Fast Digit-Index Permutations

DOROTHY BOLLMAN[1], JAIME SEGUEL[1], AND JOHN FEO[2]

[1]*Department of Mathematics, University of Puerto Rico, Mayaguez, PR 00681*
[2]*Computer Research Group, Lawrence Livermore National Laboratory, Livermore, CA 94550*

ABSTRACT

We introduce a tensor sum which is useful for the design and analysis of digit-index permutations (DIPs) algorithms. Using this operation we obtain a new high-performance algorithm for the family of DIPs. We discuss an implementation in the applicative language Sisal and show how different choices of parameters yield different DIPs. The efficiency of the special case of digit reversal is illustrated with performance results on a Cray C-90. © 1996 John Wiley & Sons, Inc.

1 INTRODUCTION

Kronecker's matrix product is a powerful tool for designing and adapting fast Fourier transform (FFT) algorithms to different multiprocessing environments [6, 9, 10]. This is especially true for implementations in an applicative language. Starting with an FFT algorithm expressed in tensor notation, one can use the laws of tensor algebra to obtain tensor formulas that take into account a given architecture and are easily translated into the language. The Kronecker product also provides a unifying framework for seemingly different FFT algorithms. Under Kronecker's formulation, for instance, Pease's parallel algorithm and the Korn-Lambiote vector FFT are simple variants of the original Cooley-Tukey FFT. In [2] the authors have exploited these ideas to obtain a high-performance FFT written in the applicative language Sisal.

The Cooley-Tukey FFT, as well as the Pease and the Korn-Lambiote algorithms, consists of a "combine" phase and a separate "ordering" phase in which the components of a vector need to be written in bit-reversed order. An efficient way to implement bit reversal on a vector machine such as a Cray is by computing a vector index of bit-reversed indices and then using gather/scatter to effect the reordering. Programmers have used various algorithms to compute the index vector; however, the development of these algorithms has been somewhat ad hoc and there is no general tool comparable to the Kronecker product, which is so useful for the combine phase, for designing and analyzing bit-reversal algorithms.

In this article we develop a tensor sum whose role in the ordering phase is similar to the role of the tensor product in the combine phase. Just as the tensor product serves as a unifying framework for the combine phase of various FFT algorithms, the tensor sum serves as a unifying framework for digit-reversal algorithms. We show how different known digit-reversal algorithms can be expressed in terms of the tensor sum. This observation allows us to obtain a new algorithm for the general class of digit-index permutations (DIPs) whose parallel-time complexity is $O(\log \log n)$. We give a Sisal implementation for this

Received April 1995
Revised June 1995

Scientific Programming, Vol. 5, pp. 137–146 (1996)
CCC 1058-9244/96/020137-10

algorithm and illustrate how different choices of parameters yield different members of the DIP family.

The results presented here provide a complete set of tools for designing, modifying, and implementing radix-r FFT permutations in a functional language such as Sisal. In other work, which will appear, the authors exploit the tensor product to develop a similar set of tools for the combine phases of radix-r FFTs. In this latter work we show that there are just four basic functions, in the sense that the combine phase of each of the radix-r FFTs can be expressed as the composition of some of the basic functions. The totality of these results allows us to develop high-performance Sisal implementations for the entire family of radix-r FFTs.

2 AN OVERVIEW OF SISAL

Sisal (Streams and Iteration in a Single-Assignment Language) is an applicative language developed primarily by Lawrence Livermore National Laboratory and Colorado State University for large-scale scientific programming. It is a "single-assignment" language—a name can receive a value only once in each name scope. Sisal has been successively implemented on shared memory machines such as the Cray family and work is currently underway to implement it on distributed memory machines.

Sisal is a very convenient language for prototyping, developing, and evaluating parallel algorithms since it allows the programmer to write programs that are very close to their original mathematical formulations. One can program and test individual functions and compose them with the guarantee of not introducing race conditions and side effects. Furthermore, there is growing evidence [3] that Sisal can perform on a par with traditional languages such as Fortran.

The authors [2, 5] have successively used Sisal to develop FFTs that outperform their Fortran counterparts on Crays. We express the algorithms developed in this work in Sisal, not only because these algorithms complement the ideas in [2], but also because the use of Sisal greatly simplifies the expression of these algorithms as opposed to an imperative language such as Fortran.

In this section we present only sufficient detail of the language to understand the algorithms presented in this work. A much more complete and very readable account of Sisal is given in [8].

Sisal comments begin with a % character and continue to the end of the line. The most important data type in Sisal for the ideas considered here is the **array** which is a one-dimensional collection of homogeneous values indexed by simple integer expressions. Multiple dimensional arrays are arrays of arrays. The size of an array is dynamic. Array operations include array construction, replacement, selection, and operations to redefine the lower bound, **array_setl**(A,lo), to add a new element at the beginning of the array, **array_addl**(A,v), and to add a new element at the end of the array, **array_addh**(A,v).

There are two types of loop expressions in Sisal: **for initial** and **for**. The former is an iterative construct having four parts: initialization, body, test, and returns clause. The initialization clause is the loop's first iteration and defines loop constants and the initial values of all loop carried names. The body redefines the values of all loop-carried names. The values of the names on the previous iteration can be accessed as **old** *name*. Thus, the **for initial** expression supports iteration while preserving single-assignment semantics. The test clause may be either **while** *Boolean expression* or **until** *Boolean expression*. The test clause can appear either before or after the body. The **returns** clause determines the value of the **for initial** expression. The **returns** clause has various forms, including: **returns value of** *expression* that returns the expression's value on the last iteration; **returns array of** *expression* that returns an array of the expression's value on each iteration; and **returns value of catenate** *expression* that returns the catenation of the arrays defined by *expression*.

The syntax of a **for** expression has three parts: generator, body, and returns clause. An instance of the body is executed for every element in the "generator." The generator can have the form **for** *name* **in** *lo,hi* that specifies that *name* takes on the values *lo* to *hi*, inclusive, and an instance of the body is executed for each value. Another useful form of the generator is **for** *name* **in** *array_name*. Here, *name* takes on the values of the array's elements, and an instance of the body is executed for each element.

The semantics of the **for** expression are such that the loop bodies are data independent and may be executed in parallel. The reduction operations specified in the returns clause are implemented determinantly by the run-time system. The **for** expression is the major source of parallelism and vectorization in Sisal programs on most commercial computer systems.

3 A TENSOR ADDITION

We shall develop a tensor addition whose role in the ordering phase of an FFT is similar to the tensor product in the combine phase. Although occasional references to tensor products can be found in early work on FFTs, it is only in the last several years [6, 9, 10] that its importance to FFTs, especially in matching specific FFT algorithms to specific architectures, has been fully appreciated. Here we remark only briefly on its role.

Let $A = [a_{kl}]$ and B be matrices of arbitrary sizes. The tensor product of A by B is defined to be the block matrix

$$A \otimes B = [a_{kl}B]$$

Different sparse-matrix factorizations of the N-point discrete Fourier transform (DFT) matrix F_N, formed by different Kronecker product decompositions, yield different computational schemes for the DFT. The main use of these Kronecker factors in matching algorithms and architectures comes from the identity $A \otimes B = (A \otimes I_n)(I_m \otimes B)$ where I_s is the $s \times s$ identity matrix, and A and B are $m \times m$ and $n \times n$ matrices, respectively. The factor $I_m \otimes B$ is then regarded as m parallel multiplications of B on n-point vectors, while the factor $A \otimes I_n$ is regarded as A multiplying a set of m-point vectors in a pipeline of length n. The so-called commutation formula $P(nm, n)(A \otimes B)P(nm, m) = B \otimes A$, where $P(nm, s)$ is the stride s permutation, allows the change from a parallel to a pipelined interpretation and vice versa. For example, the decomposition $F_{2^{2r}} = (F_{2^r} \otimes I_{2^r})T(I_{2^r} \otimes F_{2^r})P(2^{2r}, 2^r)$, where T is a diagonal matrix of the so-called "twiddle factors," is turned into a parallel algorithm by commuting the leftmost factor. The same formula can be turned into a vector algorithm by commuting the rightmost factor. Kronecker formulation thus provides a mathematical framework for FFT dataflow modification and identification of core operations.

We shall see that our tensor sum is a very natural companion to the tensor product. We define the *tensor sum* $u \oplus v$ of u and v as follows:

For any scalars u and v, define $u \oplus v = u + v$.

For any vector $u = [\alpha_0, \alpha_1, \ldots, \alpha_{r-1}]$ and any scalar v define $u \oplus v = v \oplus u = [\alpha_0 + v, \alpha_1 + v, \ldots, \alpha_{r-1} + v]$.

For any vectors $u = [\alpha_0, \alpha_1, \ldots, \alpha_{r-1}]$ and $v = [\beta_0, \beta_1, \ldots, \beta_{s-1}]$ define

$$u \oplus v = [u \oplus \beta_0, u \oplus \beta_1, \ldots, u \oplus \beta_{s-1}]. \quad (1)$$

For our purposes here, we assume that all of the scalars in the definition are integers.

The subvectors $u \oplus \beta_i$ in the above definition can be computed in parallel. On a vector machine each of these operations is a vector operation. An implementation in Sisal is

```
type OneDim = array[integer]
function oplus(u, v: OneDim returns OneDim)
for comp_v in v returns value of catenate
    for comp_u in u
    returns array of comp_u + comp_v
    end for
end for
end function
```

It is easy to see that the tensor sum satisfies:

Associativity: $v \oplus (u \oplus z) = (v \oplus u) \oplus z$ for all vectors or scalars v, u, and z.

Distributivity: $\alpha(v \oplus u) = \alpha v \oplus \alpha u$, for v and u vectors or scalars and α scalar

Identity: $v \oplus 0 = 0 \oplus v = v$ for all vector or scalar v.

Also, the tensor sum is:

Noninvertible: unless both terms are scalars.

Noncommutative: unless one of the terms is a scalar.

For $u_0, \ldots, u_{k-1}$, vectors in C^r, we use the notation

$$\bigoplus_{i=0}^{k-1} u_i = u_0 \oplus \cdots \oplus u_{k-1} \quad (2)$$

For any nonnegative integer r, let $V_r = [0, 1, \cdots, r-1]$. As an example of this notation, note that

$$V_3 \oplus 3V_3 \oplus 9V_3 = [0, 1, 2] \oplus [0, 3, 6] \oplus [0, 9, 18] = V_{27}.$$

In general,

$$V_{r^k} = V_{r^{k-1}} \oplus r^{k-1}V_r \quad (3)$$

and by induction on k we have

$$V_{r^k} = r^0V_r \oplus r^1V_r \oplus \cdots \oplus r^{k-1}V_r \quad (4)$$

4 DIPs

We shall sometimes describe permutations in terms of cyclic permutations, which are permuta-

tions of the form $p(a_1) = a_2$, $p(a_2) = a_3$, $\cdots$ $p(a_{m-1}) = a_m$, $p(a_m) = a_1$. Such a permutation is denoted by the cycle $(a_1, a_2, \cdots, a_m)$. By a well-known result from elementary group theory, every permutation can be written as a product of disjoint cycles (i.e., no two cycles alter the same symbol).

Let $p: N_k \to N_k$ be a fixed but arbitrary permutation, where for any positive integer k, N_k denotes the set $\{0, 1, 2, \cdots, k-1\}$. The permutation defined by

$$P: \qquad N_r{}^k \to N_r{}^k \tag{5}$$

$$j = \Sigma_{i=0}^{k-1} j_i r^i \to P(j) = \Sigma_{i=0}^{k-1} j_i r^{p(i)} \tag{6}$$

is said to be the DIP on $N_r{}^k$ induced by p. Thus, the image of an integer $j \in N_r{}^k$ is obtained by expressing j in base r and using p to permute the base r digits of j.

The following result provides the basis for a generic algorithm for the DIPs:

Theorem. Let P be a DIP on $N_r{}^k$ induced by p: $N_k \to N_k$. Then

$$P(V_r{}^k) = \bigoplus_{i=0}^{k-1} r^{p(i)} V_r \tag{7}$$

Proof. We show that $P(j)$ is equal to the j-th element of $\bigoplus_{i=0}^{k-1} r^{p(i)} V_r$. Let us first define what we mean by the antilexicographical order on a Cartesian product of ordered sets. For any ordered set A, the antilexicographical order on A is the order on A. If A and B are ordered sets with respective cardinalities m and n, and if a_i and b_i are the i-th elements of A and B, respectively, then the antilexicographical order on $A \times B$ is defined as follows: The first element of $A \times B$ is $\langle a_1, b_1 \rangle$. If $\langle a_r, b_s \rangle$ is the i-th element, where $1 \le i < mn$, then the $i+1$-th element is $\langle a_{r+1}, b_s \rangle$ if $r < m$ and is $\langle a_1, b_{s+1} \rangle$ if $r = m$. The antilexicographical order on $\times_{i=1}^{m} A_i$ where each A_i is an ordered set is the antilexicographical order on $(\times_{i=1}^{m-1} A_i) \times A_m$ and where we identify each $\langle \langle c_1, \ldots, c_{m-1} \rangle, c_m \rangle$ with $\langle c_i, \ldots, c_m \rangle$.

For any vectors $u_j = [u_j[0], u_j[1], \cdots, u_j[r-1]]$, $j = 0, \ldots, k-1$, the j-th component of $\bigoplus_{i=0}^{k} u_i$ is $u_0(j_0) + i_1(j_1) + \cdots + u_{k-1})$ where $\langle j_0, j_1, \ldots, j_{k-1} \rangle$ is the j-th element in the antilexicographical order of $\times_{j=0}^{k-1} N_r$. Thus, the j-th component of $\bigoplus_{i=0}^{k-1} r^{p(i)} V_r$ is

$$\sum_{i=0}^{k-1} r^{p(i)} V_r(j_i) = \sum_{i=0}^{k-1} j_i r^{p(i)} = P(j) \quad \square$$

It is shown in [1] that the set of all DIPs on $N_r{}^k$ is precisely the group of permutations generated by the generalized "shuffles," i.e., permutations GS_{r^k, r^i} induced by the cycles $c_i = (i, i+1, \ldots, k-1)$. Thus, a permutation on $N_r{}^k$ is a DIP if and only if it can be expressed as a product of GS_{r^k, r^i}'s. In actual practice, the most convenient description of a DIP on $N_r{}^k$ is in terms of the permutation p on N_k that induces it. In what follows we examine some examples of DIPs of special interest and give their vector representations based on Equation 7.

Example 1. Digit Reversal

These are perhaps the best known DIPs. Such a permutation $R_r{}^k : N_r{}^k \to N_r{}^k$, is induced by

$$p(i) = k - i - 1, \quad 0 \le i \le k-1. \tag{8}$$

In terms of Equation 7, digit reversal on $V_r{}^k$ is expressed as

$$R_r{}^k(V_r{}^k) = r^{k-1} V_r \oplus r^{k-2} V_r \oplus \cdots \oplus r^0 V_r. \tag{9}$$

For example, digit reversal for $r = 3$, $k = 2$ is induced by the permutation $(0, 1)$ and its vector representation is

$$R_{3^2}(V_{3^2} = 3[0, 1, 2] \oplus [0, 1, 2] \tag{10}$$

$$= [0, 3, 6, 1, 4, 7, 2, 5, 8] \tag{11}$$

An important special case arises when $r = 2$, i.e., "bit reversal," which we denote simply by B_k.

Example 2. Stride Permutations

A stride-r^i on $N_r{}^k$ is induced by a left cyclic shift of length i on N_k. We consider some special cases in the examples that follow.

Example 3. Even-Odd Sort

This is the stride 2 permutation, E_k, on N_{2^k}. Thus, E_k groups all even indices and all odd indices and is induced by $(0, 1, 2, \cdots, k-1)$. By the theorem we have,

$$E_k(V_{2^k} = 2V_2 \oplus 2^2 V_2 \oplus \cdots \oplus 2^{k-1} V_2 \oplus V_2 \tag{12}$$

For example, E_3 is induced by $(0, 1, 2)$ and its vector representation is

$$E_3(V_{2^3} = 2[0, 1] \oplus 2^2[0, 1] \oplus [0, 1] \tag{13}$$

$$= [0, 2, 4, 6, 1, 3, 5, 7] \tag{14}$$

Example 4. Perfect Shuffles
These are the permutations S_{r^k} or stride-r^{k-1} permutations, which are induced by the left cyclic shifts $(0, k-1, k-2, \cdots, 1)$. Thus

$$S_{r^k}(V_{r^k}) = r^{k-1}V_r \oplus V_r \oplus rV_r \oplus \cdots \oplus r^{k-2}V_r. \quad (15)$$

For example, if $r = 2$ and $k = 3$, then S_8 is induced by $(0, 2, 1)$ and

$$S_{2^3}(V_8) = 2^2[0, 1] \oplus [0, 1] \oplus 2[0, 1] \quad (16)$$

$$= [0, 4, 1, 5, 2, 6, 3, 7] \quad (17)$$

Example 5. Matrix Transpositions
Given an $m \times n$ matrix A which is stored in row major order (as in Sisal), the transpose of A can be regarded as a stride-n permutation on V_{mn}. For square matrices of order, say, n, transposition can be represented as the DIP on N_{n^2} induced by the permutation $(0, 1)$. Thus,

$$Transpose(V_{n^2}) = nV_n \oplus V_n \quad (18)$$

For example, if

$$A = \begin{bmatrix} a[0,0] & a[0,1] & a[0,2] & a[0,3] \\ a[1,0] & a[1,1] & a[1,2] & a[1,3] \\ a[2,0] & a[2,1] & a[2,2] & a[2,3] \\ a[3,0] & a[3,1] & a[3,2] & a[3,3] \end{bmatrix}, \quad (19)$$

$$\begin{bmatrix} a[0] & a[4] & a[8] & a[12] \\ a[1] & a[5] & a[9] & a[13] \\ a[2] & a[6] & a[10] & a[14] \\ a[3] & a[7] & a[11] & a[15] \end{bmatrix} \quad (20)$$

then the one-dimensional array a occupied by A can be envisioned by

$$\begin{bmatrix} a[0] & a[1] & a[2] & a[3] \\ a[4] & a[5] & a[6] & a[7] \\ a[8] & a[9] & a[10] & a[11] \\ a[12] & a[13] & a[14] & a[15] \end{bmatrix} \quad (21)$$

The first row of A is indexed by V_4, the second by $V_4 \oplus 4$, and, in general, the j-th row is indexed by $V_4 \oplus 4j$. So, V_4 is associated with the rows while $4V_4$ is associated with the columns. Thus, although $V_4 \oplus 4V_4$ is a one-dimensional indexing vector, it preserves, by means of the tensor sum decomposition, the two-dimensional features of the original array. The indexing set $4V_4 \oplus V_4$ of the transpose can also be interpreted as a two-dimensional indexing set by associating $4V_4$ with rows and V_4 with columns. The resulting array is then

$$\begin{bmatrix} a[0] & a[4] & a[8] & a[12] \\ a[1] & a[5] & a[9] & a[13] \\ a[2] & a[6] & a[10] & a[14] \\ a[3] & a[7] & a[11] & a[15] \end{bmatrix} \quad (22)$$

the transpose of A.

Example 6. A DIP for 2D-FFTs
From its definition it follows that the 2D-DFT of size $M \times L$ can be computed in terms of one-dimensional transforms, as follows:
Step 1. Compute the L-point transforms of each of the M rows
Step 2. Compute the M-point transforms of each of the L columns of the result of Equation 1.
Assuming that x is stored in row major order, as in Sisal, a common approach to this problem is to perform a transpose after Step 1, then again compute transforms of rows, and then another transpose.

An efficient way to effect the transposes in the above method would be to combine them with the ordering phases of Steps 1 and 2. For example, suppose that the Gentleman-Sande algorithm (where digit reversal follows the combine phase) is used to compute the row transforms. Then the four separate permutations, two row-wise digit reversals and two transposes, could be replaced by two single DIPs, each consisting of the composition of row-wise digit reversal with transposition.

For example, if $M = 2^2 = 4$ and $L = 2^3 = 8$, then row-wise bit reversal is represented by the permutation on N_{2^5} which is induced by $(0, 2)(1)(3)(4)$ on N_5, while the transposition of a 4×8 matrix is represented by the permutation on N_{2^5} which is induced by $(0, 3, 1, 4, 2)$ on N_5. The DIP representing the composition of row-wise bit reversal and transposition of the $M \times L$ matrix is then the permutation P_1 on N_{2^5} induced by $(0, 3, 1, 4, 2)(0, 2) = (0, 3, 1, 4)(2)$. Similarly, the DIP representing the composition of row-wise bit reversal and transposition of the $L \times M$ transpose is the permutation P_2 on N_{2^5} which is induced by $(0, 2, 4, 1, 3)(0, 1) = (0, 2, 4)(1, 3)$. Thus, the vector representations for the two DIPs needed for computing the two-dimensional 4×8 FFT by the above method are

$$P_1(V_{2^5}) = 2^3V_2 \oplus 2^4V_2 \oplus 2^2V_2 \oplus 2V_2 \oplus V_2 \quad (23)$$

and

$$P_2(V_{2^5}) = 2^2V_2 \oplus 2^3V_2 \oplus 2^4V_2 \oplus 2V_2 \oplus V_2 \quad (24)$$

5 ALGORITHM DESIGN

Several algorithms for computing $B_k(V_{2^k})$ have been designed. Among them, the better known are the "Double-Add-One" (DAO) and Elster's algorithm (EA) [4]. Both algorithms, as well as their generalizations to R_{r^k}, can be interpreted as simple variants of Equation 7. For this purpose let us first observe that

$$\alpha(u \oplus v) = \alpha u \oplus \alpha v \tag{25}$$

for any scalar α. The distributivity of $\oplus$ allows the "Horner-like" nesting of (7). This nested computation of $B_k(V_{2^k})$ constitutes the DAO algorithm. For example, since

$$\begin{aligned} B_4(V_{16}) &= 2^3[0,1] \oplus 2^2[0,1] \oplus 2[0,1] \oplus 2^0[0,1] \\ &= 2(2(2[0,1] \oplus [0,1]) \oplus [0,1]) \oplus [0,1] \end{aligned}$$

we have

$$\begin{aligned} B_4 &= 2(2[0,2,1,3] \oplus [0,1]) \oplus [0,1] \\ &= 2[0,4,2,6,1,5,3,7] \oplus [0,1] \\ &= [0,8,4,12,2,10,6,14,1,9,5,13,3, \\ &\quad 11,7,15]. \end{aligned}$$

In general,

$$B_i(V_{2^i}) = 2B_{i-1}(V_{2^{i-1}}) \oplus V_2 \tag{26}$$

and $B_k(V_{2^k})$ can be computed in $k-1$ sequential steps. The maximum number of scalar operations occurs in the last step and is $2^{k-1} + 2^{k-1} = n$.

EA, in turn, is based on associating to the left in Equation 7. For example, since

$$\begin{aligned} B_4(V_{16}) &= 2^3[0,1] \oplus 2^2[0,1] \oplus 2[0,1] \oplus 2^0[0,1] \\ &= (([0,8] \oplus [0,4]) \oplus [0,2]) \oplus [0,1] \end{aligned}$$

we have

$$\begin{aligned} B_4 &= ([0,8,4,12] \oplus [0,2]) \oplus [0,1] \\ &= [0,8,4,12,2,10,6,14] \oplus [0,1] \\ &= [0,8,4,12,2,10,6,14,1,9,5,13,3, \\ &\quad 11,7,15]. \end{aligned}$$

Let us compare these algorithms within the PRAM framework. We assume that such a model consists of p processors connected to a shared random access memory M. The input for an algorithm is assumed to be in some designated memory cells and the output is to be placed in some other designated cells. All processors run the same program. Each time step has three phases: The read phase in which each processor may read from a memory cell, the computation phase in which each processor does a computation, and the write phase in which each processor may write to a memory cell. We assume the "CREW" model which allows concurrent reads but only exclusive writes. The parallel complexity or parallel time is the total number of steps executed, expressed, as a function of n, to a solve a problem of size n.

A **PRAM** can compute $u \oplus v$ in one time step and using rs processors, where r and s are the respective sizes of u and v, as follows: Let $u = [\alpha_0, \alpha_1, \ldots, \alpha_{r-1}]$ and $v = [\beta_0, \beta_1, \ldots, \beta_{s-1}]$. For each $i = 1, \ldots, s$, each of r processors p_{ij} reads an α_j, $j = 0, 1, \ldots, r-1$ and β_i, and each of the rs processors p_{ij} then adds its α_j and β_i and writes its result in a distinct memory cell.

It is easy to see that both DAO and EA require $O(\log n)$ parallel steps and $O(n)$ processors. However, by performing successive pairwise sums in Equation 7, we can asymptotically reduce the complexity to $(O(\log \log n), O(n))$ much in the same way that the parallel time of computing an associative sum of $n = 2^k$ elements is reduced from $O(n)$ to $O(\log n)$ in the familiar prefix sums algorithm. For example,

$$\begin{aligned} B_4 &= 2^3[0,1] \oplus 2^2[0,1] \oplus 2^1[0,1] \oplus 2^0[0,1] \\ &= ([0,8] \oplus [0,4]) \oplus ([0,2] \oplus [0,1]) \\ &= [0,8,4,12] \oplus [0,2,1,3] \\ &= [0,8,4,12,2,10,6,14,1,9,5,13,3, \\ &\quad 11,7,15]. \end{aligned}$$

Hence, the pairwise computation of Equation 7 can be represented by a tree of height $\lceil \log k \rceil$ where each node represents a (tensor addition) time step. This gives an algorithm of parallel complexity $O(\log k) = O(\log \log n)$, where $n = r^k$. The maximum number of processors used, which occurs in the last step, is n.

One of the problems of implementing the $O(\log \log n)$-parallel time algorithm is the representation of the variable number of vectors in each of the iterative steps. One possibility is to represent each

such set of vectors by their concatenation, but this vector is of variable length. Furthermore, keeping track of the indices of the components of the individual vectors within the large vectors is very difficult. This problem can be completely eliminated in Sisal by representing the set of vectors by a dynamically sized array of dynamically sized arrays.

6 SISAL IMPLEMENTATIONS

The following Sisal function T ("tree") computes the tensor sum (7) by repeated pairwise applications of $\oplus$. We assume the data type *OneDim* as previously defined.

```
type TwoDim =array[OneDim]
function T(r,k: integer; p: vector
          returns OneDim)
for initial
    j := k;
% The rows of Bj are the initial terms of the tree
    Bj := for c in p cross i in 0, r - 1
          returns array of exp(r,c) * i
          end for
while j> 1 repeat
    j := (old j + 1)/2;
% The following expression executes a tree reduction
% of the rows of Bj under the oplus reduction
    Bj := for i in 0, (old j/2) - 1
          returns array of
           oplus(old Bj[2 * i], old Bj[2 * i + 1])
          end for
          || % catenate on odd row if it exists
          if odd(old j) then
           array[0: old Bj[old j - 1]]
          else
           array TwoDim[ ]
          end if
returns value of Bj[0]
end for
end function
```

The function T provides a generic algorithm for computing the vector representation of any DIP P in parallel time $O(\log \log n)$, $n = r^k$. One only needs to correctly supply the values of the permutation p on N_k that induces P. For digit reversal we use $p = [k-1, k-2, \cdots, 1, 0]$:

```
function digit_reversal(r,k: integer
                        returns OneDim)
let
    p := for i in 0, k - 1
         returns array of k - i - 1
         end for
in
    T(r,k,p)
end let
end function
```

For a perfect shuffle or stride r^{k-1} we need to compute $p = [k-1, 0, 1, \cdots, k-2]$:

```
function shuffle(r,k: integer returns OneDim)
let
    p := for i in 1, k - 1
         returns array of i - 1
         end for
in
    T(r,k,array_addl(p,k - 1))
end let
end function
```

To obtain "even-odd" sort we need $p = [1, 2, \cdots, k-1, 0]$:

```
function even_odd(r, k: integer returns OneDim)
let
    p := for i in 0, k - 2
         returns array of i + 1
         end for
in
    T(r, k, array_addh(p, 0))
end let
end function
```

7 DIGIT REVERSAL

We thus see that the function T constitutes a $O(\log \log n)$ parallel time algorithm for the whole family of DIPs. In particular, we obtain a digit-reversal algorithm, which we term "tree-digit-reversal" ("TDR") by computing T with $p(i) = r^{k-i}$, $i = 1, 2, \cdots, k$.

In previous work [7] we developed another $O(\log \log n)$ time algorithm, "recursive divide and conquer" ("RDC") for computing digit reversal. We conclude with a comparison between these two algorithms.

It follows from Equation 7 that

$$R_{r^k} = r^{k_1} R_{r^{k_2}} \oplus R_{r^{k_1}} \tag{27}$$

where $k_1 = k_2 = \dfrac{k}{2}$ if k is even and $k_1 = \dfrac{k+1}{2}$ and $k_2 = \dfrac{k-1}{2}$ if k is odd. Algorithm RDC results by repeated application of this recurrence. For example, $R_{2^5} = B_5$ can be computed in the following three steps:

$$B_2 = 2^1 \cdot B_1 \oplus B_1$$
$$B_3 = 2^2 \cdot B_1 \oplus B_2$$
$$B_5 = 2^3 \cdot B_2 \oplus B_3$$

In general, R_{r^k} can be computed by applying recurrence (27) $\lceil \log k \rceil$ times. Thus, RDC has the same parallel complexity and uses the same number of processors as TDR. However, RDC uses slightly fewer scalar operations than TDR. This is because RDC makes use of the fact that in each of the sequential steps the vectors $r^{k_1} R_{r^{k_2}}$ and $R_{r^{k_1}}$ can be obtained from the previous step. This is true even in case $k_1 \neq k_2$ since $r^{k_1} R_{r^{k_2}}$ consists of the first r^{k_2} components of $r^{k_2} R_{r^{k_1}}$.

Let us examine the relationship between the number $N_{\mathrm{TDR}}(n)$ of scalar operations used by TDR and the number $N_{\mathrm{RDC}}(n)$ used by RDC in a little more detail. This is easiest to see when k is a power of 2, say $k = 2^m$. The computation done by TDR can be visualized by a complete binary tree T of height m where each node represents a tensor sum. Numbering the levels $0, 1, \cdots, m-1$ from leaves to root, on level i TDR computes 2^{m-i} tensor sums of vectors, each of length $r^{2^{i-1}}$. Thus

$$N_{\mathrm{TDR}} = \sum_{i=1}^{m} 2^{m-i} r^{2^i} \tag{28}$$

On the other hand, RDC computes the product of a scalar by a vector of length $r^{2^{i-1}}$ and just one tensor sum of vectors of length $r^{2^{i-1}}$ on each level of the tree and hence

$$N_{\mathrm{RDC}} = \sum_{i=1}^{m} (r^{2^{i-1}} + r^{2^i}) \tag{29}$$

It is straightforward to show that for fixed r, both N_{TDR} and N_{RDC} are $O(n)$. The exact values are very nearly equal. For example,

$$N_{\mathrm{TDR}}(3^{16}) = 2^3 \cdot 3^2 + 2^2 \cdot 3^{2^2} + 2 \cdot 3^{2^3} + 3^{2^4}$$
$$= 43066890,$$

whereas

$$N_{\mathrm{RDC}}(3^{16}) = (3^2 + 3) + (3^{2^2} + 3^2) + (3^{2^3} + 3^{2^2})$$
$$+ (3^{2^4} + 3^{2^3}) = 43060026$$

Actually, we could reduce the number of scalar operations used in both RDC and TDR. This is because each application of the tensor sum in each of the algorithms involves vectors whose first components are zero. For two such vectors of length r, we could save r operations alone by just noting that $u \oplus [0, v_1, \cdots, v_{r-1}] = [u, u \oplus v_1, \cdots, u \oplus v_{r-1}]$.

Taking advantage of this economy requires a slightly more complicated version of the Sisal function *oplus*. However, such a change has little noticeable effect.

First, the change has very little effect on the total number of scalar operations needed. Recalculating Equations 28 and 29 we obtain

$$N'_{\mathrm{TDR}} = \sum_{i=1}^{m} 2^{m-i} r^{2^{i-1}} - 1) \tag{30}$$

and

$$N'_{\mathrm{RDC}} = \sum_{i=1}^{m} r^{2^i} \tag{31}$$

N'_{TDR} is still slightly greater than N'_{RDC}, but the difference is even smaller than the difference between N_{TDR} and N_{RDC}. For example, $N'_{\mathrm{TDR}} = 43053456$ and $N'_{\mathrm{RDC}} = 43053372$.

Second, and most importantly, experiments confirm that this change has virtually no effect on the running times of either RDC or TDR, nor does it have any noticeable effect on EA.

The following Sisal function implements RDC. We have eliminated recursion by using a function *ks* to precompute vectors that give the values of k_1 and k_2, respectively.

```
function rdc(r,k: integer returns OneDim)
for initial
    i := lg(k,2);
    k1, k2 := ks(k,i);
    v := for i in 0, r - 1
         returns array of i end for
while i > 1 repeat
    i := old i - 1;
    x := exp(r,k2[old i]);
    u := for vi in old v
         returns array of x * vi end for;
    v := array_setl(oplus(u, old v),0)
returns value of v
end for
end function
```

8 PERFORMANCE

TDR and RDC are both excellent parallel algorithms. The last step ($i = m$ in Equations 28 and 29) comprises almost all the work. In the experiments we ran, 99% of the execution time was spent on the last step in function *oplus*. The overhead of the tree reduction and management of the data structures is inconsequential.

Table 1 gives the execution times in seconds for the Elster (EA), TDR, and RDC algorithms on a 16-processor Cray C-90 with 2GB of main memory. For $r = 2$, TDR and RDC are approximately twice as fast as EA, and the speedup for all three algorithms is nearly linear. RDC is slightly faster than TDR, but the difference is insignificant. Both algorithms execute approximately 285 million integer operations per second per processor. Since the Sisal version of TDR comprises a triply-nested **for** expression,

Table 1. Performance Execution Times on C-90

r^k	Algorithm	P1	P2	P3	P4
	EA	.1177	.0605	.0410	.0322
2^{24}	RDC	.0598	.0301	.0203	.0177
	TDR	.0602	.0307	.0222	.0169
	EA	.2337	.1187	.0796	.0602
2^{25}	RDC	.1193	.0603	.0403	.0313
	TDR	.1183	.0605	.0404	.0345
	EA	.4652	.2369	.1571	.1183
2^{26}	RDC	.2347	.1187	.0786	.0592
	TDR	.2343	.1178	.0791	.0596
	EA	.6759	.3392	.2270	.1732
2^{27}	RDC	.4696	.2350	.1569	.1181
	TDR	.4666	.2339	.1565	.1179
	EA	.2406	.1235	.0810	.0651
3^{16}	RDC	.1644	.0829	.0560	.0431
	TDR	.1620	.0825	.0579	.0459
	EA	.6759	.3392	.2269	.1724
3^{17}	RDC	.4638	.2328	.1555	.1172
	TDR	.6027	.3769	.2042	.1920

for *each pair of row in Bj*
  for *each element in the second row of the pair*
    for *each element in the first row of the pair*

and there is only one pair of rows on the last step, we compiled the code with the $-n^2$ flag to force the compiler to parallelize the first two levels of the nested expression. Note that the Cray Sisal compiler always vectorizes the innermost **for** expression or slices thereof.

For $r = 3$, performance of the three algorithms is similar to $r = 2$; however, TDR performs poorly for 3^{17}. The algorithm's poor speedup on two and four processors for this data size is an effect of static loop slicing. For $k = 17$, the algorithm builds five instances of the *Bj* array with 17, 9, 5, 3, and 2 rows, respectively. On the last step, the first row has 3^{16} and the second row has three elements. Since we instructed the compiler to parallelize only the first two levels of the triply-nested **for** expression, concurrency is realized only across the three elements of the second row; thus, the code suffers from poor load balance on two processors and insufficient parallelism on four processors. We can improve the performance by instructing the compiler to parallelize all three levels, but the correct general solution to this problem is for the Sisal compiler to support dynamic loop slicing.

9 SUMMARY AND CONCLUSIONS

We have defined a vector operation, tensor sum, that plays a very important role in the design and analysis of DIPs and in particular in digit-reversal permutations. We derived a new $O(\log \log n)$ parallel-time algorithm for the family of DIPs and developed a Sisal implementation for this algorithm. Different choices of the parameter p of the Sisal function T give different DIPs. For the special case of bit-reversal (TDR with $r = 2$) we showed that on a Cray C-90 our algorithm runs up to twice as fast than that of Elster's bit-reversal algorithm and differs only very slightly from the authors' RDC algorithm.

Work is currently underway to apply the methods presented here to a more general class of permutations which include mixed-radix digit-reversal permutations and prime-factor FFT permutations.

ACKNOWLEDGMENTS

Supported in part by the Computational Mathematics Group of Puerto Rico EP-SCoR II grant and by NSF grant HRD-9450448 (D.B., J.S.) and by the Lawrence Livermore National Laboratory under DOE contract W–7405–ENG–48 and the Department of Energy Technology Transfer Initiative (J.F.) The authors thank Cray Research Incorporated for the dedicated computer time they provided in support of this work.

REFERENCES

[1] D. Bollman, "Digit-index permutations," *Congressus Numerantium*, vol. 91, pp. 213–217, 1992.

[2] D. Bollman, F. Sanmiguel, and J. Seguel, "Implementing FFTs in Sisal," *Proceedings of the Second Sisal Users' Conference*, San Diego, 1992, pp. 56–62.

[3] D. Cann, "Retire Fortran? A debate rekindled," *Comm. ACM*, vol. 30, pp. 81–89, 1992.

[4] A. C. Elster, "Fast bit-reversal algorithms," in *Proceedings of the International Conference on Acoustic, Speech and Signal Processing*, IEEE, 1989, pp. 1099–1102.

[5] J. Feo and D. Cann, "Developing a high-performance FFT algorithm in Sisal for a vector supercomputer," in *Proceedings of the Third Sisal User's Conference*, San Diego, 1993, p. 164–171.

[6] J. Johnson, R. Johnson, D. Rodriguez, and R. Tolmieri, "A methodology for designing, modifying and implementing Fourier transform algorithms on various architectures," *Circuits, Systems Signal Processing*, vol. 9, pp. 449–500, 1990.

[7] J. Seguel and D. Bollman, "Fast digit-reversal algorithms on a shared-memory machine," *Parallel Comput.*, vol. 20, pp. 93–99, 1994.

[8] S. Skedielewski, "Sisal," in *Parallel Functional Languages and Compilers*, B. Szymanski, Ed. New York: ACM Press, 1991, pp. 105–157.

[9] R. Tolimeri, M. An, and C. Lu, *Algorithms for Discrete Fourier Transforms and Convolution.* New York: Springer-Verlag, 1989.

[10] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform.* Philadelphia: SIAM, 1992.