_{l,m}(x) = (A_{l+m-1}x + B_{l+m-1})Q_{l,m-1}(x) + C_{l+m-1}Q_{l,m-2}(x),\\
&\quad Q_{l,0}(x) = 1, \quad Q_{l,1}(x) = A_l x + B_l,\\
&R_{l,m}(x) = (A_{l+m-1}x + B_{l+m-1})R_{l,m-1}(x) + C_{l+m-1}R_{l,m-2}(x),\\
&\quad R_{l,0}(x) = 0, \quad R_{l,1}(x) = C_l.
\end{aligned} \tag{4.10}$$

LEMMA 4.7 (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* $m \geq 0$,

$$p_{l+m} = Q_{l,m} \cdot p_l + R_{l,m} \cdot p_{l-1}. \tag{4.11}$$

PROOF. Equation (4.11) follows by induction on m, with the case $m = 1$ being the original three-term recurrence (4.9). ☐

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.

4.2.2. Derivation of the Driscoll-Healy algorithm. The Driscoll-Healy algorithm [**21, 22**] 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 $1/N$. For simplicity we restrict ourselves to this special case, and furthermore we assume that N is a power of 2. In Appendix C, 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 (4.1) 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

$$f(x_j^N) = f_j, \quad j = 0, \ldots, N-1. \tag{4.12}$$

Using this notation and the relation

$$f \cdot p_{l+m} = Q_{l,m} \cdot (f \cdot p_l) + R_{l,m} \cdot (f \cdot p_{l-1}), \tag{4.13}$$

derived from the three-term recurrence (4.11), 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 (4.13) with $l = 1$ and $m = N/2 - 1$ or $m = N/2$, to compute
  $f \cdot p_{\frac{N}{2}} = Q_{1,\frac{N}{2}-1} \cdot (f \cdot p_1) + R_{1,\frac{N}{2}-1} \cdot (f \cdot p_0)$ 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 (4.13) with $l = 2q(N/2^k) + 1$, $0 \leq q < 2^{k-1}$, and $m = N/2^k - 1$ or $N/2^k$, to compute the polynomial pairs
  $f \cdot p_{\frac{N}{2^k}},\ f \cdot p_{\frac{N}{2^k}+1};\ f \cdot p_{\frac{3N}{2^k}},\ f \cdot p_{\frac{3N}{2^k}+1};\ \cdots\ ; f \cdot p_{\frac{(2^k-1)N}{2^k}}, f \cdot p_{\frac{(2^k-1)N}{2^k}+1}$.

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

$$\mathcal{T}_n f = \sum_{k=0}^{n-1} b_k T_k. \tag{4.14}$$

The important properties of $\mathcal{T}_n$ are given in Lemma 4.8.

LEMMA 4.8. *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 4.6, $T_k \cdot Q$ 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 4.8, 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 4.9. *Let f be the unique polynomial of degree less than N such that $f(x_j^N) = f_j$, $0 \le j < N$. Let $\{p_k\}$ be an orthogonal polynomial sequence. Then*

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

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

Proof. This follows from the definition of discrete orthogonal polynomial transforms, the Gaussian quadrature rule (4.8) for Chebyshev polynomials applied to the function $f \cdot p_l$, and Lemma 4.8,

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

□

The key property of the truncation operators $\mathcal{T}_n$ is the 'aliasing' property (Part 3 of Lemma 4.8), 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 4.8 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

$$Z_l^K = \mathcal{T}_K(f \cdot p_l) \tag{4.15}$$

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 (4.13) directly, each stage uses truncated versions. Specifically, (4.13) and Part 3 of Lemma 4.8 imply the following recurrences for the Z_l^K:

$$Z_{l+m-1}^{K-m} = \mathcal{T}_{K-m}[Z_l^K \cdot Q_{l,m-1} + Z_{l-1}^K \cdot R_{l,m-1}], \tag{4.16}$$

$$Z_{l+m}^{K-m} = \mathcal{T}_{K-m}[Z_l^K \cdot Q_{l,m} + Z_{l-1}^K \cdot R_{l,m}], \tag{4.17}$$

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

$$Z_{l+K-1}^{K} = \mathcal{T}_{K}[Z_l^{2K} \cdot Q_{l,K-1} + Z_{l-1}^{2K} \cdot R_{l,K-1}], \tag{4.18}$$

$$Z_{l+K}^{K} = \mathcal{T}_{K}[Z_l^{2K} \cdot Q_{l,K} + Z_{l-1}^{2K} \cdot R_{l,K}]. \tag{4.19}$$

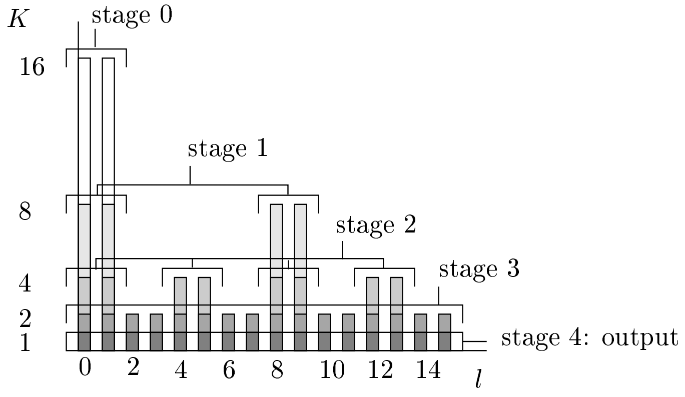


FIGURE 4.1. The computation of Z_l^K for $N = 16$.

The algorithm proceeds in $\log_2 N + 1$ stages, as shown in Algorithm 4.1. The organization of the computation is illustrated in Figure 4.1. The vertical lines represent 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.

Algorithm 4.1 Polynomial version of the Driscoll-Healy algorithm.

```
INPUT  $(f_0, \ldots, f_{N-1})$: Polynomial defined by $f_j = f(x_j^N)$; $N$ is a power of 2.
OUTPUT  $(\hat{f}_0, \ldots, \hat{f}_{N-1})$: Transformed polynomial with $\hat{f}_l = \mathcal{T}_1(f \cdot p_l) = Z_l^1$.
STAGES
    0. Compute $Z_0^N \leftarrow f \cdot p_0$ and $Z_1^N \leftarrow \mathcal{T}_N(f \cdot p_1)$.
    $k$. for $k = 1$ to $\log_2 N - 1$ do
            $K \leftarrow \frac{N}{2^k}$
            for $l = 1$ to $N - 2K + 1$ step $2K$ do
              (a) Use recurrence (4.18) and (4.19) to compute new polynomials.
                  $Z_{l+K-1}^K \leftarrow \mathcal{T}_K \left( Z_l^{2K} \cdot Q_{l,K-1} + Z_{l-1}^{2K} \cdot R_{l,K-1} \right)$
                  $Z_{l+K}^K \leftarrow \mathcal{T}_K \left( Z_l^{2K} \cdot Q_{l,K} + Z_{l-1}^{2K} \cdot R_{l,K} \right)$
              (b) Truncate old polynomials.
                  $Z_{l-1}^K \leftarrow \mathcal{T}_K Z_{l-1}^{2K}$
                  $Z_l^K \leftarrow \mathcal{T}_K Z_l^{2K}$
 $\log_2 N$. for $l = 0$ to $N - 1$ do
            $\hat{f}_l \leftarrow Z_l^1$
```

4.2.3. 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.

4.2.3.1. *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

$$Z_l^K = \sum_{n=0}^{K-1} z_n^l T_n, \tag{4.20}$$

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

The input polynomial f of degree less than N is given as the vector $\mathbf{f} = (f_0, \ldots, f_{N-1})$ of values $f_j = f(x_j^N)$. This is called the *point value representation* of f. In stage 0, we convert $Z_0^N = \mathcal{T}_N(f \cdot p_0) = f \cdot p_0$ and $Z_1^N = \mathcal{T}_N(f \cdot p_1)$ to their Chebyshev 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 4.10 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.

4.2.3.2. *Recurrence using Chebyshev transforms.* To apply the recurrences (4.18) and (4.19) efficiently, we do the following.

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

This procedure replaces the polynomial multiplications in the recurrences (4.18) and (4.19) by a slightly different operation. Because the multiplications are made in only $2K$ points whereas the degree of the resulting polynomial could be $3K-1$, we must verify that the end result is the same. To describe the operation formally, we introduce the *Lagrange interpolation operators* $\mathcal{S}_K$, 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 $x_0^K, \dots, x_{K-1}^K$. The important properties of $\mathcal{S}_K$ are given in Lemma 4.10.

LEMMA 4.10. *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* $K \geq m$. *If* $\deg h \leq K + m$, *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 $\deg h \leq K + m$. 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 4.8, 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 (4.18) and (4.19) and Part 3 of Lemma 4.10 with $2K$ instead of K and $m = K$ it follows that

$$Z_{l+K-1}^K = \mathcal{T}_K[\mathcal{S}_{2K}(Z_l^{2K} \cdot Q_{l,K-1}) + \mathcal{S}_{2K}(Z_{l-1}^{2K} \cdot R_{l,K-1})], \tag{4.21}$$

$$Z_{l+K}^K = \mathcal{T}_K[\mathcal{S}_{2K}(Z_l^{2K} \cdot Q_{l,K}) + \mathcal{S}_{2K}(Z_{l-1}^{2K} \cdot R_{l,K})]. \tag{4.22}$$

These equations are exactly the procedure described above. The inner loop of stage k of Algorithm 4.1 becomes:

```
(a) Compute the Chebyshev representation of $Z^K_{l+K-1}$ and $Z^K_{l+K}$.
      $(z_0^{l+K-1}, \dots, z_{K-1}^{l+K-1};\ z_0^{l+K}, \dots, z_{K-1}^{l+K})$
            $\leftarrow \mathrm{Recurrence}_l^K(z_0^{l-1}, \dots, z_{2K-1}^{l-1};\ z_0^{l}, \dots, z_{2K-1}^{l})$
(b) Compute the Chebyshev representation of $Z^K_{l-1}$ and $Z^K_l$.
      Discard $(z_K^{l-1}, \dots, z_{2K-1}^{l-1})$ and $(z_K^{l}, \dots, z_{2K-1}^{l})$.
```

Algorithm 4.2 describes in detail the recurrence procedure, which costs

$$\begin{aligned} C_{\mathrm{Recurrence}}(K) &= 4(\alpha \cdot 2K \log_2 2K + \beta \cdot 2K) + 12K \\ &= 8\alpha K \log_2 K + (8\alpha + 8\beta + 12)K. \end{aligned} \tag{4.23}$$

Algorithm 4.2 Recurrence procedure using the Chebyshev transform.

```
CALL Recurrence$_l^K(\tilde f_0, \ldots, \tilde f_{2K-1};\ \tilde g_0, \ldots, \tilde g_{2K-1})$.
INPUT $\tilde{\mathbf{f}} = (\tilde f_0, \ldots, \tilde f_{2K-1})$ and $\tilde{\mathbf{g}} = (\tilde g_0, \ldots, \tilde g_{2K-1})$: First $2K$ Chebyshev coefficients of
    input polynomials $Z_{l-1}^{2K}$ and $Z_l^{2K}$; $K$ is a power of 2.
OUTPUT $\tilde{\mathbf{u}} = (\tilde u_0, \ldots, \tilde u_{K-1})$ and $\tilde{\mathbf{v}} = (\tilde v_0, \ldots, \tilde v_{K-1})$: First $K$ Chebyshev coefficients of
    output polynomials $Z_{l+K-1}^K$ and $Z_{l+K}^K$.
STEPS
    1. Transform $\tilde{\mathbf{f}}$ and $\tilde{\mathbf{g}}$ to point-value representation.
       $(f_0, \ldots, f_{2K-1}) \leftarrow \mathrm{Chebyshev}^{-1}(\tilde f_0, \ldots, \tilde f_{2K-1})$
       $(g_0, \ldots, g_{2K-1}) \leftarrow \mathrm{Chebyshev}^{-1}(\tilde g_0, \ldots, \tilde g_{2K-1})$
    2. Perform the recurrence.
       for $j = 0$ to $2K - 1$ do
           $u_j \leftarrow Q_{l,K-1}(x_j^{2K})\, g_j + R_{l,K-1}(x_j^{2K})\, f_j$
           $v_j \leftarrow Q_{l,K}(x_j^{2K})\, g_j + R_{l,K}(x_j^{2K})\, f_j$
    3. Transform $\mathbf{u}$ and $\mathbf{v}$ to Chebyshev representation.
       $(\tilde u_0, \ldots, \tilde u_{2K-1}) \leftarrow \mathrm{Chebyshev}(u_0, \ldots, u_{2K-1})$
       $(\tilde v_0, \ldots, \tilde v_{2K-1}) \leftarrow \mathrm{Chebyshev}(v_0, \ldots, v_{2K-1})$
    4. Discard $(\tilde u_K, \ldots, \tilde u_{2K-1})$ and $(\tilde v_K, \ldots, \tilde v_{2K-1})$.
```

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

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

$$Q_{l,m} = \sum_{n=0}^{m} q_{l,m}^n T_n, \quad R_{l,m} = \sum_{n=0}^{m-1} r_{l,m}^n T_n. \tag{4.24}$$

The problem of finishing the computation at the end of stage $k = \log_2(N/M)$, when $K = M$, is equivalent to finding $\hat f_l = z_0^l$, for $0 \le l < N$, given the data z_n^l, z_n^{l-1}, $0 \le n < M$, $l = 1, M+1, 2M+1, \ldots, N-M+1$. Our method of finishing the computation is to use Part 1 of Lemma 4.11, which follows. Part 2 of this lemma can be used to halve the number of computations in the common case where the polynomial recurrence (4.9) has a coefficient $B_k = 0$ for all k.

LEMMA 4.11. 1. *If $l \ge 1$ and $0 \le m < M$, then*

$$\hat f_{l+m} = \sum_{n=0}^{m} \frac{1}{\epsilon_n}(z_n^l q_{l,m}^n + z_n^{l-1} r_{l,m}^n). \tag{4.25}$$

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*

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

Proof. By (4.17) 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 (4.20) and (4.24), and rewrite the product of sums by using the identity $T_j \cdot T_k = \frac{1}{2}(T_{|j-k|} + T_{j+k})$. For the second part, we assume that p_l satisfies the given recurrence. Then $Q_{l,m}$ is odd or even according to whether m is odd or even, and $R_{l,m}$ is even or odd according to whether m is odd or even, which can be verified by induction on m. This implies that the Chebyshev expansion of $Q_{l,m}$ must contain only odd or even coefficients, respectively, and the reverse must hold for $R_{l,m}$. ☐

Assuming that the assumptions of the Part 2 of the lemma are valid, i.e., each term of (4.25) 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 needed to compute $\hat{f}_{l+m}$ is $2m+1$.

4.2.5. Complexity of the algorithm. Algorithm 4.3 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 $C_{\text{Recurrence}}(K)$, cf. (4.23), so that the total cost of that stage is

$$4\alpha N \log_2 K + (4\alpha + 4\beta + 6)N \text{ flops.}$$

Adding the costs for $K = N/2, \dots, M$ gives

$$2\alpha N[\log_2^2 N - \log_2^2 M] + (2\alpha + 4\beta + 6)N[\log_2 N - \log_2 M] \text{ flops.}$$

In the last stage, output values have to be computed for $m = 1, \dots, M-2$, for each of the N/M values of l. This gives a total of

$$\frac{N}{M} \sum_{m=1}^{M-2} (2m+1) = NM - 2N \text{ flops.}$$

Summing the costs gives

$$\begin{aligned} C_{\text{Driscoll-Healy}}(N) = N[&2\alpha(\log_2^2 N - \log_2^2 M) + (4\alpha + 4\beta + 6)\log_2 N \\ &- (2\alpha + 4\beta + 6)\log_2 M + M + 2\beta]. \end{aligned} \tag{4.26}$$

Algorithm 4.3 Driscoll-Healy algorithm.

```
INPUT  $\mathbf{f} = (f_0, \ldots, f_{N-1})$: Real vector with $N$ a power of 2.
OUTPUT  $\hat{\mathbf{f}} = (\hat{f}_0, \ldots, \hat{f}_{N-1})$: Discrete orthogonal polynomial transform of $\mathbf{f}$.
STAGES
          0. Compute the Chebyshev representation of $Z_0^N$ and $Z_1^N$.
             (a) $(z_0^0, \ldots, z_{N-1}^0) \leftarrow \text{Chebyshev}(f_0 p_0, \ldots, f_{N-1} p_0)$.
             (b) $(z_0^1, \ldots, z_{N-1}^1) \leftarrow \text{Chebyshev}(f_0 p_1(x_0^N), \ldots, f_{N-1} p_1(x_{N-1}^N))$.
          k. for $k = 1$ to $\log_2 \frac{N}{M}$ do
                 $K \leftarrow \frac{N}{2^k}$
                 for $l = 1$ to $N - 2K + 1$ step $2K$ do
                   (a) Compute the Chebyshev representation of $Z_{l+K-1}^K$ and $Z_{l+K}^K$.
                         $(z_0^{l+K-1}, \ldots, z_{K-1}^{l+K-1};\ z_0^{l+K}, \ldots, z_{K-1}^{l+K})$
                             $\leftarrow \text{Recurrence}_l^K(z_0^{l-1}, \ldots, z_{2K-1}^{l-1};\ z_0^l, \ldots, z_{2K-1}^l)$
                   (b) Compute the Chebyshev representation of $Z_{l-1}^K$ and $Z_l^K$.
                         Discard $(z_K^{l-1}, \ldots, z_{2K-1}^{l-1})$ and $(z_K^l, \ldots, z_{2K-1}^l)$.
  $\log_2 \frac{N}{M} + 1$. Compute the remaining values.
             for $l = 1$ to $N - M + 1$ step $M$ do
                 $\hat{f}_{l-1} \leftarrow z_0^{l-1}$
                 $\hat{f}_l \leftarrow z_0^l$
                 for $m = 1$ to $M - 2$ do
                     $\hat{f}_{l+m} \leftarrow z_0^l q_{l,m}^0 + z_0^{l-1} r_{l,m}^0 + \frac{1}{2} \sum_{n=1}^{m} (z_n^l q_{l,m}^n + z_n^{l-1} r_{l,m}^n)$
```

The optimal stage at which to halt the Driscoll-Healy algorithm and complete the computation using Lemma 4.11 depends on α and β and can be obtained theoretically. The derivative of (4.26) as a function of M equals zero if and only if

$$M \ln^2 2 - 4\alpha \ln M = (2\alpha + 4\beta + 6) \ln 2. \tag{4.27}$$

In our implementation $\alpha = 2.125$ and $\beta = 5$, cf. (3.37), thus the minimum is $M = 64$. In practice, the optimal choice of M may also depend on the architecture of the machine used.

4.3. *Basic parallel algorithm and its implementation*

In the following subsections, 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.

4.3.1. Data structures and data distributions. At each stage k, with $1 \leq k \leq \log_2(N/M)$, 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

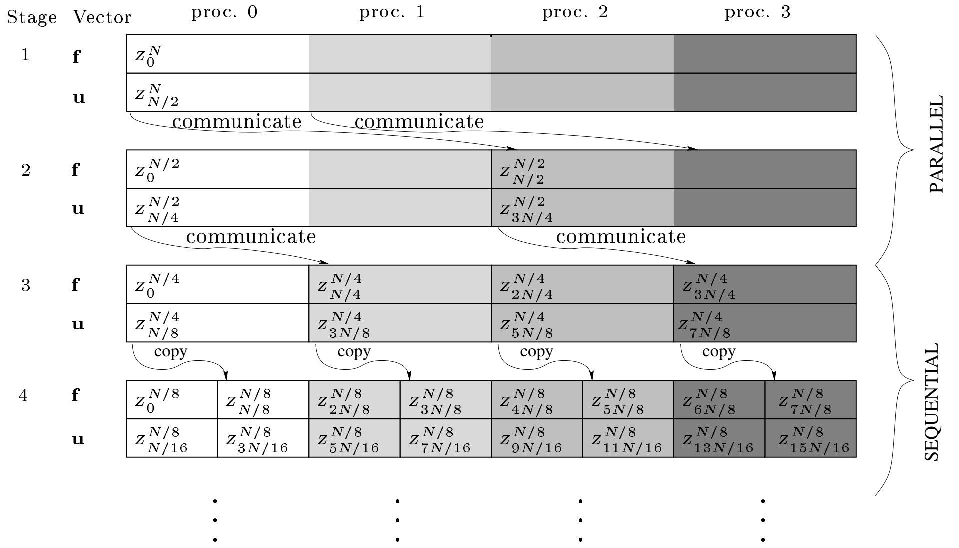


FIGURE 4.2. Main data structure and data distribution in the parallel FLT algorithm for $p = 4$. Arrays **f** and **g** contain the Chebyshev coefficients of the polynomials Z_l^{2K} and Z_{l+1}^{2K}, which are already available at the start of the stage. Arrays **u** and **v** contain Z_{l+K}^{2K} and Z_{l+K+1}^{2K}, which become available at the end of the stage. Arrays **g** and **v** are not depicted. Each array is divided into four local subarrays by using the block distribution.

array **f** to store the Chebyshev coefficients of the polynomials Z_l^{2K} and an array **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, **u** and **v**, of size N.

The data flow of the algorithm, see Figure 4.2, suggests that we distribute all the vectors by the block distribution. This works well if p is a power of two, which we will assume from now on. Since both N and p are powers of two, the block size $b = N/p$ is also a power of two.

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

$$\begin{aligned} \mathbf{Q}[2k-2, l+j] &= Q_{l+1,K-1}(x_j^{2K}), \\ \mathbf{Q}[2k-1, l+j] &= Q_{l+1,K}(x_j^{2K}), \end{aligned} \tag{4.28}$$

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

FIGURE 4.3. 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.

	$m = 1$	$m = 2$	$m = 3$	$m = 4$	$m = 5$	$m = 6$	
$l = 1$	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 4.4. 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 $r^n_{l,m}$.

for $l = 0, 2K, \ldots, N-2K$, $j = 0, 1, \ldots, 2K-1$, where $K = N/2^k$. Thus, polynomials $Q_{l+1,K-1}$ are stored in row $2k-2$ and polynomials $Q_{l+1,K}$ in row $2k-1$. This is shown in Figure 4.3. The polynomials $R_{l+1,K-1}$ and $R_{l+1,K}$ are stored in the same way in array $\mathbf{R}$. Note that the indexing of the implementation arrays starts at zero. Each row of $\mathbf{Q}$ and $\mathbf{R}$ is distributed by the block distribution, i.e., $\mathbf{Q}[i,j], \mathbf{R}[i,j] \in \mathrm{Proc}(j) = j \ \mathrm{div}\ \frac{N}{p}$, so that the recurrence is a local operation.

The termination 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 $\mathbf{T}$ 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 Part 2 of Lemma 4.11, either $q^n_{l,m} = 0$ or $r^n_{l,m} = 0$ 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 $q^n_{l,m}$'s and $r^n_{l,m}$'s. Figure 4.4 illustrates this data structure.

The termination stage is local if $M \leq N/p$. 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. Each column of $\mathbf{T}$ is in the block distribution, i.e., $\mathbf{T}[i,j] \in \mathrm{Proc}(i) = i \text{ div } \frac{N}{pM}$. As a result, the N/M rows of $\mathbf{T}$ are distributed in consecutive blocks of rows.

4.3.2. Basic parallel template. In order to formulate our basic parallel algorithm we introduce the following conventions:

- **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

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

  replaces the input vector $\mathbf{f}$ of size n by its Chebyshev transform if $sign = 1$ or by its inverse Chebyshev transform if $sign = -1$. A group of p_1 processors starting from processor s_0 work together; s_1 with $0 \leq s_1 < p_1$ denotes the local processor number within the group. For a group of size $p_1 = 1$, this subroutine reduces to the sequential fast Chebyshev transform algorithm. Since the Chebyshev transforms come in pairs, Algorithms 3.9 and 3.12 can be used in the implementation of the BSP_FChT2 algorithm. The cost of the resulting algorithm is

  $$C_{\mathrm{FChT2,par}}(N,p) = 2\alpha\frac{N}{p}\log_2 N + 2\beta\frac{N}{p} + 6\frac{N}{p}\cdot g + 3l, \tag{4.29}$$

  where $\alpha = 2.125$ and $\beta = 5$, cf. (3.37).

- **Truncation.** The operation

$$\mathbf{f} \leftarrow \text{BSP\_Trunc}(s, p, N, p_1, K, \mathbf{u})$$

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 p_1 processors starting from processor $s_0 = s - s_1$ work together to truncate one polynomial; $s_1 = s \bmod p_1$ denotes the local processor number within the group. When $p_1 = 1$ the block size $N/p \geq 2K$ and one processor is in charge of the truncation of one or more complete polynomials. In Figure 4.2, the truncation operation is depicted by arrows. Algorithm 4.4 describes subroutine BSP_Trunc2 which carries out two truncation operations simultaneously. In doing so, we save one synchronization. The cost of Algorithm 4.4 is

$$C_{\text{Trunc2,par}}(N, p) = 2\frac{N}{p}g + l, \tag{4.30}$$

if $p_1 > 1$. If $p_1 = 1$, the truncation is a simple copy operation which costs nothing.

Algorithm 4.4 Truncation procedure for the FLT.

CALL $(\mathbf{f}, \mathbf{g}) \leftarrow \text{BSP\_Trunc2}(s, p, N, p_1, K, \mathbf{u}, \mathbf{v})$.

ARGUMENTS

s: Processor identification; $0 \leq s < p$.
p: Total number of processors, p is a power of 2 with $p < N$.
N: Vector size; N is a power of 2.
p_1: Number of processors in group.
K: Subvector size; K is a power of 2 with $K < N$.
$\mathbf{u} = (u_0, \ldots, u_{N-1})$: Real vector of size N (block distributed).
$\mathbf{v} = (v_0, \ldots, v_{N-1})$: Real vector of size N (block distributed).

OUTPUT $(\mathbf{f}, \mathbf{g})$.

DESCRIPTION

```
          if $p_1 = 1$ then
1^comp        Sequential truncation.
              for $l = s\frac{N}{p}$ to $(s+1)\frac{N}{p} - 2K$ step $2K$ do
                  $(\mathbf{f}_{l+K}, \mathbf{g}_{l+K}) \leftarrow \text{Copy2}(K, \mathbf{u}_l, \mathbf{v}_l)$
          else
2^comm        Parallel truncation.
              if $s \bmod p_1 < \frac{p_1}{2}$ then
                  $(\mathbf{f}_{s\frac{N}{p}+K}, \mathbf{g}_{s\frac{N}{p}+K}) \leftarrow \text{Put2}(s + \frac{p_1}{2}, \frac{N}{p}, \mathbf{u}_{s\frac{N}{p}}, \mathbf{v}_{s\frac{N}{p}})$
              Synchronize
```

The basic template for the fast Legendre transform is presented as Algorithm 4.5. At each stage $k \leq \log_2(N/M)$, there are 2^{k-1} independent problems. For $k \leq \log_2 p$, there are more processors than problems, so that the processors will have to work in groups. Each group of $p_1 = p/2^{k-1} > 1$ processors handles one subvector of size $2K$, $K = N/2^k$; each processor handles a block of $2K/p_1 = N/p$ vector components. In this case, the l-loop has only one iteration, namely $l = s_0 \cdot N/p$, and the j-loop has N/p iterations, starting with $j = s_1 \cdot N/p$, so that the indices $l + j$ start with $(s_0 + s_1)N/p = s \cdot N/p$, and end with $(s_0 + s_1)N/p + N/p - 1 = (s + 1)N/p - 1$. Inter-processor communication is needed, but it occurs only in two instances:

- Inside the parallel FChTs (in supersteps 2, 5, 7), see Chapter 3.
- At the end of each stage (in supersteps 3, 8).

For $k > \log_2 p$, the length of the subvectors involved becomes $2K \leq N/p$. In that case, $p_1 = 1$, $s_0 = s$, and $s_1 = 0$, and each processor has one or more 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 4.3 are combined into one stage and then performed efficiently, as follows. First, in superstep 1, the polynomials Z_1^N, $Z_{N/2}^N$, and $Z_{N/2+1}^N$ are computed directly from the input vector $\mathbf{f}$. This is possible because the point-value representation of $Z_1^N = \mathcal{T}_N(f \cdot P_1) = \mathcal{T}_N(f \cdot x)$ needed by the recurrences is the vector of $f_j \cdot x_j^N, 0 \leq j < N$, see Subsection 4.2.3.1. In superstep 2, polynomials $Z_0^N = \mathbf{f}, Z_1^N = \mathbf{g}, Z_{N/2}^N = \mathbf{u}$, and $Z_{N/2+1}^N = \mathbf{v}$ are transformed to Chebyshev representation; then, in superstep 3, they are truncated to obtain the input for stage 2.

The main loop works as follows. In superstep 4, the polynomials Z_l^{2K}, with $K = N/2^k$ and $l = 0, 2K, \ldots, N - 2K$, are copied from the array $\mathbf{f}$ into the auxiliary array $\mathbf{u}$, where they are transformed into the polynomials Z_{l+K}^{2K}, in supersteps 5 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 same happens to polynomials Z_{l+K+1}^{2K} and Z_{l+1}^{2K}.

The termination procedure, superstep 9, is described separately as Algorithm 4.6. The template is a direct implementation of Lemma 4.11 using the data structure $\mathbf{T}$ described in Subsection 4.3.1. Note that it is convenient to regard each row l/M in the array $\mathbf{T}$ as a triangular matrix, and process the elements of each matrix using 2×2

Algorithm 4.5 Basic template for the parallel 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 $p < N$.
N: Transform size; N is a power of 2 with $N \ge 4$.
M: Termination block size; M is a power of 2 with $M \le \min(N/2, N/p)$.
$\mathbf{f} = (f_0, \ldots, f_{N-1})$: Real vector of size N (block distributed).

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

DESCRIPTION

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

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

3$^{\mathtt{CpCm}}$   Stage 1: Truncation.
        $(\mathbf{f}, \mathbf{g}) \leftarrow$ BSP_Trunc2$(s, p, N, p, \frac{N}{2}, \mathbf{u}, \mathbf{v})$

        for $k = 2$ to $\log_2 \frac{N}{M}$ do
            $K \leftarrow \frac{N}{2^k}$
            $p_1 \leftarrow \max(\frac{p}{2^{k-1}}, 1)$
            $s_0 \leftarrow (s \text{ div } p_1) p_1$
            $s_1 \leftarrow s \bmod p_1$
4$^{\mathtt{Comp}}$           Stage $k$: Copy.
            $(\mathbf{u}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}, \mathbf{v}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}) \leftarrow \mathrm{Copy2}(\frac{N}{p}, \mathbf{f}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}, \mathbf{g}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}})$

            for $l = s_0\frac{N}{p}$ to $(s_0 + 1)\frac{N}{p} - \frac{2K}{p_1}$ step $\frac{2K}{p_1}$ do
5$^{\mathtt{CpCm}}$               Stage $k$: Inverse Chebyshev transform.
                BSP_FChT2$(s_0, s_1, p_1, -1, 2K, \mathbf{u_l}, \mathbf{v_l})$

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

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

8$^{\mathtt{CpCm}}$           Stage $k$: Truncation.
            $(\mathbf{f}, \mathbf{g}) \leftarrow$ BSP_Trunc2$(s, p, N, p_1, K, \mathbf{u}, \mathbf{v})$

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

Algorithm 4.6 Termination procedure for the FLT.

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

DESCRIPTION

```
$h_0 \leftarrow f_0$
$h_1 \leftarrow g_0$
$b \leftarrow 0$
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
        $h_{m+1} \leftarrow h_{m+1} + \frac{1}{2}\left(f_n \cdot \mathbf{T}[l, b+n] + g_{n+1} \cdot \mathbf{T}[l, b+n+1]\right)$
        $h_{m+2} \leftarrow h_{m+2} + \frac{1}{2}\left(g_n \cdot \mathbf{T}[l, b+n+m+1] + f_{n+1} \cdot \mathbf{T}[l, b+n+m+2]\right)$
    $h_{m+2} \leftarrow h_{m+2} + \frac{1}{2} \cdot g_{m+1} \cdot \mathbf{T}[l, b+2m+2]$
    $b \leftarrow b + 2m + 3$
```

Note 1: The index b indicates where coefficient $r_{l+1,m}^0$ is stored.
Note 2: The factors 1/2 can be stored in the precomputed table.

blocks. This leads to a step size of two in the loops of the algorithm. Superstep 9 is a computation superstep, provided the condition $M \le N/p$ 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 Section 4.4.3.

The cost of the basic parallel FLT algorithm can be computed as follows. Each parallel stage k, $2 \le k \le \log_2 p$, of the algorithm costs

$$C_{\text{FLTstage,par}}(N, p, k) = 6 \cdot \frac{N}{p} + 2 \cdot C_{\text{FChT2,par}}\left(\frac{N}{2^{k-1}}, \frac{p}{2^{k-1}}\right) + C_{\text{Trunc2,par}}(N, p). \tag{4.31}$$

Note that the first term corresponds to the cost of the recurrence (superstep 6). Adding the cost of all parallel stages of the basic FLT algorithm, including the first, gives

$$\frac{N}{p} + \sum_{k=1}^{\log_2 p} C_{\text{FLTstage,par}}(N, p, k) =$$
$$[2\alpha(2\log_2 N - \log_2 p + 1) + 4\beta + 6]\frac{N}{p}\log_2 p + \frac{N}{p} + 14\frac{N}{p}\log_2 p \cdot g + 7\log_2 p \cdot l,$$

cf. 4.29. Here we had to add an extra N/p flops corresponding to the extra computations of the initialization (superstep 1). The cost of the remaining sequential stages, $k > \log_2 p$, is equal to the cost of a sequential FLT of size N/p, which is given by $C_{\text{Driscoll-Healy}}(N/p)$, cf. (4.26), but without its stage 0. The total cost of the basic FLT algorithm is

$$\begin{aligned}
C_{\text{FLT,par,basic}}(N, M, p) &= \frac{N}{p}[2\alpha(\log_2^2 N - \log_2^2 M) \\
&\quad + (2\alpha + 4\beta + 6)(\log_2 N - \log_2 M)] + \frac{NM}{p} - \frac{N}{p} \\
&\quad + 14\frac{N}{p}\log_2 p \cdot g + 7\log_2 p \cdot l \\
&\approx 4.25\frac{N}{p}(\log_2^2 N - \log_2^2 M) + 30.25\frac{N}{p}(\log_2 N - \log_2 M) + \frac{NM}{p} \\
&\quad + 14\frac{N}{p}\log_2 p \cdot g + 7\log_2 p \cdot l.
\end{aligned} \tag{4.32}$$

4.4. *Improvements of the parallel algorithm*

In this section we show how to transform the basic parallel FLT algorithm into an optimized parallel FLT algorithm. The optimizations carried out reduce the communication and synchronization costs of the algorithm to about one third.

The optimized algorithm needs two auxiliary complex vectors, both of size N. Vector $\mathbf{h}$ stores the data previously stored in vectors $\mathbf{f}$ and $\mathbf{g}$: $\mathbf{h} = \mathbf{f} + i\mathbf{g}$. Vector $\mathbf{w}$ stores the data previously stored in vectors $\mathbf{u}$ and $\mathbf{v}$: $\mathbf{w} = \mathbf{u} + i\mathbf{v}$. Besides simplifying the notation, this convention also makes it easier to construct the template for the optimized FLT algorithm by using the subroutines developed in the previous chapters.

4.4.1. Optimization of the main loop. The basic FLT algorithm is a modular algorithm based on the block distribution. Though useful for didactic purposes, the block distribution is not the best choice for the main loop of the FLT algorithm.

Breaking open the FChT module inside the main loop and merging it with the surrounding computations allows us to radically reduce the amount of communication involved in the parallel FLT algorithm.

4.4.1.1. *Moving permutations to precomputation.* The recurrence (superstep 6 of Algorithm 4.5) is an element by element operation and thus it can be performed in any data distribution as long as the recurrence coefficients are also permuted to the desired distribution. Because of this, it is possible to completely skip the permutations ϱ, cf. (3.34), which immediately precede and follow the recurrence, saving $4\frac{N}{p}g + 2l$ in communication costs for each parallel stage of the main loop.

To carry out this modification, arrays $\mathbf{Q}$ and $\mathbf{R}$ used in the template for the basic FLT algorithm must be replaced by the permuted versions $\mathbf{Q}'$ and $\mathbf{R}'$ which are defined as follows. The first pair of rows is not permuted. The coefficients necessary for the initial stage $k = 1$ can be stored in $\mathbf{Q}'$ by

$$\begin{aligned}\mathbf{Q}'[0,j] &= Q_{1,\frac{N}{2}-1}(x_j^N) \cdot x_j^N + R_{1,\frac{N}{2}-1}(x_j^N),\\ \mathbf{Q}'[1,j] &= Q_{1,\frac{N}{2}}(x_j^N) \cdot x_j^N + R_{1,\frac{N}{2}}(x_j^N), \quad 0 \le j < N.\end{aligned}$$

The first two rows of $\mathbf{R}'$ can remain empty. Each remaining pair of rows in $\mathbf{Q}'$ stores the data needed for one stage $k \ge 2$, by

$$\begin{aligned}\mathbf{Q}'[2k-2, l+j] &= Q_{l+1,K-1}(x^{2K}_{\varrho_{2K}(j)}),\\ \mathbf{Q}'[2k-1, l+j] &= Q_{l+1,K}(x^{2K}_{\varrho_{2K}(j)}),\end{aligned}$$

for $l = 0, 2K, \dots, N-2K$, $j = 0, 1, \dots, 2K-1$, where $K = N/2^k$. The polynomials $R_{l+1,K-1}$ and $R_{l+1,K}$ are stored in the same way in array $\mathbf{R}'$.

4.4.1.2. *Modifying the truncation operation.* As with the recurrence, the truncation operation does not need to be done in the block distribution. This means that we can skip the permutation back to block distribution at the end of superstep 7 of Algorithm 4.5, and proceed directly to the truncation, which now becomes merely freeing of storage.

As a result of the new truncation, at the end of stage $k-1$, with $2 \le k \le \log_2(N/M)$, the complex vectors $\mathbf{h}$ and $\mathbf{w}$ stores the Chebyshev coefficients of polynomials Z_l^{2K}, Z_{l+1}^{2K}, with $K = N/2^k$ and $l = 0, 2K, \dots, N-2K$ in the following way. Polynomials Z_l^{2K}, Z_{l+1}^{2K}, with $l = 0, 4K, \dots, N-4K$, are stored as the real and imaginary parts of the first half of subvector $\mathbf{h}_\mathbf{l}$ of size $4K$, and polynomials Z_{l+2K}^{2K}, Z_{l+2K+1}^{2K} are stored as the real and imaginary parts of the first half of subvector $\mathbf{w}_\mathbf{l}$ which is also of size $4K$. The second half of both vectors stores junk data. If $k \le \log_2 2p$, vectors $\mathbf{h}_\mathbf{l}$ and $\mathbf{w}_\mathbf{l}$ are both zig-zag cyclically distributed over a group of

$2p_1$ processors, where $p_1 = p/2^{k-1}$, with processor offset $s_0 = l \operatorname{div} \frac{N}{p}$. In other words, vectors $\mathbf{h}$ and $\mathbf{w}$ are *zig-zag cyclically distributed in* $r = p/(2p_1) = 2^{k-2}$ *groups.*

DEFINITION 4.12 (Zig-zag cyclic distribution in r groups, $\mathrm{Z}^r(p,N)$). Let r, p, and N be integers with $1 \le r \le p \le N$, such that r divides p and N. Let $\mathbf{h}$ be a vector of size N to be distributed over p processors organized in r groups. Define $M = N/r$ to be the size of the subvector of a group and $u = p/r$ to be the number of processors in a group. We say that $\mathbf{h}$ is *zig-zag cyclically distributed in* r *groups* over p processors if, for all j, the element h_j has local index $j' = (j \bmod M) \operatorname{div} u$, and is stored in processor $s_0 + s_1$, where $s_0 = (j \operatorname{div} M) \cdot u$ is the number of the first processor in the group (i.e., the processor offset) and s_1 is the processor identification within the group, which is defined by $s_1 = (j \bmod M) \bmod u$ if $j \bmod M$ is even, or by $s_1 = -(j \bmod M) \bmod u$ otherwise.

As with the r-cyclic distribution, the zig-zag cyclic distribution in r groups forms a family of distributions that contains the block distribution and the zig-zag cyclic distribution as extremes: $\mathrm{Z}^p(p,N) = \mathrm{B}(p,N)$ and $\mathrm{Z}^1(p,N) = \mathrm{Z}(p,N)$. Figure 4.5 illustrates the use of this family of distributions in the optimized FLT algorithm.

At the beginning of stage k, the input data (i.e., the Chebyshev coefficients of polynomials Z_l^{2K}, Z_{l+1}^{2K}, with $l = 0, 2K, \ldots, N-2K$) which are stored in vectors $\mathbf{h}$ and $\mathbf{w}$ must be rearranged so that the polynomials Z_l^K, Z_{l+1}^K, Z_{l+K}^K, and Z_{l+K+1}^K can be computed efficiently. The rearrange procedure consists of two phases which are illustrated in Figure 4.6 for the simple case of stage 2. Phase 1 is a local operation that prepares the polynomials Z_l^{2K}, Z_{l+1}^{2K}, with $l = 0, 2K, \ldots, N-2K$, for use in the recurrence operation. This is done in two steps: (a) copy the first half of subvector $\mathbf{w}_\mathbf{l}$ of size $4K$, with $l = 0, 4K, \ldots, N-4K$, into its second half; (b) copy the first half of subvector $\mathbf{h}_\mathbf{l}$ of size $4K$ into the first half of subvector $\mathbf{w}_\mathbf{l}$. After Phase 1, vector $\mathbf{w}$ contains all the data needed by the recurrence procedure, which will compute polynomials Z_{l+K}^K, Z_{l+K+1}^K, with $l = 0, 2K, \ldots, N-2K$.

Phase 2 computes polynomials Z_l^K, Z_{l+1}^K, with $l = 0, 2K, \ldots, N-2K$, simply by truncating Z_l^{2K}, Z_{l+1}^{2K}. Since, by now, vector $\mathbf{w}$ contains all the input data, Phase 2 consists of redistributing a partial copy of vector $\mathbf{w}$ from distribution $\mathrm{Z}^{r_2}(p,N)$, with $r_2 = \min(p, 2^{k-2})$, to distribution $\mathrm{Z}^{r_1}(p,N)$, with $r_1 = \min(p, 2^{k-1})$, and storing it in vector $\mathbf{h}$. If $k \le \log_2(2p)$, then $r_2 = r_1/2$ and this phase consists of a global permutation. Because polynomials Z_l^{2K}, Z_{l+1}^{2K} are being truncated, half the data contained in vector $\mathbf{w}$ does not need to be redistributed, which means that this permutation will only cost $\frac{N}{p}g + l$ (instead of $2\frac{N}{p}g + l$). If $k > \log_2(2p)$, then $r_2 = r_1 = p$ and the permutation reduces to a local copy operation which costs nothing.

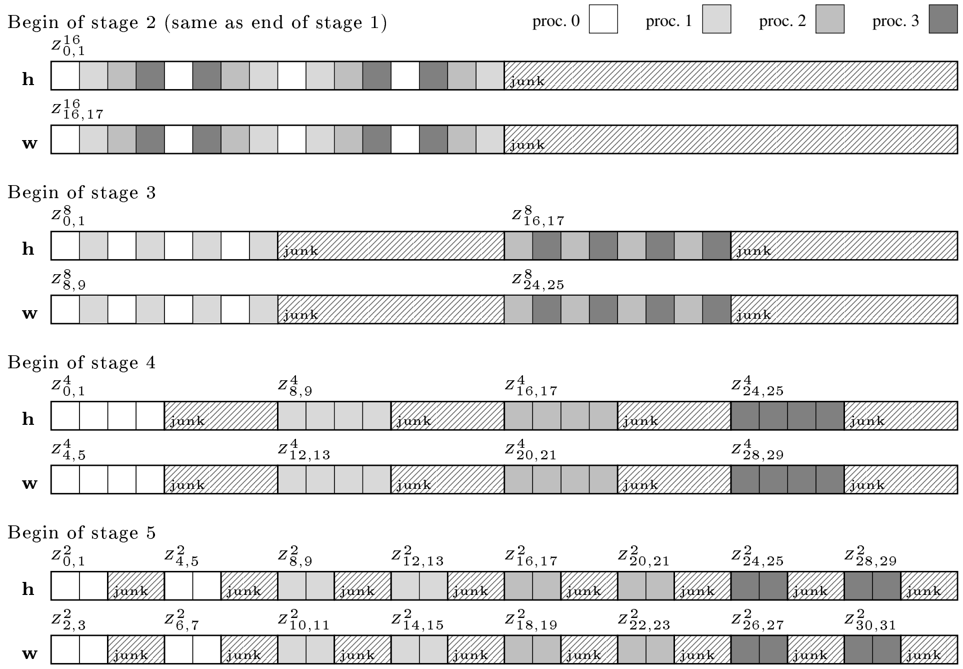


FIGURE 4.5. Use of the zig-zag cyclic distribution in r groups in the optimized FLT algorithm (logical view). Example with $p = 4$, $N = 32$, and $M = 2$. At the beginning of stage k, complex array $\mathbf{h}$ contains the Chebyshev coefficients of the polynomials Z_l^{2K}, Z_{l+1}^{2K}, with $K = N/2^k$ and $l = 0, 4K, \ldots, N - 4K$, packed as complex; and complex array $\mathbf{w}$ contains the Chebyshev coefficients of the polynomials Z_{l+2K}^{2K}, Z_{l+2K+1}^{2K} packed as complex. Both $\mathbf{h}$ and $\mathbf{w}$ are zig-zag cyclically distributed in $r = \min(p, 2^{k-2})$ groups. In the figure, $Z_l^{2K} + iZ_{l+1}^{2K}$ is depicted as $Z_{l,l+1}^{2K}$.

Subroutine BSP_Rearrange (Algorithm 4.7) is a template for the parallel part of the rearrange procedure, which is illustrated in Figure 4.6. Note that the block distribution is used throughout the template, which means that the changes of distribution are replaced by permutations. The template consists of three supersteps. Superstep 1 performs Phase 1 of the new procedure. Phase 2 is carried out by supersteps 2 and 3. Superstep 2 sends packets of data, and superstep 3 locally rearranges the data. This subroutine replaces the permutation to block distribution at the end of superstep 7, the truncation operation (superstep 8), the copy operation (superstep 4), and the permutation to zig-zag cyclic distribution at the beginning of superstep 5 of the basic

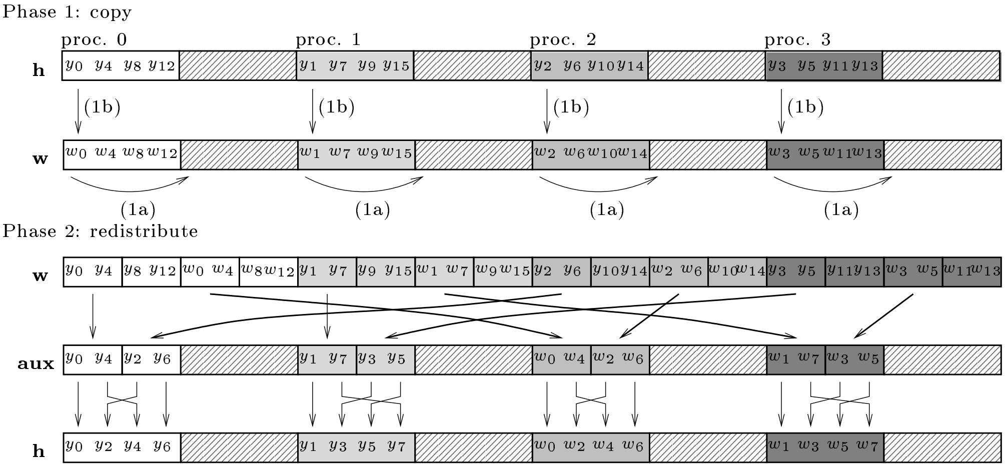


FIGURE 4.6. Rearrange operation (storage view). Example with $N = 32, p = 4, k = 2$, thus $K = 8$, $p_1 = 2$, and $l = 0$. At the beginning, subvectors $\mathbf{h_l}$ and $\mathbf{w_l}$ of size $4K$ are zig-zag cyclically distributed over $2p_1$ processors. The first half of subvector $\mathbf{h_l}$ contains polynomials Z_l^{2K}, Z_{l+1}^{2K}, and the first half of subvector $\mathbf{w_l}$ contains polynomials Z_{l+2K}^{2K}, Z_{l+2K+1}^{2K}. At the end, subvector $\mathbf{w_l}$ remains in distribution $\mathrm{Z}(2p_1, 4K)$, but now its first half contains polynomials Z_l^{2K}, Z_{l+1}^{2K} and its second half contains polynomials Z_{l+2K}^{2K}, Z_{l+2K+1}^{2K}. Subvector $\mathbf{h_l}$ is now redistributed by $\mathrm{Z}^2(2p_1, 4K)$. Its first quarter contains polynomials Z_l^K, Z_{l+1}^K, and its third quarter contains polynomials Z_{l+2K}^K, Z_{l+2K+1}^K. (The second and fourth quarters of the vector contain junk data). The thick arrows correspond to the puts in the algorithm.

algorithm, saving another $5\frac{N}{p}g + 2l$ flops in communication costs for each parallel stage of the main loop. The cost of the rearrange operation is

$$C_{\text{Rearrange,par}}(N, p, k) = \begin{cases} \frac{N}{2p} \cdot g + l, & \text{if } k = \log_2(2p), \\ \frac{N}{p} \cdot g + l, & \text{otherwise.} \end{cases} \tag{4.33}$$

4.4.1.3. *Modifying the Chebyshev transform.* After the rearrange operation, in stages $k \leq \log_2(2p)$, each subvector $\mathbf{w_l}$ of size $4K$, with $l = 0, 4K, \ldots, N - 4K$, which is distributed by $\mathrm{Z}(2p_1, 4K)$, with $p_1 = p/2^{k-1}$, contains two distinct data sets. Polynomials Z_l^{2K}, Z_{l+1}^{2K} are stored in its first half, and polynomials Z_{l+2K}^{2K}, Z_{l+2K+1}^{2K} are stored in its second half (cf. Figure 4.6). Because of this, the inverse

Algorithm 4.7 Template for the optimized redistribution operation of the FLT.

CALL BSP_Rearrange$(s, p, k, N, \mathbf{h}, \mathbf{w})$.

ARGUMENTS

s: Processor identification; $0 \le s < p$.
p: Number of processors; p is a power of 2 with $2 \le p < N$.
k: FLT stage; $2 \le k \le \log_2(2p)$.
N: Vector size; N is a power of 2 with $N \ge 4$.
$\mathbf{h} = (h_0, \ldots, h_{N-1})$: Complex vector of size N (block distributed).
$\mathbf{w} = (w_0, \ldots, w_{N-1})$: Complex vector of size N (block distributed).

OUTPUT

$\mathbf{w}$: data needed at stage k.
$\mathbf{h}$: half the data needed at stage $k+1$.

DESCRIPTION

```
$1^{\mathsf{Comp}}$  Keep the data needed at stage $k$.
      $w_{s\frac{N}{p}+\frac{N}{2p}} \leftarrow \mathrm{Copy}(\frac{N}{2p}, w_{s\frac{N}{p}})$
      $w_{s\frac{N}{p}} \leftarrow \mathrm{Copy}(\frac{N}{2p}, h_{s\frac{N}{p}})$
$2^{\mathsf{Comm}}$  Redistribute the data needed at stage $k+1$: send packets.
      $p_1 \leftarrow \frac{p}{2^{k-1}}$
      $s_1 \leftarrow s \bmod (2p_1)$
      $s_0 \leftarrow s - s_1$
      **if** $s_1 < p_1$ **then**
          $aux_{s\frac{N}{p}} \leftarrow \mathrm{Copy}(\frac{N}{4p}, w_{s\frac{N}{p}})$
          $aux_{(s+p_1)\frac{N}{p}} \leftarrow \mathrm{Put}(s+p_1, \frac{N}{4p}, w_{s\frac{N}{p}+\frac{N}{2p}})$
      **else**
          **if** $s_1 = p_1$ **then** $proc \leftarrow s_0$ **else** $proc \leftarrow s_0 + 2p_1 - s_1$
          $aux_{proc\frac{N}{p}+\frac{N}{4p}} \leftarrow \mathrm{Put}(proc, \frac{N}{4p}, w_{s\frac{N}{p}})$
          $aux_{(proc+p_1)\frac{N}{p}+\frac{N}{4p}} \leftarrow \mathrm{Put}(proc+p_1, \frac{N}{4p}, w_{s\frac{N}{p}+\frac{N}{2p}})$
      Synchronize
$3^{\mathsf{Comp}}$  Redistribute the data needed at stage $k+1$: local rearrange.
      **if** $s_1 = 0$ **or** $s_1 = p_1$ **then**
          **for** $j' = 0$ **to** $\frac{N}{4p} - 1$ **do**
              $h_{s\frac{N}{p}+2j'} \leftarrow aux_{s\frac{N}{p}+j'}$
              $h_{s\frac{N}{p}+2j'+1} \leftarrow aux_{s\frac{N}{p}+j'+\frac{N}{4p}}$
      **else**
          **for** $j' = 0$ **to** $\frac{N}{4p} - 2$ **step** 2 **do**
              $h_{s\frac{N}{p}+2j'} \leftarrow aux_{s\frac{N}{p}+j'}$
              $h_{s\frac{N}{p}+2j'+3} \leftarrow aux_{s\frac{N}{p}+j'+1}$
              $h_{s\frac{N}{p}+2j'+1} \leftarrow aux_{s\frac{N}{p}+j'+\frac{N}{4p}}$
              $h_{s\frac{N}{p}+2j'+2} \leftarrow aux_{s\frac{N}{p}+j'+1+\frac{N}{4p}}$
```

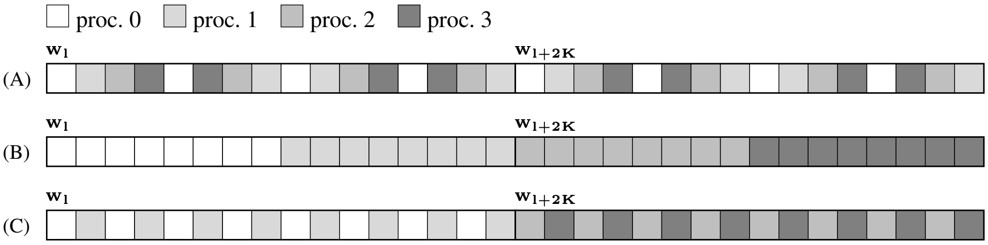


FIGURE 4.7. Data flow of the modified recurrence algorithm (logical view). Example with $2K = 16$ and $p_1 = 2$, for a certain $l \in \{0, 4K, \ldots, N-4K\}$. (A) At the beginning, vectors $\mathbf{w_l}$ and $\mathbf{w_{l+2K}}$ of size $2K$ are zig-zag cyclically distributed over the $2p_1$ processors. (B) After the permutation to block distribution the two vectors are separated. Vector $\mathbf{w_l}$ is block distributed over the first p_1 processors and vector $\mathbf{w_{l+2K}}$ is block distributed over the next p_1 processors. (C) After the recurrence and the forward FChT2, vector $\mathbf{w_l}$ is zig-zag cyclically distributed over the first p_1 processors and vector $\mathbf{w_{l+2K}}$ is zig-zag cyclically distributed over the next p_1 processors.

FChT2 algorithm must be modified so that it can handle the two distinct data sets simultaneously.

With the help of Lemma 3.2 we can prove that the pack phase (3.32) of the inverse FChT2 and the long distance butterfly phase of the inverse CFFT, starting at butterfly size $2K$ down to size $8p_1$, can always be performed locally, given that the input vector is distributed by $\mathrm{Z}(2p_1, 2K)$. For simplicity, we assume from now on that $p < \sqrt{N}$. With this assumption no medium distance butterflies are needed, because the next butterfly stage to be performed has size $4p_1 \leq 2p \leq N/p$.[1]

Algorithm 4.8 is a template for the parallel modified recurrence procedure, which consists of the backward Chebyshev transform, the recurrence itself, and the forward Chebyshev transform. Figure 4.7 illustrates the data distributions used. Since each group of $2p_1$ processors contains two distinct data sets, packed as the first and second half of subvector $\mathbf{w_l}$ (see Figure 4.7A), Algorithm 4.8 regards each input subvector $\mathbf{w_l}$ of size $4K$ as two distinct subvectors of size $2K$: subvectors $\mathbf{w_l}$ and $\mathbf{w_{l+2K}}$, both distributed by $\mathrm{Z}(2p_1, 2K)$, with processor offset $s_0 = l \operatorname{div} \frac{N}{p}$.

---

[1] Contrary to the previous chapter, here we compute as many butterfly stages as possible in the zig-zag cyclic distribution, and then change to the block distribution in order to compute the remaining stages.

Algorithm 4.8 Template for the (parallel) modified recurrence.

CALL BSP_Recurrence2$(s, p, k, N, \mathbf{w})$.

ARGUMENTS

s: Processor identification; $0 \leq s < p$.

p: Number of processors; p is a power of 2 with $2 \leq p < \sqrt{N}$.

k: FLT stage; $2 \leq k \leq \log_2(2p)$.

N: Vector size; N is a power of 2 with $N \geq 4$.

$\mathbf{w} = (w_0, \ldots, w_{N-1})$: Complex vector of size N containing the Chebyshev coefficients of the polynomial pairs Z_l^{2K}, Z_{l+1}^{2K} packed as complex. Here $K = N/2^k$ and $l = 0, 2K, \ldots, N-2K$. The vector has been permuted to distribution $\mathrm{Z}^{2^{k-2}}(p, N)$.

OUTPUT $\mathbf{w}$: Chebyshev coefficients of the polynomial pairs Z_{l+K}^K, Z_{l+K+1}^K packed as complex in subvectors of size K. Each second subvector of size K contains junk data. The vector has been permuted to distribution $\mathrm{Z}^{2^{k-1}}(p, N)$.

DESCRIPTION

```
              $K \leftarrow \frac{N}{2^k}$
              $p_1 \leftarrow \frac{p}{2^{k-1}}$
              $s_1 \leftarrow s \bmod (2p_1)$
$1^{\texttt{Comp}}$    Modified backward FChT2: pack phase and
                                       long distance butterflies (sizes $2K$ down to $8p_1$).
              if $s_1 = 0$ then
                  $w_{s\frac{N}{p}} \leftarrow \frac{1}{2} \cdot w_{s\frac{N}{p}}$
                  $w_{s\frac{N}{p}+\frac{N}{2p}} \leftarrow \frac{1}{2} \cdot w_{s\frac{N}{p}+\frac{N}{2p}}$
              FCT2_PACK2$(\frac{s_1}{2p_1}, \frac{N}{2p}, \mathbf{w}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}, \mathbf{w}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}+\frac{\mathbf{N}}{\mathbf{2p}}})$

              TBTFLY_ZIG2$(\frac{s_1}{2p_1}, \frac{N}{2p}, 4, \mathbf{w}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}}, \mathbf{w}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}+\frac{\mathbf{N}}{\mathbf{2p}}})$
$2^{\texttt{CmCp}}$    Modified backward FChT2: complex permutation to block distribution.
              BSP_BlockToZig$(s - s_1, s_1, 2p_1, -1, \frac{N}{p}, \mathrm{complex}, \mathbf{w})$
$3^{\texttt{Comp}}$    Modified backward FChT2: short distance butterflies (sizes $4p_1$ down to 2).
              TBTFLY_ZIG$(0, 4p_1, 4, \mathbf{w}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}})$
$4^{\texttt{Comp}}$    Recurrence.
              for $j = s\frac{N}{p}$ to $s\frac{N}{p} + \frac{N}{p} - 1$ do
                  $a1 \leftarrow \mathbf{Q}'[2k-2, j] \cdot \mathrm{Im}(w_j) + \mathbf{R}'[2k-2, j] \cdot \mathrm{Re}(w_j)$
                  $a2 \leftarrow \mathbf{Q}'[2k-1, j] \cdot \mathrm{Im}(w_j) + \mathbf{R}'[2k-1, j] \cdot \mathrm{Re}(w_j)$
                  $w_j \leftarrow a1 + ia2$

              $s_1 \leftarrow s \bmod p_1$
$5^{\texttt{CmCp}}$    Modified forward FChT2.
              BSP_FChT2_mod$(s - s_1, s_1, p_1, 1, K, \mathbf{w}_{(\mathbf{s}-\mathbf{s_1})\frac{\mathbf{N}}{\mathbf{p}}})$
```

Note : The call to BSP_BlockToZig was modified so that the permutation can be done in a subgroup of processors; the original template (Algorithm 3.1) must be modified to use $s - s_1 + proc$ instead of $proc$ as the destination processor when communicating.

Algorithm 4.9 Template for the modified parallel forward FChT2.

CALL BSP_FChT2_mod($s_0, s_1, p_1, sign = 1, K, \mathbf{w}$).

ARGUMENTS

s_0, s_1: Processor offset and processor identification within group; $0 \leq s_1 < p_1$.
p_1: Number of processors in the group; p_1 is a power of 2 with $1 \leq p_1 < \sqrt{2K}$.
$sign = 1$: Transform direction is forward.
$2K$: Vector size; K is a power of 2 with $K \geq 2$.
$\mathbf{w} = (w_0, \ldots, w_{2K-1})$: Complex vector of size $2K$ permuted by ϱ_{2K} (block distributed).

OUTPUT $\mathbf{w}$: truncated forward Chebyshev transform of the input vector permuted by $\zeta_{p_1,2K}$.

DESCRIPTION

```
$1^{\mathtt{Comp}}$    Short distance butterflies.
        BTFLY_ZIG($0, \frac{2K}{p_1}, 4, \mathbf{w}_{\mathbf{s_1 \frac{2K}{p_1}}}$)
$2^{\mathtt{CpCm}}$    Complex permutation to zig-zag cyclic distribution.
        BSP_BlockToZig($s_0, s_1, p_1, 1, \frac{2K}{p_1}$, complex, $\mathbf{w}$)
$3^{\mathtt{Comp}}$    Long distance butterflies.
        BTFLY_ZIG($\frac{s_1}{p_1}, \frac{2K}{p_1}, 4\frac{2K}{p_1^2}, \mathbf{w}_{\mathbf{s_1 \frac{2K}{p_1}}}$)
$4^{\mathtt{Comp}}$    Extract phase and normalization.
        if $s_1 = 0$ then
            $j_0 \leftarrow 1$
            $j_1 \leftarrow 0$
            $w_0 \leftarrow \frac{1}{2K} \cdot w_0$
        else
            $j_0 \leftarrow 0$
            $j_1 \leftarrow 1$
        for $j' = j_0$ to $\frac{K}{p_1} - 1$ do
            if $j'$ is even or $s_1 = 0$ then $j \leftarrow j' \cdot p_1 + s_1$ else $j \leftarrow j' \cdot p_1 + p_1 - s_1$
            $sum \leftarrow w_{s_1 \frac{2K}{p_1} + j'} + w_{s_1 \frac{2K}{p_1} + \frac{2K}{p_1} - j' - j_1}$
            $diff \leftarrow w_{s_1 \frac{2K}{p_1} + j'} - w_{s_1 \frac{2K}{p_1} + \frac{2K}{p_1} - j' - j_1}$
            $w_{s_1 \frac{2K}{p_1} + j'} \leftarrow \frac{1}{2K} \cdot [\cos(\frac{\pi j}{4K}) \cdot sum + i \sin(\frac{\pi j}{4K}) \cdot diff]$
```

Because of this, subroutines FCT2_PACK and TBTFLY_ZIG, which carry out the pack phase of the FChT2 and the long distance butterflies of the transposed CFFT, must be called twice, once for each data set. Note that the first element of each vector must be multiplied by two because the normalization constants of the DCT are different than the normalization constants of the Chebyshev transform.

After the long distance butterfly phase, the data sets must be permuted to the block distribution. After this permutation (Figure 4.7B) the two data sets become separated. The first data set is stored by $\mathrm{B}(p_1, 2K)$ in the first group of p_1 processors and the second data set is stored by $\mathrm{B}(p_1, 2K)$ in the second group of p_1 processors, which means that processor offset s_0 and processor identification numbers s_1 must be recomputed. The short distance butterflies are then carried out by subroutine TBTFLY_ZIG. The permutation ϱ^{-1} which combines the bit reversal with the extract phase of the FChT2 is skipped (cf. Section 4.4.1.1).

After the recurrence, subvectors $\mathbf{w_l}$ and $\mathbf{w_{l+2K}}$ must be transformed by a forward FChT2, so that the newly computed polynomials pairs Z_{l+K}^{2K}, Z_{l+K+1}^{2K}, which are stored in $\mathbf{w_l}$, and polynomial pairs Z_{l+3K}^{2K}, Z_{l+3K+1}^{2K}, which are stored in $\mathbf{w_{l+2K}}$, can be transformed to their Chebyshev representation. Since the subvectors are already separated into two groups of p_1 processors, no special care needs to be taken. Note, however, that the permutation ϱ must be skipped. Furthermore, since the second half of both vectors will be discarded, we can alter the algorithm to avoid computing them. Figure 4.7C shows the situation after the forward FChT2 is carried out. The modified forward transform is presented as a separate subroutine, see Algorithm 4.9, because it is also needed in stage 1 of the main algorithm. The total cost of Algorithm 4.8 is

$$C_{\mathrm{Recurrence2,par}}(N,p,k) = \begin{cases} 8.5\frac{N}{p}\log_2\frac{N}{2^k} + 28.5\frac{N}{p} + 2\frac{N}{p}\cdot g + l, & \text{if } k = \log_2(2p), \\ 8.5\frac{N}{p}\log_2\frac{N}{2^k} + 28.5\frac{N}{p} + 4\frac{N}{p}\cdot g + 2\cdot l, & \text{otherwise.} \end{cases} \tag{4.34}$$

The case $k = \log_2(2p)$ is special because $2K = N/p$, which means that the forward FChT2 (superstep 5) can be computed locally without communication.

4.4.2. Optimized template. The template for the optimized FLT algorithm is given as Algorithm 4.10. For simplicity, we assume that $M \leq \min(N/2, N/(2p))$. The algorithm works as follows. In stage 1, the auxiliary complex vectors $\mathbf{h}$ and $\mathbf{w}$ are initialized and transformed so that the Chebyshev coefficients of the polynomial pairs $Z_0^{N/2}$, $Z_1^{N/2}$ are stored as the real and complex parts of the first half of vector $\mathbf{h}$, and the Chebyshev coefficients of the polynomial pairs $Z_{N/2}^{N/2}$, $Z_{N/2+1}^{N/2}$ are stored as the real and complex parts of the first half of vector $\mathbf{w}$.

At each subsequent stage $k = 2, \ldots, \log_2(N/M)$, the polynomials Z_l^{2K}, Z_{l+1}^{2K}, with $K = N/2^k$ and $l = 0, 2K, \ldots, N-2K$ (which are stored in vector $\mathbf{h}$ if $l = 0, 4K, \ldots, N-4K$, otherwise they are stored in vector $\mathbf{w}$), are transformed into the polynomials Z_l^K, Z_{l+1}^K, Z_{l+K}^K, and Z_{l+K+1}^K as follows. In superstep 3 (and 5a), the polynomials Z_l^{2K}, Z_{l+1}^{2K} are copied and further truncated using subroutine BSP_Rearrange (or copy statements), so that vector $\mathbf{h}$ receives the truncated polynomials Z_l^K, Z_{l+1}^K, and vector $\mathbf{w}$ keeps a copy of the polynomials Z_l^{2K}, Z_{l+1}^{2K}. In supersteps 4 (and 5b), the polynomials Z_l^{2K}, Z_{l+1}^{2K} are transformed into Z_{l+K}^{2K}, Z_{l+K+1}^{2K} and further truncated to obtain the polynomials Z_{l+K}^K, Z_{l+K+1}^K. Note that the modified sequential recurrence (Algorithm 4.11) must be called twice, once for l and once

Algorithm 4.10 Template for the optimized parallel 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 $p < \sqrt{N}$.
N: Transform size; N is a power of 2 with $N \geq 4$.
M: Termination block size; M is a power of 2 with $M \leq \min(N/2, N/(2p))$.
$\mathbf{f} = (f_0, \ldots, f_{N-1})$: Real vector of size N (block distributed).

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

DESCRIPTION

```
$1^{\mathrm{Comp}}$    Stage 1: Initialization.
        for $j = s\frac{N}{p}$ to $s\frac{N}{p} + \frac{N}{p} - 1$ do
            $h_j \leftarrow (1 + ix_j^N) \cdot f_j$
            $w_j \leftarrow (\mathbf{Q}'[0,j] + i\mathbf{Q}'[1,j]) \cdot f_j$

$2^{\mathrm{CpCm}}$    Chebyshev Transform.
        BSP_Rho2$(s, p, \frac{N}{p}, \mathbf{h}, \mathbf{w})$
        BSP_FChT2_mod2$(0, s, p, 1, \frac{N}{2}, \mathbf{h}, \mathbf{w})$

        for $k = 2$ to $\log_2(2p)$ do
$3^{\mathrm{Comm}}$            Parallel rearrange.
            BSP_Rearrange$(s, p, k, N, \mathbf{h}, \mathbf{w})$

$4^{\mathrm{CpCm}}$            Parallel recurrence.
            BSP_Recurrence2$(s, p, k, N, \mathbf{w})$

$5^{\mathrm{Comp}}$    Sequential stages.
        for $k = \log_2(4p)$ to $\log_2 \frac{N}{M}$ do
            $K \leftarrow \frac{N}{2^k}$
            for $l = s\frac{N}{p}$ to $s\frac{N}{p} + \frac{N}{p} - 4K$ step $4K$ do
             (a). Sequential rearrange.
                 $\mathbf{w_{l+2K}} \leftarrow$ Copy$(2K, \mathbf{w_l})$
                 $\mathbf{w_l} \leftarrow$ Copy$(2K, \mathbf{h_l})$
                 $\mathbf{h_{l+2K}} \leftarrow$ Copy$(K, \mathbf{w_{l+2K}})$

             (b). Sequential recurrence.
                 Seq_Recurrence2$(l, K, \mathbf{w_l})$
                 Seq_Recurrence2$(l + 2K, K, \mathbf{w_{l+2K}})$

$6^{\mathrm{Comp}}$    Stage $\log_2 \frac{N}{M} + 1$: Termination.
        for $l = s\frac{N}{p}$ to $s\frac{N}{p} + \frac{N}{p} - 2M$ step $2M$ do
            $\mathbf{f_l} \leftarrow$ Terminate$(l, M, \mathrm{Re}(\mathbf{h_l}), \mathrm{Im}(\mathbf{h_l}))$
            $\mathbf{f_{l+M}} \leftarrow$ Terminate$(l + M, M, \mathrm{Re}(\mathbf{w_l}), \mathrm{Im}(\mathbf{w_l}))$
```

for $l + 2K$. The termination stage is carried out by subroutine Terminate (Algorithm 4.6); this subroutine must also be called twice, because the polynomials Z_l^M, Z_{l+1}^M, $l = 0, 2M, \ldots, N-2M$, are stored in the auxiliary vector $\mathbf{h}$, and the polynomials Z_{l+M}^M, Z_{l+M+1}^M are stored in the auxiliary vector $\mathbf{w}$.

Algorithm 4.11 Template for the (sequential) modified recurrence.

CALL Seq_Recurrence2($l, K, \mathbf{w}$).

ARGUMENTS

l, K: Polynomial parameters.

$\mathbf{w} = (w_0, \ldots, w_{2K-1})$: Complex vector of size $2K$ containing the Chebyshev coefficients of the polynomial pair Z_l^{2K}, Z_{l+1}^{2K}, packed as complex.

OUTPUT $\mathbf{w}$: Chebyshev coefficients of the polynomial pair Z_{l+K}^K, Z_{l+K+1}^K, packed as complex. The second half of the vector contains junk data.

DESCRIPTION

```
1. Backward modified FChT2.
   $w_0 \leftarrow \frac{1}{2} \cdot w_0$
   FCT2_PACK$(0, 2K, \mathbf{w})$
   TBTFLY_ZIG$(0, 2K, 4, \mathbf{w})$

2. Recurrence.
   for $j = 0$ to $2K - 1$ do
       $a1 \leftarrow \mathbf{Q}'[2k-2, l+j] \cdot \mathrm{Im}(w_j) + \mathbf{R}'[2k-2, l+j] \cdot \mathrm{Re}(w_j)$
       $a2 \leftarrow \mathbf{Q}'[2k-1, l+j] \cdot \mathrm{Im}(w_j) + \mathbf{R}'[2k-1, l+j] \cdot \mathrm{Re}(w_j)$
       $w_j \leftarrow a1 + ia2$

3. Forward modified FChT2.
   BTFLY_ZIG$(0, 2K, 4, \mathbf{w})$
   $w_0 \leftarrow \frac{1}{2K} \cdot w_0$
   for $j = 1$ to $K - 1$ do
       $w_j \leftarrow \frac{1}{2K} \cdot [\cos(\frac{\pi j}{4K}) \cdot (w_j + w_{2K-j}) + i \sin(\frac{\pi j}{4K}) \cdot (w_j - w_{2K-j})]$
```

After all the optimizations, only three communication supersteps remain: the permutation from zig-zag cyclic to block distribution inside the inverse FFT, the permutation from block to zig-zag cyclic distribution inside the FFT, and the rearrange operation. The total communication cost is $(5\frac{N}{p}\log_2 p + 5.5\frac{N}{p}) \cdot g + 3 \cdot (\log_2 p + 1) \cdot l$. Algorithm 4.10 is the template for the optimized FLT. Its approximate cost is

$$\begin{aligned} C_{\mathrm{FLT,par}}(N, M, p) \approx\ & 4.25\frac{N}{p}(\log_2^2 N - \log_2^2 M) + 24.25\frac{N}{p}(\log_2 N - \log_2 M) + 2\frac{NM}{p} \\ & + 5\frac{N}{p}\log_2 p \cdot g + 3\log_2 p \cdot l. \end{aligned} \tag{4.35}$$

Note that there is relatively less communication in the parallel FLT algorithm compared with the parallel FFT algorithm, for $p < \sqrt{N}$, cf. (2.38). This means that one should expect a better scalability behavior for the FLT algorithm.

4.4.3. 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 [**20**], which may be faster for small problem sizes. The termination as expressed by Lemma 4.11 is similar to the multiplication of a dense lower triangular matrix and a vector.

4.4.3.1. *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 $\mathbf{h} = L\mathbf{x}$. Assume for simplicity that p is square. A parallel algorithm for matrix-vector multiplication was proposed in [**10**]. This algorithm is based on a two-dimensional distribution of the matrix over the processors, which are numbered (s, t), $0 \leq s < p_0$, $0 \leq t < p_1$, 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 $(s, *)$, 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 processor $(i \bmod \sqrt{p}, (j \operatorname{div} \sqrt{p}) \bmod \sqrt{p})$. 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 processors $(*, (j \operatorname{div} \sqrt{p}) \bmod \sqrt{p})$, 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 $\frac{n^2}{p} + 2\frac{n}{\sqrt{p}} g + 2l$.

4.4.3.2. *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 to $p_1 = \frac{pM}{N}$ and redistributes the data to the input distribution of the termination. We assume, for simplicity of exposition, that p_1 is square. We adapt the lower triangular matrix-vector multiplication algorithm to the context of the termination, as follows. Let the termination index $l \geq 1$ be fixed. We replace n by $M - 1$ and p by p_1, and define L using Lemma 4.11, for instance by $L_{ij} = q^j_{l,i}/2$ for $i \geq j$, $i - j$ even, and $j > 0$. Here, we include the trivial case $i = 0$. The two-dimensional processor numbering is created by the identification $(s, t) \equiv s0 + s + t\sqrt{p_1}$, where the offset $s0$ denotes the first processor of the group that handles the termination for l. Figure 4.8 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

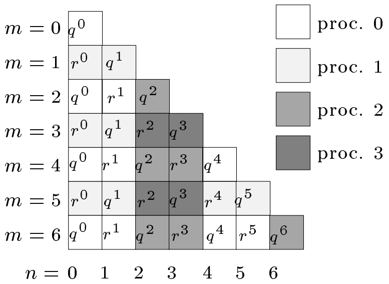

$m = 0$ q^0
$m = 1$ r^0 q^1
$m = 2$ q^0 r^1 q^2
$m = 3$ r^0 q^1 r^2 q^3
$m = 4$ q^0 r^1 q^2 r^3 q^4
$m = 5$ r^0 q^1 r^2 q^3 r^4 q^5
$m = 6$ q^0 r^1 q^2 r^3 q^4 r^5 q^6
$n = 0$ 1 2 3 4 5 6
proc. 0
proc. 1
proc. 2
proc. 3

FIGURE 4.8. 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 $p_1 = 4$ processors. The coefficients $q^n = q^n_{l,m}$ and $r^n = r^n_{l,m}$ are stored in a lower triangular matrix fashion.

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

$$C_{\text{term, par}}(N, M, p) \approx \frac{MN}{p} + \frac{2\sqrt{MN}}{\sqrt{p}} g + 2l. \tag{4.36}$$

4.5. *Experimental results and discussion*

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

Our implementations follow the same conventions as described in Section 2.5, and were tested on the same machine, i.e., a Cray T3E with up to 64 processors, with double precision (64-bit) accuracy of 1.0×10^{-15}. Accuracy results are also given using the more commonly used IEEE 754 floating point arithmetic for which the double precision accuracy is 2.2×10^{-16}.

4.5.1. Accuracy. We tested the accuracy of our implementation by measuring the relative error (2.46) obtained when transforming a random input vector $\mathbf{f}$ with elements uniformly distributed between 0 and 1. The exact DLT was computed by (4.1), using the stable three-term recurrence (4.2) and quadruple precision.

Table 4.1 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. This could not be done for the Cray T3E

TABLE 4.1. Relative errors for the sequential FLT algorithm. (QP indicates that the precomputation is carried out in quadruple precision.)

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

because quadruple precision is not available there. Note, however, that it is possible to precompute the values on another computer. The results show that the error of the FLT algorithm is comparable with the error of the DLT provided that the precomputed values are accurate. Therefore it is best to perform the precomputation in increased precision. This can be done at little extra cost, because the precomputation is done only once and its cost can be amortized over many FLTs. 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 **Q** and **R** can be halved. We will not address this issue here. See [**30, 29**] 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.

4.5.2. Efficiency of the sequential implementation. We measured the efficiency of our optimized FLT algorithm by comparing its execution time with the execution time of the direct DLT algorithm (i.e., a matrix-vector multiplication). Table 4.2 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 for $N \leq 8192$; $M = N/2$ is the maximum termination value that our program can handle, and the resulting algorithm is similar to the semi-naive algorithm [**20**]. 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.

TABLE 4.2. Execution time (in ms) of various Legendre transform algorithms on one processor of a CRAY T3E.

N	DLT	FLT $M = N/2$	FLT $M = 64$	FLT $M = 2$
16	0.0176	0.0317	$--$	0.0880
32	0.0668	0.0647	$--$	0.2018
64	0.2836	0.1263	$--$	0.4697
128	1.4640	0.3563	0.3563	1.1362
256	6.8996	1.4302	1.1369	2.5533
512	27.4952	5.7674	3.4048	6.2110

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

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

Tables 4.3 and 4.4 show the timing results obtained for the sequential and parallel versions executed on up to 64 processors, with $p < \sqrt{N}$, for $M = 2, 64, 128$. The termination parameters $M = 64$ and $M = 128$ are the empirically determined values that make the algorithm perform best for $N \leq 8192$ and $N > 8192$, respectively. These empirical values are in accordance with the theoretical optimum $M = 64$, see Section 4.2.4.

TABLE 4.3. Execution times (in ms) for the pure FLT algorithm ($M = 2$) on a Cray T3E.

N	*seq*	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$
512	6.23	6.47	3.72	2.77	1.81	2.16	$--$	$--$
1024	14.68	15.03	8.36	4.72	3.23	2.54	$--$	$--$
2048	35.52	34.95	18.66	10.30	6.01	4.37	3.88	$--$
4096	83.79	81.57	43.53	22.61	12.85	7.63	5.94	$--$
8192	212.79	211.54	105.71	54.46	29.15	16.05	9.72	9.10
16384	669.61	669.10	336.68	161.18	82.74	44.87	23.69	15.77
32768	1841.50	1843.20	941.74	469.40	209.09	108.20	55.28	32.09
65536	4560.50	4558.70	2345.00	1206.70	569.25	247.63	128.73	67.84

TABLE 4.4. Execution times (in ms) for the FLT algorithm with optimal early termination values on a Cray T3E. $M = 64$ for $N \leq 8192$, $M = 128$ for$N > 8192$.

N	*seq*	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$
512	3.40	3.62	2.32	1.70	$--$	$--$	$--$	$--$
1024	8.79	9.06	5.51	3.26	2.43	$--$	$--$	$--$
2048	22.06	22.49	12.50	7.41	4.57	3.65	$--$	$--$
4096	54.30	56.52	30.57	16.49	9.82	6.31	5.13	$--$
8192	154.29	157.31	78.37	41.41	22.10	13.21	8.39	8.26
16384	494.25	504.03	253.94	116.88	62.28	32.86	18.89	13.15
32768	1400.30	1434.90	731.54	358.12	155.87	81.84	42.78	26.19
65536	3621.00	3673.10	1900.50	967.43	454.33	190.45	100.97	54.60

Figure 4.9 shows the absolute speedups and absolute efficiencies obtained for various input sizes with $M = 2$ on up to 64 processors. The speedups for $M = 64, 128$ (not shown) are somewhat lower than for $M = 2$ because sequential early termination does not reduce the parallel overhead of the algorithm; it improves only the computation part.

The DLT is normally used as part of a two-dimensional Legendre transform or as part of a three-dimensional spherical harmonic transform. This means that, in practical applications, many independent FLTs of small size will be performed by a group of processors which could be only slightly larger than the number of transforms. For this reason it is important that the FLT algorithm scales well for small N on a small number of processors. Figure 4.9 (A) and (B) show that our algorithm scales well to very well on up to 8 processors with N as small as 512. When p is larger, our FLT implementation scales well for intermediate problem sizes ($4096 \leq N \leq 16384$), and very well for large problem sizes ($N \geq 32768$).

Since the parallel FLT algorithm has relatively less communication than the parallel FFT algorithm, the FLT implementation should scale better than the FFT implementation. Comparing the scalability results of both algorithms (cf. Figure 2.4 and Figure 4.9), we verify that the FLT indeed scales better than the FFT for completely in-cache computations, i.e. $N \leq 4096$. However, for out-of-cache computations, i.e., $N > 4096$, the FFT implementation achieves efficiencies up to 1.5 times larger than the ideal efficiency, while the maximum efficiency obtained by the FLT implementation is 1.15. This happens because the FLT algorithm is more cache-friendly than the FFT algorithm. At each stage k of the FLT algorithm, the size of the subproblems

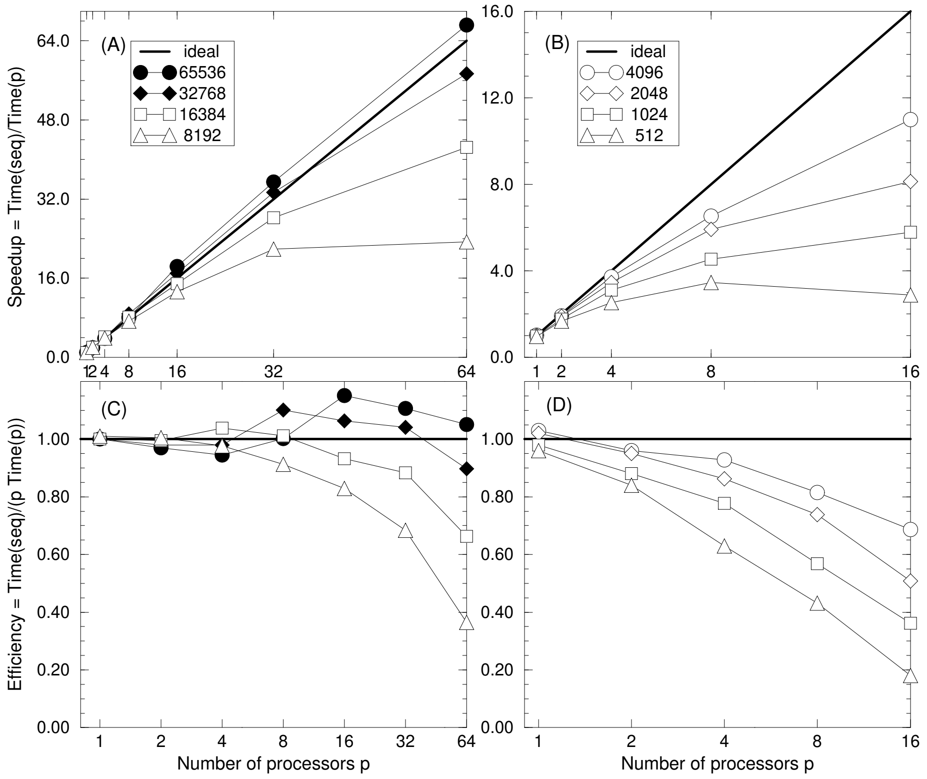


FIGURE 4.9. Scalability of the FLT on a Cray T3E.

halves as the number of subproblems doubles. After a certain stage k^*, the subproblem size will be small enough to fit in cache. This means that both the sequential and the parallel FLT implementations use the cache memory more efficiently and therefore the ratio Time(seq)/Time(p) will be smaller making the cache effect less pronunciated.

4.6. *Conclusions and future work*

In this chapter, we developed and implemented a sequential algorithm for the discrete Legendre transform, based on the Driscoll-Healy algorithm. This implementation is competitive for large problem sizes. Its complexity $O(N \log^2 N)$ is considerably lower than the $O(N^2)$ matrix-vector multiplication algorithms which are still much in use today for the computation of Legendre transforms. Its accuracy is similar, provided the precomputation is performed in increased precision. The new algorithm promises to be useful in compute-intensive applications such as weather forecasting.

To obtain an efficient parallel algorithm we started with a basic algorithm, and then optimized this algorithm by removing permutations and redistributions wherever possible. To carry out the optimizations we used all the tools developed in the previous chapters. In particular, the choice of the FCT2 algorithm introduced in Section 3.4 simplified the optimization work, because the pack-transform-extract structure of the FCT2 algorithm leads to a simpler parallel algorithm than the algorithm obtained for single FCTs. 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 experimental results show that the performance of our parallel algorithm scales very well with the number of processors, for medium to large problem sizes. For small problem sizes the FLT scales very well on a small number of processors, say $p \leq 8, 16$. This situation is the most likely to occur in practical applications, since the DLT is most used as part of a two-dimensional Legendre transform or as part of a spherical harmonic transform.

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 [**22**] 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.

Final Remarks and Future Work

The subject of this thesis was the development of new parallel algorithms for computing discrete transforms. More specifically, we derived and implemented parallel algorithms for the fast Fourier transform (of complex data – CFFT, and real data – RFFT), the fast cosine transform (FCT), and the fast Legendre transform (FLT). These algorithms compute the corresponding discrete transform of an input vector of size N in $O(N \log N)$ arithmetic operations (CFFT, RFFT, FCT), or $O(N \log^2 N)$ arithmetic operations (FLT).

In Chapter 1, we introduced the BSP model and discussed relevant aspects of parallel computing. In Chapters 2 to 4 we presented the algorithms. Besides deriving the algorithms, we presented them in the form of templates, which give a complete description of the algorithms and facilitate their implementation. We also presented results concerning their accuracy and scalability. Here, we conclude the thesis with some general remarks; specific conclusions can be found at the end of each chapter.

We chose the BSP model because it gives a simple and effective way of producing 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. This choice turned out to be the right one for the kind of algorithms discussed in this thesis.

We presented our parallel algorithms using templates in which the block distribution is maintained throughout the procedure: changes in distribution are effectuated by permutations. In this way the algorithms can be readily implemented, because it is easy to compute where each data element is stored. For example, the j-th element of a vector of size N which is distributed over p processors is always stored in processor $\mathrm{Proc}(j) = j \,\mathrm{div}\, \frac{N}{p}$ and has local index $j' = j \bmod \frac{N}{p}$. The way we presented our algorithms can be viewed as a methodology for formulating parallel algorithms.

In most cases, our computation subroutines are local subroutines which are parametrized by a parameter α. This parametrization is useful when designing cache-friendly algorithms (sequential or parallel), since the resulting cache-friendly algorithms can use the same computation subroutines, although with a different α.

All the algorithms were implemented and tested on a Cray T3E with up to 64 processors. The tests showed that the algorithms scale well to very well as long as the ratio problem size/number of processors is not very small (N/p should be at least larger than 128). Note that such good results were only possible because we communicate data in large packets. In principle, it is possible to run our implementations on a large variety of parallel computers without having to change one line of program code. Though we did not test our implementations on other parallel machines, the BSP cost function can be used to predict their performance as long as the BSP parameters g, l, and v are available.

The Bulk Synchronous Parallel Fast Transform package (BSPFTpack) contains the implementations of the algorithms discussed in this thesis. This package was written in ANSI C and uses BSPlib as communication library, which is freely available. Though this package was originaly written using BSPlib, it can be easily adapted to use another communication library such as MPI. In that case only the communication supersteps would have to be modified.

Though this package is ready and will soon be available on the World Wide Web,[2] there is still much work to be done. The parallel FFT is optimized to a large extent, and works for any p that is a power of two. The parallel RFFT, FCT, FCT2, and FLT algorithms have additional restrictions. The mixed-radix parallel FFT algorithm should be implemented. Cache-friendly versions of the same algorithms should be developed as well. The possibility of implementing mixed-radix parallel algorithms for the RFFT, FCT, FCT2, and FLT should also be investigated. In the specific case of the FLT, an inverse FLT must be developed and the FLT must be adapted to the more general case of the spherical harmonic transform.

---

[2] `http://www.math.uu.nl/people/bisseling/software.html`

Appendix A

BSP Parameters on the Cray T3E

The BSP parameters we use were measured using a modified version of the program `bspbench` that comes with the package `BSPEDUpack`.[1] The modifications we made are the following.

- We use `bsp_hpput` instead of `bsp_put`.
- We communicate data using packets instead of single elements.
- We use h-relations with $h = 1, 32, 64, \ldots, 8495$, thus at intervals of size 31.

Table A.1 shows the values of s, g, and l as a function of the number of processors.

TABLE A.1. BSP parameters for the CRAY T3E, with $s = 34.9$ Mflops/s. The value of s is based on in-cache dot product computations.

p	g		l	
	(flops)	(μs)	(flops)	(μs)
1	0.28	0.00804	3	0.09
2	1.14	0.03279	479	13.72
4	1.46	0.04175	858	24.57
8	2.14	0.06133	1377	39.48
16	2.30	0.06600	1754	50.26
32	2.77	0.07940	2024	58.00
64	3.05	0.08758	3861	110.88

---

[1] `http://www.math.uu.nl/people/bisseling/software.html`

Appendix B

Lookup Table

Our implementations use the array w defined below as a lookup table for sines and cosines. Let L be a power of two and define the array w of size $L+1$ by:

$$\mathsf{w}[l] = \cos(\frac{\pi \cdot l}{2L}), \quad \text{for } l = 0, \ldots, L, \tag{B.1}$$

(by symmetry, $\mathsf{w}[L-l] = \sin(\frac{\pi \cdot l}{2L})$). This table contains all the weights needed by the CFFT and RFFT of size N, provided that $L \geq N/4$, and all the weights needed by the FCT, FCT2, and FLT of size N, provided that $L \geq N$.

As an example, we compute the index corresponding to weight

$$w_k^{j+\alpha} = \cos(2\pi(j+\alpha)/k) + i \sin(2\pi(j+\alpha)/k),$$

with $0 \leq j < k/2$, needed by a forward generalized 2-butterfly, where $\alpha = (s \bmod u)/u$, for some u needed in Algorithm 2.2. Define $s_1 = s \bmod u$. Then the angle in question is

$$\frac{2\pi(j + s_1/u)}{k} = \frac{4\pi(u \cdot j + s_1)}{2ku} = \frac{\pi \cdot l}{2L},$$

where $l = (u \cdot j + s_1) \cdot 4L/(ku)$. The fact that $ku \leq N \leq 4L$ and that ku and $4L$ are both powers of two implies that $4L/(ku)$ is an integer. Furthermore, $u \cdot j + s_1 < ku/2$, and this implies $l \in \{0, 1, \ldots, 2L-1\}$. Therefore, the required values can be obtained directly from the table:

$$\cos(\frac{\pi \cdot l}{2L}) = \mathsf{w}[l], \text{ and } \sin(\frac{\pi \cdot l}{2L}) = \mathsf{w}[L-l], \text{ for } l \leq L, \tag{B.2}$$

and, by symmetry of the trigonometric functions,

$$\cos(\frac{\pi \cdot l}{2L}) = -\mathsf{w}[2L-l], \text{ and } \sin(\frac{\pi \cdot l}{2L}) = \mathsf{w}[l-L], \text{ for } l > L. \tag{B.3}$$

It is easy to access this lookup table by means of a stride. The index of the first needed weight, $w_k^{s_1/u}$, is $s_1 \cdot 4L/(ku)$. To compute the index l of the next needed

phase 0	$u = 1,\ s_1 = 0$		
$K = \ 4 \cdot 1$	$w_4^0,$	$w_4^0,$	$w_4^0,$
$K = 16 \cdot 1$	$w_{16}^0,$	$w_{16}^0,$	$w_{16}^0,$
	$w_{16}^1,$	$w_{16}^2,$	$w_{16}^3,$
	$w_{16}^2,$	$w_{16}^4,$	$w_{16}^6,$
	$w_{16}^3,$	$w_{16}^6,$	$w_{16}^9,$
phase 1	$u = 16,\ s_1 = 3$		
$K = 4 \cdot 16$	$w_4^{\frac{3}{16}},$	$w_4^{\frac{3}{16}},$	$w_4^{\frac{3}{16}},$
$K = 16 \cdot 16$	$w_{16}^{\frac{3}{16}},$	$w_{16}^{\frac{3}{16}},$	$w_{16}^{\frac{3}{16}},$
	$w_{16}^{1+\frac{3}{16}},$	$w_{16}^{2+\frac{3}{16}},$	$w_{16}^{3+\frac{3}{16}},$
	$w_{16}^{2(1+\frac{3}{16})},$	$w_{16}^{2(2+\frac{3}{16})},$	$w_{16}^{2(3+\frac{3}{16})},$
	$w_{16}^{3(1+\frac{3}{16})},$	$w_{16}^{3(2+\frac{3}{16})},$	$w_{16}^{3(3+\frac{3}{16})},$
phase 2	$u = 32,\ s_1 = 19$		
$K = 16 \cdot 32$	$w_{32}^{\frac{19}{32}},$	$w_{32}^{1+\frac{19}{32}},$	$w_{32}^{2+\frac{19}{32}},$
	$w_{32}^{3+\frac{19}{32}},$	$w_{32}^{4+\frac{19}{32}},$	$w_{32}^{5+\frac{19}{32}},$
	$w_{32}^{6+\frac{19}{32}},$	$w_{32}^{7+\frac{19}{32}}$	

FIGURE B.1. Alternative lookup table for computing a parallel FFT of size $N = 512$ with $p = 32$ and $s = 19$.

weight just add the stride $4L/k$ to the preceding index (this follows from the relation $l = (s_1 + u \cdot j) \cdot 4L/(ku)$). To find the value needed use (B.2)–(B.3). The indices of the weights used in the other algorithms can be computed and accessed in a similar way.

The table described above is simple to use, and it provides all the weights needed for any CFFT or RFFT of size $N \leq 4L$, and for any number of processors $p < N$, provided that N and p are powers of two. It can also be used for FCTs, FCT2s, and FLTs, provided that the problem size does not exceed L. In the case of the FLT, the property that it can be used for any p and N is useful, because the parallel FLT algorithm needs to compute FFTs for various combinations of p and N.

The wide applicability of this table does not come without a price. The table is not scalable, since its size depends only on N, and does not decrease with p.

Nevertheless, it is possible to construct a scalable table for the parallel algorithms of this book. For example, a simple way of constructing a table for the FFT is to store the weights one after another in the order they are needed, so that they can be easily accessed, see Figure B.1. Suppose, for simplicity, that N/p is an even power of two. Since a generalized 4-butterfly stage D_k^α needs $3k/4$ complex weights, each complete butterfly phase (i.e., $k = 4, 16, \ldots, N/p$) needs $N/p-1$ weight values.[1] Since there are a total of $H = \lceil \log_{\frac{N}{p}} N \rceil$ phases, the total size of the table cannot exceed $H(N/p-1)$ complex values. Note that for phase 0 it is possible to use table w, with $L = N/(4p)$.

---

[1] If N/p is not an even power of two, we arrive at the same result.

Appendix C

Related Polynomial Transforms and Algorithms

The derivation of the Driscoll-Healy algorithm given in Section 4.2 has the feature that it only depends on the properties of truncation operators $\mathcal{T}_K$ given in Lemma 4.8, and on the existence of an efficient algorithm for applying the truncation operators. In particular, Lemma 4.8 and Lemma 4.10 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 which starts with the polynomial 1, and the Lagrange interpolation operators are defined using the roots of the polynomials from the sequence. In theory, this can be used to develop new algorithms for computing orthogonal polynomial transforms, though with different sample weights w_j. In practice, however, the existence of efficient Chebyshev and cosine transform algorithms makes these the only reasonable choice in the definition of the truncation operators. This situation may change with the advent of other fast transforms.

Theoretically, the basic algorithm works, with minor modifications, in the following general situation. We are given operators $\mathcal{T}_M^K$, for $1 \leq M \leq K$, 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 $M \leq L \leq K$ then $\mathcal{T}_M^L \mathcal{T}_L^K = \mathcal{T}_M^K$.
3. If $\deg Q \leq m \leq K \leq L$ 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 $0 \leq l < N$, where $\{p_l\}$ is a sequence of orthogonal polynomials. This problem may be treated using the same algorithms as in Section 4.2, but with the truncation operators $\mathcal{T}_M$ replaced by $\mathcal{T}_M^K$, where $K \leq N$ 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

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

cf. (4.16) and (4.17).

This generalization of the approach we have presented may be used to derive the original algorithm of Driscoll and Healy in the exact form it was presented [**21**], which uses the cosine transforms in the points $\cos(j\pi/K)$.

Driscoll, Healy, and Rockmore [**22**] 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 D

Precomputation Algorithm for the FLT

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

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

LEMMA D.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*

$$\begin{aligned} Q_{l,j+k} &= Q_{l+k,j} Q_{l,k} + R_{l+k,j} Q_{l,k-1}, \\ R_{l,j+k} &= Q_{l+k,j} R_{l,k} + R_{l+k,j} R_{l,k-1}. \end{aligned} \tag{D.1}$$

PROOF. By induction on j. The proof for $j = 0$ follows immediately from the definition (4.10), 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

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

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 D.1 is to start with polynomials of degree $0, 1$, given in only one point, and then repeatedly double the number of points by performing a Chebyshev transform, adding zero terms to the Chebyshev expansion, and transforming back, and also double the maximum degree of the polynomials by applying the lemma, with $j = K-1, K$ and $k = K$.

Algorithm D.1 Precomputation of the point values needed by the FLT.

INPUT N: a power of 2.

OUTPUT $Q_{l,m}(x_j^{2^k})$, $R_{l,m}(x_j^{2^k})$, for $1 \leq k \leq \log_2 N$, $0 \leq j < 2^k$, $m = 2^{k-1}, 2^{k-1}-1$, 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 $\deg R_{l,m} \leq m-1$, so the Chebyshev coefficients $r_{l,m}^n$ with $n \geq m$ are zero, which means that the polynomial is fully represented by its first m Chebyshev coefficients. In the case of the $Q_{l,m}$, the coefficients are zero for $n > m$. If $n = m$, however, the coefficient is nonzero, and this is a problem if $m = K$. The K-th coefficient which was set to zero must then be corrected and set to its true value, which can be computed easily by using (4.10) and (4.3).

The point values needed can be retrieved as follows. The FLT algorithm requires the numbers

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

for $l = r \cdot 2K + 1$, $0 \leq r < \frac{N}{2K}$, for all K with $M \leq K \leq N/2$. After the m-loop in stage $k = \log_2 K + 1$ of Algorithm D.1, we have obtained these values for $l = rK + 1$, $0 \leq r < N/K$. 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

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

Comparing with the cost (4.26) 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 $q_{l,m}^n, r_{l,m}^n$ required to terminate the Driscoll-Healy algorithm early, as in Lemma 4.11, is based on the following recurrences.

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

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

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

PROOF. These recurrences are the shifted three-term recurrences (4.10) 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 $q_{l,m}^n$ and $r_{l,m}^n$ by increasing m, starting with the known values for $m = 0, 1$ and finishing with $m = M - 2$. For each m, we only need to compute the $q_{l,m}^n$ with $n \leq m$, and the $r_{l,m}^n$ with $n < m$. The total number of flops of the precomputation of the Chebyshev coefficients in the general case is

$$C_{\text{precomp, term}}(M) = 7M^2 - 16M - 15. \tag{D.3}$$

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. 4.4. 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 D.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.

Bibliography

[1] R. C. Agarwal and J. W. Cooley. Vectorized mixed radix discrete Fourier transform algorithms. *Proceedings of the IEEE*, 75(9):1283–1292, 1987.

[2] N. Ahmed, T. Natarajan, and K. R. Rao. Discrete cosine transforms. *IEEE Trans. Comput.*, 23:90–93, 1974.

[3] B. K. Alpert and V. Rokhlin. A fast algorithm for the evaluation of Legendre expansions. *SIAM J. Sci. Statist. Comput.*, 12(1):158–179, 1991.

[4] M. Ashworth and A. G. Lyne. A segmented FFT algorithm for vector computers. *Parallel Computing*, 6:217–224, 1988.

[5] A. Averbuch, E. Gabber, B. Gordissky, and Y. Medan. A parallel FFT on an MIMD machine. *Parallel Computing*, 15:61–74, 1990.

[6] S. R. M. Barros and T. Kauranne. On the parallelization of global weather models. *Parallel Computing*, 20:1335–1356, 1994.

[7] P. Barrucand and D. Dickinson. On the associated Legendre polynomials. In D. T. Haimo, editor, *Orthogonal Expansions and Their Continuous Analogues. Proceedings of the Conference held at Southern Illinois University, April 27–29, 1967*, pages 43–50. Southern Illinois University Press, Carbondale, IL, USA, 1968.

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

[9] R. H. Bisseling. Basic techniques for numerical linear algebra on bulk synchronous parallel computers. In L. Vulkov, J. Waśniewski, and P. Yalamov, editors, *Workshop Numerical Analysis and its Applications 1996*, volume 1196 of *Lecture Notes in Computer Science*, pages 46–57. Springer-Verlag, Berlin, 1997.

[10] R. H. Bisseling and W. F. McColl. Scientific computing on bulk synchronous parallel architectures. In B. Pehrson and I. Simon, editors, *Technology and Foundations: Information Processing '94, Vol. I*, volume 51 of *IFIP Transactions A*, pages 509–514. Elsevier Science Publishers, Amsterdam, 1994.

[11] G. Bongiovanni, P. Corsini, and G. Frosini. One-dimensional and two-dimensional generalized-discrete Fourier transforms. *IEEE Trans. ASSP*, ASSP-24:97–99, 1976.

[12] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) library – design, implementation and performance. In *13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, San Juan, Puerto Rico, April 12 – April 16 1999.

[13] W. L. Briggs and V. E. Henson. *The DFT: An Owner's Manual for the Discrete Fourier Transform.* SIAM, Philadelphia, PA, 1995.

[14] 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*, 117:1058–1075, 1989.

[15] E. Chu and A. George. FFT algorithms and their adaptation to parallel processing. *Linear Algebra and its Applications*, 284:95–124, 1998.

[16] J. W. Cooley and J. W. Tukey. An algorithm for machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[17] T. H. Cormen and D. M. Nicol. Performing out-of-core FFTs on parallel disk systems. *Parallel Computing*, 24(1):5–20, 1998.

[18] P. Corsini and G. Frosini. Properties of the multidimensional generalized discrete Fourier transform. *IEEE Trans. Comput.*, c-28(11):819–830, 1979.

[19] P. L. DeVries. *A First Course in Computational Physics*. John Wiley & Sons, New York, 1994.

[20] G. A. Dilts. Computation of spherical harmonic expansion coefficients via FFTs. *J. Comput. Phys.*, 57(3):439–453, 1985.

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

[22] J. R. Driscoll, D. M. Healy, Jr, and D. Rockmore. Fast discrete polynomial transforms with applications to data analysis for distance transitive graphs. *SIAM J. Comput.*, 26(4):1066–1099, 1997.

[23] A. Dubey, M. Zubair, and C. E. Grosch. A general purpose subroutine for Fast Fourier Transform on a distributed memory parallel machine. *Parallel Computing*, 20:1697–1710, 1994.

[24] B. S. Duncan and A. J. Olson. Approximation and characterization of molecular surfaces. *Biopolymers*, 33:219–229, 1993.

[25] A. Gupta and V. Kumar. The scalability of FFT on parallel computers. *IEEE Transactions on parallel and distributed systems*, 4(8):922–932, 1993.

[26] O. Haan. A parallel one-dimensional FFT for Cray T3E. In H. Lederer and F. Hertweck, editors, *Proceedings of the Fourth European SGI/Cray MPP Workshop*, pages 188–198, IPP, Garching, Germany, Sept. 10–11 1998.

[27] D. M. Healy, Jr and P. T. Kim. Spherical deconvolution with application to geometric quality assurance. Technical report, Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH, USA, 1993.

[28] D. M. Healy, Jr. and P. T. Kim. An empirical Bayes approach to directional data and efficient computation on the sphere. *Ann. Statist.*, 24(1):232–254, 1996.

[29] D. M. Healy, Jr, S. S. B. Moore, and D. Rockmore. FFTs for the 2-sphere - improvements and variations. Technical report PCS-TR96-292, Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH, USA, 1996.

[30] D. M. Healy, Jr, S. S. B. Moore, and D. N. 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, USA, 1993.

[31] M. Hegland. Real and complex fast Fourier transforms on the Fujitsu VPP 500. *Parallel Computing*, 22:539–553, 1995.

[32] 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*, 24:1947–1980, 1998.

[33] M. A. Inda, R. H. Bisseling, and D. K. Maslen. Parallel fast Legendre transform. In W. Zwieflhofer and N. Kreitz, editors, *Towards Teracomputing. Proceedings of the Eighth ECMWF Workshop on the Use of Parallel Processors in Meteorology, Reading, UK, 16–20 Nov, 1998*, pages 87–108. World Scientific, Singapore, 1999.

[34] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994.

[35] D. K. Maslen. A polynomial approach to orthogonal polynomial transforms. Technical report MPI/95-9, Max-Planck-Institut für Mathematik, Bonn, Germany, June 1994.

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

[37] M. J. Narasimha and A. M. Peterson. On the computation of the discrete cosine transform. *IEEE Trans. Commun.*, COM-26(6):934–936, 1978.

[38] T. Ooura, 1998. `http : //momonga.t.u − tokyo.ac.jp/∼ooura/fft.html`.

[39] S. A. Orszag. Fast eigenfunction transforms. In G.-C. Rota, editor, *Science and Computers*, volume 10 of *Advances in Mathematics Supplementary Studies*, pages 23–30. Academic Press, NY, 1986.

[40] R. B. Pelz. Parallel compact FFTs for real sequences. *SIAM Journal of Scientific Computing*, 14(4):914–935, 1993.

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

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

[43] P. A. Regalia and S. K. Mitra. Kronecker products, unary matrices and signal processing applications. *SIAM Review*, 31(4):586–613, 1989.

[44] N. Shalaby. Parallel discrete cosine transforms: theory and practice. Cambridge, MA TR-34-95, Center for Research in Computing Technology, Harvard University, Hampton, VA, 1995.

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

[46] R. C. Singleton. An algorithm for computing the mixed radix fast fourier transform. *IEEE Transactions on Audio and Electroacoustics*, AU-17(2):93–103, 1969.

[47] G. Steidl and M. Tasche. A polynomial approach to fast algorithms for discrete Fourier-cosine and Fourier-sine transforms. *Mathematics of Computation*, 56(193):281–296, 1991.

[48] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis.* Springer-Verlag, New York, 1980.

[49] P. N. Swarztrauber. Symmetric FFTs. *Mathematics of Computation*, 47(175):323–346, 1986.

[50] P. N. Swarztrauber. Multiprocessor FFTs. *Parallel Computing*, 5:197–210, 1987.

[51] C. Temperton. Fast mixed-radix real Fourier transforms. *J. Comput. Phys.*, 52:340–350, 1983.

[52] C. Temperton. Self-sorting mixed-radix fast Fourier transforms. *J. Comput. Phys.*, 52(1):1–23, 1983.

[53] C. Temperton. Self-sorting in-place fast Fourier transforms. *SIAM J. Sci. Statist. Comput*, 12(4):808–823, 1991.

[54] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[55] C. van Loan. *Computational Frameworks for the Fast Fourier Transform.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1992.

[56] S. Zoldi, V. Ruban, A. Zenchuk, and S. Burtsev. Parallel implementation of the split-step Fourier method for solving nonlinear Schödinger systems. *SIAM News*, January/February:8–9, 1999.

Summary

In this thesis we develop new parallel algorithms for discrete transforms. More specifically, we derive parallel algorithms for the fast Fourier transform (of complex data – CFFT, and real data – RFFT), the fast cosine transform (FCT), and the fast Legendre transform (FLT). These algorithms compute the corresponding discrete transform of an input vector of size N in $O(N \log N)$ arithmetic operations (CFFT, RFFT, FCT) or $O(N \log^2 N)$ arithmetic operations (FLT). A discrete transform can be seen as a matrix-vector multiplication of the form $\hat{\mathbf{f}} = W_N \cdot \mathbf{f}$, where $\mathbf{f} = (f_0, \ldots, f_{N-1})$ and $\hat{\mathbf{f}} = (\hat{f}_0, \ldots, \hat{f}_{N-1})$ are column vectors of size N and $W_N = \{w_N^{(k,j)}\}$ is an $N \times N$ matrix. The vector $\mathbf{f}$ contains the data to be transformed and the vector $\hat{\mathbf{f}}$ contains the transformed data. In the case of the Fourier transform, $W_N = F_N$ is known as the Fourier matrix and has elements $w_N^{(k,j)} = \exp(2\pi ijk/N)$. In the case of the cosine transform $w_N^{(k,j)} = \cos(\pi(j + \frac{1}{2})k/N)$, and in the case of the Legendre transform $w_N^{(j,k)} = \frac{1}{N} P_k(\cos(\pi(j + \frac{1}{2})/N))$, where P_k is the k-th Legendre polynomial.

We design our parallel algorithms using the *bulk synchronous parallel* (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 BSP model [**54**], 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. To ensure the correct execution of the algorithm, global synchronization barriers (i.e., places of the algorithm where all processors must wait for each other) precede and/or follow a communication superstep. Using supersteps imposes a sequential structure on parallel algorithms, and this greatly simplifies the design process.

In Chapter 1 we describe the BSP model in more detail and also discuss aspects relevant to parallel computing. Chapters 2 to 4 are dedicated to deriving the parallel discrete transform algorithms. Each algorithm is presented in the form of a template, which is a high level of detail algorithm that can be readily implemented. Furthermore, results concerning accuracy and scalability are presented.

Our CFFT algorithm is a parallel version of the well-known *radix-2 FFT* algorithm [**16, 55**], which assumes N to be a power of two. The sequential radix-2 FFT algorithm starts with a bit reversal permutation of the input vector, and proceeds in $\log_2 N$ butterfly stages, numbered $K = 2, 4, \ldots, N$. Each butterfly stage consists of N/K times a butterfly computation, which modifies $K/2$ pairs $(f_j, f_{j+K/2})$ at distance $K/2$ by the following computation:

$$\begin{pmatrix} f_j \\ f_{j+K/2} \end{pmatrix} \leftarrow \begin{pmatrix} f_j + e^{2\pi ij/K} \cdot f_{j+K/2} \\ f_j - e^{2\pi ij/K} \cdot f_{j+K/2} \end{pmatrix}.$$

Assuming that the number of processors $p < N$ is a power of two, and that the input/output vector is in the block distribution (i.e., divide the vector in p subvectors of the same size and give one subvector to each processor), the parallel algorithm starts with a parallel bit reversal permutation (which is a communication superstep) and carries out the butterfly stages by interleaving communication and computation supersteps. The communication supersteps redistribute the data vector so that the data needed by the next computation superstep is local. The distributions used in the algorithm are members of a family which we call the *group-cyclic distribution family* $\mathrm{C}^r(p, N)$. This family includes the well-known cyclic and the block distribution as extreme cases, $\mathrm{C}^1(p, N)$ and $\mathrm{C}^p(p, N)$. Each computation superstep, which consists of at most $\log_2(N/p)$ butterfly stages, is called a *butterfly phase*. A total of $H = \lceil \log_2 N / \log_2(N/p) \rceil = \lceil \log_{\frac{N}{p}} N \rceil$ butterfly phases is performed. (H is the largest integer for which $(N/p)^{H-1} \leq N$.) In phase 0, the first $\log_2(N/p)$ stages are performed in the block distribution. This phase consists of butterfly stages with $K \leq N/p$ (which we call short distance butterflies). Afterwards, in each intermediate phase J, $1 \leq J < H-1$, a group of $\log_2(N/p)$ butterfly stages (the medium distance butterflies) is performed in the cyclic distribution restricted to a subgroup of processors of size $(N/p)^J$. Note that, if $p \leq N/p$, then $\log_{\frac{N}{p}} H \leq 2$, which means that no intermediate phase is performed. Finally, in phase $H - 1$, the remaining long distance butterflies are performed in the cyclic distribution over p processors. This process is illustrated in Figure 1.

In Chapter 2 we derive the FFT algorithm sketched above. Furthermore, we show how to modify the algorithm to accept vectors which are not in the block distribution; specifically, in the case of the cyclic distribution we show that the communication cost can be drastically reduced. We also show how to obtain a *cache-friendly* version of our algorithm, that is, an algorithm that takes advantage of the cache memory of a computer (i.e. a small but very fast memory) by breaking up the computations

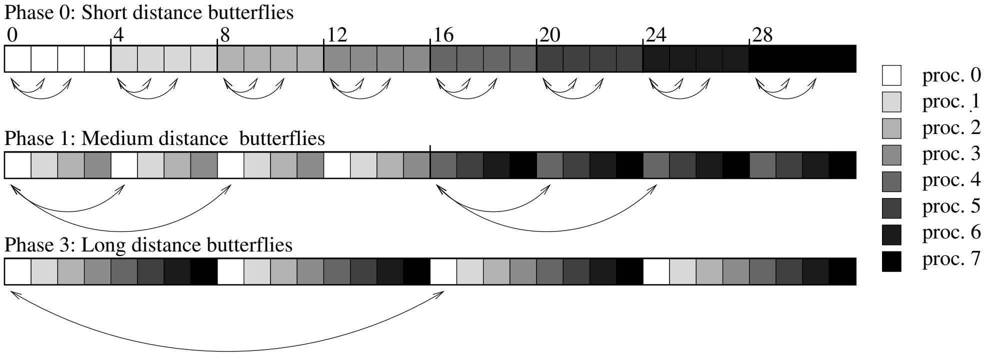

Phase 0: Short distance butterflies
0
4
8
12
16
20
24
28
Phase 1: Medium distance butterflies
Phase 3: Long distance butterflies
proc. 0
proc. 1
proc. 2
proc. 3
proc. 4
proc. 5
proc. 6
proc. 7

FIGURE 1. Butterfly operations using the group-cyclic distribution family $\mathrm{C}^r(p,N) = \mathrm{C}^r(8,32)$. The short distance butterflies are carried out using the block distribution. The medium distance butterflies are performed using the $\mathrm{C}^2(8,32)$ distribution. The long distance butterflies are performed using the cyclic distribution. For clarity, not all butterfly pairs are shown.

in small sections in such a way that the data stored in the cache is completely used before new data is brought in. As a final topic in this chapter we introduce the *parallel mixed-radix FFT algorithm* which is based on the work of Agarwal and Cooley [**1**] for vector computers. This last algorithm is most promising, since it can solve FFTs of any combination of p and $N = N_0 N_1 \dots N_{H-1}$ as long as p divides each N_l and sequential FFTs of size N_l are available.

Algorithms for the RFFT and for the FCT can be derived by: (1) packing the N real input data elements into a complex vector of size $N/2$, (2) transforming the complex vector using a CFFT of size $N/2$, and then (3) extracting the desired transform from the transformed data. Such algorithms have a computational cost of roughly half the computational cost of a CFFT of the same size. The packing phase of both algorithms is a permutation, which can be carried out without extra communication cost by combining it with the bit reversal permutation of the CFFT. The extract operation is a pair-wise operation that combines element pairs $(j, N-j)$. In order to perform this operation without any extra communication cost we perform the long distance butterflies in the *zig-zag cyclic distribution*, which is a variant of the cyclic distribution that contains element pairs $(j, N-j)$ in the same processor. In Chapter 3 we describe the parallel RFFT and FCT algorithms for the case that $p < \sqrt{N/2}$. We also show how to invert the algorithms and we derive a new algorithm that computes the FCT of two vectors at the same time.

In 1989, Driscoll and Healy introduced a fast polynomial transform algorithm that computes the discrete Legendre transform in $O(N \log^2 N)$ arithmetic operations [**21, 22**]. The algorithm starts by computing the Chebyshev coefficients of the input vector by a discrete Chebyshev transform, and then it proceeds in $\log_2 N$ stages, assuming that N is a power of two. (The discrete Chebyshev transform is similar to the DCT and can be computed in $O(N \log N)$ arithmetic operations by using the FCT algorithm.) Each stage k, $1 \leq k < \log_2 N$, of the FLT algorithm, takes a set of 2^{k-1} polynomials of degree less than $N/2^{k-1}$ as input data and computes a new set of polynomials by means of a recurrence procedure which involves forward and backward FCTs. At the end of the stage, both the input polynomials and the newly computed ones are truncated to half their original size so that the total amount of data remains constant. Designing a basic parallel version of this algorithm is straightforward [**30, 45**]. In the initial stages, $k \leq \log_2 p$, there are $2^{k-1} < p$ independent problems and a group of $p/2^{k-1}$ processors must deal with one problem. Inter-processor communication is needed, but it occurs only in two instances: (1) inside the parallel FCTs, and (2) during the truncation. In the final stages, $k > \log_2 p$, there are $2^{k-1} \geq p$ independent problems and one processor must deal with a group of problems, which means that no inter-processor communication is needed. In Chapter 4 we derive the sequential FLT algorithm and the basic parallel algorithm. Having the parallel algorithm sketched above as basis, we derive an improved version which reduces the communication cost by a factor of three.

Samenvatting

In dit proefschrift ontwikkelen we nieuwe parallelle algoritmen voor discrete transformaties. In het bijzonder leiden we algoritmen af voor de snelle Fourier transformatie (van complexe gegevens - CFFT, en reële gegevens - RFFT), de snelle cosinus transformatie (FCT) en de snelle Legendre transformatie (FLT). Deze algoritmen berekenen de bijbehorende discrete transformatie in $O(N \log N)$ rekenkundige operaties (CFFT, RFFT, FCT) of $O(N \log^2 N)$ rekenkundige operaties (FLT). Een discrete transformatie kan beschouwd worden als een matrix-vector vermenigvuldiging van de vorm $\hat{\mathbf{f}} = W_N \mathbf{f}$, waarbij $\mathbf{f} = (f_0, \ldots, f_{N-1})$ en $\hat{\mathbf{f}} = (\hat{f}_0, \ldots, \hat{f}_{N-1})$ vectoren zijn van lengte N en $W_N = \{w_N^{(k,j)}\}$ een $N \times N$ matrix is. De vector $\mathbf{f}$ bevat de gegevens die getransformeerd moeten worden en de vector $\hat{\mathbf{f}}$ bevat de getransformeerde gegevens. In het geval van de Fourier transformatie is $W_N = F_N$ de zogenaamde Fourier matrix met elementen $w_N^{(k,j)} = \exp(2\pi ijk/N)$. In het geval van de cosinus transformatie is $w_N^{(k,j)} = \cos(\pi(j + \frac{1}{2})k/N)$ en in het geval van de Legendre transformatie is $w_N^{(k,j)} = \frac{1}{N} P_k(\cos(\pi(j + \frac{1}{2})/N))$, waarbij P_k het k-de Legendre polynoom is.

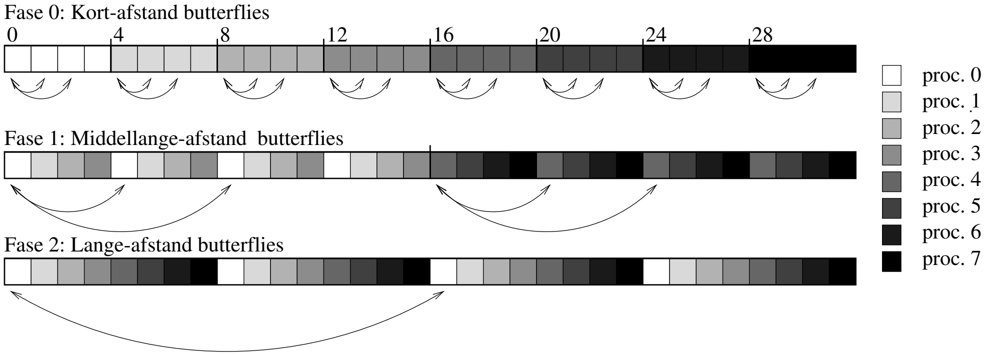
We hebben onze parallelle algoritmen ontworpen met behulp van het *bulk synchrone parallelle* (BSP) model. Het BSP model geeft ons een eenvoudige en effectieve manier om overdraagbare parallelle algoritmen te construeren: het model hangt niet af van een specifieke computerarchitectuur en het bevat een eenvoudige kostenfunctie die het mogelijk maakt tussen algoritmen te kiezen zonder deze daadwerkelijk te moeten implementeren. In het BSP model [**54**] bestaat een computer uit een collectie van p processoren, ieder met een eigen geheugen, die verbonden zijn door een communicatienetwerk dat de processoren toegang verleent tot de geheugens van de andere processoren. In dit model bestaan algoritmen uit een aantal opeenvolgende superstappen. In de modelvariant die wij gebruiken bestaat een *superstap* ofwel uit een aantal rekenstappen, ofwel uit een aantal communicatiestappen. Om de correcte uitvoering van het algoritme te garanderen wordt elke communicatiesuperstap voorafgegaan en/of gevolgd door een globale synchronisatiebarrière, d.w.z. een moment in het algoritme waar de processoren op elkaar moeten wachten. Het gebruik van superstappen legt een sequentiële structuur op aan parallelle algoritmen. Dit vereenvoudigt het ontwerpproces aanmerkelijk.

In Hoofdstuk 1 beschrijven we het BSP model meer gedetailleerd en tevens behandelen we de meest relevante aspecten van parallel rekenen. Hoofdstukken 2 t/m 4 zijn gewijd aan het afleiden van parallelle algoritmen voor discrete transformaties. In de afleiding van ieder algoritme worden de aspecten aan de orde gesteld die leiden tot een efficiënte implementatie. Ieder algoritme wordt gepresenteerd in de vorm van een sjabloon, een zeer gedetailleerd algoritme dat zó geïmplementeerd kan worden. Verder worden er resultaten betreffende nauwkeurigheid, efficiëntie en schaalbaarheid gepresenteerd.

Ons CFFT algoritme is een parallelle versie van het bekende radix-2 FFT algoritme [**16, 55**], dat aanneemt dat N een macht van twee is. Het sequentiële radix-2 FFT algoritme begint met een bitomkeringspermutatie van de invoervector en vervolgt met $\log_2 N$ zogenaamde butterfly stappen. Elk van deze stappen bestaat uit N/K maal een butterfly berekening die $K/2$ paren $(f_j, f_{j+K/2})$ op afstand $K/2$ wijzigt volgens de formule

$$\begin{pmatrix} f_j \\ f_{j+K/2} \end{pmatrix} \leftarrow \begin{pmatrix} f_j + e^{2\pi ij/K} \cdot f_{j+K/2} \\ f_j - e^{2\pi ij/K} \cdot f_{j+K/2} \end{pmatrix}.$$

We nemen aan dat het aantal processoren $p < N$ een macht van twee is en dat de invoer/uitvoer vector in de blokdistributie is (d.w.z. de vector is in p stukken van gelijke lengte opgehakt en elk stuk is aan een processor toegekend). Het parallelle algoritme begint in deze situatie met een parallelle bitomkeringspermutatie (die een communicatiesuperstap vormt) en vervolgens voert het de butterfly stappen uit door reken- en communicatie-superstappen af te wisselen. In een communicatiesuperstap worden de gegevens hergedistribueerd zodanig dat de daaropvolgende rekensuperstap lokaal is. De distributies die in het algoritme voorkomen behoren tot de familie die wij de *groep-cyclische distributiefamilie* hebben genoemd, genoteerd als $\mathrm{C}^r(p, N)$. Deze familie bevat de bekende cyclische en blok-distributie als speciale gevallen, $\mathrm{C}^1(p, N)$ en $\mathrm{C}^p(p, N)$. Elke rekensuperstap is een fase; deze bevat tot $\log_2(N/p)$ butterfly stappen. In totaal worden er $H = \lceil \log_2 N / \log_2(N/p) \rceil = \lceil \log_{\frac{N}{p}} N \rceil$ fases uitgevoerd. (Het aantal fases H is het grootste gehele getal waarvoor $(N/p)^{H-1} \leq N$.) In fase 0 worden de eerste $\log_2(N/p)$ stappen uitgevoerd in de blokdistributie. Deze fase bestaat uit de butterfly stappen met $K \leq N/p$ (die we de korte-afstand butterflies noemen). Daarna wordt in elke middenfase $1 \leq J < H - 1$ een serie van $\log_2(N/p)$ butterfly stappen met middellange-afstand butterflies uitgevoerd in de cyclische distributie beperkt tot een groep van telkens $(N/p)^J$ processoren. Tot slot worden in fase $H-1$ de resterende


Fase 0: Kort-afstand butterflies
0
4
8
12
16
20
24
28
Fase 1: Middellange-afstand butterflies
Fase 2: Lange-afstand butterflies
proc. 0
proc. 1
proc. 2
proc. 3
proc. 4
proc. 5
proc. 6
proc. 7

FIGUUR 1. Butterfly operaties met gebruik van de groep-cyclische distributie familie $\mathrm{C}^r(p, N) = \mathrm{C}^r(8, 32)$. De korte-afstand butterflies worden uitgevoerd in de blokdistributie, de middellange-afstand butterflies in de $\mathrm{C}^2(8, 32)$ distributie en de lange-afstand butterflies in de cyclische distributie. Voor de duidelijkheid worden slechts een aantal butterfly-paren getoond.

lange-afstand butterflies uitgevoerd in de cyclische distributie over p processoren. Dit proces wordt geïllustreerd in Figuur 1.

In Hoofdstuk 2 leiden we het bovengeschetste algoritme af. Verder laten we zien hoe het algoritme gewijzigd kan worden om invoervectoren te accepteren die niet in de blokdistributie zijn. In het bijzonder laten we in het geval van de cyclische distributie zien dat de communicatiekosten drastisch gereduceerd kunnen worden. We behandelen tevens de vraag hoe we een cache-vriendelijke versie van ons algoritme kunnen verkrijgen, d.w.z. een algoritme dat de cache (een klein maar snel extra geheugen) van de computer optimaal benut door de berekening in kleine stukjes op te breken zodat de gegevens die in de cache zijn volledig worden benut voordat nieuwe gegevens worden binnengebracht. Als laatste onderwerp in dit hoofdstuk introduceren we een nieuw *gemengde-radix parallel FFT algoritme* dat gebaseerd is op werk van Agarwal en Cooley [**1**] voor vectorcomputers. Dit algoritme is veelbelovend omdat het FFT's met elke combinatie van p en $N = N_0 N_1 \dots N_{H-1}$ aankan mits p een deler is van iedere factor N_l en sequentiële FFT's van lengte N_l beschikbaar zijn.

Algoritmen voor de RFFT en de FCT kunnen afgeleid worden door: (1) de N reële invoergegevens in te pakken in een complexe vector van lengte $N/2$, (2) deze vector te transformeren door middel van een CFFT van lengte $N/2$, en daarna (3) de gewenste uitvoer uit de getransformeerde data te halen. Dergelijke algoritmen hebben rekenkosten die ruwweg de helft zijn van die van een CFFT van lengte N. De inpakfase van beide algoritmen is een permutatie, die uitgevoerd kan worden zonder extra communicatiekosten door deze te combineren met de bitomkeringsoperatie van de CFFT. De uitpakfase is een paarsgewijze operatie die elementenparen $(j, N-j)$ combineert. Om deze operatie zonder extra communicatiekosten uit te voeren laten we de lange-afstand butterflies geschieden in de *zig-zag cyclische distributie*, een variant van de cyclische distributie die elementenparen $(j, N-j)$ op dezelfde processor zet. In Hoofdstuk 3 beschrijven we parallelle RFFT en FCT algoritmen voor het geval $p < \sqrt{N/2}$. We laten ook zien hoe we de transformaties kunnen inverteren en we leiden een nieuw algoritme af dat de FCT van twee vectoren tegelijkertijd berekent.

Driscoll en Healy introduceerden in 1989 een snel polynomiaal transformatie-algoritme dat de discrete Legendre transformatie berekent in $O(N \log^2 N)$ rekenkundige operaties [**21, 22**]. Het algoritme start met het berekenen van de Chebyshev coëfficiënten van de invoervector door middel van een discrete Chebyshev transformatie en voert daarna de eigenlijke berekening uit in $\log_2 N$ stappen, waarbij N een macht van twee is. (De discrete Chebyshev transformatie lijkt op de DCT en kan in $O(N \log N)$ rekenkundige operaties berekend worden gebruik makend van een FCT.) Elk stap k, $1 \leq k < \log_2 N$, van het FLT algoritme heeft als invoer 2^{k-1} polynomen van graad minder dan $N/2^{k-1}$ en berekent als uitvoer een even grote nieuwe collectie van polynomen. Dit gebeurt door middel van een recurrentie procedure, een FCT en een inverse FCT. Aan het eind van elk stap worden de oude en de nieuwe polynomen afgekapt tot de helft van de oorspronkelijke grootte. De totale hoeveelheid gegevens blijft dus constant. Een basis parallelle versie kan nu op de voor de hand liggende wijze worden ontworpen [**30, 45**]. In de eerste stappen, met $k \leq \log_2 p$, zijn er $2^{k-1} < p$ onafhankelijke problemen zodat een groep van $p/2^{k-1}$ processoren zich met een probleem moet bezighouden. Communicatie tussen de processoren is nodig maar beperkt zich tot twee instanties: (1) in de parallelle FCT's en (2) bij het afkappen. In de latere stappen, met $k > \log_2 p$, zijn er $2^{k-1} \geq p$ onafhankelijke problemen zodat elke processor zich kan wijden aan zijn eigen problemen, zonder te hoeven communiceren. In Hoofdstuk 4 leiden we het sequentiële en basis parallelle FLT algoritme af en vervolgens verbeteren we de parallelle versie door de communicatie met een factor drie te verminderen.

Curriculum Vitae

Márcia Alves de Inda was born in August 19, 1969 in Porto Alegre, one of the main cities of the south of Brazil. In 1987, she finished high school obtaining the title of Technician in Human Nutrition. During her high school years, she developed a taste for chemistry that led her to ingress, in 1987, at the Universidade Federal do Rio Grande do Sul (UFRGS, Brazil) as a student in chemistry. In 1990, aware of the importance of mathematics as a fundamental tool for the understanding of chemical and physical phenomena, she decided to redirect her carrier towards mathematics, and in 1993 she finished her B.Sc. in Computational and Applied Mathematics at the same university. In 1994, interested in applying her knowledge in mathematics to chemistry, she started an interdisciplinary research in these fields supervised by Prof Dr Mark Thompson (from the Mathematics Department, UFRGS) and Prof Dr Dimitrios Samios (from the Chemistry Department, UFRGS). In the following year, she completed this work with a master dissertation entitled "Monte Carlo simulation of the cluster-cluster aggregation model of Samios and Netz." At the same year (1995) she was awarded a fellowship from the Fundação Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) and joined the group of Prof Dr Henk A. van der Vorst at the Mathematics Department of the Utrecht University (Utrecht, The Netherlands) as a Ph.D. student. Under daily supervision of Dr Rob H. Bisseling, she carried out research on Monte Carlo simulation of polymers and on the development of parallel algorithms for discrete transforms.

List of Publications

1) M. A. Inda, M. Thompson, and D. Samios, Algoritmo Monte Carlo para estudo de processos reativos. In *Anais do XVIII Congresso Nacional de Matemática Aplicada e Computacional, CNMAC95, Volume I*, 28 August – 1 September 1995, Curitiba, Brazil, pages 338–342.
2) D. Samios, J. Schifino, M. A. Inda, and M. Thompson, Network formation as a cluster-cluster diffusion-limited process, 2: Modeling the polyethylene crosslinking process using Monte Carlo and graph techniques. *Macromol. Theory Simul.* 7:349–353, 1998.
3) M. A. Inda, R. H. Bisseling, and D. K. Maslen, Parallel fast Legendre transform. in W. Zwieflhfer and N. Kreitz, editors, *Towards Teracomputing. Proceedings of the eighth ECMWF workshop on the use of parallel processors in meteorology*, Reading, UK, 16 – 20 November 1998, pages 87–108. World Scientific, Singapore, 1999.
4) M. A. Inda, R. H. Bisseling, and D. K. Maslen, On the efficient parallel computation of Legendre transforms. Submitted to *SIAM Journal on Scientific Computing.*
5) M. A. Inda and R. H. Bisseling, A simple and efficient parallel FFT algorithm using the bulk synchronous parallel model. Part of Chapter 1 of this thesis, will be published elsewhere.
6) M. A. Inda and R. H. Bisseling, Mixed-Radix parallel fast Fourier transform. Part of Chapter 1 of this thesis, will be published elsewhere.
7) M. A. Inda and R. H. Bisseling, Communication efficient parallel real-valued FFT and fast cosine transforms. Chapter 2 of this thesis, will be published elsewhere.
8) M. A. Inda and D. Frenkel, Enhancing the Rosenbluth sampling using the multiple histogram method. In preparation.

Acknowledgments

I am much indebted to Dr Rob H. Bisseling from whom I have learned a great deal. His dedication, tolerance, and friendship meant a lot to me. Furthermore, his keen and painstaking revision of my thesis is priceless. I also wish to thank Prof Dr Henk A. van der Vorst for welcoming me into his group; Dr David K. Maslen for his collaboration in the FLT problem; Prof Dr Richard D. Gill for helping me with statistical methods; Prof Dr Daan Frenkel (AMOLF) for the work in the multiple histogram method; and Dr Igal Szleifer and Dr Javier E. Satulovsky (Purdue University, USA) for the work in tethered polymers. Thanks to Prof Dr A. W. Heemink (TUDelft), Prof Dr N. Petkov (RuG), Prof Dr P. Sloot (UvA), Dr C. Temperton (ECMWF, UK), and Dr J. Lukkien of the reading committee for the useful comments on the thesis.

Doing a Ph.D. abroad was rewarding experience. Here I had the opportunity to meet and interact with people of different countries and with different backgrounds. This interaction made me learn a little bit about each of those different cultures and a lot about my own culture. This would have been impossible without the financial support of CAPES, Brazil, and, in the last seven months, of the Department of Mathematics of Utrecht University. I also would like to thank the NCF for funding the computer time on the Cray T3E, and the HPαC, Delft, for the technical support.

Working with the staff of the Mathematics Department of the Utrecht University was very pleasant. In special, I would like to mention my roommates Wim Bomhof, Roderik Lindenbergh, and Ellen Meijerink. I also would like to thank my former roommate Dr. Martin van Gijzen. His friendship was very important throughout this period. He was also kind enough to proofread my thesis. As I write these words, my husband Paulo Dani is giving a final look at my thesis to see if he can catch any remaining error. I cannot express in words how much his companionship and support helped me to achieve this goal.

Márcia Alves de Inda
Utrecht, February 18, 2000