SIAM J. SCI. COMPUT.
Vol. 14, No. 2, pp. 349–367, March 1993

006

SUPERPARALLEL FFTS*

HANS MUNTHE-KAAS†

Abstract. Fast Fourier transform (FFT) algorithms for single instruction multiple data (SIMD) machines are developed which simultaneously solve any combination of FFTs of different sizes and even different spatial dimensionalities. The only restrictions are that all the periods must be powers of two and that the initial data must satisfy some alignment requirements with the address space in the computer. The degree of parallelism is equal to the sum of the sizes of all the subproblems and the (parallel) solution time is proportional to $\log_2(m)$, where m is the number of points in the largest subsystem. It is shown that the task of unscrambling the data can be both executed and scheduled efficiently in parallel. Finally, implementations on the MasPar computer are described. The codes can be quickly and easily employed in solving complicated problems, and the interface for the routines may therefore be interesting for sequential FFT codes as well.

Key words. FFT, parallel computing, SIMD computers, superparallel algorithms

AMS(MOS) subject classifications. 65T20, 65Y05, 65Y10

1. Introduction.

Fast Fourier transforms (FFTs) constitute one of the most important classes of algorithms in scientific computations. Several papers deal with the problems of implementing FFTs on vector and multiprocessor systems; see [1], [26], and [27]. Various aspects of mapping FFT algorithms to Boolean cubes are discussed, for example, in [11] and [23].

In this paper we are concerned with the effective implementation of FFTs on massively parallel computer systems. Primarily, we have in mind fine-grained parallel machines of the single instruction multiple data (SIMD) stream type, often also called *data parallel computers.* SIMD machines are now commercially available with a number of processors ranging from a few thousand to almost a hundred thousand. Increasingly, we are faced with the luxury of having "too many processors."

Suppose that we wish to solve a mathematical problem of some characteristic size n (n is, e.g., the number of points in a grid, the number of pixels in an image, or some other size indicator). Parallel algorithms for solving the problem can typically be grouped into three main categories.

1. $p \ll n$. The number of processors is much smaller than the size of the problem. This is a typical situation for implementations on coarse-grained parallel systems with a moderate number of processors.

2. $p \sim n$. The number of processors matches the size of the problem. This is not as common in practice; it mainly occurs for benchmark model problems or for special-purpose machines built for a specific task.

3. $p \gg n$. The number of processors is much larger than the size of the problem. This problem class is becoming increasingly important as the number of processors grows, and is our concern in this paper.

As the maximum number of processors that can possibly be employed in solving a given problem is limited by (some function of) n, it is not likely that problem class 3 has efficient algorithms in the case where only a single instance of the problem is to be solved. On the other hand, it is often the case that we want to simultaneously

*Received by the editors May 22, 1991; accepted for publication (in revised form) February 27, 1992. This work was sponsored in part by the Norwegian Science Foundation NAVF through the project "Partielle Differensiallikninger i Storskalaberegninger" and by the University of Bergen.

†Department of Informatics, University of Bergen, N-5020 Bergen, Norway. A large portion of this work was done while the author was a visiting researcher at Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique (CERFACS), Toulouse, France.

solve multiple occurrences of the same (or closely related) problems. Parkinson [22] uses the term *superparallel algorithms* to denote algorithms that in a SIMD fashion can solve multiple instances of similar problems (of possibly different sizes), with a degree of parallelism that is on the order of the sum of the sizes of all the subproblems. Superparallel algorithms are immensely important for the efficient utilization of massively parallel SIMD machines, as they in a sense make the SIMD machine behave as a MIMD (multiple instruction multiple data) stream machine, solving different problems simultaneously. Compared to a parallel scheduling of different jobs on a MIMD computer, superparallel algorithms have several advantages, since they require no load balancing or synchronizations.

Before we describe the superparallel FFT algorithms in detail, we will briefly address the main differences between sequential and data parallel programming, since this is a key issue for understanding the fundamental ideas. In a sequential programming style, the geometry information of a problem is typically controlled through the instruction stream (i.e., through the program code). For instance, in grid problems the size of the problem usually appears in the range of some looping variables. Data parallel programming [8], on the other hand, seeks to define geometries through the data streams. For example, grid problems are defined by mapping the grid onto the processors, and boundary effects are taken into account by modifying the data set near the boundary points. Since SIMD machines have many data streams, but only one instruction stream, defining the geometries through the data stream is the only way of obtaining superparallel algorithms.

Superparallel algorithms were previously known for problems such as the solution of linear tridiagonal equation systems and linear recurrences [22]. Superparallel FFT algorithms have, on the other hand, not been known. The reason for this is probably that it is less obvious how to define the geometries of FFTs through the data stream. Parkinson [22] writes: "[For the FFT algorithm], there does not appear to be a variable which would allow us to easily extend the algorithm to have the Super Parallel property."

In the following sections we will demonstrate that such a variable exists, and construct superparallel FFT algorithms. It came as a pleasant surprise that in many ways the corresponding codes are both simpler to program and simpler to use than their sequential counterparts.

2. The superparallel FFT.

2.1. Background and fundamental ideas.

Given an n-periodic vector

$$x(p+n) = x(p); \qquad p = 0, 1, \ldots, n-1,$$

the domain of this vector may be identified with the n-cyclic group $\mathcal{G} = \mathcal{Z}_n$, i.e., the integers $0, 1, \ldots, n-1$ under addition modulo n.

The Fourier transform on $\mathcal{Z}_n$ is defined as

$$\hat{x} = \sum_{q \in \mathcal{G}} x(q) \cdot e^{2\pi i p q/n},$$

or equivalently, in matrix form,

$$\hat{x} = F_n \cdot x,$$

where F_n is the Fourier matrix of order n

$$(F_n)_{p,q} = e^{2\pi i p q/n}. \tag{1}$$

The domain of $\hat{x}$ is the dual group $\hat{\mathcal{G}}$, which is isomorphic to $\mathcal{G}$. Or, in simpler language, $\hat{x}$ is also an n-periodic vector, and there is a natural 1–1 correspondence between the elements of $\hat{x}$ and the elements of x.

Now given a general finite Abel group $\mathcal{G}$, it can always be written as a direct product of cyclic groups $\mathcal{Z}_{n_i}$

$$\mathcal{G} = \mathcal{Z}_{n_{k-1}} \otimes \mathcal{Z}_{n_{k-2}} \otimes \cdots \otimes \mathcal{Z}_{n_1} \otimes \mathcal{Z}_{n_0}.$$

$\mathcal{G}$ can be identified with the set of all integer k-tuples

$$(l_{k-1}, l_{k-2}, \ldots, l_0); \qquad 0 \le l_i < n_i - 1,$$

under addition modulo $(n_{k-1}, n_{k-2}, \ldots, n_0)$. One can simply think of $\mathcal{G}$ as being a k-dimensional grid, periodic of period n_i in the direction i. The Fourier transform on $\mathcal{G}$ is given by

$$\hat{x}(p) = \sum_{q \in \mathcal{G}} x(q) \cdot e^{2\pi i \langle p,q \rangle / n},$$

where $\langle p, q \rangle$ is the bilinear pairing

$$\langle p, q \rangle = \frac{p_{k-1}q_{k-1}}{n_{k-1}} + \frac{p_{k-2}q_{k-1}}{n_{k-2}} + \cdots + \frac{p_1 q_1}{n_1} + \frac{p_0 q_0}{n_0}.$$

The transform can equivalently be written in matrix form

$$\hat{x} = (F_{n_{k-1}} \otimes F_{n_{k-2}} \otimes \cdots \otimes F_{n_0}) \cdot x,$$

where F_n is given by (1) and $\otimes$ denotes the matrix tensor product (see, e.g., [7] for its definition and properties). The domain of $\hat{x}$ is the dual group $\hat{\mathcal{G}}$, which again is isomorphic to the original group $\mathcal{G}$.

In the rest of the paper we make the assumption that all periods n_i are powers of two. It is our aim to develop superparallel FFTs that can simultaneously handle any collection of FFTs of (possibly) different sizes and different number of dimensions k, the only restriction being that all the periods are powers of two. Our approach is based on Cooley–Tukey FFTs [6].

The idea behind the radix-2 Cooley–Tukey FFT is an attempt to factor the group $\mathcal{Z}_n$ in a direct product of binary groups $\mathcal{Z}_2$. But since $\mathcal{Z}_n$ and $\mathcal{Z}_2 \otimes \mathcal{Z}_{n/2}$ are in general nonisomorphic groups, we cannot factor the matrix F_n as $F_2 \otimes F_{n/2}$. The closest we can get to a factorization of this kind is the following formula (described in §2.2):

$$P \cdot F_n = (I_2 \otimes F_{n/2}) \cdot T \cdot (F_2 \otimes I_{n/2}), \tag{2}$$

where P is a permutation matrix, called the *scrambling matrix*. I_k are identity matrices of order k, and T is a diagonal matrix containing the so-called *twiddle factors*.

Since $(I_2 \otimes F_{n/2}) \cdot (F_2 \otimes I_{n/2}) = (F_2 \otimes F_{n/2})$ is the FFT on the group $Z_2 \otimes Z_{n/2}$, we observe that apart from the twiddle factors T, and the scrambling matrix P, the FFTs on $\mathcal{Z}_n$ and $\mathcal{Z}_2 \otimes \mathcal{Z}_{n/2}$ are identical. We can continue to factor out binary groups and arrive at an equivalency between the FFT on

$$\mathcal{Z}_{n_{k-1}} \otimes \mathcal{Z}_{n_{k-2}} \otimes \cdots \otimes \mathcal{Z}_{n_1} \otimes \mathcal{Z}_{n_0}$$

and on the d-dimensional binary group

$$\mathcal{Z}_2 \otimes \mathcal{Z}_2 \otimes \cdots \otimes \mathcal{Z}_2,$$

where $d = \log_2(n_{k-1} \cdot n_{k-2} \cdots n_0)$, the only differences between the two transforms being the values of the twiddle factors and the scrambling of the results.

Thus, since all the geometric information about the transforms is contained in the twiddle factors (and in the scrambling of the results), it is possible to code the geometric information of FFT*s in the data stream instead of in the instruction stream. This is the basis for the development of superparallel* FFT*s.*

2.2. The basic algorithm. To give a precise meaning to the splitting formula (2), we must define a matching between the elements of different groups. This is done through the *bit representation* of the elements of a group, defined by stacking together the binary representation of each component of the group element.

For example, suppose we want to identify the elements of a group $\mathcal{G}_1 = \mathcal{Z}_8 \otimes \mathcal{Z}_4 \otimes \mathcal{Z}_4$ with the elements of the group $\mathcal{G}_2 = \mathcal{Z}_2 \otimes \mathcal{Z}_4 \otimes \mathcal{Z}_4 \otimes \mathcal{Z}_4$. An element, say $g = (5, 1, 2) \in \mathcal{G}_1$, has the bit representation $(1, 0, 1,\ 0, 1,\ 1, 0)$. (Note that the second component is represented by the *two bits* $0, 1$ since two bits are required to represent all numbers in $\mathcal{Z}_4$.) This element is identified with the element $(1, 1, 1, 2) \in \mathcal{G}_2$, which has the same bit representation.

Thus the bit representation identifies a group element in $\mathcal{G}$ with an element of the binary group

$$\mathcal{G}_b = \mathcal{Z}_2 \otimes \cdots \otimes \mathcal{Z}_2,$$

with the same cardinality as $\mathcal{G}$.

The binary representation will be the most used component-wise representation of a group element, *so from now on, unless otherwise stated,* $(g_{k-1}, g_{k-2}, \ldots, g_0)$ *refers to the bit representation of a group element* g.

The *perfect shuffle permutation* of the elements in a cyclic group $\mathcal{Z}_n$ is given by the transformation

$$\sigma(g) = \sigma((g_{k-1}, g_{k-2}, \ldots, g_0)) = (g_{k-2}, g_{k-3}, \ldots, g_0, g_{k-1}),$$

i.e., a cyclic rotation towards the left of its bit representation. The perfect shuffle permutation of a vector defined on a group $\mathcal{Z}_n$ is defined as the matrix S_n acting by

$$S_n \cdot x(q) = x(\sigma(q)). \tag{3}$$

Note 1. Permutation matrices defined in this way multiply together in a way opposite to what one may first think: if π_1 and π_2 are two permutations of the domain, and if the corresponding permutation matrices P_1 and P_2 are defined as in (3), we obtain

$$P_2 \cdot P_1 \cdot x(q) = P_1 x(\pi_2(q)) = x(\pi_1 \circ \pi_2(q)),$$

where $\circ$ denotes function composition.

The Cooley–Tukey splitting of the FFT matrix is then given by the following lemma.

LEMMA 2.1. *Let* S_n *be the perfect shuffle. Then the Fourier matrix* F_n *can be factored as*

$$S_n \cdot F_n = (I_2 \otimes F_{n/2}) \cdot T_n \cdot (F_2 \otimes I_{n/2}),$$

where T_n, the twiddle matrix, is diagonal. T_n acts on vectors in $\mathcal{G} = \mathcal{Z}_2 \otimes \mathcal{Z}_{n/2}$ according to

$$T_n \cdot x((g_1, g_0)) = \begin{cases} x((g_1, g_0)) & \text{if } g_1 = 0, \\ e^{2\pi i g_0/n} \cdot x((g_1, g_0)) & \text{if } g_1 = 1, \end{cases}$$

for all $g \in \mathcal{G}$, where (g_1, g_0) denotes the components of $g \in \mathcal{Z}_2 \otimes \mathcal{Z}_{n/2}$.

Proof. This is checked by a straightforward computation. For a survey of similar results, see, e.g., [24]. □

The Cooley–Tukey FFT algorithm applies this formula recursively. Starting with a matrix F_n, we first multiply from the left by S_n, then by $I_2 \otimes S_{n/2}$, $I_4 \otimes S_{n/4}$, and so on down to $I_{n/4} \otimes S_4$. This gives the complete binary factorization of F_n:

$$\begin{aligned} &(I_{n/4} \otimes S_4) \times (I_{n/8} \otimes S_8) \cdots (I_2 \otimes S_{n/2}) \cdot S_n \times F_n \\ &\quad = (I_{n/2} \otimes F_2) \times \big(I_{n/4} \otimes [T_4 \times (F_2 \otimes I_2)]\big) \cdots \big(I_2 \otimes [T_{n/2} \times (F_2 \otimes I_{n/4})]\big) \\ &\qquad \times T_n \times (F_2 \otimes I_{n/2}). \end{aligned} \tag{4}$$

A multiplication of a vector by $I_{2^k} \otimes F_2 \otimes I_{2^l}$ is called a *butterfly* operation, and multiplication by $I_{2^k} \otimes T \otimes I_{2^l}$ a *twiddle* operation. The right-hand side in (4) is thus a series of butterfly and twiddle operations.

Let σ_l denote the perfect shuffle of the right-most l bits, i.e.,

$$\sigma_l(g) = (g_{k-1}, g_{k-2}, \ldots, g_l, g_{l-2}, g_{l-3}, \ldots, g_0, g_{l-1}).$$

Since $(I_{n/2^l} \otimes S_{2^l}) \cdot x(g) = x(\sigma_l(g))$, we find that

$$\begin{aligned} (I_{n/4} \otimes S_4) \cdot (I_{n/8} \otimes S_8) \cdots (I_2 \otimes S_{n/2}) \cdot S_n \cdot\ x(g) &= x(\sigma_{k-1} \circ \sigma_{k-2} \circ \cdots \circ \sigma_2(g)) \\ &= x((g_0, g_1, \ldots, g_{k-1})). \end{aligned}$$

Thus, as is well known, the results appear in *bit-reversed* order.

In the following we will describe how (4) is turned into a superparallel algorithm. This is best explained through pseudocodes.

The instruction stream of the superparallel FFT routine tells all the processors that they should compute the matrix–vector product as in the right-hand side of (4). It is, however, left to each processor to decide whether it should participate in each butterfly and twiddle operation, or if it should rest idle instead. Each processor must also decide which values it should use for the twiddle factors. By these decisions, (4) can be tuned to produce any matrix product of the form

$$I_{n_l} \otimes F_{n_{l-1}} \otimes I_{n_{l-2}} \otimes F_{n_{l-3}} \otimes \cdots \otimes F_{n_1} \otimes I_{n_0}, \tag{5}$$

where n_i are powers of two. We will see later that it is very useful to be able to include identity matrices in the product as well.

When the superparallel FFT routine is initially called, each processor must, of course, know which product it wants to compute, and this information must be *consistent*. For example, if a processor wants to participate in a butterfly operation, then its corresponding neighbor must want the same.

The type of matrix product each processor wants to compute is stored *locally* in an *active_list* containing *active groups*, i.e., a list of beginning bits and end bits for each term F_{n_i} in (5).

Example 1. Suppose the address space of the computer is 12 bits. If a processor wants to participate in the product

$$I_{2^3} \otimes F_{2^4} \otimes I_{2^2} \otimes F_{2^3},$$

then it creates the following list, containing two active groups

$$\text{active\_list} = \left\{ \begin{matrix} (8,5) \\ (2,0) \end{matrix} \right\},$$

indicating that F_{2^4} affects bits 8 to 5 and F_{2^3} affects bits 2 to 0. (It is most natural to start from the left since the reduction starts with the left-most bit first.)

See §3 for some practical examples.

Given the active list of each processor, the pseudocode in Algorithm 1 describes the main part of the algorithm. First, we present some preliminaries.

1. We assume that there are 2^k processors in the network and altogether 2^k data points. Initially, each processor stores the data point $x(g)$, where $g = (g_{k-1}, \ldots, g_0)$ is the binary address of the processor. In §2.3 we discuss the case where there are more data points.

2. The symbol $\oplus$ denotes *exclusive or* (xor), i.e.,

$$\begin{aligned} 0 \oplus 0 &= 0, \\ 0 \oplus 1 &= 1, \\ 1 \oplus 0 &= 1, \\ 1 \oplus 1 &= 0. \end{aligned}$$

3. The *if*'s in the pseudocode denote parallel if's. That is, the processors that do not fulfill the condition in the test are masked as inactive and wait idle until the others are finished.

4. The final answers appear in bit-reversed order within each active group, e.g., in Example 1, processor $(g_{11}, g_{10}, \ldots, g_0)$ will end up with the answer

$$\hat{x}((g_{11}, g_{10}, g_9, g_5, g_6, g_7, g_8, g_4, g_3, g_0, g_1, g_2)). \tag{6}$$

ALGORITHM 1.

```
# Basic version of superparallel FFT code
for i = k − 1, 0, −1 # loop over all bits from left to right
   if (i ∈ active_group) then
      # next line contains the interprocessor communication
      tmp := x((g_{k−1}, ..., g_{i+1}, g_i ⊕ 1, g_{i−1}, ..., g_0))
      if (g_i = 0) then
         x := x + tmp
      else
         x := tmp − x
         tw := twiddle_factor(i,active_group,g)
         x := tw * x
      endif
   endif
endfor
```

The algorithm takes $\mathcal{O}(p)$ parallel steps, where p is the number of bits that are active for some of the processors. For example, if the algorithm computes a collection

of equally sized transforms, p is given as the $\log_2$ of the number of points in each transform.

From Lemma 2.1 we find that the twiddle factors are computed in the following ways.

ALGORITHM 2.

```
# Computation of twiddle factors
function twiddle_factor(i, active_group,g)
   ra := position_of_rightmost_bit_in_active_group
   # The right-hand side below is the binary representation of a
   real number
   θ = 0.g_{i-1} g_{i-2} ... g_{ra}
   tw := e^{2πθ√-1}
   return(tw)
endfunction
```

It is amazing that such a simple code can simultaneously handle any collection of different-sized FFTs, even of different dimensionalities! The pseudocode is even simpler than the standard one-dimensional radix-2 FFT for a sequential computer. The simplicity of the code results from the inherent natural parallelism contained in the FFT.

2.3. Doubling the speed with bitswaps. The basic code has two flaws that should be corrected. First of all, it assumes that the total number of data points exactly matches the number of processors. The code must also be able to handle the case where more data points are available. For certain machines (e.g., the Connection Machine) the user is free to configure the number of *virtual processors*, i.e., the machine can, in a transparent way, pretend that it has more processors than it physically has. This is done in microcode. For other machines (e.g., the MasPar), the mapping of multiple points to a single processor is taken care of by either a high-level language compiler or, explicitly, by the programmer. We shall see that in the latter case it is possible to correct the second major drawback of the basic code, which is its efficiency.

Suppose the basic code is used to compute one large one-dimensional FFT, i.e., all the processors have a single active set covering all the address bits. First, all the processors swap one number with another processor; afterwards half of the processors compute a sum (while the rest are idle); and finally the second half of the processors compute a difference and perform the twiddle multiplication (while the first half is idle). Thus in a large part of the code, half of the machine is doing nothing.

Now assume that we have at least twice as many data points as processors. The way to handle this is to introduce some extra bits which refer to different locations in the memory of each processor. For example, if there are four times as many data points as processors, each processor stores four numbers and we get two extra address bits, which we call *memory bits*. Hereafter, we assume that the *left-most* bit is always a memory bit. The location of the other memory bits is of no importance, and, to simplify the exposition, we forget the rest of the memory bits and assume that we have *exactly* twice as many data points as we have processors. We define a *bitswap permutation* as a swapping of the left-most bit with another bit

$$B^i \cdot x((g_{k-1}, \ldots, g_i, \ldots,)) = x((g_i, \ldots, g_{k-1}, \ldots,)).$$

Note that B^i is its own inverse. The bitswap can be performed by the following pseudocode.

ALGORITHM 3.

```
# Computation of x := B^i·x
function bitswap(i, x)
   if (g_{k-1} ≠ g_i) then
      x((g_{k-1}, ..., g_i, ...)) := x((g_{k-1} ⊕ 1, ..., g_i ⊕ 1, ...))
   endif
endfunction
```

Note that the condition is true for exactly one of the two numbers in each processor, thus every processor is active and swaps exactly one number with another processor. The bitswap transfers all the "action" from bit i to the left-most bit by:

$$I_{n/2^i} \otimes [T_{2^i} \cdot (F_2 \otimes I_{2^{i-1}})] = B^i \cdot \tilde{T}_{2^i} \cdot (F_2 \otimes I_{n/2}) \cdot B^i, \tag{7}$$

where

$$\tilde{T}_{2^i} = B^i \cdot (I_{n/2^i} \otimes T_{2^i}) \cdot B^i$$

is a diagonal matrix of order n. Equation (7) can be substituted into (4). After some consideration, one realizes that the left-most bitswap in each substituted term commutes with everything to the left of it, and can be pulled out to the left and multiplied over to the other side of the equation. This gives the following "bitswap version" of (4):

$$\begin{aligned} & B^0 \cdot B^1 \cdots B^{k-1} \cdot (I_{n/4} \otimes S_4) \cdot (I_{n/8} \otimes S_8) \cdots (I_2 \otimes S_{n/2}) \cdot S_n \cdot F_n \\ & = \tilde{T}_2 \cdot (F_2 \otimes I_{n/2}) \cdot B^0 \cdot \tilde{T}_4 \cdot (F_2 \otimes I_{n/2}) \cdot B^1 \cdots \tilde{T}_{n/2} \cdot (F_2 \otimes I_{n/2}) \\ & \quad \times B^{k-2} \cdot \tilde{T}_n \cdot (F_2 \otimes I_{n/2}) \cdot B^{k-1}, \end{aligned} \tag{8}$$

where $\tilde{T}_2$ and B^{k-1} are identity matrices included to make the formula more symmetric.

In the basic variant of the algorithm, the data points were shuffled locally, i.e., they were only permuted to other processors with the same active_list as the processor where they started. In the bitswap version of the algorithm, data points are shuffled more globally, to processors that initially contained a different active_list. This means that in order to compute twiddle factors correctly, the active_list must also be bitswapped. Furthermore, if *any* processor wants to do a bitswap, we must force *all* processors to participate, even if they are not inside an active group. This is to ensure that the bitswaps define permutations of the data set. We use the word "all" to emphasize that no processor is allowed to be idle, and the word "any" to test if a parallel logical expression is true for at least one processor. This leads to the bitswap variant of the superparallel FFT.

ALGORITHM 4.

```
# Bitswap version of superparallel FFT code
for i = k − 1, 0, −1 # loop over all bits from left to right
   if any (i∈ active_group) then
      all
         bitswap(i, x)
         bitswap(i,active_list)
      end all
   endif
   if (i∈ active_group) then
      tmp := x((0, g_{k-2}, ...))
```

```
        x((0, g_{k-2}, ...)) := tmp + x((1, g_{k-2}, ...))
        x((1, g_{k-2}, ...)) := tmp - x((1, g_{k-2}, ...))
        tw := twiddle_factor (i, active_group,g)
        x((1, g_{k-2}, ...)) := tw * x((1, g_{k-2}, ...))
    endif
endfor
```

Note that the function twiddle_factor is the same as in the basic version of the algorithm.

In the computational part, this code computes two points in the same time that it takes the basic version to compute one point. If we neglect the bitswap of the active_list, each processor swaps one number for each i. Since each processor is computing twice as many numbers as in the basic version, we have also reduced the communication time by a factor of two. In practice, the code is divided into two parts, a symbolic preprocessing stage where the twiddle factors are computed, and a numerical phase where the actual transforms are done. In many situations the same geometries are used for many different transforms. Then the symbolic computation can be done once, and the cost of bitswapping the active_list and computing tw can be neglected. Thus, *on a* SIMD *machine, the bitswap version of the superparallel* FFT *is twice as fast as the basic version.*

The main drawback of the bitswap version is the more complicated scrambling of the results. As we shall see in §4, the bitswap version leaves the data in a much more disordered state than the basic version. For many applications, it is not necessary to unscramble the data. It suffices to know the identities of the results in each processor, and this is straightforward to compute. In other applications it is, however, desirable to unscramble the results. As we shall see in §4, efficient algorithms exist for unscrambling the results of both the basic and the bitswap version.

3. Defining problems and checking consistency. Formally, the only restriction of the possible active_lists is that if a processor wants to participate in a multiplication by F_2, then its partner should want to do the same. However, this requirement alone allows some rather pathological cases, such as a two-dimensional field of numbers where every column wants to first do an FFT in the y-direction, but only the first column wants to afterwards do a transform in the x-direction as well. We cannot see any practical use for such a possibility. For debugging, it is useful to have the possibility of checking the active_lists for inconsistencies of this kind. In such situations, the unscrambling algorithms in the next section will give unpredictable results. We therefore want stricter rules for the possible active_lists. The following rule is general enough to allow all possible interesting transforms.

DEFINITION 1 (consistent active_lists). Given a point with active_list **al**, the set of bits not in any of the active groups is called *inactive.* The collection of all the active_lists is *consistent* if all points with the same values in the inactive bits also share the same active_list **al**.

It is a straightforward matter to check for consistency in this sense: simply mimic the FFT code, but instead of performing the butterfly and twiddle multiplications, check whether or not the active_lists of the neighbors are equal.

DEFINITION 2. A group of points sharing the same active_list and the same values for the inactive bits is called an FFT *chunk.* The values of the inactive bits define the *site* of the chunk.

The number of points in a chunk is given as 2^k, where k is the total number of *active* bits (bits inside the active groups).

We will now see some examples showing that the active_lists can easily be set up to solve rather complicated problems.

Example 2. Given a two-dimensional field of 512×256 points, suppose we want to divide the field into 8×8 square tiles, each of size 64×32, and perform two-dimensional FFTs on each tile. This is simply done by giving each point the active list

$$\text{active\_list} = \left\{ \begin{matrix} (13,8) \\ (4,0) \end{matrix} \right\}$$

and calling the FFT routine.

This example also easily extends to the case where the domain is divided into rectangles of different sizes (as long as the sizes are powers of 2). For example, if we want to merge two tiles neighboring each other in the x-direction into one tile, we simply modify the active lists of these points to

$$\text{active\_list} = \left\{ \begin{matrix} (13,8) \\ (5,0) \end{matrix} \right\}.$$

Example 3. Suppose we have a $64 \times 64 \times 64$ three-dimensional grid and we want to do an FFT on each two-dimensional $y-z$ plane. This is a practical problem arising from using spectral methods to simulate flow between plates. This problem is defined by giving each point the active list

$$\text{active\_list} = \left\{ \begin{matrix} (17,12) \\ (11,6) \end{matrix} \right\}.$$

Compared to the more standard approach (for sequential computers) of solving these problems by looping over the independent FFTs, calling one-dimensional FFTs, transposing the data, looping and calling one-dimensional FFTs again, and finally transposing the matrix again, it is amazing how easily the superparallel FFT can be called to solve relatively complicated problems.

Now suppose we have a collection of different-sized problems, and that we are free to choose their initial positions as we like. We will show that it is always possible to solve these problems in an address space of size the smallest power of 2 larger than or equal to the sum of the sizes of all the subproblems. If there are no geometric relations between the different chunks, it is natural to store each chunk in a contiguous part of the address space. We are, however, not always allowed to pack the different chunks tightly together. Since the lowest address in a chunk is obtained by setting all the active bits to 0, the following *alignment requirement* must be satisfied.

LEMMA 3.1 (memory alignment). *An* FFT *chunk, with a total of k active bits, stored contiguously in the address space, must start in a memory location dividing 2^k, i.e., the right-most k bits are zero.*

If the alignment requirement is fulfilled, then the address succeeding the chunk must also divide 2^k. We conclude with the following theorem.

THEOREM 3.2. *If different-sized chunks are ordered in decreasing order according to their size, they can be stacked together contiguously with no interleave in the address space.*

COROLLARY 3.3. *Less than* 50 *percent of the address space is wasted due to the memory alignment requirement.*

Proof. The memory space needed is the smallest power of 2 containing all subproblems. □

Example 4. Suppose we have three one-dimensional FFTs of lengths 2, 4, and 8. Then we need four bits to define the problems:

chunk 1:
`site :` $q_3 = 0$
`active_list :` {(2,0)}
chunk 2:
`site :` $q_3 = 1;\ q_2 = 0$
`active_list :` {(1,0)}
chunk 3:
`site :` $q_3 = 1;\ q_2 = 1;\ q_1 = 0$
`active_list :` {(0,0)}

The two points $(1,1,1,0)$ and $(1,1,1,1)$ are left unused.

4. Unscrambling the results. For many applications it is not necessary to unscramble the results. A call to an FFT is often followed with a call to an inverse FFT, and in between, the Fourier coefficients are often just multiplied by some diagonal matrix. If one has a pair of transforms, the FFT from ordered to scrambled, and the inverse working backwards from scrambled to unscrambled, then all one usually needs to know is the identities of each Fourier coefficient. After performing the (symbolic) FFT, this can easily be computed in parallel by computing backwards, using the information contained in the active_lists of the results and the information about which bits have been bitswapped.

Example 5. In a point $(g_{11}, g_{10}, \ldots, g_0)$ the active_list is as in example (1) after the FFT (with bitswaps). Without bitswaps the point would contain the result as in (6). Suppose all bits have been swapped in the FFT; then the point contains the number

$$\begin{aligned} &B^{k-1} \cdots B^1 \cdot B^0 \cdot \hat{x}((g_{11}, g_{10}, g_9, g_5, g_6, g_7, g_8, g_4, g_3, g_0, g_1, g_2)) \\ &\quad = \hat{x}((g_2, g_{11}, g_{10}, g_9, g_5, g_6, g_7, g_8, g_4, g_3, g_0, g_1)). \end{aligned}$$

Note that a sequence of bitswaps produces an inverse perfect shuffle.

There are also, however, many applications where it is necessary to obtain the the results in unscrambled order (see, e.g., [3] and [16]). In this section we will derive efficient unscrambling algorithms. To study permutation algorithms it is necessary to assume a model for the computer interconnection network. A particularly useful network is the Beneš network [2], [13], [19]. This is one of the simplest networks capable of performing any permutation of 2^k objects. A permutation algorithm for a Beneš network can easily be transformed into efficient permutation algorithms for a variety of different networks (see the comments below, and for more details, [17]).

A (masked) *bit-exchange* permutation of bit i is given by

$$x((q_{k-1}, q_{k-2}, \ldots, q_i, \ldots, q_0)) \to x((q_{k-1}, q_{k-2}, \ldots, q_i \oplus \chi, \ldots, q_0)),$$

where $\chi(q) \in \{0,1\}$ is a Boolean function. We require this to be a permutation. This is equivalent to the condition

$$\chi((q_{k-1}, q_{k-2}, \ldots, q_i, \ldots, q_0)) = \chi((q_{k-1}, q_{k-2}, \ldots, q_i \oplus 1, \ldots, q_0)),$$

i.e., χ is a function independent of the value of bit q_i. The Beneš network on 2^k data points performs a series of $2k-1$ bit exchanges, where the bits are exchanged in the fixed order

$$q_{k-1}, q_{k-2}, \cdots, q_1, q_0, q_1, \cdots, q_{k-2}, q_{k-1}.$$

By choosing the functions χ_i (i.e., setting the switches in the network) in the right manner, it is known that the Beneš network can perform any permutation of its 2^k inputs. The major problem is the computation of the functions χ_i. This is called the *B-setting* problem. Given a B-setting for a permutation, it is straightforward to solve permutation problems for a variety of different interconnection topologies (see [17] for details).

• *The "many points per processor problem."* Given a network with $p = 2^d$ processors, $d < k$, where there are 2^m, $m = k - d$, data points per processor, suppose we know how to perform a general permutation of data spread by one point per processor. For example, on the MasPar the cheapest way to perform one-point-per-processor permutations is to call a "router" routine (black box permutation algorithm). The problem is: *How few router calls are necessary to perform a given permutation of the complete data set?* If the address bits are ordered so that the left-most m bits are memory bits, we see that the sequence of exchanges

$$q_{k-1}, \cdots, q_{k-m}$$

only involves a local reordering of the data. The sequence

$$q_{k-m-1}, \cdots, q_1, q_0, q_1, \cdots, q_{k-m-1}$$

can be collapsed into one permutation and performed by 2^m calls to the router (one for each memory location). Finally the sequence

$$q_{k-m}, \cdots, q_{k-1}$$

is again only a local reordering. Thus a B-setting provides us with an algorithm that can perform a general permutation by only 2^m router calls. For many permutations it is impossible to do this with fewer calls.

• *Shuffle-exchange networks* [25] can directly perform the same permutations as a Beneš network, by running it forward k steps and backward $k-1$ steps. Alternatively, the B-settings can be transformed into algorithms for performing the permutation in $3k-1$ forward steps [19], or by a somewhat more complex algorithm, in $3k-3$ forward steps [9].

• *Butterfly and omega networks* are equivalent to k forward loops in a shuffle-exchange network. Thus the comments above apply to computers based on these networks.

• *Hypercubes* are extensions of Beneš networks, where *any* of the k bits can be exchanged in each step. Thus a hypercube network can emulate a Beneš network simply by exchanging the bits in a fixed order. This will, however, only use one out of the possible k wires extending from each processor in each step. The main problem is how to efficiently use the full bandwidth of the network. In the case of 2^m points per processor, the technique above splits the permutation task into 2^m independent tasks. The full bandwidth of the network can be utilized by running k of these tasks in parallel over different wires. This can be done by cyclically shifting the order

of the bits in each independent permutation, i.e., k different permutations are done simultaneously by exchanging the bits in the following order:

Task 0:	q_{k-1}	q_{k-2}	$\cdots$	q_1	q_0	q_1	$\cdots$	q_{k-1}
Task 1:	q_{k-2}	q_{k-3}	$\cdots$	q_0	q_{k-1}	q_0	$\cdots$	q_{k-2}
Task 2:	q_{k-3}	q_{k-4}	$\cdots$	q_{k-1}	q_{k-2}	q_{k-1}	$\cdots$	q_{k-3}
$\vdots$								
Task $k-1$:	q_0	q_{k-1}	$\cdots$	q_2	q_1	q_2	$\cdots$	q_0

If there is only one point per processor, this technique may be used, provided that each point is associated with so much data that it can be split into k different parts. Each part can then be permuted on a different set of wires.

• *Two-dimensional mesh-connected computers.* There are several ways to emulate a Beneš network on a two-dimensional mesh. These schemes involve $\mathcal{O}(N^{1/2})$ time for performing permutations. See [12] for special tricks related to the MasPar hardware.

• *Multistage crossbar networks,* where each crossbar performs a general permutation of 2^s wires, can be programmed by collapsing s consecutive bit exchanges into general permutations of 2^s points. For example, the hardware underlying the MasPar "router" construct is a three-stage crossbar. A B-setting of a permutation can, in principle, be used to increase the speed of this kind of hardware. However, we have not tried this approach for the MasPar, since there is no high-level language access to programming the router hardware.

Let $N = 2^k$. The best known algorithms for solving the general B-setting problem takes $\mathcal{O}(N \log(N))$ time on a sequential computer, $\mathcal{O}(N^{1/2})$ time on a mesh-connected parallel computer with $N^{1/2} \times N^{1/2}$ processors, $\mathcal{O}(k \log^3(N))$ time on a hypercube-connected computer with $\mathcal{O}(N^{1+1/k})$ processors where $1 \leq k \leq \log(N)$, and $\mathcal{O}(\log^2(N))$ time on an N-processor shared-memory computer [19]. Thus, unless the FFTs are to be performed many times with the same geometries, the general B-setting algorithms will be too expensive compared to the time of running the numerical parts of the FFTs. The high cost of the general B-setting algorithms has led to a search for classes of permutations that can be B-set *efficiently* on a parallel computer (i.e., where solving the B-setting problem takes no more time than actually performing the permutation). See [17], [13], [19], and [10] for fast B-setting algorithms. Unscrambling the results of the basic superparallel FFT (Algorithm 1) belongs to the classes of permutations that can be solved by the algorithms in these papers. For the more difficult problem of unscrambling the results of the bitswap version (Algorithm 4) these algorithms will generally fail, and a new algorithm is needed.

Before attacking this general unscrambling problem, we study two simpler problems:

1. Finding B-settings for unscrambling the results of the basic version of the superparallel FFT.

2. Finding B-settings for a product of bitswap permutations.

We define the shorthand notation

$$q_i := q_i \oplus \chi$$

to denote the permutation

$$x((\ldots, q_i, \ldots)) \to x((\ldots, q_i \oplus \chi, \ldots)) \,.$$

A commonly used basic permutation is the xor of two different bits

$$q_i := q_i \oplus q_j.$$

As long as $i \neq j$, this defines a permutation. (When $i = j$ it is *not* a permutation, since $q_i \oplus q_i = 0$—two different numbers are collapsed into a single memory address.)

Let $\bar{q}_i$ denote the original values of the bits. A bit reversal, e.g.,

$$x((q_3, q_2, q_1, q_0)) \to x((q_0, q_1, q_2, q_3)),$$

can be performed by the following B-setting:

ALGORITHM 5. UNSCRAMBLING BIT-REVERSED DATA (example with four bits).

$$\begin{aligned}
q_3 &:= q_3 \oplus q_0 = \bar{q}_3 \oplus \bar{q}_0 \\
q_2 &:= q_2 \oplus q_1 = \bar{q}_2 \oplus \bar{q}_1 \\
q_1 &:= q_1 \oplus q_2 = \bar{q}_1 \oplus \bar{q}_2 \oplus \bar{q}_1 = \bar{q}_2 \\
q_0 &:= q_0 \oplus q_3 = \bar{q}_0 \oplus \bar{q}_3 \oplus \bar{q}_0 = \bar{q}_3 \\
q_1 &:= q_1 = \bar{q}_2 \\
q_2 &:= q_2 \oplus q_1 = \bar{q}_2 \oplus \bar{q}_1 \oplus \bar{q}_2 = \bar{q}_1 \\
q_3 &:= q_3 \oplus q_0 = \bar{q}_3 \oplus \bar{q}_0 \oplus \bar{q}_3 = \bar{q}_0
\end{aligned}$$

Note that when there is an odd number of bits, there will be a bit in the middle that is left unchanged. This algorithm extends readily to the more general case where several active groups are to be bit reversed. Even the situation where there are several different FFT chunks with different active groups to be bit reversed can be handled by this method, since each chunk is to be unscrambled within its own site, and since the above method only modifies the active bits of each chunk, i.e., there is no interference between different chunks. Thus this process solves the unscrambling problem for the basic algorithm.

Now to the problem of finding a B-setting for a product of bitswaps: For simplicity, we assume that all the bits are to be bit-swapped. (If some of the bits should not be bit-swapped, we simply ignore them and proceed as below.)

As noted in Example 5, a product of bitswaps amounts to doing an inverse perfect shuffle. Let

$$\bigoplus_{i=0}^{k-1} q_i = q_0 \oplus q_1 \oplus \cdots \oplus q_{k-1};$$

then

$$q_j := \bigoplus_{i=0}^{k-1} q_i \tag{9}$$

is a permutation. By performing this permutation cyclically, one obtains the perfect shuffle.

ALGORITHM 6. PERFECT SHUFFLE PERMUTATION.

$$q_{k-1} := \bigoplus_{i=0}^{k-1} q_i = \bigoplus_{i=0}^{k-1} \bar{q}_i$$
$$q_{k-2} := \bigoplus_{i=0}^{k-1} q_i = \bigoplus_{i=0}^{k-2} \bar{q}_i \oplus \bigoplus_{i=0}^{k-1} \bar{q}_i = \bar{q}_{k-1}$$
$$q_{k-3} := \bigoplus_{i=0}^{k-1} q_i = \bigoplus_{i=0}^{k-3} \bar{q}_i \oplus \bigoplus_{i=0}^{k-1} \bar{q}_i \oplus \bar{q}_{k-1} = \bar{q}_{k-2}$$
$$\vdots$$
$$q_j := \bigoplus_{i=0}^{k-1} q_i = \ldots = \bar{q}_{j+1}$$
$$\vdots$$
$$q_1 := \bigoplus_{i=0}^{k-1} q_i = \bar{q}_2$$
$$q_0 := \bigoplus_{i=0}^{k-1} q_i = \bar{q}_1$$

on return only q_{k-1} needs correction :

$$q_{k-1} := \bigoplus_{i=0}^{k-1} q_i = \bar{q}_0$$

Now we are prepared to study the general problem of unscrambling the results of the bitswap version of the superparallel FFT. The permutation to be performed is a product of a local bit reversal of each active group (performed differently for each chunk), and a global inverse perfect shuffle of all the chunks. In a given chunk, there are some bits that can be exchanged without changing the site of the chunk. There are other bits that are more critical, since modifying them will change the site of the chunk. We call these bits *site defining*. A modification of the site-defining bits must be coordinated with all the other chunks where the bits are site defining. Note that the site-defining bits may change during the course of the algorithm. At some stage even a linear combination of all the bits in a chunk may be site defining.

Consider an example where the address space consists of five bits. Given a chunk with $q_4 = 1$, $q_0 = 0$, and active group $(3, 1)$, Table 1 shows the site-defining bits during an inverse perfect shuffle.

TABLE 1

Step	Permutation	Site (after permutation)	
0	none	$q_4 = 1;$	$q_0 = 0$
1	$q_4 := \bigoplus_{i=0}^{4} q_i$	$\bigoplus_{i=0}^{4} q_i = 1;$	$q_0 = 0$
2	$q_3 := \bigoplus_{i=0}^{4} q_i = \bar{q}_4$	$q_3 = 1;$	$q_0 = 0$
3	$q_2 := \bigoplus_{i=0}^{4} q_i = \bar{q}_3$	$q_3 = 1;$	$q_0 = 0$
4	$q_1 := \bigoplus_{i=0}^{4} q_i = \bar{q}_2$	$q_3 = 1;$	$q_0 = 0$
5	$q_0 := \bigoplus_{i=0}^{4} q_i = \bar{q}_1$	$q_3 = 1;$	$\bigoplus_{i=0}^{4} q_i = 0$
6	$q_4 := \bigoplus_{i=0}^{4} q_i = \bar{q}_0$	$q_3 = 1;$	$q_4 = 0$

Steps 1, 2, 5, and 6 are critical, and since they change the site, they must be coordinated with the other chunks. Let the motion defined in (9) be denoted the *collective motion*. The idea of the unscrambling algorithm is to force all the chunks to follow the collective motion when they are in a critical section. Outside critical sections we impose a local motion within a chunk in such a manner that the active groups finally become bit reversed.

There is a critical section two bits long when entering an active group (the last bit outside, and the first bit inside, the active group), and one two bits long when

exiting (the two bits succeeding the active group). Thus the movement in the two bits following the active group is forced according to the collective motion. The main problem is to arrange the local movement in the active group such that the two succeeding bits come out correctly when they participate in the collective motion.

The following algorithm solves this problem; thus it solves the problem of unscrambling the bitswap version of the superparallel FFT. For proof of correctness of this algorithm, see [15].

ALGORITHM 7. UNSCRAMBLING OF BITSWAP FFT RESULTS (local motion passing an active group (r, s)). The chunk wants to perform the motion:

$$\begin{array}{c}(\ldots, q_{r+1}, q_r, q_{r-1}, \ldots, \ldots, q_s, q_{s-1}, \ldots, \ldots)\\ \downarrow\\ (\ldots, \ldots, q_{r+1}, q_s, q_{s+1}, \ldots, q_{r-1}, q_r, q_{s-1}, \ldots)\end{array}$$

1. The bits q_{r+1}, q_r, q_{s-1} and q_{s-2} are critical sections. They follow the collective motion:
$$q_j := \bigoplus_{i=0}^{k-1} q_i.$$
2. The bits $q_{r-1}, q_{r-2}, \ldots, q_s$ are to be bit reversed. This is done as in Algorithm 5, with the following modifications:
   - On the way "down," the bits q_{r-1} and q_s are exchanged according to:
$$\begin{aligned} q_{r-1} &:= \textstyle\bigoplus_{i=0}^{k-1} q_i \oplus q_s,\\ q_s &:= \textstyle\bigoplus_{i=0}^{k-1} q_i \oplus q_{r-1}.\end{aligned}$$
   - On return, the bits q_s and q_{r-1} are corrected to their final values according to:
$$\begin{aligned} q_s &:= \textstyle\bigoplus_{i=s-1}^{p} q_i \quad \text{where } p = \lfloor (r+s)/2 \rfloor - 1,\\ q_{r-1} &:= q_{r-1} \oplus q_{s-1}.\end{aligned}$$

The computation of B-settings for the unscrambling of bitswap FFT results takes $\mathcal{O}(p)$ parallel steps, where p is the number of bits that are active for some of the processors.

We illustrate the algorithm with an example.

Example 6. $r = 9; s = 2$. The symmetry point is $p = 5$. The motion of this chunk passing the active group is done as:

$$\begin{aligned} q_9 &:= \textstyle\bigoplus_{i=0}^{k-1} q_i = \bar{q}_{10}\\ q_8 &:= \textstyle\bigoplus_{i=0}^{k-1} q_i \oplus q_2 = \bar{q}_2 \oplus \bar{q}_9\\ q_7 &:= q_7 \oplus q_3 = \bar{q}_3 \oplus \bar{q}_7\\ q_6 &:= q_6 \oplus q_4 = \bar{q}_4 \oplus \bar{q}_6\\ q_5 &:= q_5 = \bar{q}_5\\ q_4 &:= q_4 \oplus q_6 = \bar{q}_6\\ q_3 &:= q_3 \oplus q_7 = \bar{q}_7\end{aligned}$$

$$q_2 := \bigoplus_{i=0}^{k-1} q_i \oplus q_8 = \bar{q}_8 \oplus \bar{q}_6 \oplus \bar{q}_7 \oplus \bar{q}_9$$
$$q_1 := \bigoplus_{i=0}^{k-1} q_i = \bar{q}_9$$
$$q_0 := \bigoplus_{i=0}^{k-1} q_i = \bar{q}_1$$

on return the bits are corrected as

$$q_2 := q_1 \oplus q_2 \oplus q_3 \oplus q_4 = \bar{q}_8$$
$$q_6 := q_6 \oplus q_4 = \bar{q}_4$$
$$q_7 := q_7 \oplus q_3 = \bar{q}_3$$
$$q_8 := q_8 \oplus q_1 = \bar{q}_2$$

In this example $r - s$ is an odd integer, and the bit q_5 is left invariant. When $r - s$ is even the algorithm is similar, but without an invariant bit in the middle.

5. Implementations on MasPar. MasPar is a SIMD machine with up to 16,384 processors arranged in a two-dimensional array, with toroidal wraparound on the edges. The machine used in this study is a $64 \times 128 = 8192$-processor machine at the University of Bergen, Norway. There are essentially three different ways of permuting a data set spread by one element per processor:

1. The xnet. This is a mechanism for sending the data set a given distance d in one of the eight directions: north, northwest, west, and so on. There are no restrictions on the distance d. This mechanism is very fast for short distances, but the time grows proportionally with d; thus long xnets are costly.

2. Piped xnet. This is a fast version of xnet, where the time is (almost) independent on d. It can, however, only be used for sparse data sets, where all the processors between the senders and the receivers are idle. It is very useful for computations of inner products and log sums, but not for our FFTs.

3. The router. This is a general (black box) construct for performing arbitrary permutations of the data set (spread by one element per processor). The underlying hardware is a three-stage crossbar switch. The time for a router call depends on the permutation, but is comparable to an xnet of the longest distance (64). For general permutations it is definitely the cheapest mechanism.

A more detailed general description of the MasPar MP-1 computer can be found in [4], [5], and [20].

The MasPar implementation of the superparallel FFTs is based on the bitswap version of the algorithm. For more details on its use and performance, see [18]. The code is divided into three parts: a symbolic preprocessing phase, the actual transform phase, and the scrambling/unscrambling phase. The symbolic phase computes the twiddle factors and schedules the unscrambling of the results. If the user only wants the results in scrambled order, it computes the necessary pointers for referencing the Fourier coefficients. The symbolic phase needs to be called only once for each configuration of geometries, and the same data is used both for the forward and the inverse FFTs.

In Table 2, we show the elapsed time for various combinations of transforms having combined lengths of 16,384 and 262,144, using 32-bit arithmetic. For simplicity, all transforms have the same length, but we emphasize that any combination of transforms can be processed. These timings *exclude* the time for the symbolic preprocessing. The present version of the symbolic part of the code is not optimized, and uses between 5 and 10 times as much time as the actual transforms. There is, however, substantial room for improvement of this part of the code.

TABLE 2
Time in milliseconds and computational speed for FFT *algorithm* (8192-*processor* MP-1).

Length	#Transforms	Time (ms) without unscrambling	Mflops without unscrambling	Time (ms) with unscrambling	Mflops with unscrambling
Tot. no. of points: 16384					
8	2048	1.1	229	2.2	108
64	256	2.5	198	3.5	139
512	32	4.0	185	5.4	137
8192	2	5.4	199	7.2	147
16384	1	6.1	187	8.1	141
Tot. no. of points: 262144					
8	32768	14.5	270	33.8	117
8192	32	84	203	105	162
262144	1	124	191	175	135

The speed of transforms without unscrambling is, to a moderate degree, dependent on the geometries; when a lot of small transforms are processed, the communication distances are shorter and the speed somewhat higher than for long transforms. The cost of the unscrambling is dominated by the cost of calling the router. This is done once for every set of 8192 points, and takes between 0.5 and 1 millisecond per call (with some special tricks employed to increase the speed). Thus the cost of unscrambling grows linearly in N. This explains the phenomenon that, whereas short transforms are faster than long without unscrambling, long transforms are faster than short with unscrambling.

Acknowledgments. The author would like to thank Professor Petter Bjørstad for suggesting improvements to an early version of the paper and the CERFACS researchers for their hospitality and support.

REFERENCES

[1] D.H. BAILEY, FFTs *in external or hierarchical memory*, J. Supercomput., 4 (1990), pp. 23–35.
[2] V.E. BENEŠ, *Mathematical Theory of Connecting Networks and Telephone Traffic*, Academic Press, New York, 1965.
[3] P. BJØRSTAD, J. COOK, H. MUNTHE-KAAS, AND T. SØREVIK, *Implementation of a* SAR *processing algorithm on MasPar* MP-1208, Rep. No. 57, Dept. of Informatics, Univ. of Bergen, Norway, 1991.
[4] T. BLANK, *The Mas Par* MP-1 *Architecture*, Proc. IEEE Compcon, Spring 1990, IEEE, Feb. 1990.
[5] P. CHRISTY, *Software to support massively parallel computing on the MasPar* MP-1, Proc. of IEEE Compcon, Spring 1990, IEEE, Feb. 1990.
[6] J.W. COOLEY AND J.W. TUKEY, *An algorithm for machine calculations of complex Fourier series*, Math. Comp., 19 (1965), pp. 297–301.
[7] A. GRAHAM, *Kroneker Products and Matrix Calculus: With Applications*, John Wiley, New York, 1981.
[8] D.W. HILLIS AND G.L. STEELE, JR., *Data Parallel Algorithms*, Comm. ACM, 29 (1986), pp. 1170–1189.
[9] S.-T. HUANG AND S.K. TRIPATHI, *Finite state model and compatibility theory: New analysis tools for permutation networks*, IEEE Trans. Comput., C-35 (1986), pp. 591–601.
[10] ———, *Self-routing technique in perfect-shuffle networks using control tags*, IEEE Trans. Comput., 37 (1988), pp. 251–256.
[11] S.L. JOHNSSON, M. JACQUEMIN, AND C.-T. HO, *High radix* FFT *on Boolean cube networks*, Tech. Rep. NA89-7, Thinking Machines Corporation, Cambridge, MA, 1989.
[12] C.L. KUSZMAUL, FFT *communications requirement optimizations on massively parallel architectures with local and global interprocessor communications capabilities*, MasPar report

TW009.0790, presented at International Society for Optical Engineering, July 8–13, San Diego, CA, 1990.

[13] J. LENFANT, *Parallel Permutations of Data: A Beneš network control algorithm for frequently used permutations*, IEEE Trans. Comput., C-27 (1978), pp. 637–647.

[14] H. MUNTHE-KAAS, *Topics in linear algebra for vector and parallel computers*, Ph.D. thesis, Department of Mathematical Sciences, Univ. of Trondheim, Norway, 1989.

[15] ———, *Super Parallel* FFT*s*, Rep. No. 52, Dept. of Informatics, Univ. of Bergen, Norway, 1991.

[16] ———, *Super Parallel* FFT*s with applications to seismic processing*, Proc. IEEE Workshop on Parallel Architechtures for Seismic Data Processing, Glasgow, Scotland, 1991.

[17] ———, *Practical parallel permutation procedures*, Dept. of Informatics, Univ. of Bergen, Norway, 1991.

[18] ———, *MasPar* SPFFT *users guide*, Dept. of Informatics, Univ. of Bergen, Norway, 1992.

[19] D. NASSIMI AND S. SAHNI, *A self-routing Beneš network and parallel permutation algorithms*, IEEE Trans. Comput., C-30 (1981), pp. 332–340.

[20] J. NICKOLLS, *The design of the MasPar* MP-1, *A cost effective massively parallel computer*, Proc. IEEE Compcon, Spring 1990, IEEE, Feb. 1990.

[21] S.D. PARKER, *Notes on shuffle/exchange-type switching networks*, IEEE Trans. Comput., C-29 (1980), pp. 213–222.

[22] D. PARKINSON, *Super parallel algorithms*, in Supercomputing, NATO ASI series F, Vol. 62, Springer-Verlag, Berlin, New York, 1989.

[23] M.C. PEASE, *The indirect binary n-cube microprocessor array*, IEEE Trans. Comput., C-26 (1977), pp. 548–573.

[24] D.J. ROSE, *Matrix identities of the fast Fourier transform*, Linear Algebra Appl., 29 (1980), pp. 423–443.

[25] H.S. STONE, *Parallel processing with the perfect shuffle*, IEEE Trans. Comput., C-20 (1971), pp. 153–161.

[26] P.N. SCHWARZTRAUBER, FFT *algorithms for vector computers*, Parallel Comput., 1(1984), pp. 45–63.

[27] ———, *Multiprocessor* FFTs, Parallel Comput., 5(1987), pp. 197–210.

[28] C. WU AND T. FENG, *The universality of the shuffle-exchange network*, IEEE Trans. Comput., C-30 (1981), pp. 324–331.