A simple and efficient parallel FFT algorithm using the BSP model[⋆]

Márcia A. Inda

FOM Institute for Atomic and Molecular Physics (AMOLF), Kruislaan 407, 1098 SJ Amsterdam, The Netherlands

and

Instituto de Matemática, Universidade Federal do Rio Grande do Sul, Av. Bento Gonçalves 9500, 91509-900 Porto Alegre, RS, Brazil

Rob H. Bisseling

Department of Mathematics, Utrecht University, PO Box 80010, 3508 TA Utrecht, The Netherlands

Abstract

We present a new parallel radix-4 FFT algorithm based on the BSP model. Our parallel algorithm uses the group-cyclic distribution family, which makes it simple to understand and easy to implement. We show how to reduce the communication cost of the algorithm by a factor of three, in the case that the input/output vector is in the cyclic distribution. We also show how to reduce computation time on computers with a cache-based architecture. We present performance results on a Cray T3E with up to 64 processors, obtaining reasonable efficiency levels for local problem sizes as small as 256 and very good efficiency levels for local sizes larger than 2048.

Key words: fast Fourier transform, bulk synchronous parallel
1991 MSC: 65T20, 65Y05

⋆ Inda supported by a doctoral fellowship from CAPES, Brazil. Computer time on the Cray T3E of HPaC, Delft, funded by NCF, The Netherlands
Email addresses: `inda@amolf.nl` (Márcia A. Inda), `Rob.Bisseling@math.uu.nl` (Rob H. Bisseling).
URL: `http://www.math.uu.nl/people/bisseling` (Rob H. Bisseling).

1 Introduction

The *discrete Fourier transform* (DFT) plays an important role in computational science. DFT applications range from solving numerical differential equations to signal processing. (For an introduction to DFT applications, see e.g. [1].) The widespread use of DFTs 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 [2] 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 [3]. 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. [3–14]).

The lack of a unified parallel computing model and the existence of many different parallel architectures have 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 and the related transpose approach (see e.g. [5,6,9–11]). Those algorithms regard the input vector of size $N = N_0N_1$ as an $N_0 \times N_1$ matrix, and they carry out the computations in a similar way as done for two-dimensional FFTs. Those algorithms require $p \leq \min(N_0, N_1)$ and hence in particular $p \leq \sqrt{N}$ must hold.

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 $N = 2^{qr}$ and $p = 2^{(q-1)s}$ with q, r, s integers with $s \leq r$ [12, Chap. 10.3] or even $s = r$ [13]. Furthermore, to our knowledge none of those algorithms were implemented.

Our main aim in this paper is to present a new parallel FFT algorithm and its implementation. Our parallel algorithm works for any $p < N$ as long as both N and p are powers of two, which is required because of the radix-2 framework. (A mixed-radix framework is discussed in [15].) In Section 2, we briefly introduce the basic framework of radix-2 and radix-4 FFTs and the bulk synchronous parallel (BSP) model. In Section 3, 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 distributed memory parallel FFT algorithm which is easy to implement. In Section 4, we present a set of subroutines that can be used in the implementation of the algorithm. In Section 5, 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 takes advantage of the cache memory of a computer 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 6, we present results regarding the performance of our implementation and discuss aspects such as the *cache effect*. In Section 7, we draw our conclusions and discuss future work.

2 Background

2.1 *The fast Fourier transform*

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

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

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

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

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

$$\mathbf{Z} = F_N \cdot \mathbf{z}. \tag{3}$$

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

$$w_N = \mathrm{e}^{\frac{2\pi \mathrm{i}}{N}}. \tag{4}$$

For simplicity, we will restrict our discussion to values of N that are powers of two, which is a requirement of the radix-2 framework. The sequential iterative radix-2 FFT algorithm starts with the so-called *bit reversal* permutation of the input vector (see Section 4.2), and proceeds in $\log_2 N$ butterfly stages, $A_{K,N}$ (numbered $K = 2, 4, 8, \ldots, N$) as described in Algorithm 1.

Algorithm 1 *Sequential radix-2 FFT algorithm.*
Input $\mathbf{y} = (y_0^{\text{in}}, \ldots, y_{N-1}^{\text{in}})$*: Complex vector of size* N*;* N *is a power of* 2 *with* $N \geq 2$.
Output $\mathbf{y} \leftarrow (y_0^{\text{out}}, \ldots, y_{N-1}^{\text{out}})$*, where* $y_k^{\text{out}} = \sum_{j=0}^{N-1} y_j^{\text{in}} \exp(2\pi \mathrm{i} jk/N)$.

```
Step 1.  Perform a bit reversal on $\mathbf{y}$.
Step 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
                     $\begin{pmatrix} z_{t+j} \\ z_{t+j+K/2} \end{pmatrix} \leftarrow \begin{pmatrix} z_{t+j} + w_K^j \cdot z_{t+j+K/2} \\ z_{t+j} - w_K^j \cdot z_{t+j+K/2} \end{pmatrix}$
             $K \leftarrow 2 \cdot K$
```

Each butterfly stage consists of $(K/2) \cdot (N/K)$ pairwise butterfly computations. These operations cost one complex multiplication and two additions, or 10 real floating point operations (flops), per pair. The total flop count of the radix-2 FFT is therefore

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

Following Van Loan's matrix approach [3], Algorithm 1 can be described as a sequence of sparse matrix-vector multiplications which corresponds to the following decomposition of the Fourier matrix [1]

$$F_N = A_{N,N} \cdots A_{8,N} A_{4,N} A_{2,N} P_N, \tag{5}$$

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

$$A_{K,N} = I_{N/K} \otimes B_K, \tag{6}$$

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

which is shorthand for a block-diagonal matrix $\mathrm{diag}(B_K, \ldots, B_K)$ with N/K copies of the $K \times K$ matrix B_K on the diagonal. The 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 inner loop of step 2 of Algorithm 1. This matrix can be written as

$$B_K = \begin{bmatrix} I_{K/2} & \Omega_{K/2} \\ I_{K/2} & -\Omega_{K/2} \end{bmatrix}. \tag{7}$$

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

$$\Omega_{K/2} = \mathrm{diag}(w_K^0, w_K^1, \ldots, w_K^{K/2-1}). \tag{8}$$

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

$$A_{K,N}^{\alpha} = I_{N/K} \otimes B_K^{\alpha}, \tag{9}$$

where B_K^{α} is the *generalized* $K \times K$ *2-butterfly matrix* [6,16,17]

$$B_K^{\alpha} = \begin{bmatrix} I_{K/2} & \Omega_{K/2}^{\alpha} \\ I_{K/2} & -\Omega_{K/2}^{\alpha} \end{bmatrix}, \tag{10}$$

which has the same form as the original B_K, but with the weights w_K^j in (8) 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-butterfly* stages which involve quadruples of vector components instead of pairs. Since 34 flops are performed per quadruple, this brings the flop count down to

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

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

$$D_K = B_K(I_2 \otimes B_{K/2}) = \begin{bmatrix} I_{K/4} & \Lambda^2_{K/4} & \Lambda_{K/4} & \Lambda^3_{K/4} \\ I_{K/4} & -\Lambda^2_{K/4} & \mathrm{i}\Lambda_{K/4} & -\mathrm{i}\Lambda^3_{K/4} \\ I_{K/4} & \Lambda^2_{K/4} & -\Lambda_{K/4} & -\Lambda^3_{K/4} \\ I_{K/4} & -\Lambda^2_{K/4} & -\mathrm{i}\Lambda_{K/4} & \mathrm{i}\Lambda^3_{K/4} \end{bmatrix}, \tag{11}$$

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

$$\Lambda_{K/4} = \mathrm{diag}(w_K^0, w_K^1, \ldots, w_K^{K/4-1}). \tag{12}$$

This matrix is a symmetrically permuted version of the radix-4 butterfly matrix [3]. [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.

An algorithm for the inverse FFT is obtained using the following property:

$$F_N^{-1} = \frac{1}{N}\bar{F}_N = \frac{1}{N}\bar{A}_{N,N}\cdots\bar{A}_{4,N}\bar{A}_{2,N}P_N. \tag{13}$$

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 is rescaled.

Now, we define the direct product of two matrices and give some properties that will be used in the course of this paper.

Definition 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*

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

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

As one would expect, the direct product is associative, but it is not commutative. Lemma 2 summarizes some direct product properties that follow directly from the definition. (See [3,18] for other useful properties).

Lemma 2 (Properties of the direct product) *The following holds.*

(1) $(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$, *provided the products* AC *and* BD *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* $(I_m \otimes A)(I_m \otimes B) = (I_m \otimes B)(I_m \otimes A)$.

2.2 The bulk synchronous parallel model

The BSP model [19] 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. 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 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.

A BSP computer can be characterized by four global parameters:

- p, the number of processors;
- v, the single-processor 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 has been 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), where h is the maximum number of data elements sent or received by any processor. The cost function of an algorithm can be obtained by adding the flops of the separate supersteps. This yields an expression of the form $a + bg + cl$. For further details and some basic techniques, see [20]. BSPlib [21] is a standard library defined in May 1997 which enables parallel programming in BSP style. The Paderborn University BSP (PUB) library [22] is another library for BSP programming; it provides the extra feature of subset synchronization.

2.3 Parallel radix-2 FFTs

Since the introduction of parallel computers, and even before that, methods for parallelizing FFT algorithms have been proposed [23]. The earliest methods produced parallel algorithms with a communication cost of $O(\log p(\frac{N}{p}g + l))$, see e.g. [7,8,14]. Such methods appeared as a direct consequence of the divide-and-conquer structure of the radix-2 FFT algorithm. Chu and George [7] discuss several parallel algorithms of this type. Restricting the vector size to powers of two, they present a common framework in which all the algorithms they discuss are reorderings of 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 $j = (j_{m-1}, \ldots, j_0)_2$, where $m = \log_2 N$, we observe that elements j and $j + K/2$ differ only in bit $\log_2 K - 1$ and that $w_K^{j \bmod K} = w_K^{(j_{\log_2 K-1},\ldots,j_0)_2}$. 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. If the new ordering can be represented using a permutation of the original bits, it is easy to know which elements to combine and which 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 current butterfly stage is one of the $\log_2(N/p)$ bits that represent the local index, then that stage is local, otherwise communication is needed.

Swarztrauber [14] carries out a similar discussion. He starts with a more general formulation of the problem, where N is not restricted to powers of two, but when discussing the distributed memory framework, he only considers FFTs on a hypercube, restricting both p and N to powers of two. A disadvantage of the algorithms discussed in [7,8,14] 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 are needed, each of cost $O(\frac{N}{p}g + l)$. A less expensive approach is to exchange all the processor bits with a group of local bits corresponding to butterfly stages that have already been performed. Since the communication cost of the permutation that exchanges many bits is of the same order $O(\frac{N}{p}g + l)$ as the cost for exchanging one bit, the reduction in the communication cost is huge.

The basic idea for such algorithms already appears in the original paper by Cooley and Tukey [2]. 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 (1) as

$$Z_{k_1,k_0} = \sum_{j_1=0}^{N_1-1} \left(\sum_{j_0=0}^{N_0-1} z_{j_0,j_1} w_N^{j_0 k_0 N_1} \right) w_N^{j_1(k_1 N_0 + k_0)}, \quad 0 \le k_1 < N_1, 0 \le k_0 < N_0, \tag{14}$$

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 (14) corresponds to a DFT of size N_0, which in turn can be computed by the same procedure as before if N_0 is not prime. The outer sum is similar to a DFT of size N_1. They remark that this procedure can be applied to any possible factorization of N, $N = N_0 \dots N_{H-1}$ 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 of the FFT into a sequence of shorter FFTs that, in the parallel case, can be spread over the processors. This is what happens in our FFT algorithm presented in the next section and in algorithms based on the 6-pass approach and the related transpose approach (see e.g. [5,6,9–11]).

3 The parallel algorithm

3.1 Group-cyclic distribution family and the parallel FFT

Since our parallel FFT algorithm is based on the radix-2 decomposition (5) 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 a power of two. We also assume that $N > p$. Our parallel FFT algorithm makes use of the data distributions defined below.

Definition 3 (Cyclic distribution in r groups, $\mathrm{C}^r(p,N)$) *Let r, p, and N be integers with $1 \leq r \leq p \leq N$, such that r divides 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 the well-known *cyclic distribution* $\mathrm{C}^1(p,N)$ and *block distribution* $\mathrm{C}^p(p,N)$ as extreme cases.

The parallel FFT algorithm works as follows. 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 is the largest integer H for which $(N/p)^{H-1} < N$.) In phase 0, the algorithm uses the block distribution to perform the first $\log_2(N/p)$ butterfly stages, i.e., those involving butterflies with $K \leq N/p$ (which we call short distance butterflies). Afterwards, in each intermediate phase $1 \leq J < H-1$ the r-cyclic distribution is used, with $r = p/(N/p)^J$, to perform a group of $\log_2(N/p)$ butterfly stages with $(N/p)^J < K \leq (N/p)^{J+1}$ (the medium distance butterflies). Finally, in phase $H-1$, the cyclic distribution is used to perform the remaining butterfly stages, i.e., those involving butterflies with $K > (N/p)^{H-1}$ (the long distance butterflies).

Figure 1 illustrates the use of the group-cyclic distribution family in our parallel FFT. The same operations are illustrated in two ways: (A) using the *logical view*, and (B) using 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

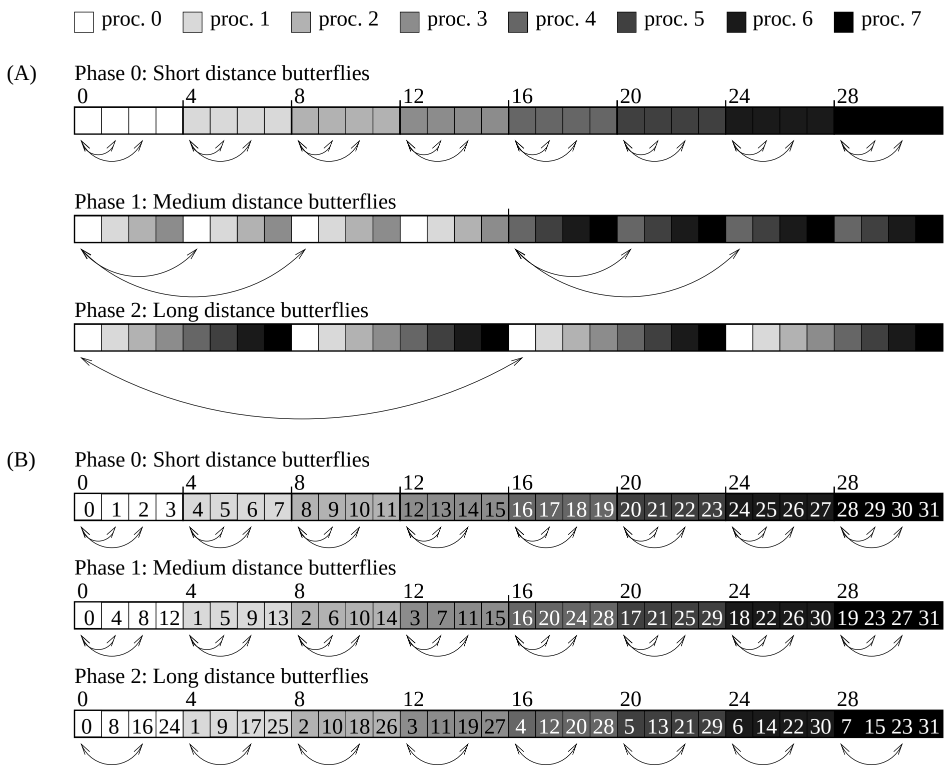

Fig. 1. Butterfly operations using the group-cyclic distribution family $\mathrm{C}^r(p, N) = \mathrm{C}^r(8, 32)$. (A) Logical view, (B) storage view. The short distance butterflies ($A_{2,32}$ and $A_{4,32}$) are performed using the $\mathrm{C}^8(8, 32)$ distribution (block distribution). The medium 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.

are the same.

Our algorithm has only $O(\log N/\log(N/p))$ communication supersteps, each of cost $O(\frac{N}{p}g + l)$, which is a significant improvement over the $O(\log p)$ communication supersteps, also of cost $O(\frac{N}{p}g + l)$, of the algorithms discussed in the previous section. McColl [13] 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 components of the data vector, increasing the communication costs unnecessarily.

3.2 Permutations and permutation matrices

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

$$\begin{aligned}
&\gamma_{u,v,N} : \{0, \ldots, N-1\} \rightarrow \{0, \ldots, N-1\} \\
&j = j_0 \cdot M + j_1 \cdot u + j_2 \mapsto l = j_0 \cdot M + j_2 \cdot \frac{N}{v} + j_1,
\end{aligned} \tag{15}$$

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 $\gamma_{u,v,N}^{-1} = \gamma_{\frac{N}{v},\frac{N}{u},N}$ and that $\gamma_{1,v,N} = \gamma_{u,N,N}$ is the identity permutation. Permuting a vector of size N by $\gamma_{u,v,N}$ can be achieved by dividing the vector into v/u subvectors of size M and then performing a (perfect) shuffle permutation $\sigma_{u,M}$:

$$\begin{aligned}
&\sigma_{u,M} : \{0, \ldots, M-1\} \rightarrow \{0, \ldots, M-1\} \\
&\qquad\qquad j \mapsto k = (j \bmod u) \cdot \frac{M}{u} + j \operatorname{div} u,
\end{aligned} \tag{16}$$

on each of the subvectors. This relation can be expressed by

$$\gamma_{u,v,N}(j) = (j \operatorname{div} M) \cdot M + \sigma_{u,M}(j \bmod M), \tag{17}$$

which implies that $\gamma_{u,u,N} = \sigma_{u,N}$. The shuffle permutations $\sigma_{p,N}$ and $\sigma_{\frac{N}{p},N}$ can be used to permute a vector of size N distributed over p processors from block to cyclic distribution and vice-versa.

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 can be expressed by the $N \times N$ permutation matrix:

$$(\Gamma_{u,p,N})_{lj} = \begin{cases} 1, \text{ if } l = \gamma_{u,p,N}(j), \\ 0, \text{ otherwise.} \end{cases} \tag{18}$$

We also define $S_{u,N} = \Gamma_{u,u,N}$.

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. Note that $\Gamma_{u,p,N} = I_r \otimes S_{u,\frac{N}{p}u}$, cf. (17). 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.

To obtain the parallel FFT algorithm, we modify the original radix-2 decomposition of the Fourier matrix (5) by inserting identity permutation matrices $I_N = \Gamma_u^{-1}\Gamma_u$ corresponding to the changes of distribution, and regrouping the matrices in the resulting decomposition. In the case that $1 < p \leq \sqrt{N}$ and hence $H = 2$, this is done as follows:

$$\begin{aligned} F_N &= \Gamma_p^{-1}\Gamma_p \cdot A_{N,N} \cdot \Gamma_p^{-1}\Gamma_p \cdots \Gamma_p^{-1}\Gamma_p \cdot A_{2\frac{N}{p},N} \cdot \Gamma_p^{-1}\Gamma_p \cdot A_{\frac{N}{p},N} \ldots A_{2,N}P_N \\ &= \Gamma_p^{-1}(\Gamma_p A_{N,N}\Gamma_p^{-1}) \cdots (\Gamma_p A_{2\frac{N}{p},N}\Gamma_p^{-1})\Gamma_p \cdot A_{\frac{N}{p},N} \ldots A_{2,N}P_N. \end{aligned} \tag{19}$$

By defining

$$\hat{A}_{k,u,p,N} = \Gamma_{u,p,N} A_{ku,N} \Gamma_{u,p,N}^{-1} \tag{20}$$

and using the fact that Γ_1 is the identity matrix, we can rewrite (19) as

$$F_N = \Gamma_p^{-1} \cdot \underbrace{\hat{A}_{\frac{N}{p},p,p,N} \cdots \hat{A}_{2\frac{N}{p^2},p,p,N}}_{\text{phase 1}} \cdot \Gamma_p \Gamma_1^{-1} \cdot \underbrace{\hat{A}_{\frac{N}{p},1,p,N} \cdots \hat{A}_{2,1,p,N}}_{\text{phase 0}} \cdot \Gamma_1 P_N. \tag{21}$$

From now on, we denote $\hat{A}_{k,u,p,N}$ by $\hat{A}_{k,u}$, reserving the indices p and N for situations where they are not obvious from the context and for stand-alone definitions. In the general case, not restricted to $p \leq \sqrt{N}$, we arrive at the following decomposition of the Fourier matrix:

$$\begin{aligned} F_N = \Gamma_p^{-1} & \underbrace{\hat{A}_{\frac{N}{p},p} \ldots \hat{A}_{2\frac{(N/p)^{H-1}}{p},p}}_{\text{phase } H-1} \Gamma_p \cdot \Gamma_{(\frac{N}{p})^{H-2}}^{-1} \underbrace{\hat{A}_{\frac{N}{p},(\frac{N}{p})^{H-2}} \ldots \hat{A}_{2,(\frac{N}{p})^{H-2}}}_{\text{phase } H-2} \cdot \\ & \Gamma_{(\frac{N}{p})^{H-2}} \cdots \Gamma_{\frac{N}{p}}^{-1} \underbrace{\hat{A}_{\frac{N}{p},\frac{N}{p}} \ldots \hat{A}_{2,\frac{N}{p}}}_{\text{phase 1}} \Gamma_{\frac{N}{p}} \cdot \Gamma_1^{-1} \underbrace{\hat{A}_{\frac{N}{p},1} \ldots \hat{A}_{2,1}}_{\text{phase 0}} \Gamma_1 \cdot P_N \end{aligned} \tag{22}$$

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

$$\hat{A}_{k,u,p,N} = I_r \otimes \operatorname{diag}(A_{k,n}^{0/u}, A_{k,n}^{1/u}, \ldots, A_{k,n}^{(u-1)/u}), \tag{23}$$

where $r = p/u$, $n = N/p$, and $A_{k,n}^{\alpha}$ was defined previously, cf. (9). We shall formally state this as Corollary 5 which follows from Theorem 4. Figure 2 exemplifies the structure of the matrix $\hat{A}_{k,u}$.

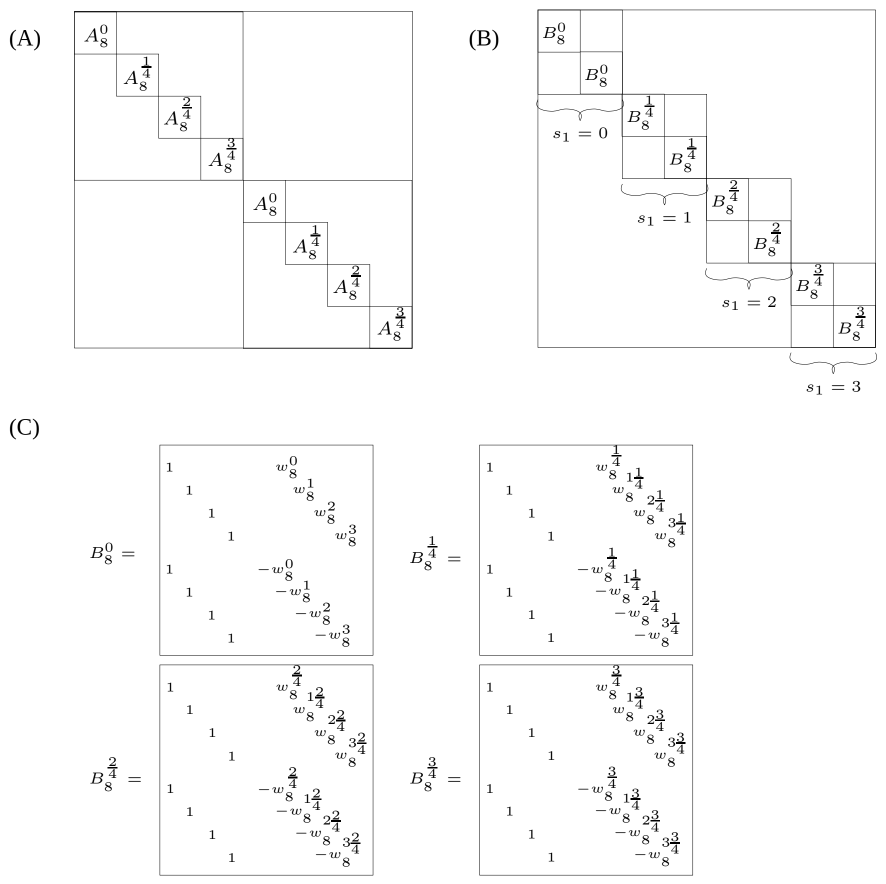

Fig. 2. (A) Structure of the $N \times N$ matrix $\hat{A}_{k,u} = I_r \otimes \mathrm{diag}(A_{k,n}^{0/u}, A_{k,n}^{1/u}, \ldots, A_{k,n}^{(u-1)/u})$. Example with $N = 128$, $p = 8$, $r = 2$, $k = 8$, and hence $u = 4$ and $n = 16$. (For clarity, the $A_{k,n}^{\alpha}$ are depicted as A_k^{α}) (B) Matrix $\mathrm{diag}(A_{k,n}^{0/u}, A_{k,n}^{1/u}, \ldots, A_{k,n}^{(u-1)/u})$. (C) Matrices $B_k^{s_1/u}$, with $s_1 = 0, \ldots, u-1$.

Theorem 4 *Let u, M, and k be powers of two such that $2 \le k \le M/u$. Define $n = M/u$. Then*

$$S_{u,M} A_{ku,M} S_{u,M}^{-1} = \mathrm{diag}(A_{k,n}^{0/u}, A_{k,n}^{1/u}, \ldots, A_{k,n}^{(u-1)/u}).$$

PROOF. See Appendix A. $\square$

Corollary 5 *Let r, p, and N be powers of two with $1 \le r \le p < N$. Define $u = p/r$ and $n = N/p$. Let k be a power of two with $2 \le k \le n$. Then*

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

PROOF. Define $M = N/r$. Then

$$\hat{A}_{k,u,p,N} = \Gamma_{u,p,N} A_{ku,N} \Gamma_{u,p,N}^{-1} = (I_r \otimes S_{u,M})(I_r \otimes A_{ku,M})(I_r \otimes S_{u,M}^{-1})$$

$$= I_r \otimes (S_{u,M} A_{ku,M} S_{u,M}^{-1}) = I_r \otimes \mathrm{diag}(A_{k,n}^{0/u}, A_{k,n}^{1/u}, \ldots, A_{k,n}^{(u-1)/u}).$$

□

Starting from the Fourier matrix decomposition (22), 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, every 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 (22) separates computation and communication completely: each generalized butterfly phase $(\hat{A}_{\frac{N}{p},u} \ldots \hat{A}_{2,u}) \cdot \mathbf{y}$ is a computation superstep, whereas each permutation $\Gamma_u \cdot \mathbf{y}$ is a communication superstep. In the next section we give a complete description of the resulting parallel algorithm.

4 Implementation of the parallel algorithm

We describe our parallel algorithms with a high level of detail. As a result, they are ready to be implemented, though not necessarily completely optimized. The following list introduces the terminology used in describing our parallel algorithms.

- **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*.
- **Indexing.** All the indices of vectors are global. This means that vector 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 vectors in an unambiguous manner, even though the vector is distributed and each processor has only part of it.
- **Communication.** Communication between processors is indicated using

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

  This operation puts n elements of vector $\mathbf{f}$, starting from element i, into processor *pid* and stores them there in vector $\mathbf{g}$ starting from element j. Subscripts are not needed when the first element of the vector 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 a scalar. All puts are assumed to be buffered, so that they are safely carried out, even if $\mathbf{f}$ and $\mathbf{g}$ happen to be the same.

Algorithm 2 is a direct implementation of the matrix decomposition (22). Note that to obtain the normalized inverse transform, the output vector must be divided by N. The subroutines used in the FFT algorithm are described in the following subsections.

Algorithm 2 *Parallel fast Fourier transform.*
Call BSP_FFT$(s, p, sign, N, \mathbf{y})$.

Input $\mathbf{y} = (y_0^{\text{in}}, \dots, y_{N-1}^{\text{in}})$*: Complex vector of size* N*, block distributed over* p *processors;* p *and* N *are powers of two with* $p < N$*;* s *is the processor identification number with* $0 \le s < p$*; sign is the transform direction.*

Output $\mathbf{y} \leftarrow (y_0^{\text{out}}, \dots, y_{N-1}^{\text{out}})$*, where* $y_k^{\text{out}} = \sum_{j=0}^{N-1} y_j^{\text{in}} \exp(sign \cdot 2\pi \mathrm{i} jk/N)$.

```
1^CpCm   Parallel bit reversal permutation: $\mathbf{y} \leftarrow P_N \cdot \mathbf{y}$.
         BSP_BitRev$(s, p, N, \mathbf{y})$
2^Comp   Phase 0, short distance butterflies: $\mathbf{y} \leftarrow (A_{\frac{N}{p},N} \dots A_{2,N}) \cdot \mathbf{y}$.
         BTFLY$(0, sign, \frac{N}{p}, 4, \mathbf{y}_{\mathbf{s}\frac{\mathbf{N}}{\mathbf{p}}})$
3^CpCm   Permutation to the r-cyclic distribution: $\mathbf{y} \leftarrow \Gamma_{p/r} \cdot \mathbf{y}$,
         with $r = \max(1, p/(N/p))$.
         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^Comp       Phase J, medium distance butterflies: $\mathbf{y} \leftarrow (\hat{A}_{\frac{N}{p},(\frac{N}{p})^J} \dots \hat{A}_{2,(\frac{N}{p})^J}) \cdot \mathbf{y}$.
             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^Comm       Permutation to the r-cyclic distribution: $\mathbf{y} \leftarrow (\Gamma_{\frac{p}{r}} \Gamma^{-1}_{(\frac{N}{p})^J}) \cdot \mathbf{y}$,
             with $r = \max(1, p/(N/p)^{J+1})$.
             BSP_CyclicToCyclic$(s, p, N, (N/p)^J, \min(p, (N/p)^{J+1}), \mathbf{y})$
6^Comp   Phase H − 1, long distance butterflies: $\mathbf{y} \leftarrow (\hat{A}_{\frac{N}{p},p} \dots \hat{A}_{2\frac{(N/p)^{H-1}}{p},p}) \cdot \mathbf{y}$.
         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^CpCm   Permutation to block distribution: $\mathbf{y} \leftarrow \Gamma_p^{-1} \cdot \mathbf{y}$.
         BSP_CyclicToBlock$(0, s, p, \frac{N}{p}, \mathbf{y})$
```

4.1 Generalized butterflies

The sequential subroutine BTFLY, which multiplies the input vector by $A^{\alpha}_{n,n} \dots A^{\alpha}_{k_0,n} A^{\alpha}_{k_0/2,n}$ is described in Algorithm 3.

Algorithm 3 *Sequential generalized butterfly operations.*

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

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

Output $\mathbf{y} \leftarrow A^{\alpha}_{n,n} \cdots A^{\alpha}_{k_0,n} A^{\alpha}_{k_0/2,n} \mathbf{y}$ *for forward* ($sign = 1$)*, and*
$\mathbf{y} \leftarrow \bar{A}^{\alpha}_{n,n} \cdots \bar{A}^{\alpha}_{k_0,n} \bar{A}^{\alpha}_{k_0/2,n} \mathbf{y}$ *for backward* ($sign = -1$)*.*

```
Step 1. Perform pairs of butterfly stages $A^{\alpha}_{k,n} A^{\alpha}_{k/2,n}$.
        $k \leftarrow k_0$
        while $k \le n$ do
            for $t = 0$ to $n - k$ step $k$ do
                Perform 4-butterfly $D^{\alpha}_k = B^{\alpha}_k (I_2 \otimes B^{\alpha}_{k/2})$.
                for $j = 0$ to $k/4 - 1$ do
                    $a \leftarrow y_{t+j} + w_k^{sign \cdot 2(j+\alpha)} \cdot y_{t+j+k/4}$
                    $b \leftarrow y_{t+j} - w_k^{sign \cdot 2(j+\alpha)} \cdot y_{t+j+k/4}$
                    $c \leftarrow w_k^{sign \cdot (j+\alpha)} \cdot y_{t+j+k/2} + w_k^{sign \cdot 3(j+\alpha)} \cdot y_{t+j+3k/4}$
                    $d \leftarrow w_k^{sign \cdot (j+\alpha)} \cdot y_{t+j+k/2} - w_k^{sign \cdot 3(j+\alpha)} \cdot y_{t+j+3k/4}$
                    $y_{t+j} \leftarrow a + c$
                    $y_{t+j+k/4} \leftarrow b + sign \cdot d\mathrm{i}$
                    $y_{t+j+k/2} \leftarrow a - c$
                    $y_{t+j+3k/4} \leftarrow b - sign \cdot d\mathrm{i}$
            $k \leftarrow 4 \cdot k$
Step 2. Perform the last butterfly stage $A^{\alpha}_{n,n}$.
        if $k = 2n$ then
            for $j = 0$ to $n/2 - 1$ do
                $a \leftarrow w_n^{sign \cdot (j+\alpha)} \cdot y_{j+n/2}$
                $y_{j+n/2} \leftarrow y_j - a$
                $y_j \leftarrow y_j + a$
```

If the needed weights are stored in a lookup table, the cost of Algorithm 3 is

$$C_{\mathrm{BTFLY}}(n, k_0) = \frac{17}{4} n \cdot \log_2 \frac{4n}{k_0} + \frac{3}{4} n \cdot \left(\log_2 \frac{4n}{k_0} \bmod 2 \right). \tag{24}$$

The FFT algorithm computes the desired (short, medium, or long distance) butterfly stages corresponding to phase J, $0 \le J < H$, 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 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).

The total computation cost of our parallel FFT, Algorithm 2, is obtained by adding the costs $C_{\mathrm{BTFLY}}(\frac{N}{p}, 4)$ of phases $J = 0$ to $H-2$, where $H = \lceil \log_{\frac{N}{p}} N \rceil$, and $C_{\mathrm{BTFLY}}(\frac{N}{p}, 4\frac{(N/p)^{H-1}}{p})$ of the last phase $H-1$. (If $H = \log_{\frac{N}{p}} N$, then $(N/p)^{H-1} = p$, and the cost of the last phase is also $C_{\mathrm{BTFLY}}(\frac{N}{p}, 4)$.) This gives a total cost of

$$C_{\mathrm{FFT,par,Comp}}(N,p) = \frac{17}{4}\frac{N}{p}\log_2 N + \frac{3}{4}\frac{N}{p}[(\log_2 \frac{N}{p} \bmod 2)\lfloor \log_{\frac{N}{p}} N \rfloor + (\log_2 N \bmod \log_2 \frac{N}{p}) \bmod 2], \quad (25)$$

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 4.5 after we discuss the parallel permutation subroutines.

4.2 Parallel bit reversal

The bit reversal matrix P_N is defined by

$$(P_N)_{jk} = \begin{cases} 1, \text{ if } j = \mathrm{rev}_N(k), \\ 0, \text{ otherwise.} \end{cases} \quad (26)$$

Here, rev_N is the bit reversal permutation

$$\begin{aligned} \mathrm{rev}_N : \{0,\ldots,N-1\} &\rightarrow \{0,\ldots,N-1\} \\ j = \sum_{l=0}^{m-1} b_l 2^l &\mapsto k = \sum_{l=0}^{m-1} b_{m-l-1} 2^l, \end{aligned} \quad (27)$$

where $m = \log_2 N$ and $(b_{m-1}\ldots b_0)_2$ 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 6 *Let $u = 2^q$ and $N = 2^m$, with $q \leq m$. Then*

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

PROOF. Straightforward. □

Corollary 7 *Let $u \leq N$ be powers of two. Then $P_N = (I_u \otimes P_{\frac{N}{u}})(P_u \otimes I_{\frac{N}{u}})S_{u,N}$.*

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

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

□

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 7 with $u = p$, it is possible to split the parallel bit reversal permutation into two parts as shown in Algorithm 4. The first part sends the elements to the final destination processors, but with the 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 a basis the block distribution, we use from now on $\mathrm{Proc}(k) = k \operatorname{div} \frac{N}{p}$ 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. The second part permutes the local indices t':

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

Algorithm 4 *Parallel bit reversal.*
Call BSP_BitRev$(s, p, N, \mathbf{y})$.

Input $\mathbf{y} = (y_0, \ldots, y_{N-1})$: *Complex vector of size N, block distributed over p processors; p and N are powers of two with $p < N$; s is the processor identification number with $0 \leq s < p$.*

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

```
1^Comm  Global permutation: $\mathbf{y} \leftarrow (P_p \otimes I_{\frac{N}{p}})S_{p,N} \cdot \mathbf{y}$.
        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: $\mathbf{y} \leftarrow (I_p \otimes P_{\frac{N}{p}}) \cdot \mathbf{y}$.
        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 we combine the local bit reversal (superstep 2 of Algorithm 4) with the short distance butterfly phase (superstep 2 of Algorithm 2), we obtain 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. If $p < N/p$, it is possible to optimize superstep 1 of Algorithm 4 by sending packets of data. This is done in a similar way as when permuting from block to cyclic distribution (see Section 4.4); the only difference is in the destination processor, which is $\mathrm{rev}_p(j \bmod p)$ instead of $j \bmod p$.

4.3 Permutations within the group-cyclic distribution 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 sizes, not necessarily powers of two, 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 the two permutations into one:

$$\begin{aligned} \gamma_{u_1}^{u_2} : \{0, \ldots, N-1\} &\rightarrow \{0, \ldots, N-1\} \\ j &\mapsto l = \gamma_{u_2}(\gamma_{u_1}^{-1}(j)). \end{aligned} \tag{28}$$

(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. Algorithm 5 implements this case.

Algorithm 5 *Parallel permutation from r_1-cyclic to r_2-cyclic distribution.*
Call BSP_CyclicToCyclic$(s, p, N, u_1, u_2, \mathbf{y})$.

Input $\mathbf{y} = (y_0, \ldots, y_{N-1})$*: Complex vector of size N, block distributed over p processors; s is the processor identification number with $0 \leq s < p$; u_1 and u_2 are the number of processors in the old group, $u_1 = p/r_1$, and in the new group, $u_2 = p/r_2$, respectively.*

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

```
1^Comm  Global permutation $\gamma_{u_1}^{u_2}$.
```

```
    for $j = s\frac{N}{p}$ to $s\frac{N}{p} + \frac{N}{p} - 1$ do
        $l \leftarrow \gamma_{u_1}^{u_2}(j)$
        $y_l \leftarrow \mathrm{Put}(l \,\mathrm{div}\, \frac{N}{p}, 1, y_j)$
    Synchronize
```

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 and is discussed in Section 4.4.

4.4 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 $p < N/p$, both $\sigma_{p,N}$ and $\sigma_{p,N}^{-1}$ can be optimized by sending packets of size N/p^2, where we assume that p^2 divides N.

For $\sigma_{p,N}$, this is done as follows. Let $b = N/p$. First, we perform a local permutation $\sigma_{p,b}$ on the local index j',

$$j' \to t' = j' \bmod p \cdot \frac{b}{p} + j' \,\mathrm{div}\, p.$$

Then we perform a global *cyclic permutation of packets* on the global index $t = t_0 \cdot b + t_1 \cdot \frac{b}{p} + t_2$,

$$t \to k = \underbrace{t_1}_{\mathrm{Proc}(k)} \cdot b + \underbrace{t_0 \cdot \frac{b}{p} + t_2}_{k'} . \tag{29}$$

This method is illustrated in Figure 3. To verify that it indeed achieves the desired permutation, we substitute $t_0 = j \,\mathrm{div}\, b$, $t_1 = (j \bmod b) \bmod p$, and $t_2 = (j \bmod b) \,\mathrm{div}\, p$ into (29), obtaining

$$\begin{aligned} k &= (j \bmod b) \bmod p \cdot b + j \,\mathrm{div}\, b \cdot \frac{b}{p} + (j \bmod b) \,\mathrm{div}\, p \\ &= j \bmod p \cdot b + (j \,\mathrm{div}\, b \cdot b + j \bmod b) \,\mathrm{div}\, p \\ &= j \bmod p \cdot b + j \,\mathrm{div}\, p \\ &= \sigma_{p,N}(j). \end{aligned}$$

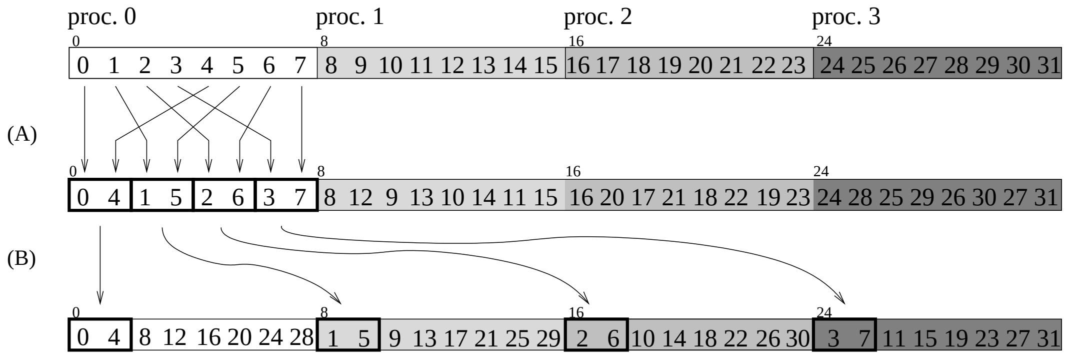

Fig. 3. Two-stage permutation from block to cyclic distribution (storage view). Example with $N = 32$ and $p = 4$. (A) Local $\sigma_{4,8}$ permutation. (B) Global cyclic permutation of packets of size 2.

Algorithm 6 uses the idea described above to permute from block to cyclic distribution within a group of u processors, numbered $s_0, s_0+1, \ldots, s_0+u-1$. The corresponding subroutine is called with $u = \min(p, N/p)$ in superstep 3 of the FFT algorithm, where it performs the permutation for all the $p/u = r$ groups simultaneously. This achieves the desired permutation by $\gamma_{u,p,N}$, because permuting a vector of size N by $\gamma_{u,p,N}$ is the same as dividing it into r subvectors of size $M = N/r$ and then performing a shuffle permutation $\sigma_{u,M}$ on each of the subvectors, cf. (17).

Algorithm 6 *Parallel permutation from block to cyclic distribution within a group of processors.*
Call BSP_BlockToCyclic($s_0, s_1, u, b, \mathbf{y}$).

Input s_0, s_1: *Processor offset and processor identification within group;* $0 \le s_1 < u$.
$\mathbf{y} = (y_0, \ldots, y_{ub-1})$: *Complex vector of size ub, block distributed within a group of u processors; the local block size, b, is a multiple of u, if* $u < b$.

Output $\mathbf{y} \leftarrow S_{u,ub}\mathbf{y}$.

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

```
        $y_{proc\cdot b+s_1\cdot\frac{b}{u}} \leftarrow \mathrm{Put}(s_0+proc, \frac{b}{u}, x_{s_1\cdot b+proc\cdot\frac{b}{u}})$
    Synchronize
```

Subroutine BSP_CyclicToBlock, which carries out $\gamma^{-1}_{u,p,N}$ is obtained by inverting subroutine BSP_BlockToCyclic. The algorithm starts by performing a global cyclic permutation of packets within a group of u processors, and then it carries out a local permutation $\sigma^{-1}_{u,b}$. (The algorithm is not presented here; see [15] for more details.)

4.5 BSP cost

To compute the total cost of our parallel FFT algorithm (Algorithm 2) we need to sum the computation, communication, and synchronization costs. The computation costs were already obtained in Section 4.1. To simplify the final result, we only include the higher order term of the total computation cost (25),

$$C_{\mathrm{FFT,par,Comp}}(N,p) = \frac{17}{4}\frac{N}{p}\log_2 N, \tag{30}$$

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, one synchronization is needed, giving a total cost of

$$C_{\mathrm{permut}}(N,p) = 2\frac{N}{p}\cdot g + l \tag{31}$$

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

$$C_{\mathrm{FFT,par}}(N,p) = \frac{17}{4}\frac{N}{p}\log_2 N + 2\frac{N}{p}(\lceil \log_{\frac{N}{p}} N\rceil + 1)\cdot g + (\lceil \log_{\frac{N}{p}} N\rceil + 1)\cdot l. \tag{32}$$

With this cost function, we can answer questions such as: is it ever worthwhile to use more than $p=\sqrt{N}$ processors? The answer would already be positive if

$$C_{\mathrm{FFT,par}}(p^2,p) > C_{\mathrm{FFT,par}}(p^2,2p). \tag{33}$$

One should realise that the values of g and l may grow with p, so that appropriate values $g(p)$ and $l(p)$ must be used in the evaluation of (32). For the interesting case $p \geq 8$, the criterion becomes

$$\frac{17}{4} p \log_2 p + p(6g(p) - 4g(2p)) + 3l(p) - 4l(2p) > 0. \tag{34}$$

Note that if g does not grow too fast with p, i.e., if $g(2p) < 1.5g(p)$, then the communication time decreases. The number of synchronizations increases from three to four, and if $l(2p) \geq l(p)$, which is most likely, this will cause an increase in synchronization time. Still, if l does not grow too fast and p and N are sufficently large, the savings in computation time will be larger than the additional synchronization time.

In Section 6, we discuss the validity of cost function (32) as an accurate estimator of the true cost of the FFT algorithm, and we also examine the use of more than $\sqrt{N}$ processors for our test machine.

5 Variants of the algorithm

5.1 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. FFT applications exist, however, where a different distribution of the I/O vector is preferred or where the distribution can be freely chosen (see e.g. [10,24,25]). 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 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, the algorithm is modified 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 replacing 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 already 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. This saves two thirds of the total communication cost in the common case that $p \leq N/p$, leaving only one permutation in the middle of the computation. (The idea of skipping permutations to save communication time or to reduce the overhead caused by local permutations is known. Cooley and Tukey [2] already suggested this to save local bit reversals. Other authors [10,24,25] give examples where skipping permutations saves communication time.)

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 so that the cyclic distribution is a natural input distribution, i.e., a distribution that does not involve any communication as the first superstep. This is done as follows. The first three supersteps of Algorithm 2 are described by the matrix decomposition

$$\Gamma_u \cdot A_{\frac{N}{p},N} \dots A_{2,N} \cdot P_N, \tag{35}$$

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 7), we rewrite matrix (35) as

$$\begin{aligned} &\Gamma_u \cdot (I_p \otimes A_{\frac{N}{p},\frac{N}{p}}) \dots (I_p \otimes A_{2,\frac{N}{p}}) \cdot (I_p \otimes P_{\frac{N}{p}}) \cdot (P_p \otimes I_{\frac{N}{p}}) \cdot S_{p,N} \\ =&\Gamma_u \cdot (I_p \otimes F_{\frac{N}{p}}) \cdot (P_p \otimes I_{\frac{N}{p}}) \cdot S_{p,N} \\ =&\Gamma_u \cdot (P_p \otimes I_{\frac{N}{p}}) \cdot (I_p \otimes F_{\frac{N}{p}}) \cdot S_{p,N}. \end{aligned} \tag{36}$$

Here we used Lemma 2. The first three supersteps of the parallel FFT algorithm derived from this new decomposition are: (1^{Comm}) permutation from block to cyclic distribution, (2^{Comp}) local FFT, (3^{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.

5.2 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 gives an algorithm with a minimum number 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 corresponds 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 (so that all factors of N have approximately the same size). This would enhance the performance on a cache-sensitive computer (see the discussion in Section 6). An even more effective way of enhancing the performance on a cache-sensitive computer is to reduce the butterfly sizes so that the butterflies always fit completely in the cache. We suggest a method in the following subsection.

5.3 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 $2 \le l \le n$, and

$$R^{\alpha}_{l,n} = A^{\alpha}_{n,n} \cdots A^{\alpha}_{2l,n} A^{\alpha}_{l,n}, \tag{37}$$

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 (37) into a sequence of smaller butterfly phases of size less than or equal to n/v which can be carried out independently from each other, we can fully exploit 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 $(\frac{n}{v})^j < l \le (\frac{n}{v})^{j+1}$. Similarly to (22), if we denote $\Gamma_{u,v,n}$ by Γ_u, we can write

$$\begin{aligned} R^{\alpha}_{l,n} = \Gamma_v^{-1} \underbrace{\hat{A}^{\alpha}_{\frac{n}{v},v} \cdots \hat{A}^{\alpha}_{2\frac{(n/v)^{h-1}}{v},v}}_{\text{phase } h-j-1} \Gamma_v \cdot \Gamma^{-1}_{(\frac{n}{v})^{h-2}} \underbrace{\hat{A}^{\alpha}_{\frac{n}{v},(\frac{n}{v})^{h-2}} \cdots \hat{A}^{\alpha}_{2,(\frac{n}{v})^{h-2}}}_{\text{phase } h-j-2} \Gamma_{(\frac{n}{v})^{h-2}} \cdot \ldots \\ \ldots \cdot \Gamma^{-1}_{(\frac{n}{v})^{j+1}} \underbrace{\hat{A}^{\alpha}_{\frac{n}{v},(\frac{n}{v})^{j+1}} \cdots \hat{A}^{\alpha}_{2,(\frac{n}{v})^{j+1}}}_{\text{phase } 1} \Gamma_{(\frac{n}{v})^{j+1}} \cdot \Gamma^{-1}_{(\frac{n}{v})^{j}} \underbrace{\hat{A}^{\alpha}_{\frac{n}{v},(\frac{n}{v})^{j}} \cdots \hat{A}^{\alpha}_{\frac{l}{(n/v)^j},(\frac{n}{v})^{j}}}_{\text{phase } 0} \Gamma_{(\frac{n}{v})^{j}}, \end{aligned} \tag{38}$$

where $\hat{A}^{\alpha}_{k,u}$ is an abbreviation for the $n \times n$ matrix $\hat{A}^{\alpha}_{k,u,v,n} = \Gamma_{u,v,n} A^{\alpha}_{ku,n} \Gamma^{-1}_{u,v,n}$. Generalized versions of Theorem 4 and Corollary 5 can be used to prove that

$$\hat{A}^{\alpha}_{k,u,v,n} = I_{\frac{v}{u}} \otimes \operatorname{diag}(A^{\alpha/u}_{k,\frac{n}{v}}, A^{(\alpha+1)/u}_{k,\frac{n}{v}}, \ldots, A^{(\alpha+u-1)/u}_{k,\frac{n}{v}}). \tag{39}$$

The matrix decomposition (38) can be used to construct an alternative (cache-friendly) algorithm for the computation of the generalized butterfly phases. Note that if $\alpha = 0$ then the resulting algorithm can be used to construct a cache-friendly sequential FFT algorithm.

6 Performance results and discussion

In this section, we present results on the performance of our implementation of the FFT. We implemented the FFT algorithm for the block distribution in ANSI C using the BSPlib communications library [21]. 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 such as workstations. 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 a normalized backward FFT.

6.1 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 $||\mathbf{F}^* - \mathbf{F}||_2/||\mathbf{F}||_2$, 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. Here, $|| \cdot ||_2$ indicates the L^2-norm.

Table 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. This validates the parallel algorithm. The results also indicate that IEEE arithmetic is superior to the CRAY-specific arithmetic.

Table 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}

6.2 Performance of the sequential implementation

Our sequential FFT algorithm was implemented using Algorithm 3 with $\alpha = 0$. Its efficiency can be analyzed by looking at execution times or *FFT flop rates*:

$$\mathrm{FFT}^{\mathrm{rate}}(seq, N) = \frac{5N \log_2 N}{\mathrm{Time}(seq, N)}, \tag{40}$$

where $\mathrm{Time}(seq, N)$ is the execution time of the sequential implementation. 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 to evaluate the overall performance of the algorithm as a function of N.

Table 2
Timing results (in ms) and FFT flop rates (in Mflop/s) of the sequential FFT on the Cray T3E.

N	Time	FFT^{rate}
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

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

6.3 Scalability of the parallel implementation

The timing results obtained by our parallel algorithm are summarized in Table 3. We also present the theoretical predictions using the cost function (32) and the values of the BSP parameters v, g, and l listed in Table 4 which were obtained using a modified version of the benchmark program from BSPpack[3] [15].

Except for the out-of-cache computations (boldface entries in Table 3), the timings show that the BSP cost function predicts the behavior of the parallel implementation well. The discrepancy between predicted and measured results for out-of-cache computations is to be expected, since the computation speed, which we assumed to be constant, suddenly drops when the computations cannot be done completely in cache. These results show that the BSP model is a valid tool for analyzing and predicting parallel performance.

Table 3 shows that using $\sqrt{N}$ processors or more is not advantageous on our test machine and with our limited number of processors, cf. the timings for $N = 512, 1024$ with $p = 32, 64$ and for $N = 2048, 4096$ with $p = 64$. Table 4 tells us that the growth of g satisfies $g(2p) < 1.5g(p)$ for all p, so that the communication time decreases with p for all problem sizes N, cf. (34). For small problems, the increase in synchronization time dominates the decrease in computation and communication time. For larger problem sizes and a larger number of processors, we expect this to be the reverse, but we do not have enough processors available to observe this phenomenon.

A way of analyzing the scalability of a parallel implementation is to look at its *absolute efficiency*

$$E^{abs}(p, N) = \frac{\text{Time}(seq, N)}{p\text{Time}(p, N)}, \tag{41}$$

as done in Figure 4. In theory, $E^{abs}(p, N) \leq 1$, and our goal is to achieve efficiencies as close to one as possible. The figure shows moderate efficiencies for small problem sizes ($N \leq 4096$). For $N \geq 8192$, efficiencies above one are achieved. Such amazing efficiencies 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.

Note that there is a sudden rise in the flop rate when the local problem size becomes small enough to fit in the 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 efficiency decreases notably in going from one to two processors, then it is more or less constant up to 8–16 processors and after that it decreases steadily. For

[3] Available at `http://www.math.uu.nl/people/bisseling/software.html`

Table 3. Predicted and measured execution times (in ms) for the FFT on a Cray T3E. Boldface entries indicate out-of-cache computations.

p	$N = 512$		$N = 1024$		$N = 2048$		$N = 4096$		$N = 8192$		$N = 16384$		$N = 32768$		$N = 65536$	
	pred	meas	pred	meas	pred	meas	pred	meas	pred	meas	pred	meas	pred	meas	pred	meas
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

Table 4
BSP parameters for the CRAY T3E, with $v = 34.9$ Mflop/s. The value of v is based on in-cache dot product computations.

p	g		l	
	(flops)	(μs)	(flops)	(μs)
1	0.28	0.008	3	0.09
2	1.14	0.033	479	13.72
4	1.46	0.042	858	24.57
8	2.14	0.061	1377	39.48
16	2.30	0.066	1754	50.26
32	2.77	0.079	2024	58.00
64	3.05	0.088	3861	110.88

large sizes ($N > 16384$) the flop rate is nearly constant, both before and after the transition out-of-cache/in-cache, indicating a good scalability. The cases $N = 8192, 16384$ are intermediate cases with an efficiency increase in going from one to two processors, but a decrease when the number of processors becomes too large.

We can also examine the scalability of our parallel algorithm by increasing the problem size together with the number of processors [12,26], for instance by maintaining the local problem size N/p constant and increasing p. In doing so, we can learn about the asymptotic behavior of our algorithm. Figure 5 shows the predicted and measured efficiencies as a function of p for various values of N/p. The predicted values converge to a horizontal line as N/p increases which means that asymptotically the efficiency can be maintained at a constant level if N/p is maintained constant. The measured values 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 $N/p = 4096$.

7 Conclusions and future work

In this work, we present a new parallel FFT algorithm, Algorithm 2, which is a mixed radix-2 and radix-4 FFT. It was derived from the matrix decomposi-

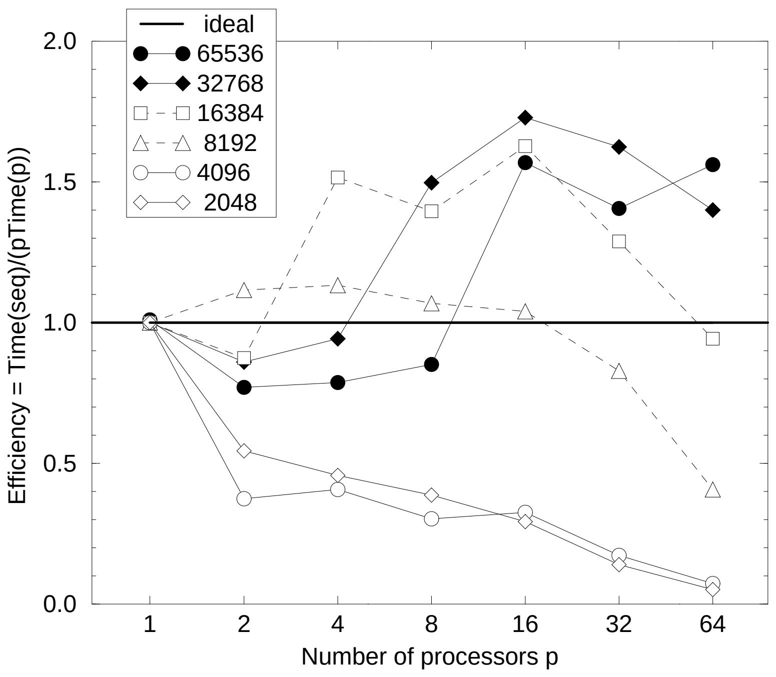

Fig. 4. Absolute efficiencies of the FFT on a Cray T3E.

tion 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 easy to understand and implement.

The use of matrix notation proved to be a powerful tool for deriving parallel FFT algorithms and adapting them to our needs. With the help of matrix notation, we showed how to modify our original algorithm to accept I/O vectors that are not block distributed, without incurring extra communication cost. For the cyclic distribution, we showed how to eliminate the first and the last permutation altogether, reducing the communication to one third of the original cost. Since the cyclic distribution is simple and widely used, this property can be exploited to accelerate many applications. A prime application of the cyclic distribution would be in the field of quantum molecular dynamics where a potential energy operator is applied to the input vector representing a wave packet and a kinetic energy operator is applied to the output vector [27]. Since both operations are componentwise, every distribution can be chosen, including the best one, the cyclic distribution.

We measured the performance of our implementation 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 favorable results obtained for

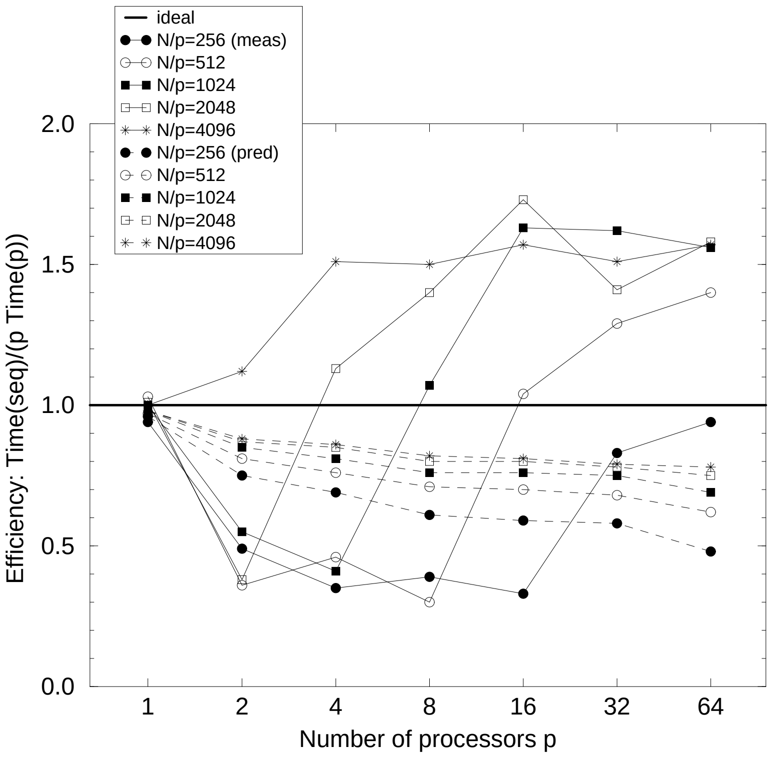

Fig. 5. Efficiency as a function of p for a constant problem size N/p on a Cray T3E. Solid lines: measured values. Dashed lines: predicted values.

larger N are due to the cache effect. To exclude this effect, we also analyzed our results in terms of FFT flop rate per processor, which confirmed the scalability of our algorithm.

Our algorithm is applicable for the case that $p > \sqrt{N}$. Using such a relatively large number of processors becomes worthwhile in the case that g and l grow slowly enough with p, provided that N is sufficiently large. The number of processors, $p = 64$, of our test machine was too small to allow us to observe gains in the range $p > \sqrt{N}$. This may be different on a larger machine, or a machine with faster communication and synchronization. Also note that our implementation uses the block distribution for the I/O vector. We could have improved our experimental results by using the cyclic distribution instead, as outlined in Section 5.1, and this would have reduced the total amount of communication and synchronization, thus making it easier to observe the gains. Of course, in certain situations it might be worthwhile to use our algorithm with more than $\sqrt{N}$ processors even when this would be slower, for instance, for reasons of memory space, or as part of a larger application with other computationally-intensive parts that would benefit from additional processors.

Because the cache-based architecture of the Cray T3E influences our results so much, and many other computers have 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 replacing the processors by virtual processors. It is also possible to derive a cache-friendly parallel algorithm by writing each generalized butterfly phase as a sequence of smaller generalized butterfly phases. We expect such an algorithm to scale just as well as the algorithm we implemented.

A Proof of Theorem 4

The proof uses the following lemma.

Lemma 8 *Let u, M, and k be powers of two such that $2 \le k \le M/u$. 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

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

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

□

Proof of Theorem 4.

PROOF. Define $K = ku$. To prove the theorem, it is sufficient to prove that

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

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

$$\begin{cases} (A_{K,M}\mathbf{x})_j & = x_j + w_K^{j \bmod K} x_{j+K/2}, \\ (A_{K,M}\mathbf{x})_{j+K/2} & = x_j - w_K^{j \bmod K} x_{j+K/2}, \quad 0 \le j \bmod K < K/2. \end{cases} \tag{A.1}$$

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

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

Defining $l = \sigma_{u,M}(j)$ and applying Lemma 8 to j gives the following. Part 1 of Lemma 8 says that $j \bmod K < K/2$ implies $l \bmod k < k/2$. Furthermore, by Part 2, $\sigma_{u,M}(j + K/2) = l + k/2$. Finally, applying Part 3 to l gives $w_K^{j \bmod K} = w_K^{\sigma_{u,M}^{-1}(l) \bmod K} = w_K^{u(l' \bmod k)+s_1} = w_k^{l' \bmod k + s_1/u}$, where $l' = l \bmod n$ and $s_1 = l \operatorname{div} n$.

Substituting the above results into (A.2) gives the following description of vector $\mathbf{z} = S_{u,M}A_{K,M}S_{u,M}^{-1}\mathbf{y}$:

$$\begin{cases} z_l & = y_l + w_k^{l' \bmod k + s_1/u} y_{l+k/2}, \\ z_{l+k/2} & = y_l - w_k^{l' \bmod k + s_1/u} y_{l+k/2}, \quad 0 \le l \bmod k < k/2. \end{cases} \tag{A.3}$$

Writing the index $l = s_1 \cdot n + (l' \operatorname{div} k) \cdot k + l' \bmod k$, it is easy to see that $z_l = (\operatorname{diag}(A_{k,n}^{0/u}, A_{k,n}^{1/u}, \ldots, A_{k,n}^{(u-1)/u}) \cdot \mathbf{y})_l$. The corresponding matrix structure is illustrated in Figure 2(B). $\square$

References

[1] W. L. Briggs, V. E. Henson, The DFT: An Owner's Manual for the Discrete Fourier Transform, SIAM, Philadelphia, PA, 1995.

[2] J. W. Cooley, J. W. Tukey, An algorithm for the machine calculation of complex Fourier series, Mathematics of Computation 19 (1965) 297–301.

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

[4] R. C. Agarwal, J. W. Cooley, Vectorized mixed radix discrete Fourier transform algorithms, Proceedings of the IEEE 75 (9) (1987) 1283–1292.

[5] M. Ashworth, A. G. Lyne, A segmented FFT algorithm for vector computers, Parallel Computing 6 (1988) 217–224.

[6] A. Averbuch, E. Gabber, B. Gordissky, Y. Medan, A parallel FFT on an MIMD machine, Parallel Computing 15 (1990) 61–74.

[7] E. Chu, A. George, FFT algorithms and their adaptation to parallel processing, Linear Algebra and its Applications 284 (1998) 95–124.

[8] A. Dubey, M. Zubair, C. E. Grosch, A general purpose subroutine for Fast Fourier Transform on a distributed memory parallel machine, Parallel Computing 20 (1994) 1697–1710.

[9] A. Gupta, V. Kumar, The scalability of FFT on parallel computers, IEEE Transactions on Parallel and Distributed Systems 4 (8) (1993) 922–932.

[10] O. Haan, A parallel one-dimensional FFT for Cray T3E, in: H. Lederer, F. Hertweck (Eds.), Proceedings of the Fourth European SGI/Cray MPP Workshop, IPP, Garching, Germany, 1998, pp. 188–198.

[11] M. Hegland, Real and complex fast Fourier transforms on the Fujitsu VPP 500, Parallel Computing 22 (1996) 539–553.

[12] V. Kumar, A. Grama, A. Gupta, G. Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

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

[14] P. N. Swarztrauber, Multiprocessor FFTs, Parallel Computing 5 (1987) 197–210.

[15] M. A. Inda, Constructing parallel algorithms for discrete transforms: From FFTs to fast Legendre transforms, Ph.D. thesis, Department of Mathematics, Utrecht University, Utrecht, The Netherlands (March 2000).

[16] G. Bongiovanni, P. Corsini, G. Frosini, One-dimensional and two-dimensional generalized discrete Fourier transforms, IEEE Transactions on Acoustics, Speech, and Signal Processing ASSP-24 (1976) 97–99.

[17] P. Corsini, G. Frosini, Properties of the multidimensional generalized discrete Fourier transform, IEEE Transactions on Computers c-28 (11) (1979) 819–830.

[18] P. A. Regalia, S. K. Mitra, Kronecker products, unitary matrices and signal processing applications, SIAM Review 31 (4) (1989) 586–613.

[19] L. G. Valiant, A bridging model for parallel computation, Communications of the ACM 33 (8) (1990) 103–111.

[20] R. H. Bisseling, Basic techniques for numerical linear algebra on bulk synchronous parallel computers, in: L. Vulkov, J. Waśniewski, P. Yalamov (Eds.), Workshop Numerical Analysis and its Applications 1996, Vol. 1196 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1997, pp. 46–57.

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

[22] O. Bonorden, B. Juurlink, I. von Otte, 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, 1999.

[23] M. C. Pease, An adaptation of the fast Fourier transform for parallel processing, Journal of the ACM 15 (2) (1968) 252–264.

[24] M. A. Inda, R. H. Bisseling, D. K. Maslen, On the efficient parallel computation of Legendre transforms, SIAM Journal on Scientific Computing (2001) in press.

[25] S. Zoldi, V. Ruban, A. Zenchuk, S. Burtsev, Parallel implementation of the split-step Fourier method for solving nonlinear Schödinger systems, SIAM News January/February (1999) 8–9.

[26] J. L. Gustafson, Reevaluating Amdahl's law, Communications of the ACM 31 (5) (1988) 532–533.

[27] R. Kosloff, Quantum molecular dynamics on grids, in: R. E. Wyatt, J. Z. Zhang (Eds.), Dynamics of Molecules and Chemical Reactions, Marcel Dekker, New York, 1996, pp. 185–230.