# Matrix Algebra and Applicative Programming

by

David S. Wise

Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT No. 222

Matrix Algebra and Applicative Programming

by

David S. Wise

June, 1987

To appear in *Proceedings of the 3rd Functional Programming Languages and Computer Architecture Conference*, Portland, Oregon.

Research reported herein was supported in part by the National Science Foundation, under Grant Number DCR 84-05241.

# MATRIX ALGEBRA AND APPLICATIVE PROGRAMMING

**David S. Wise***

**Computer Science Department, Indiana University**

**101 Lindley Hall, Bloomington, IN 47405-4101**

dswise@iuvax.cs.indiana.edu

**CR categories and Subject Descriptors:**
C.1.2 [**Multiple Data Stream Architectures (Multiprocessors)**]: Array and vector processors, Parallel processors; D.1.1 [**Applicative (Functional) Programming Techniques**]; G.1.3 [**Numerical Linear Algebra**]: Sparse and very large systems; E.1 [**Data Structures**]: Trees; F.2.1 [**Numerical Algorithms and Problems**]: Computation of fast Fourier transform.
**General Term:** Algorithms.

**Abstract**

The broad problem of matrix algebra is taken up from the perspective of functional programming. A key question is how arrays should be represented in order to admit good implementations of well-known efficient algorithms, and whether functional architecture sheds any new light on these or other solutions. It relates directly to disarming the "aggregate update" problem.

The major thesis is that $2^d$-ary trees should be used to represent $d$-dimensional arrays; examples are matrix operations ($d = 2$), and a particularly interesting vector ($d = 1$) algorithm. Sparse and dense matrices are represented homogeneously, but at some overhead that appears tolerable; encouraging results are reviewed and extended. A Pivot Step algorithm is described which offers optimal stability at no extra cost for searching. The new results include proposed sparseness measures for matrices, improved performance of stable matrix inversion through repeated pivoting while deep within a matrix-tree (extendible to solving linear systems), and a clean matrix derivation of the vector algorithm for the fast Fourier transform. Running code is offered in the appendices.

This work is particularly important because of the importance of this family of problems. Progress would be of simultaneous use in decomposing algorithms over traditional vector multiprocessors, as well as motivate practical interest in highly parallel functional architectures.

---

* Research reported herein was sponsored, in part, by the National Science Foundation under Grant Number DCR 84-05241.

## Section 1. Introduction

Various functional programming languages and architectures have been explored through recent years, and we are close to implementations that will join them to deliver the parallelism long foreseen by many researchers.

Before we reach this goal, however, another component of functional programming productivity must be developed: a style for algorithms—or, rather, versions of known algorithms—suitable to the anticipated parallel environment. In recognition of this need and as a test of the "applicative thesis" (that functional languages and architectures are necessary to accomplish highly parallel performance), a familiar class of problems, matrix algebra, is under study [18, 20, 21], with the goal of developing pure, functional algorithms to mimic the parallel performance of this class of well-studied programs [22]. The salient feature of this work is the use of recursive block decomposition to represent *all* arrays, forcing a rigorously recursive (functional) decomposition of the usual algorithms along the boundaries of naturally arising subtrees.

This paper briefly reviews some results from that effort that have already appeared elsewhere, and newly presents related ones. This class of problems was originally selected for study because it was thought to be "understood," in the sense that much study had already been invested into hardware and software for it.

While the applicative thesis, as well as most computer science, is aimed at general problems, matrix algebra has always been a problem important enough to demand special-purpose languages and hardware. It is, therefore, doubly worthy of study in this context, because progress over existing matrix techniques, alone, would be motivation for support of applicative systems. Results here will better justify the investment needed to construct "special-purpose functional machines" under the disguise of "special-purpose vector/matrix machines."

The remainder of this paper is in six parts. Section 2 defines the proposed representation of arrays (of any dimension) with particular emphasis on matrices (two dimensional arrays). The next reviews costs for representing sparse/dense matrices under this convention; a notable feature of this representation is that only one homogeneous representation (and program) handles both sparse and dense matrices. The fourth section reviews algorithms for elementary arithmetic, including optimally-stable matrix inversion through pivoting (Gaussian elimination.). That algorithm is new, an unforeseen dividend from this effort. The next section newly describes how pivotings may be localized in one subtree (processor), so that more progress toward the result is made for each process creation/rendezvous. The sixth section reviews Pease's derivation of the fast Fourier transform, which uses quadtree matrices to develop a vector (as binary tree) algorithm, factoring the usual bit-reversal permutation. The last section offers conclusions and

open problems raised by these results, which indicate need for continuing work on Functional Programming and Architectures.

## Section 2. Quadtree Representation

*Dimension* refers to the number of subscripts on an array. *Order* of a square matrix means the number of its rows or columns when written as the conventional tableau. Similarly, the *size* of a vector is the number of elements when it is expressed in the conventional tuple formulation.

Let any $d$-dimensional array be represented as a $2^d$-ary tree. Here only matrices and vectors are considered, where $d = 2$ suggests quadtrees, and $d = 1$ suggests binary trees.

Matrix algorithms will be arranged so that we may (without loss) perceive any nonzero scalar, $x$, as a diagonal matrix of arbitrary order, entirely of zeroes except for $x$'s on the main diagonal; that is, $x = [x\delta_{i,j}]$. Thus, a domain is postulated that coalesces scalars and matrices, with every scalar-like object conforming also as a matrix of any order. Of particular interest is the scalar 1, which is at once the *unique* multiplicative identity for scalar/matrix arithmetic. The additive identity, 0, is represented by the null pointer, **NIL** (using PASCAL notation), which is particularly helpful in reducing resources used for sparse matrices.

A matrix (of otherwise-known order) is either a 'scalar' or it is a quadruple of four equally-ordered submatrices. So that this recursive cleaving works smoothly, we embed a matrix of size $n \times n$ in a $2^{\lceil \lg\, n \rceil} \times 2^{\lceil \lg\, n \rceil}$ matrix, justified at the lower, right (southeast) corner with zero padding to the north and west. Padding with **NIL** minimizes the space consumed in padding. The matrix is justified to the southeast, rather than the northwest, so to help with computation of eliminants [1].

This prescribes a *normal form* for quadtrees: no scalar entry is ever 0, four quadrants cannot all be **NIL**, and if the southwest and northeast are **NIL** then the northwest and southeast cannot be the same scalar. Similarly, **NIL** as a vector refers to the zero vector, and any non-zero scalar $x$ is interpreted as a vector of arbitrary size, each of whose elements is $x$; this normal form for vectors precludes any entry from being 0 and any brothers from both being **NIL** or the same scalar.

Inferring the conventional meaning from such a matrix now requires additional information (*viz.* its order), but we can proceed quite far without size information; it only becomes critical upon Input or Output. One must acknowledge that the I/O conversions are non-trivial algorithms [20], but because they consume little processor resource—and are restrained, also, by communication bandwidth—we eschew them here. Like floating-point number conversions, they are an irritating impediment to one who would experiment with the algorithms discussed below.

A "header" above each matrix quadtree should contain its size, necessary for output translation and needed for better control of certain algorithms, like pivoting on singular matrices. Here, particularly, size is the proper length of the main diagonal—exclusive of any padding and normalization. This value also might be used for run-time conformability checking.

Another useful annotation is to include a bit within each pointer, indicating that the referenced tree structure is to be interpreted as transposed, recursively interchanging southwest\northeast quadrants upon any access. Call this the *transposed* flag. With it, not only does quadtree representation allow us to transpose an entire matrix in constant time—building a new root with that flag inverted—but also it allows row and column traversal at equally high efficiency, at the cost of symmetric-order traversal of the appropriately projected binary tree.

This enforced block-decomposition representation of matrices recalls several results from the literature. Binary decomposition of vectors is implicit in the fast Fourier transform, and shows up explicitly in Pease's development [15] of a quadtree decomposition of the discrete Fourier transform matrix.

The second originates with McKellar and Coffman [11], who study the storage of submatrices in a demand-paging environment in order to reduce page faults. They arrive at the square-block decomposition, extended to block-specific algorithms by Fischer and Probert [4].

The third is George's *nested dissection* method [3] for matrix problems. A description is given by George and Liu [6].

The last, a recent problem impacted by this representation, is that of updating an aggregate structure in a functional language. Hudak [7] proposes specific primitives for sequentially stored aggregates, which are avoided by O'Donnell in his tree-like machine [13]. Block decomposition moots sequential storage, by providing access and reconstruction (recopying using shared references) in time and space logarithmic, rather than linear, in the order of the structure. The cost in space becomes irrelevant, however, when storage management on acyclic structures is free [19]. Moreover, as we shall see, many of the necessary algorithms distribute naturally across the tree so that any reconstruction is local to a substructure, rather than global over the aggregate.

## Section 3. Measures of Sparsity and Density

Duff states in his authoritative survey, "In quantitative terms, the density of a matrix is defined as the percentage of the number of nonzeros to the total number of entries in the matrix. The term sparsity for the complement of this quantity is rarely used. [3, p. 500]" Rather, he suggests that sparsity of a matrix has as much to do with the distribution of zero elements as with their relative population.

His perspective is reflected in analysis of the total space and access-time for familiar kinds of "sparse" matrices, represented as quadtrees [21]. Significant overhead is caused by the nonterminal nodes of a quadtree, a cost that does not appear in conventional, sequential storage of matrices as vectors at contiguous memory addresses. Closed-form results have been computed for total-space and for expected-depth for dense, symmetric, triangular, banded, and permutation matrices. These results are summarized below in Table 1.

Based on Duff's *caveat* and these numbers, I here propose measures of both density and sparsity that are motivated by results on quadtrees, but are expressed independently of any particular representation. Examples appear in Table 1, also.

*Density* of a particular matrix is the ratio between the space it occupies, and the space occupied by a dense matrix of the same order. *Non-sparsity* of a particular matrix is the ratio between the expected time to access a random element (path length if considering quadtrees [20]), and the expected access time within a dense matrix of the same order. *Sparsity* is the difference between one and this non-sparsity measure.

Both density and sparsity are measured on a scale from zero to one. For the conventional row-major, sequential representation of matrices, the density measure corresponds precisely with Duff's. The sparsity measure is uniformly near zero there, consistent with the observation that this representation offers no special advantage for sparse matrix manipulation.

Let us consider $n \times n$ matrices. Table 1 presents closed-form and asymptotic results for space, density, expected path length (root to terminal node), and sparsity for some familiar patterns of matrices. In all cases, a matrix is presumed to be completely dense, except where the indicated pattern requires zeros (or shared storage in the case of symmetry). The only strange ones are the shuffle permutation and the "FFT permutation" matrix, both of which occur when considering fast Fourier transform (FFT) algorithms; the latter is the so-called "bit-reversal" permutation that arises as the first or last step in the usual FFT. (Both are discussed in Section 6. If, as was done with symmetric matrices, sharing is allowed, the space needed to represent an $n \times n$ shuffle permutation shrinks to $4 \lg n - 2$.)

The remarkable entry in the table is for the FFT permutation, which measures out to be neither dense nor sparse, in spite of the fact that it only contains $n$ non-zeroes of $n^2$ entries. This is remarkably consistent with Duff's observation that patterning is essential to sparseness; the bit-reversal permutation is characterized by its lack of local patterning! Permutation matrices, in general, measure out this badly [21] as quadtrees, so it is fortunate that we already prefer an alternative representation that is not dense: as vectors of integers.

| | Space | Density* | Expected Path | Sparsity* |
|---|---|---|---|---|
| Dense | $\frac{4}{3}(n^2 - \frac{1}{4})$ | 1 | $\lg n + 1$ | 0 |
| Symmetric | $\frac{2}{3}(n+2)(n-\frac{1}{2})$ | $\frac{1}{2}$ | $\lg n + 1$ | 0 |
| Triangular | $\frac{2}{3}(n+2)(n-\frac{1}{2})$ | $\frac{1}{2}$ | $\frac{\lg n}{2} + \frac{3}{2} - \frac{1}{2n}$ | $\frac{1}{2} - \frac{1}{\lg n}$ |
| FFT permutation | $\frac{n \lg n}{2} + \frac{4n}{3} - \frac{1}{3}$ | $\frac{3}{8}\frac{\lg n}{n}$ | $\frac{\lg n}{2} + \frac{4}{3} - \frac{1}{3n}$ | $\frac{1}{2} - \frac{5}{6 \lg n}$ |
| Tridiagonal | $6n - 2\lg n - 5$ | 0 | $\frac{10}{3} - \frac{3}{n} + \frac{2}{3n^2}$ | $1 - \frac{10}{3 \lg n}$ |
| Pentadiagonal | $8n - 2\lg n - 9$ | 0 | $\frac{10}{3} - \frac{1}{n} - \frac{10}{3n^2}$ | $1 - \frac{10}{3 \lg n}$ |
| Heptadiagonal | $11n - 2\lg n - 19$ | 0 | $\frac{10}{3} - \frac{5}{n} - \frac{76}{3n^2}$ | $1 - \frac{10}{3 \lg n}$ |
| Enneadiagonal | $13n - 2\lg n - 26$ | 0 | $\frac{10}{3} - \frac{7}{n} - \frac{100}{3n^2}$ | $1 - \frac{10}{3 \lg n}$ |
| Shuffle permutation | $3(n-1)$ | 0 | $3(1 - \frac{1}{n})$ | $1 - \frac{3}{\lg n}$ |
| Identity | 1 | 0 | 1 | $1 - \frac{1}{\lg n}$ |

Table 1. Measures of patterned and unpatterned matrices as quadtrees.

**Density* is accurate within a term of $\Theta(n^{-1})$. *Sparsity* is accurate within a term of $\Theta((\lg n)^{-2})$.

This measure of sparsity has not been well studied; it is being defined here for the first time. Moreover, the utility of these measures must yet be demonstrated in analytical or in experimental analyses of the behavior of algorithms on arguments of various measures. For instance, it is left as an analytic exercise to show that matrix addition of two $n \times n$ matrices, respectively of density $d_1$ and $d_2$ and of sparsity $s_1$ and $s_2$, yields an answer with space, processor-time, density, and sparsity within the bounds indicated below:

$$n^2|d_1 - d_2| \leq \textbf{space} \leq n^2 \cdot \min(1, d_1 + d_2);$$

$$\max(1, n^2 \lg n(1 - (s_1 + s_2))) \leq \textbf{uniprocessor time} \leq n^2 \lg n(1 - \max(s_1, s_2));$$

$$|d_1 - d_2| \leq \textbf{density} \leq \min(1, d_1 + d_2);$$

$$\max(0, s_1 + s_2 - 1) \leq \textbf{sparsity} \leq 1 - |s_1 - s_2|.$$

In general, however, analytical results are difficult and so the utility of these measures ultimately must be established or denied by experimentation on real data.

Although nontrivial in comparison with constant-time access into conventional matrices, path length is an artificial measure in a couple of ways. First of all, it may be irrelevant in the algorithms described below, which typically involve recursive descent. That is, rather than accessing elements of a matrix from the root of the tree (analogously to indexing through a conventional array based from a single memory address), these algorithms recurse to nested and successively shallower subtrees, so that an entire path from the root is rarely traversed just to manipulate a *single* element.

Secondly, even if the complete path were traversed upon every probe of an array, the time spent to traverse that path might be recovered in other ways. More specifically, if a heap were spread across very many memory banks and several processors were performing similar algorithms on one globally accessible quadtree-matrix (whose nodes mapped across those banks in a random order), then the aggregate performance of those processors might actually improve over correspondingly similar algorithms on matrices stored sequentially. The improvement arises from the random pattern mitigating the problem of two, or more, processors falling into the same access pattern and addressing the same banks simultaneously and repeatedly. The coincidence of regular access patterns to regularly allocated arrays, even from regular offsets within different matrices, is likely to become an ever increasing problem with more processors. Randomization available from this kind of heap [8] would not prevent the first contention between two algorithms, but it would certainly help prevent a first from being immediately followed by more. Thus, the tree structure would allow many coprocessors to run with less memory contention, and to absorb the cost of repeated path traversals.

Both efficiency of access and sparse matrices are of high interest in parallel processing. Many caching strategies fail under parallelism, and bus or switch contention compounds the problem of wait states. Many problems sufficiently large and important enough to justify parallel computation also exhibit sparseness. Therefore, these results are of great interest in designing parallel algorithms and parallel computers [2, 16].

## Section 4. Arithmetic Algorithms

The recursive definition of quaternary trees molds the recursive structure of programs that manipulate them. Moreover, the bifurcation of tree composition leads naturally to more stable algorithms. For instance, each addend in the sum of a vector of size $2^p$ (as a binary tree of depth $p$) naturally participates in no more than $p$ binary additions; if the vector were instead stored in consecutive memory locations, the "natural" algorithm has each addend participating in up to $2^p - 1$ additions. This is important in floating-point algorithms which, though stable over multiplicative operations, become unstable with addition and subtraction.

The algorithm for matrix addition [18] and subtraction decomposes naturally into four quadrant additions, separate and independent processes. Because of their mutual independence, these four are naturally computed in parallel within a shared memory, or distributed to independent processors with private memory. In the latter case, the tree structure of a matrix guides its mapping onto a tree within private memories. And the division extends naturally to 16, 64, 256, *etc.* processors, or—by splitting the sums in half, rather than in quarters—to 2, 8, 128 *etc.* as well.

Whenever either addend is **NIL**, then their sum is efficiently represented as a shared reference to the root of the other addend, without need for any further traversal.

Matrix multiplication decomposes two ways (again treating the product as two halves), four ways (the four quadrants of the answer), and eight ways (the eight quadrant products in Strassen's decomposition [17] of Gaussian matrix multiplication.) Whenever a factor is either **NIL** or 1, the product is directly available, either as **NIL**, or as a shared reference to the other factor. The former case occurs particularly often with sparse factors, and annihilates the recursion not only of quadrant multiplication, but also of the addition of quadrant-products that follows.

Solutions to linear systems and matrix inversion have been reduced to a Pivot Step algorithm [10], where the "independent" problem of a stable choice for the pivot element folds naturally onto the tree [20]. Each nonterminal node in the quadtree is decorated with a nonnegative number and two bits: the absolute value of the largest uneliminated value in the tree, and an indicator toward the quadrant in which it resides. Initially all elements are uneliminated, and the tree is decorated bottom-up with four-way, local maxima.

These tree decorations are sufficient to identify the path from the root to the largest uneliminated element, which is the next pivot. During pivoting, every entry in the pivot row and in the pivot column will be visited, and the tree is then reconstructed with that row and column eliminated from the maxima—by treating their contents as zero for the purposes of recomputing all local decorations. By redecorating during each pivoting, the conventional search-for-pivot is distributed back into the previous pivoting without need for any additional traversal (and the interprocessor communication required for parallel searching.)

Knuth describes the Pivot Step as the transformation of the matrix

$$\begin{pmatrix} & \vdots & & \vdots & \\ \cdots & a & \cdots & b & \cdots \\ & \vdots & & \vdots & \\ \cdots & c & \cdots & d & \cdots \\ & \vdots & & \vdots & \end{pmatrix}, \tag{1}$$

where $a$ is the selected pivot element, into the matrix

$$\begin{pmatrix} & \vdots & & \vdots & \\ \cdots & \frac{1}{a} & \cdots & \frac{b}{a} & \cdots \\ & \vdots & & \vdots & \\ \cdots & -\frac{c}{a} & \cdots & d-\frac{cb}{a} & \cdots \\ & \vdots & & \vdots & \end{pmatrix}. \tag{2}$$

This sketch is necessarily different from Knuth's; where he uses the numerator $bc$, I use $cb$ to provide for multiplication when $c$ and $b$ are matrices. This detail arises whenever multiplication does not commute [1].

In this description, the off-pivot entry $d$ is typical of most of the matrix. Particularly in sparse matrices where either $b$ or $c$ is likely to be zero, whole quadrants of values, $d$, will not change over a Pivot Step; the decorations in those quadrants *do not change either.* In this way, full (total, complete [3]) pivoting may be achieved without the cost of an $O(n^2)$ search for selecting every pivot.

The Pivot Step algorithm is described elsewhere [20]; the code treats each quadrant of each nontrivial, decorated matrix in one of four ways, two of which are presented in Appendix B. That Daisy [5, 9] program specifies data dependencies, which implies some order of evaluation, not entirely apparent at coding time. Because Daisy is entirely lazy, it is difficult to foresee the order of creation for recursively dependent data structures. On the other hand, uncertainty of the algorithm's author on such unimportant details later becomes an advantage for automatic scheduling of processes, or during reimplementation in a strict (manually scheduled) language. This is an example of how programming in a lazy language relaxes mandatory scheduling, leaving more freedom for subsequent site-specific implementation/scheduling.

Just as $a$ is preselected, the four quadrants sort themselves into one of four types: *PIV* wherein $a$ lies, *ROW* alongside *PIV*, *COL* above/beneath *PIV*, and *OFF* diagonal from *PIV*. *PIV* either is the scalar $a$, or decomposes again into four quadrants of these types. If *ROW* decomposes, it is made up of two *ROW*s and two *OFF*s; *COL* decomposes to two *COL*s and two *OFF*s. *OFF* decomposes into four *OFF*s.

The code provides four functions, for extending the pivot transformation on a quadrant of each of these types. When a decomposition is necessary, the four recurrences are specified to proceed in parallel, and in the case of *OFF* decomposition—the most frequent—it actually can run in parallel. Appendix B gives the complete, running Daisy code for `PIVOT`, the most intricate of the four, and `OFF` which is the simplest.

However, pivoting an *OFF* quadrant requires the contents of the pivot row which is a result from pivoting the adjacent *ROW* quadrant, as well as the pivot column from pivoting the neighboring *COL*. Similarly, *ROW* (and *COL*) depends on the portion of the pivot column (respectively, pivot row) that is extracted from pivoting its sibling *PIV*. The dependence suggests that (recursively) *PIV* must be processed first, followed by *ROW* and *COL* simultaneously, with components of *OFF* computable as soon as coordinating pieces of *ROW* and *COL* are available. The operational effect is that pivoting descends the tree as a uniprocessor algorithm until $a$ is encountered, whereupon the recurrence backs out, spreading the transformation to sibling quadrants. (The exact behavior, however, depends on the scheduler. For lazy languages—like Daisy—the precise order of evaluation is most obscure.)

It is this same dependence of the transformation of $OFF$ on the pivot row and column vectors that reveals an advantage of the normal form for binary trees. When either is entirely zeros it is represented as NIL, easily tested so that the function OFF quickly returns its argument matrix unchanged.

### Section 5. Repeating Pivots Locally

A severe cost in multiprocessing is process allocation and deallocation. The previous section shows how quadrants may be transformed in parallel under Pivot Step, but there is little explicit effort to minimize the number of process dispatches. A very good way to minimize process dispatch is to perform repeated pivots within a single $PIV$ quadrant, and then to dispatch the sibling transformations in a single clump.

There are two ways that repeated pivots may be dispatched within $PIV$ without sacrificing any stability. One is implicit in the normalized representation of matrices, and one requires a modification on the Pivot Step as described [20].

Consider the case where $PIV = a$, a nonzero scalar, but some of $ROW$, $COL$, $OFF$ are not trivial. That is, $a$ turns up to be a scalar at a level above its siblings. In this case, $PIV$ is a normalized form of

$$\begin{pmatrix} a & 0 \\ 0 & a \end{pmatrix}$$

or perhaps even a $4 \times 4$ or larger matrix. Here, the conformable multiplier of $ROW$ is

$$\begin{pmatrix} 1/a & 0 \\ 0 & 1/a \end{pmatrix},$$

rather than the scalar, $\frac{1}{a}$, the multiplier for $COL$ is

$$\begin{pmatrix} -1/a & 0 \\ 0 & -1/a \end{pmatrix},$$

and the postimage of $OFF$ will be

$$OFF - COL \cdot \begin{pmatrix} 1/a & 0 \\ 0 & 1/a \end{pmatrix} \cdot ROW.$$

Interpreting the implicit scalar multiplication as explicit matrix multiplication does yield the correct transformations, effecting two pivots in this case; if one expands the four sums implicit in this matrix multiplication he will have the effect of both Pivot Steps (subtractions) implied by the $2 \times 2$ conformable form of $PIV$. In fact, if we merely implement matrix product/difference wherever scalar product/difference is specified, recognizing that a scalar may be conformable with any non-trivial matrix, then we get the effect of pivoting on larger blocks. The normal form works!

Yet another improvement is possible, but it is not accomplished so easily. Suppose that $PIV$ with pivot value $a$ has been passed through Pivot Step to become $PIV'$, whose largest uneliminated element is now $a'$. Under what circumstances can we, without loss of stability, perform an immediate local pivot on $a'$ to get $PIV''$ without first propagating the first onto $ROW$, $COL$, and $OFF$? In that case we later propagate two (or even more) pivots through sibling quadrants for the cost of only one rendezvous, interprocessor communication, or process dispatch.

We can, indeed, anticipate and make this determination, based only on one extra numeric parameter to Pivot Step; at the outermost level this argument will have value zero. Let the parameter be called *threshold*. At each level, let $\hat{b}$ be the integer decorating the root of $ROW$, let $\hat{c}$ be the decoration on $COL$, and let $\hat{d}$ decorate $OFF$. As we invoke Pivot Step on $PIV$, pass in the new argument,

$$\max(2\hat{b}, 2\hat{c}, \hat{d} + \tfrac{\hat{c}\hat{b}}{a}, \mathit{threshold}).$$

In this way, when the scalar $a$ is encountered deep in the tree, the current value of *threshold* is the maximum of $2\hat{b}, 2\hat{c}, \hat{d} + \frac{\hat{c}\hat{b}}{a}$, for all coordinating $\hat{b}, \hat{c}, \hat{d}$ labeled according to (1).

**Theorem:** If $a' \geq \mathit{threshold}$ then $a'$ is the next pivot element.

**Proof:** Recall that $a \geq \hat{b}, \hat{c}, \hat{d}$ by definition of the decoration. For all $b, d \in ROW$ and all $c \in PIV$.

$$a' \geq 2\hat{b} \geq \hat{b} + \tfrac{a\hat{b}}{a} \geq |d| + |\tfrac{cb}{a}| \geq |d - \tfrac{cb}{a}|$$

Thus, $a'$ exceeds any candidate for the next pivot element that may arise from $ROW$. Similarly, for all $c, d \in COL$ and $b \in PIV$

$$a' \geq 2\hat{c} \geq \hat{c} + \tfrac{\hat{c}a}{a} \geq |d| + |\tfrac{cb}{a}| \geq |d - \tfrac{cb}{a}|;$$

for all $b \in ROW, c \in COL$ and $d \in OFF$

$$a' \geq \hat{d} + \tfrac{\hat{c}\hat{b}}{a} \geq |d| + |\tfrac{cb}{a}| \geq |d - \tfrac{cb}{a}|.$$

Thus, $a'$ also exceeds any other candidate for pivot, and so it must be the next one. ∎

Therefore, it is correct to repeat Pivot Step on $PIV'$, returning multiple tuples of results (pivot rows, pivot columns, *etc.*) to be processed through the sibling quadrants ($ROW$, $COL$, and $OFF$) in sequence. In fact, the test can be made repeatedly for $2 \cdot \mathit{threshold}$, $4 \cdot \mathit{threshold}$, etc. With favorable results from purely local testing of decorations, it is possible to repeat local pivots with confidence of maintaining stability.

The suggestions in this section are similar to those of Peters [14], who is concerned with identifying multiple pivotings that do not generate mutual processing conflicts. His method is not necessarily stable. This section, rather, locally reveals multiple, stable pivotings within a quadrant, which are then propagated together to synchronize their interdependence.

## Section 6. Fast Fourier Transform

This section deals explicitly with a vector algorithm, although it derives the algorithm through matrix manipulation—again using quadrant decomposition. Pease [15] *derives* the Fast Fourier Transform (FFT) in this way from the ordinary Discrete Fourier Transform. His derivation, as well as the ordinary "butterfly" explication [12] of this important algorithm, however, is characterized by separation of necessary permutations from the FFT, itself. After the derivation is presented, the significance of retaining the permutations in the derivation will be discussed.

Let $n = 2^p$ for integer $p$, and let $\omega$ be a principal $n^{\text{th}}$ root of unity. The vector $\vec{y}$ is defined in terms of the vector $\vec{x}$ according to the component-wise formula

$$y_k = \sum_{m=0}^{n-1} x_m \omega^{mk}.$$

We can express this as a matrix multiplication:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 & \cdots & \omega^{n-1} \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2n-2} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{n-1} & \omega^{2n-2} & \omega^{3n-3} & \cdots & \omega^{n^2-2n+1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{pmatrix}$$

Pease names this matrix **T**; we subscript it $T_{p,\omega}$ and rewrite it in quadrants using the fact that $\omega^n = 1$.

$$T_{p,\omega} = \left(\begin{array}{ccccc|ccc} \omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \cdots & \omega^{\frac{n}{2}-1} & \omega^{\frac{n}{2}} & \cdots & \omega^{n-1} \\ \omega^0 & \omega^2 & \omega^4 & \cdots & \omega^{n-2} & \omega^0 & \cdots & \omega^{n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{\frac{n}{2}-1} & \omega^{n-2} & \cdots & \omega^1 & \omega^{\frac{n}{2}} & \cdots & \omega^{\frac{n}{2}+1} \\ \hline \omega^0 & \omega^{\frac{n}{2}} & \omega^0 & \cdots & \omega^{\frac{n}{2}} & \omega^0 & \cdots & \omega^{\frac{n}{2}} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{n-1} & \omega^{n-2} & \cdots & \omega^{\frac{n}{2}+1} & \omega^{\frac{n}{2}} & \cdots & \omega^1 \end{array}\right) \tag{3}$$

Now let us introduce two permutation matrices, $D_p$ and $S_p$, which are here called *deal* and *shuffle*, respectively. Multiplying a vector of size $n$ by $D_p$ on the left will have the effect of reordering its elements as if the vector were a deck of cards, which was dealt into two full hands of $n/2$ cards which were then stacked. A typical layout of $D_p$ appears below. It is characterized by ones appearing in "knight's moves," first descending from the far northwest entry to central-east, and also ascending from the far southeast entry to central-west.

$$D_p = \left(\begin{array}{ccccccc|ccc} 1 & 0 & 0 & 0 & 0 & \cdots & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & \cdots & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \cdots & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & \cdots & 1 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 & \cdots & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & \cdots & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & \cdots & 0 & 1 \end{array}\right)$$

$S_p$ is the inverse of $D_p$; Pease names $S_p$ as $\mathbf{P}$, again without specifying its order. Multiplying a vector on the left by $S_p$ has the effect of performing a perfect riffle-shuffle on the elements of the vector. Multiplying on the right by these permutations reorders the columns similarly.

Consider what happens when we multiply $T_{p,\omega}$ on the left by $D_p$:

$$U_{p,\omega} = D_p T_{p,\omega} = \left(\begin{array}{ccccc|ccc} \omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^0 & \omega^2 & \omega^4 & \cdots & \omega^{n-2} & \omega^0 & \cdots & \omega^{n-2} \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{\frac{n}{2}} & \omega^0 & \ddots & \omega^{\frac{n}{2}} & \omega^0 & \cdots & \omega^{\frac{n}{2}} \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{n-2} & \omega^{n-4} & \cdots & \omega^2 & \omega^0 & \cdots & \omega^2 \\ \hline \omega^0 & \omega^1 & \omega^2 & \cdots & \omega^{\frac{n}{2}-1} & \omega^{\frac{n}{2}} & \cdots & \omega^{n-1} \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{\frac{n}{2}-1} & \omega^{n-2} & \ddots & \omega^1 & \omega^{\frac{n}{2}} & \cdots & \omega^{\frac{n}{2}+1} \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{n-1} & \omega^{n-2} & \cdots & \omega^{\frac{n}{2}+1} & \omega^{\frac{n}{2}} & \cdots & \omega^1 \end{array}\right). \tag{4}$$

$U_{p,\omega}$ has familiar patterns, notably $T_{(p-1),\omega^2}$ as both upper quadrants! Pease develops this observation in terms of $\mathbf{T}'$, to be introduced at (11) below. He then presents the essence of the following recurrence for $p > 0$:

$$\begin{aligned} T_{0,\omega} &= 1; \\ T_{p,\omega} &= 1T_{p,\omega} = S_p(D_p T_{p,\omega}) = S_p U_{p,\omega} \\ &= S_p \left(\begin{array}{c|c} T_{p-1,\omega^2} & T_{p-1,\omega^2} \\ \hline (T_{p-1,\omega^2})K_{p-1,\omega} & -(T_{p-1,\omega^2})K_{p-1,\omega} \end{array}\right) \end{aligned} \tag{5}$$

where

$$K_{p-1,\omega} = \left(\begin{array}{ccccc} \omega^0 & 0 & 0 & \cdots & 0 \\ 0 & \omega^1 & 0 & \cdots & 0 \\ 0 & 0 & \omega^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \omega^{\frac{n}{2}-1} \end{array}\right) = -\left(\begin{array}{ccccc} \omega^{\frac{n}{2}} & 0 & 0 & \cdots & 0 \\ 0 & \omega^{\frac{n}{2}+1} & 0 & \cdots & 0 \\ 0 & 0 & \omega^{\frac{n}{2}+2} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \omega^{n-1} \end{array}\right)$$

is Pease's K, and 1 is Pease's I.

Factoring,

$$T_{p,\omega} = S_p \left( \begin{array}{c|c} T_{p-1,\omega^2} & 0 \\ \hline 0 & T_{p-1,\omega^2} \end{array} \right) \left( \begin{array}{c|c} 1 & 0 \\ \hline 0 & K_{p-1,\omega} \end{array} \right) \left( \begin{array}{c|c} 1 & 1 \\ \hline 1 & -1 \end{array} \right). \qquad (6)$$

This recurrence is the conventional FFT using "decimation in frequency."

With $\vec{x}$ represented as a binary tree and its two halves readily accessible, an efficient recursive algorithm can be easily characterized. Running Daisy code is offered in Appendix A. One interpretation associates the above products to the right, and begins with addition and subtraction of the two halves of $\vec{x}$ (probably in parallel). Then the difference is multiplied, elementwise (again in parallel) by the powers of $\omega$ from the appropriate $K$, while the sum is unaltered. Finally, the algorithm is applied recursively (in parallel?) to each of the two resulting vectors, and the entire result is shuffled. (Once again, lazy evaluation on a uniprocessor confounds the serial order of evaluation—as described—but that same confoundedness admits lots of parallelism.)

The inverse of the FFT can be read right to left from Equation (6) when $p > 0$:

$$T^{-1}_{p,\omega} = \left( \begin{array}{c|c} 1 & 1 \\ \hline 1 & -1 \end{array} \right) \frac{1}{2} \left( \begin{array}{c|c} 1 & 0 \\ \hline 0 & K_{p-1,\omega^{n-1}} \end{array} \right) \left( \begin{array}{c|c} T^{-1}_{p-1,\omega^2} & 0 \\ \hline 0 & T^{-1}_{p-1,\omega^2} \end{array} \right) D_p. \qquad (7)$$

The decimation in time recurrence of the FFT, as well as its inverse, follows from transposing both sides of Equations (6) and (7), because $T_{p,\omega}$ is symmetric; see Equation (3):

$$\begin{aligned} T_{p,\omega} &= T^{\mathrm{T}}_{p,\omega} = \\ &= \left( \begin{array}{c|c} 1 & 1 \\ \hline 1 & -1 \end{array} \right) \left( \begin{array}{c|c} 1 & 0 \\ \hline 0 & K_{p-1,\omega^1} \end{array} \right) \left( \begin{array}{c|c} T_{p-1,\omega^2} & 0 \\ \hline 0 & T_{p-1,\omega^2} \end{array} \right) D_p. \end{aligned} \qquad (8)$$

Equation (8) can also be obtained from expanding $(T_{p,\omega} S_p) D_p$ similarly to Equations (5) and (6).

The FFT permutation ($p$-bit-reversal), here called $F_p$, arises as a consequence of the recursions in Equations (6), (7), or (8). If the product of nested permutations implied by those recurrences is expanded, then the permutation $F_p$, described below, results. It is listed as the "FFT permutation" in Table 1, and is called "bit-reversal" permutation because it exchanges $x_i$ and $x_{b(i)}$ in permuting $\vec{x}$, where $b$ is a function on natural numbers less than $2^p$ that reverses the $p$-bit strings that represent them. Since $b$ is its own inverse, $F_p$ is a symmetric permutation matrix.

$$F_p = S_p \left( \begin{array}{c|c} S_{p-1} & 0 \\ \hline 0 & S_{p-1} \end{array} \right) \left( \begin{array}{c|c} \left( \begin{array}{c|c} S_{p-2} & 0 \\ \hline 0 & S_{p-2} \end{array} \right) & 0 \\ \hline 0 & \left( \begin{array}{c|c} S_{p-2} & 0 \\ \hline 0 & S_{p-2} \end{array} \right) \end{array} \right) \cdots 1; \tag{9}$$

$$= 1 \cdots \left( \begin{array}{c|c} \left( \begin{array}{c|c} D_{p-2} & 0 \\ \hline 0 & D_{p-2} \end{array} \right) & 0 \\ \hline 0 & \left( \begin{array}{c|c} D_{p-2} & 0 \\ \hline 0 & D_{p-2} \end{array} \right) \end{array} \right) \left( \begin{array}{c|c} D_{p-1} & 0 \\ \hline 0 & D_{p-1} \end{array} \right) D_p; \tag{10}$$

$$T_{p,\omega} = F_p T'_{p,\omega}. \tag{11}$$

Equation (11) reveals Pease's $\mathbf{T}'$, the factor of $T_{p,\omega}$ exclusive of all permutations. Nothing here improves on him [15], except perhaps the facility of coding when language primitives provide vector bifurcation. He addresses radices other than two and, indeed, much can also be said in favor ternary and quinary trees when confronted with an FFT of order 360. However, whereas the permutations are here included as $T_{p,\omega}$ is factored, he excludes them as he factors $\mathbf{T}'$.

It is easy to overlook the important role of the permutations when using the FFT in a cyclic convolution, because they cancel. If $\vec{x}$ and $\vec{y}$ are both of size $2^p$, then their cyclic convolution (decimation in frequency) is given by

$$T_{p,\omega}^{-1}(T_{p,\omega}\vec{x} \cdot T_{p,\omega}\vec{y}) = U_{p,\omega}^{-1} D_p((S_p U_{p,\omega}\vec{x}) \cdot (S_p U_{p,\omega}\vec{y})) \tag{12}$$

$$= U_{p,\omega}^{-1}(D_p S_p)(U_{p,\omega}\vec{x} \cdot U_{p,\omega}\vec{y}) \tag{13}$$

$$= U_{p,\omega}^{-1}(U_{p,\omega}\vec{x} \cdot U_{p,\omega}\vec{y}).$$

$$= T_{p,\omega}'^{-1}(T'_{p,\omega}\vec{x} \cdot T'_{p,\omega}\vec{y}). \tag{14}$$

The dot in Equations (12)–(14) indicates component-wise product, across which any permutation distributes in (13), so that (14) looks much like the left side of (12) in form, except without permutations. Other reasons that permutations get overlooked are that the correct "plugboard" circuit—*if available*—can effect any vector permutation in constant time, or that a uniprocessor can elide the permutation with (repeated) clever indexing into sequentially-stored vectors, each access still proceeding at full memory speed.

However, we would do well to take **some** permutations more seriously when the vector is distributed across shared memory of a multiprocessor. In the case of the FFT permutation, an attempt to do the clever indexing *in parallel* will likely saturate shared paths from processors to memory, as if the memory switch were suddenly forced to behave like the the appropriate

"plugboard." The result is that each processor must endure memory delays dependent on the problem size, contrary to past intuition and unlike experience on uniprocessors.

Table 1 indicates that $F_p$ measures surprisingly bad with respect to both density and sparsity. If those measures accurately reflect the resource required for actually permuting in parallel, then Table 1 also suggests a viable alternative to $F_p$. The two factorizations of $F_p$, (9) and (10), indicate that nested Shuffles (or Deals) suffice where we are accustomed to using the bit-reversal FFT permutation, $F_p$. Each factorization of $F_p$ has factors nested along diagonal subtrees. Table 1 shows that $S_p$ (and similarly $D_p$) has sparsity of nearly $1 - 3/p$ and, therefore, it may be cheaper to perform nested shuffles/deals on localized subproblems, building up to one simple, global $S_p$ permutation, rather than to use the complicated global $F_p$ permutation. Results on the resources needed for sparse matrix multiplication would establish which is better. In any event, it is certainly easy to code the fast Fourier transform using nested Shuffles; see Appendix A. (Direct coding of an applicative FFT permutation is left as a challenge to the reader.)

## Section 7. Conclusions

This paper presents a case for array representation as trees. Not only does the logarithmic access path relax most of the difficulties from the aggregate-update problem, but also the tree structure leads to a natural decomposition of matrix problems and, thereby, to natural parallelism in their implementation. Not surprisingly, the expressiveness of functional languages helps to reveal both.

The space overhead that arises from the nonterminal nodes of a tree (compared to sequential representation that requires space only for terminal nodes) is not too burdensome. While they do take space, that cost is reasonable for reasonably sparse matrices, and it provides a natural decomposition of data space across local memories where it is necessary. Moreover, it unifies algorithms for both sparse and dense matrix techniques. Therefore, it seems to be an ideal strategy for multiprocessor matrix algebra—even if each processor is a high-powered vector processor for sequentially allocated submatrices under a certain order. It remains to be seen whether such a hybrid scheme, already tractable on existing architectures, will be adopted.

It is particularly surprising to uncover new variants of old, well-studied algorithms, like the folding of full-matrix search into the Pivot Step algorithm. While the decomposition of the FFT is not new, exposing both the *shuffle* and the *deal* decompositions of the bit-reversal permutation, each precisely following the nesting of the FFT recurrence pattern, is a useful insight on their interactions. Are other such "improvements" available from arrays-as-trees?

Issues of architecture follow immediately upon the idea that an array is actually a tree. It suggests—as some have long accepted—that much data is better organized if linked within a heap

memory, than if allocated sequentially. Many of the advantages and of the problems (*e.g.* "hot spots" in memory space) of sequential addressing are mooted by the kind of algorithms described here. As this idea gains acceptance, the efficient implementation of a multiprocessor heap will become an ever more significant goal [19].

The fast Fourier transform, itself, is an artifact of hardware for many. Yet, Pease's derivation is done without schematics. Is this just another example of synthesizing complicated hardware [9] through applicative languages?

One challenge to this work is that I have *only* considered examples that are "naturally recursive," or whose block-decomposition proceeds gracefully. While the FFT was chosen for this reason, I assert that the choice of matrix problems is general, although as yet covering only elementary matrix algebra. However, the choice of solutions was, indeed, restricted to those suitable to tree decomposition. Lots of row-decomposition algorithms for solving linear systems were put aside; the ones presented here are survivors. A better response to this challenge is to ask whether there are *any* highly parallel algorithms admitting graceful scheduling (decomposition) that are not "naturally recursive."

As stated early on, this class of problems was selected to test the "applicative thesis" because (as was thought) its parallelism was already well understood. This brief exploration has been far more successful than was originally expected and, therefore, the restrictions imposed by tree-representation of arrays can hardly be judged to be artificial! That any of these algorithms is really "a natural," however, will only be established if these algorithms improve performance of large multiprocessors.

**Acknowledgement:**

John Franco contributed the results on sparseness of random permutation matrices. Conversations with Dorothy Bollman led to our factoring the FFT, and Morven Gentleman pointed out its significant precedent. I thank John Williams for encouraging me toward further exploration of functionals over binary trees, and the students in a graduate seminar at Indiana University for grappling with some algorithms that turned out to be poorly suited to quadtree decomposition.

## References


1. S. K. Abdali. & D. D. Saunders. Transitive closure and related semiring properties via eliminants. *Theoretical Computer Science* **40**, 2,3 (1985), 257-274.

2. P. J. Denning Parallel computing and its evolution. *Comm. ACM* **29**, 12 (December, 1986), 1163-1167.

3. I. S. Duff. A survey of sparse matrix research. *Proc. IEEE* **65**, 4 (April, 1977), 500-535.

4. P. C. Fischer & R. L. Probert. Storage reorganization techniques for matrix computation in a paging environment. *Comm. ACM* **22**, 7 (July, 1979), 405-415.

5. D. P. Friedman & D. S. Wise. Aspects of applicative programming for parallel processing. *IEEE Trans. Comput.* **C-27**, 4 (April, 1978), 289-296.

6. A. George & J. W-H Liu. *Computer Solution of Large Sparse Positive Definite Systems,* Englewood Cliffs, NJ, Prentice-Hall (1981), Chapter 8.

7. P. Hudak. Arrays, non-determinism, side-effects, and parallelism: a functional perspective (Extended Abstract). *Proc. LANL/MCC Graph Reduction Workshop* Sante Fe, September, 1986, *Lecture Notes in Computer Science,* New York, Springer (to appear).

8. S. D. Johnson. "Storage Allocation for List Multiprocessing", Indiana University Computer Science Dept. Technical Report No. 168, (March, 1985).

9. S. D. Johnson. *Synthesis of Digital Designs from Recursion Equations,* Cambridge, MA, M.I.T. Press (1984).

10. D. E. Knuth. *The Art of Computer Programming,* **I**, *Fundamental Algorithms*, 2nd Ed., Reading, MA, Addison-Wesley (1975), 299-318 + 401, 556.

11. A. C. McKellar & E. G. Coffman, Jr. Organizing matrices and matrix operations for paged memory systems. *Comm. ACM* **12**, 3 (March, 1969), 153-165.

12. H. J. Nussbaumber. *Fast Fourier Transforms and Convolution Algorithms*, Berlin, Springer (1982).

13. J. T. O'Donnell. An architecture that efficiently updates associative aggregates in applicative programming languages. In Jean-Pierre Jouannaud (ed.), *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science* **201**, Berlin, Springer (1985), 164-189.

14. F. J. Peters. Parallel pivoting algorithms for sparse symmetric matrices. *Parallel Computing* **1**, 1 (August, 1984), 99-110.

15. M. C. Pease. An adaptation of the fast Fourier transform for parallel processing. *J. ACM* **15**, 2 (April, 1968), 252-264.

16. R. Rettberg & R. Thomas. Contention is no obstacle to shared-memory multiprocessing. *Comm. ACM* **29**, 12 (December, 1986), 1202-1212.

17. V. Strassen. Gaussian elimination is not optimal. *Numer. Math.* **13**, 4 (August, 1969), 354-356.

18. D. S. Wise. Representing matrices as quadtrees for parallel processors. *Information Processing Letters* **20** (May, 1985), 195-199.

19. D. S. Wise. Design for a Multiprocessing Heap with On-Board Reference Counting. In Jean-Pierre Jouannaud (ed.), *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science* **201**, Berlin, Springer (1985), 289-304.

20. D. S. Wise. Parallel decomposition of matrix inversion using quadtrees. *Proc. 1986 International Conference on Parallel Processing* (IEEE Cat. No. 86CH2355-6), 92-99.

21. D. S. Wise & J. Franco. Costs of quadtree representation of sparsely patterned matrices (in preparation.)

22. M. F. Young. A functional language and modular arithmetic for scientific computing. In Jean-Pierre Jouannaud (ed.), *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science* **201**, Berlin, Springer (1985), 305-318.

```
|       Appendix A        Daisy code for FFT--decimation in frequency

| Vertical-bar is the comment character.  Backslash is 'lambda'.
| Infix colon is application.  Infix exclamation point is 'cons'.

fft = \[x lgsize]. rec:[
        ffdo
        \[x wtree lgsize]. if:<
                zero?:lgsize x
                    shuffle:
                    <ffdo                *                              >:<
                    <vec+vec:x           vecXvec:<wtree vec-vec:x>>
                    <0:deal:wtree        *                              >
                    <dcr:lgsize          *                              > >
                >
        ffdo:<x buildAtree:dcr:lgsize lgsize>   ]

shuffle = \pair. if:<
                Scalar?:0:pair pair
                <shuffle*>:pair >
deal = \pair. if:<
                Scalar?:0:pair pair
                <\x.x *> :<     \[x].deal:x     *>:     <
                                pair                    > >

buildwtree  =  \lgsize. rec:[
        bin-ww
        \lgsize. if:<
                zero?:lgsize <1 omega>
                let:[   [bin ww]        bin-ww:dcr:lgsize
                        <       <bin scaXvec:<ww bin> >
                                square:ww
                        >] >
        0:bin-ww:lgsize ]



|       Appendix B       Daisy code for OFF and PIVOT

| A "decorated matrix" has the template,        [[max ibit jbit] ! undecrtdMtx]
| An "undecorated matrix" has the template,                       [NW NE SW SE]

IDENT = \a.a                    SWAP= \[a b].<b a>

OFF = \[decorMtx elimrowcol pcolrow index]. if:<
        any?: <\[p].VOID?:p *>:<pcolrow>  decorMtx       | Pivot row or col is 0.
        one?:index decorDIFFERENCE:<elimrowcol           |Indexed to pivot level.
                                decorMtx decorPRODUCT:<elimrowcol ! pcolrow> >
        decorNORMALIZE:<elimrowcol
                            <OFF                OFF     OFF     OFF     >: <
    if:<Scalar?:decorMtx    <decorMtx           0       0       decorMtx>
                                                                tail:decorMtx>
                                    spread4:elimrowcol
                                    <IDENT*>:SWAP:<IDENT*>:spread4:SWAP:pcolrow
                            <   div:<index 2>                   *       >   > >>

spread4 = \[erow ecol]. let:[
    [                   [eleft eright]                          [etop ebot]   ]
    <if:<Scalar?:erow   <erow  erow  >
          isAtm?:erow   <erow  erow  >
                            erow        >
                                        if:<Scalar?:ecol        <ecol ecol>
                                             isAtm?:ecol        <ecol ecol>
                                                                  ecol      > >
    <<eleft etop>       <eright etop>   <eleft ebot>    <eright ebot>>]
```

```
PIVOT = \[decorMtx elimrowcol permutT]. if:<
        |returns
        |     [decorMtx [elimrow elimcol] permutT [pivotrow pivotcol][ipos jpos]]
                        | permutT is a permutation matrix---normalized
                        | elimrow and elimcol  are boolean vectors;
                        |       [] => that row is eliminated
                        | pivotrow and pivotcol
                        |       are vectors of numbers, respectively
                        |     b/a  and  +c  entries
                        |       except for pivot elt, which is  1/a  or -1/a
                        | ipos and jpos are inverse-bit rel indices of pivot elt
        Scalar?:decorMtx        let:[ reciprocal  scalarRECIPROCAL:decorMtx
            < reciprocal [[] []] 1 <scalarNEGATE:reciprocal reciprocal> [1 1]> ]
                        |   Pivotrow entry is -1/a for pivot column construction
                        |   Pivotcol entry is +1/a for pivot row construction
                        |   No i,j coordinates here
                        |       because logarithms are zero; base case.
        let:[  [permutT [[max ibit jbit] ! mtx]
                        [pII pIII pIV stet incr] ]
               <if:<isNUMERAL?:permutT <permutT 0 0 permutT> permutT>
                decorMtx
                <\[i ii iii iv].<ii i iv iii>   |pII    permute NE quad to NW
                 \[i ii iii iv].<iii iv i ii>   |pIII   permute SW quad to NW
                 \[i ii iii iv].<iv iii ii i>   |pIV    permute SE quad to NW
                 \n.add:<n n>                   |stet   index builder
                 \n.inc:add:<n n>               |incr   index builder
                >
               >
        let:[                           [perMTX perPT perROW perCOL tweaki tweakj]
                        |  These are all permutations for arguments &/or results
          if:<          | Make all cases uniformly manipulate upper-left quad
                all?:<zero?:ibit zero?:jbit> <IDENT IDENT IDENT IDENT stet stet>
                zero?:ibit                   <pII   pIII  IDENT SWAP  stet incr>
                zero?:jbit                   <pIII  pII   SWAP  IDENT incr stet>
                                             <pIV   pIV   SWAP  SWAP  incr incr>
             >
        let:[[ [epivot [] [] [eright ebot]]     [permutHEAD!permutTAIL] ]
             < perMTX:spread4:elimrowcol        perPT:permutT           >
        rec:[
            [   [i [eleft etop] permutPIVOT [pleft ptop] [ipos jpos]]
                    [ii pright]  [iii pbot] iv]
                <PIVOT          ROW             COL             OFF          >:<
                perMTX:mtx
                <epivot         <eright etop>   <eleft ebot>    <eright ebot>>
                <permutHEAD     ptop            pleft           <pbot pright>>
                <[]             ipos            jpos            *            > >
        let:[ [ elimrowcol pivotrowcol]
              < < if:<all?:<nil?:eleft nil?:eright> [] perCOL:<eleft eright>>
                  if:<all?:<nil?:etop  nil?:ebot  > [] perROW:<etop  ebot  >> >
                <perCOL:<pleft pright>          perROW:<ptop pbot>           >
             >
        <decorNORMALIZE:<elimrowcol   perMTX:<i ii iii iv> >
            elimrowcol
            Normalize:perPT:<permutPIVOT ! permutTAIL>
            pivotrowcol                 <tweaki:ipos tweakj:jpos>       >
        ]]]]  ]>
```