

Matrix Algorithms using Quadrees
Invited Talk, ATABLE-92*
Technical Report 357

David S. Wise[†]
Computer Science Department
Indiana University
Bloomington, Indiana 47405-4101, USA
dswise@cs.indiana.edu

June, 1992

Abstract

Many scheduling and synchronization problems for large-scale multiprocessing can be overcome using functional (or applicative) programming. With this observation, it is strange that so much attention within the functional programming community has focused on the “aggregate update problem” [10]: essentially how to implement FORTRAN arrays. This situation is strange because in-place updating of aggregates belongs more to uniprocessing than to mathematics.

Several years ago functional style drew me to treatment of d -dimensional arrays as 2^d -ary trees; in particular, matrices become quaternary trees or quadrees. This convention yields efficient recopying-*cum*-update of any array; recursive, algebraic decomposition of conventional arithmetic algorithms; and uniform representations and algorithms for both dense and sparse matrices. For instance, any nonsingular subtree is a candidate as the pivot block for Gaussian elimination; the restriction actually helps identification of pivot blocks, because searching a tree is easy.

A new block-decomposition formulation for LU decomposition, called $(L + U)$, D' decomposition, is particularly well suited to quadrees. It provides efficient representation of intermediate matrices when pivoting on blocks of various sizes, *i.e.* during “undulant-block” elimination. A given matrix, A is decomposed into two matrices, plus two permutations. The

*To appear in *Proc. Intl. Workshop on Arrays, Functional Languages, and Parallel Systems*, Montreal (1992).

[†]Research reported herein was sponsored, in part, by the National Science Foundation under Grant Number CCR 90-02797.

permutations, P and Q , are the row/column rearrangements usual to full pivoting (one of which is I under partial pivoting). The principal results are $(L+U)$ and $(QDP)^{-1}$, where L , respectively U , is proper lower—and respectively proper upper—triangular, D is quasi-diagonal following zero blocks in $L+U$, and $PAQ = (I+L)D(U+I)$.

It is offered as a challenge for functional programming styles, (both HASKELL's and APL's) as the kind of algorithm that is desirable under parallel and distributed processing, and which should be lucidly expressed in such languages. Programmers must be encouraged to express both the parallelism and the sharing that is implicit here. Mathematics allows both; efficient computation requires them; programs should provide them.

1991 Mathematics Subject Classification: 65F05.

CR categories and Subject Descriptors:

E.1 [Data Structures]: Trees; G.1.3 [Numerical Linear Algebra]: Linear systems, Matrix inversion, Sparse and very large systems; D.1.1 [Applicative (Functional) Programming Techniques].

General Term: Algorithms.

Additional Key Words and Phrases: Gaussian elimination, block algorithm, LU factorization, quadtree matrices, quaternary trees, exact arithmetic.

0 Introduction

Iverson built many novel attributes into APL; almost all of them appeared in other languages since then. Interactive dialog, novel in the era of batch processing, is now necessary to personal—and personalized—computers. Overloading of operators not only is one of the pillars of Object Oriented Programming, but also has been extensively analyzed in the development of polymorphic type checkers, as in ML or HASKELL, for instance. A significant contribution from APL is its rich vocabulary for array operations, which has become the heritage of this meeting.

From the pure-functional programming camp I come to assail some constraining prejudices that are part of that heritage: APL's aggregate/matrix operations that, regrettably, focus on row/column decomposition. If one asserts, since the underlying model for memory in a modern computer is an array, that row- or vector-processing is closest to “natural” computation, I want to undo that prejudice, as well. My model of memory is a heap of relatively small nodes, linked into trees, dags, and graphs.

Lucid expression of parallelism requires a comfortable vocabulary and style for mapping functions. Mapping or “spreading” of application across data structure is the best applicative analog of parallelism, and is particularly important in a functional language because the mapped function is necessarily independent in all its applications. (Collateral argument evaluation is the simplest and most common case of such mapping.) When mapping occurs early in the

divide-and-conquer decomposition of a problem, each application represents a desirably large granule of computation that can be further mapped. APL maps across arrays, which is insufficient; HASKELL [11] maps (or (cringe) `zipWith3s`) only across lists; this is also too weak. The most important target for mapping is homogeneous tuples of relatively small size, so to control the growth of parallelism—and the scheduling problem. Much is done here with fourples.

My task is to propose and to justify a new approach to an old algorithm, Gaussian elimination—an essential result from matrix algebra. While the performance of this algorithm won't beat a Cray so far, it has ample opportunities for multiprocessing, for distributed processing, for algebraic transformation, and even for partial evaluation; that is, it exhibits lots of structured opportunities for massive parallelism, just as we expect from a functional program.

The challenge is to express array algorithms like this in your favorite array language. Since the principal data structure is a quaternary tree and no row or column operations are used, vector operations may be useless. Block decomposition and block arithmetic are important; sometimes blocks carry other decorations, aside from their constituent elements.

The remainder of this paper is in six parts, much of it condenses results from elsewhere. Section 1 presents the definitions of the quadtree representation and presents analytic results and describes experience using it as a technique to compress sparse matrices. The second section introduces quadtree operations using the HASKELL language. Section 3 contains a mathematical presentation of $L + U, D'$ decomposition, and a discussion of its motive, notably the concept of undulant-block pivoting. Section 4 considers how to select a good pivot—using tree search—and Section 5 reviews experience with scalar pivoting on a multiprocessor. The final section visits the motivations and prospects for this algorithm, and its lessons to future array languages.

1 Quadtree representation of matrices.

Dimension refers to the number of subscripts on an array. Of special interest here are square matrices. *Order* of a square matrix is the cardinality of a basis for its underlying vector space: the number of its rows or columns when it is written as the conventional tableau. Orders and sizes here will mostly be powers of two; arrays of other sizes can cheaply be padded with zeroes.

Thesis. *Any d -dimensional array is decomposed by blocks and represented as a 2^d -ary tree.*

Only vectors and matrices are considered below: where $d = 1$ yields binary trees and $d = 2$ suggests quaternary trees—or quadtrees [12].

Binary Vector Representation. *A vector of size 2^p is either homogeneously zero and represented by NIL, or $p = 0$ and its element is a non-zero scalar, and*

is represented by that value; or it is non-trivial is represented by a binary tree whose subtrees each represent vectors of size 2^{p-1} .

The subtrees are called *north* and *south*, consistently with the usual columnar representation of vectors.

Quadtree Matrix Representation. A matrix of order 2^p is either homogeneously zero, in which case it is represented *NIL*; or $p = 0$ and its element is a non-zero scalar, and is represented by that scalar; or else it is represented by an ordered quaternary tree whose subtrees each represent matrices of order 2^{p-1} .

These subtrees are respectively identified as *northwest*, *northeast*, *southwest*, and *southeast*, isomorphic to the four quadrants of a block decomposition of the conventional tableau, in the order that those names suggest.

Table 1 is borrowed [19] to display analytical results of the number of nodes, and the average path length in quadtree matrices of various patterns. The number of nodes is a measure of space; for an $n \times n$ matrix it ranges from $4/3n^2$ (33% above serial representation) down to zero. Remarkably, patterned matrices (Hankel, banded) can otherwise be represented in linear space, also require only linear space as quadtrees. Thus, the quadtree seems to be a general, uniform representation that requires only slightly more space than we might expect from other *ad hoc* data structures.

The path length is a measure of the time to access a random element of the array, starting from the root. Two remarks about this column are necessary: first, every representation for generally sparse arrays has non-constant access time. This one requires logarithmic time that improves to a very small constant in special cases. Second, any decent sparse matrix algorithm does not probe repeatedly from the root of the matrix, so this measure is excessively conservative. In the next section we shall see that ordinary algorithms follow the tree decomposition to perform significant amounts of computation from nodes interior to the tree.

Importantly, Table 1 indicates that this single representation responds well to sparse pathologies, so that we can write uniform algorithms and run them on both sparse and non-sparse problems with confidence that they will perform with reasonable efficiency in special cases.

Related results by Beckman [2] show another use of quadtree representation: to configure sparse matrices for input/output. The Harwell-Boeing test suite [6] is formatted on magnetic media to suit Fortran transput conventions; it contains both floating-point entries and location descriptors about each one's location in the matrix. The entire tape was translated to quadtree format with a byte (really only a 4-bit nybble) for each non-terminal node and eight bytes for every terminal node (the floating-point entries). The data was stored in preorder-sequential order with four bits in the single byte to identify non-*NIL* sons of each father. Testing on 283 of the 293 matrices on the tape showed that the location data compressed to about 25% of its former volume. When specific

Pattern	Space	Expected Path
Full	$\frac{4}{3}(n^2 - \frac{1}{4})$	$\lg n + 1$
Symmetric	$\frac{2}{3}(n + 2)(n - \frac{1}{2})$	$\lg n + 1$
Hankel/Toeplitz	$4n - \lg n - 3$	$\lg n + 1$
Triangular	$\frac{2}{3}(n + 2)(n - \frac{1}{2})$	$\frac{\lg n}{2} + \frac{3}{2} - \frac{1}{2n}$
fFT permutation	$\frac{n \lg n}{2} + \frac{4n}{3} - \frac{1}{3}$	$\frac{\lg n}{2} + \frac{4}{3} - \frac{1}{3n}$
Random permutation	$\frac{n \lg n}{2} + 0.9n - \frac{4}{3}$	$\frac{\lg n}{2} + 0.9$
Diagonal	$2n - 1$	$2 - \frac{1}{n}$
Tridiagonal	$6n - 2\lg n - 5$	$\frac{10}{3} - \frac{3}{n} + \frac{2}{3n^2}$
Pentadiagonal	$8n - 2\lg n - 9$	$\frac{10}{3} - \frac{1}{n} - \frac{10}{3n^2}$
Heptadiagonal	$11n - 2\lg n - 19$	$\frac{10}{3} + \frac{5}{n} - \frac{76}{3n^2}$
Enneadiagonal	$13n - 2\lg n - 27$	$\frac{10}{3} + \frac{7}{n} - \frac{100}{3n^2}$
Shuffle permutation	$3(n - 1)$	$3(1 - \frac{1}{n})$
Zero	0	0

Table 1. Costs of patterned and unpatterned matrices as quadrees.

floating values are also stored, this netted a 20% reduction in disk space across the entire set. In many instances, values for the entries are stored at all; the data is purely matrix patterns. Then the savings was around 75%

Not only is the disk-resident library thereby compressed, but also its static representation remains serially readable—for instance, to retrieve particular entries. Although this compression was done for our I/O convenience, the results might, nevertheless, be useful on any computer without room for raw representation of matrices.

2 Matrix rings in HASKELL

The code below is an introduction to HASKELL [11] and to quadtree programming. Important to understand is the `data` declaration. It declares “constructors” to be used later in pattern-matches to identify special cases. `ZeroM` is a nullary constructor, easily recognized and treated as an additive identity and multiplicative annihilator. The other constructors follow the definition of quadtrees, except for `IdentM` which allows efficient implementations of permutations as sparse matrices.

The `instance` declaration establishes overloading of the ring operations `+`, `*` so that these are now defined on quadtree matrices as well as numbers. Haskell can infer the definition of `-` from `negate` and `+`, or it can also be specified: one tree traversal rather than two. In the Gaussian elimination algorithm, there is no syntactic distinction between scalar and block multiplication—nor is there here. [9, §1.3]

Readers familiar with Strassen's algorithm [14] will recognize the quadtree as ideally suited to implement it, and HASKELL as the perfect language in which to code it. In fact, as was pointed out early on [15], this code can even beat Strassen's because it knows the algebra of ZeroM, and uses it to great advantage on sparse matrices.

```

module UndecoratedMatrices(..) where
  type Quadrants a = [Matrx a]           --list of *four* submatrices.
  data Matrx a = ZeroM | ScalarM a | Mtx (Quadrants a)
                | IdentM                --used mostly in permutations.
  instance (Num a) => Num (Matrx a) where
    fromInteger 0      = ZeroM
    fromInteger 1      = IdentM
    fromInteger _      = error "fromInteger defined only on 0/1 Matrces."
    negate 0           = 0
    negate (ScalarM x) = ScalarM (negate x)
    negate (Mtx quads) = Mtx (map negate quads)

    x      + 0      = x
    0      + y      = y
    ScalarM x + ScalarM y = case x+y of 0 -> 0
                                         z -> ScalarM z
    Mtx x   + Mtx y   = case zipWith (+) x y of [0,0,0,0] -> 0
                                                quads   -> Mtx quads
    _       + _       = error "Matrx addition undefined on IdentM"

    0      * _      = 0
    _      * 0      = 0
    1      * y      = y      --NB: multiplication accepts IdentM
    x      * 1      = x
    ScalarM x * ScalarM y = ScalarM (x*y)
    --Except with infinitesimal floats: case x*y of 0->0; z->ScalarM z
    Mtx x   * Mtx y   = (case zipWith (+)
                          (zipWith (*) (colExchange x)(offDiagSqsh y))
                          (zipWith (*)      x      (prmDiagSqsh y))
                          of [0,0,0,0] -> 0
                             quads   -> Mtx quads
                          ) where colExchange [nw,ne,sw,se] = [ne,nw,se,sw]
                                prmDiagSqsh [nw,ne,sw,se] = [nw,se,nw,se]
                                offDiagSqsh [nw,ne,sw,se] = [sw,ne,sw,ne]

    abs _ = error "abs not implemented yet on Matrces."
    signum _ = error "signum not implemented yet on Matrces."

```

In fact, experiments in REDUCE that applied these algorithms to integer and symbolic matrices [1] of sizes 4×4 up to 100×100 demonstrated the accelerations with sparsity that one would expect from Table 1. The improvement depends only on the axioms on ZeroM for identity and annihilation. A blind matrix inversion was also tested, with dramatic accelerations in sparse cases, and understandable slowdowns on denser ones. Non-singularity of every north-west quadrant was presumed, this unsatisfactory assumption is addressed with the following algorithm.

3 $(L + U), D'$ decomposition

The motivation for this algorithm is parallel processing. As parallel architectures become more available, known algorithms like Gaussian elimination will be reformulated in order to satisfy needs of parallelism that have been hitherto ignored: for instance, a extensible supply of independent processes, each of slowly receding granularity. Block algorithms provide a supply, but they raise other problems, like efficient representation and global permutations, that are also addressed here.

This terse section is abstracted from Wise [20]. The algorithm is Gaussian elimination of blocks of varying sizes. Therefore it is called “undulant” block pivoting. It is presented in the general case of full pivoting, but partial pivoting is surely available. This formulation supersedes earlier attempts at Gauss-Jordan elimination on quadrees [17, 18].

It is distinguished from pivoting on blocks of fixed size [9], that can be unstable when there is no decent pivot block of the fixed size [5]. Under undulant-block pivoting, however, one can always eliminate a troublesome, large entry as a 1×1 block before resuming elimination of larger blocks.

Definition. A matrix A is proper lower (upper) triangular if $a_{i,j} = 0$ for $i \leq j$ (respectively, $i \geq j$).

Notation. I denotes the identity matrix of any order. Similarly, 0 denotes the zero matrix of any order.

Definition. Two matrices, A and B , are said to be disjoint if

$$\forall i, j (a_{i,j} = 0 \vee b_{i,j} = 0).$$

Definition. A square matrix A is quasi-diagonal [7] if it has square submatrices (cells) along its main diagonal with its remaining elements equal to zero.

This more obscure term is used instead of *block-diagonal* to emphasize that the blocks along the diagonal can differ in size, as Faddeeva illustrates.

If D is an $n \times n$ quasi-diagonal matrix with b nonzero blocks then, following the block decomposition, one can partition the basis of the underlying vector

space, decomposing it into b mutually complementary subspaces. Thus, problems on D can be decomposed into b small, independent problems: one in each subspace.

The proper lower triangular matrix, L , associated below with a quasi-diagonal matrix, D , will have zero submatrices exactly where D has nonzero matrices. Therefore, the above decomposition on the vector space underlying D can be applied to $I + L$, as well. The same can be said for associated upper triangular matrices, U .

Definition. An $(L + U), D'$ decomposition of a nonsingular matrix A , is the quadruple, $\langle P, Q, L + U, (QDP)^{-1} \rangle$ where

- P and Q are permutations;
- L is proper lower triangular and U is proper upper triangular;
- D is quasi-diagonal and disjoint from both L and U ;
- $PAQ = (I + L)D(U + I)$.

It is trivial to separate $(L + U)$ into unit upper and unit lower triangular matrices, $I + L$ and $U + I$, but this is never necessary.

Notation. Subscripts n, m, s , should be read as “north, middle, south;” and w, c, e as “west, central, east.”

Algorithm 1 (Pivoting nonsingular A to $\langle P, Q, A', D' \rangle$.) Full, undulant-block pivoting is assumed, although no strategy for selecting pivot blocks is yet addressed. The quadruple of results from repeated pivoting on a matrix A is described recursively as follows. If A is void then the result is $\langle I, I, 0, 0 \rangle$.

Otherwise, let the block decomposition of A , isolating the $k \times k$ pivot block, A_{mc} , be labeled

$$A = \begin{matrix} & & j & k & n - k - j \\ & i & & & \\ & k & \begin{pmatrix} A_{nw} & A_{nc} & A_{ne} \\ A_{mw} & A_{mc} & A_{me} \\ A_{sw} & A_{sc} & A_{se} \end{pmatrix} & & \\ n - k - i & & & & \end{matrix}$$

where A is $n \times n$ and A_{nw} is $i \times j$. The trivial case has $i = 0 = j$ and $k = n$. Then

$$\langle P, Q, A', D' \rangle = \left\langle \left(\begin{pmatrix} 0 & I & 0 \\ P_w & 0 & P_e \end{pmatrix}, \begin{pmatrix} 0 & Q_n \\ I & 0 \\ 0 & Q_s \end{pmatrix}, \begin{pmatrix} A'_{nw} & A'_{nc} & A'_{ne} \\ A'_{mw} & 0 & A'_{me} \\ A'_{sw} & A'_{sc} & A'_{se} \end{pmatrix}, \begin{pmatrix} D'_{nw} & 0 & D'_{ne} \\ 0 & D'_{mc} & 0 \\ D'_{sw} & 0 & D'_{se} \end{pmatrix} \right) \right\rangle$$

where P_w is $(n - k) \times i$, P_e is $(n - k) \times (n - k - i)$, Q_n is $j \times (n - k)$, and Q_s is $(n - k - j) \times (n - k)$. These values can be computed as follows: first

$$D'_{mc} = A_{mc}^{-1}$$

is solved recursively (with good accuracy); then the pivot row and pivot column are completed using BLAS Level 3 operations [5],

$$\begin{aligned} (A'_{mw} \quad A'_{me}) &= D'_{mc} (A_{mw} \quad A_{me}) \\ \begin{pmatrix} A'_{nc} \\ A'_{sc} \end{pmatrix} &= \begin{pmatrix} A_{nc} \\ A_{sc} \end{pmatrix} D'_{mc}; \end{aligned}$$

finally,

$$\left\langle (P_w \quad P_e), \begin{pmatrix} Q_n \\ Q_s \end{pmatrix}, \begin{pmatrix} A'_{nw} & A'_{ne} \\ A'_{sw} & A'_{se} \end{pmatrix}, \begin{pmatrix} D'_{nw} & D'_{ne} \\ D'_{sw} & D'_{se} \end{pmatrix} \right\rangle$$

results from recursive pivoting on the $(n - k) \times (n - k)$ problem

$$\begin{pmatrix} A_{nw} & A_{ne} \\ A_{sw} & A_{se} \end{pmatrix} - \begin{pmatrix} A_{nc} \\ A_{sc} \end{pmatrix} (A'_{mw} \quad A'_{me}),$$

that can also be derived using Level 3 operations. \square

Theorem 1 . Let $\langle P, Q, A', D' \rangle$ be the result from Algorithm 1 on nonsingular input A . Then the $(L + U), D'$ decomposition of A is $\langle P, Q, PA'Q, D' \rangle$.

Corollary 1 . D' in Algorithm 1 is a rearrangement of a quasi-diagonal matrix to fit the structure of the pivots, as selected.

Algorithm 2 ($(L + U), D'$ decomposition.) Use Algorithm 1 to compute $\langle P, Q, A', D' \rangle$ from A , and apply the permutations *once* to return $\langle P, Q, PA'Q, D' \rangle$. \square

Algorithm 3 (Solving a linear system.) Solve $A\vec{x} = \vec{b}$ using this reformulation:

$$PAQ = (I + L)D(U + I)$$

implies

$$P^{-1}(I + L)Q^{-1}(QDP)P^{-1}(U + I)Q^{-1}\vec{x} = \vec{b}$$

1. Compute the $(L + U), D'$ decomposition of A using Algorithm 2.
2. [forward substitution] Solve $(I + L)\vec{y} = P\vec{b} = \vec{c}$.
 - If $L + U = 0$ then $\vec{y} = \vec{c}$.
 - Otherwise, partition¹

$$L + U = \begin{pmatrix} L_{nw} + U_{nw} & E \\ W & L_{se} + U_{se} \end{pmatrix}; \quad \vec{y} = \begin{pmatrix} \vec{y}_n \\ \vec{y}_s \end{pmatrix}; \quad \vec{c} = \begin{pmatrix} \vec{c}_n \\ \vec{c}_s \end{pmatrix}.$$

¹ E and W should be chosen for compact representation under the matrix representation; secondarily, one might choose E and W to be of about equal size, or to force either $L_{nw} + U_{nw}$ or $L_{se} + U_{se}$ to be entirely zero.

- Recursively solve $(I + L_{nw})\vec{y}_n = \vec{c}_n$.
 - Recursively solve $(I + L_{se})\vec{y}_s = \vec{c}_s - W\vec{y}_n$.
3. [backward substitution] Similarly, solve $(U + I)\vec{z} = P(D'(Q\vec{y}))$.
 4. Permute $\vec{x} = Q\vec{z}$. \square

Algorithm 4 (Matrix inversion.) Invert

$$A = P^{-1}(I + L)[Q^{-1}(QDP)P^{-1}](U + I)Q^{-1}.$$

1. Compute the $(L + U), D'$ decomposition of A using Algorithm 2.
2. If $L + U = 0$ then $A^{-1} = (QP)D'(QP)$;
3. Otherwise compute $L' = (I + L)^{-1}$ and $U' = (U + I)^{-1}$ recursively; then $A^{-1} = Q(U'(PD'Q)L')P$.
 - Partition $L + U$ as in the previous algorithm.
 - Recursively compute

$$L'_{nw} = (I + L_{nw})^{-1}; L'_{se} = (I + L_{se})^{-1};$$

$$U'_{nw} = (U_{nw} + I)^{-1}; U'_{se} = (U_{se} + I)^{-1}.$$

- Then

$$L' = \begin{pmatrix} L'_{nw} & 0 \\ -L'_{se}WL'_{nw} & L'_{se} \end{pmatrix}; \quad U' = \begin{pmatrix} U'_{nw} & -U'_{nw}EU'_{se} \\ 0 & U'_{se} \end{pmatrix}.$$

\square

The preceding presentation says nothing about matrix representation. Certainly the algorithm is intended for block decomposition, but it does not require any particular one. Blocks may be sized according to the size of pages under virtual memory, as long ago recommended [8], or be determined by the geography of a block within a distributed system. Of course, a block is here intended to be a quadrant, or a quadrant of a quadrant, *etc.*: some subtree of a quadtree.

The halving in quadtree decomposition assures some sort of balance—at least early on in the pivoting—in the divide and conquer strategy. It also assures that all represented subblocks are square and, therefore, themselves viable as candidates for elimination. Moreover, at each level of the tree there is either a “north” or a “south” band—never both, and that band is just as tall as is the “middle” one; similarly one deals exclusively with “west” or “east” and this simplifies the coding tremendously. Furthermore, both normalization (to ZeroM) and tournament contention (*v.i.*) are of minimal degree: four rather than nine as the displays above suggest.

Whatever block decomposition one chooses for data, the algorithm should decompose its control following it. In the case of quadtrees, it is intended (Should I say *required*?) that elimination descend the tree recursively—just like matrix addition—so that the array “flop”

$$\begin{pmatrix} A_{nw} & A_{ne} \\ A_{sw} & A_{se} \end{pmatrix} - \begin{pmatrix} A_{nc} \\ A_{sc} \end{pmatrix} (A'_{mw} \quad A'_{me}),$$

is dispatched not from the root of the tree, but at every level as the elimination algorithm backs up the path from the newly eliminated block to the root. Moreover, it can fork to four parallel flops as it descends off-pivot blocks sibling to that path. It is important to inject locality into implementation by distributing the flopping across siblings—just as addition is mapped across addends.

4 Pivot Selection

Any general elimination algorithm must provide for a search of what is to be eliminated; most often partial pivoting is used. Not only does such pivot selection avoid elimination on a singular block, but also steers efficacy of the algorithm toward practical goals like stability, parallelism, and sparsity. While these may not change the final answer, they certainly affect the fun of getting there.

Usually a search for the “midcentral block” occurs between applications of elimination, between recursive invocations of Algorithm 1. Here follows an argument that quadtrees distribute that pivot search across Algorithm 1 so that no extra traversal (bandwidth) is incurred for “pivoting.” (The idea is hardly new [13], but it is entertaining that its loss is blamed on FORTRAN.)

Consider, for example, the case of scalar pivoting on the element of largest magnitude. Full pivoting seems to require traversal of the entire matrix. Instead, decorate every non-terminal (Mtx) node of A with two bits and the maximal magnitude of elements in that block.² The two bits identify which of the four quadrants contains that element. Such a decoration is cheaply propagated, tournament-style, as every incarnation of A is built. After each elimination step A is decorated with bits leading toward the next element to be eliminated. Thus, no special search is ever needed; pivoting has been distributed across elimination. Moreover, when annihilation of a flop of, say, A_{sw} occurs because, say, $A_{sc} = 0$ then the value *and* the decorations within A_{sw} remain valid. This is good algebra and very good computation!

Other local properties can determine winners of the tournament: to minimize fill-in, and to maximize block size of the “eliminand.” Lest the search space become overwhelming, the only blocks that correspond to subtrees of the quadtree are pivot candidates; already encapsulated, they have addresses.

²The decorated value can be fairly coarse: *e.g.* the exponent from a floating-point number.

Eliminating a larger block increases parallelism and reduces bandwidth; two or four elements can be eliminated for the communication and scheduling costs associated with one; granularity of the pivoting process is raised. Fill-in is predicted using Markovitz vectors [4] accumulated during a former elimination. Size is isomorphic to the depth of the block within the tree; a semidecision procedure [20] is used to identify which blocks in A are non-singular.

Finally (and new here) is the observation that the pivot block can be expanded as Algorithm 1 backs up the tree that is A' . Consider the case of a non-trivial subtree $A_{\text{subtree}} = \begin{pmatrix} A_{nw} & A_{ne} \\ A_{sw} & A_{se} \end{pmatrix}$ where A_{nw} has been selected for elimination by the tournament and has *just* been eliminated, but no results have yet been propagated up the tree. Ignoring the permutations for a moment, the results would be A' and D' :

$$A'_{\text{subtree}} = \begin{pmatrix} 0 & A'_{ne} \\ A'_{sw} & A'_{se} \end{pmatrix}; \quad D'_{\text{subtree}} = \begin{pmatrix} A_{nw}^{-1} & 0 \\ 0 & 0 \end{pmatrix}.$$

If, before returning these results, it determined (by inspection) that A'_{se} is also non-singular then it can be decomposed and the local inversion of A_{subtree} completed on the spot; the results instead become

$$A'_{\text{subtree}} = 0; \quad D'_{\text{subtree}} = A_{\text{subtree}}^{-1}.$$

That is, although A_{nw} had won the tournament and was being eliminated, “alert” pivoting instead eliminated all of A_{subtree} . In this way, a quadtree decorated only for scalar elimination might, nevertheless, experience elimination of 2×2 blocks.

5 Experience

In order to test this family of algorithms, Beckman [3] has implemented them in C on the BBN Butterfly 1000 computer and tested them on twenty different matrices from the Harwell-Boeing collection [6]. Typically these were unsymmetrical, real, assembled matrices of order 1000 or so. Absolute performance must be interpreted because the system is layered over an expensive foundation: a global heap (managed by reference counting) on an architecture where memory latency is sensitive to non-local references (NUMA). This is not the recommended way to use such computers.

The results of these experiments indicate that these algorithms are viable, although perhaps not in the common case where the atomic elements are floating-point numbers. That is, the `ScalarM` elements of the quadtree would better require more than one flop in each step: *e.g.* symbolic arithmetic, exact arithmetic, or elements of hypermatrices [4]. The problem is that there is significant overhead in building nodes near the leaves of the tree that can only be amortized by an advantage in parallelism.

Experience shows that these algorithms do perform well on problems that have large subproblems. However, Crays still do far better on really dense floating arithmetic, and algorithms tuned to sparse algorithms (linear list representation) do far better on very sparse problems. The problem on large sparse problems is that there is insufficient fan-out as elimination descends the tree; sibling processes from early eliminations annihilate too quickly, with the result that the quadtree algorithm tends to behave like a slow uniprocessor. The overhead of building, decorating, and destroying leaf nodes appears to overcome whatever parallelism remains after annihilation.

On the other hand, where the parallel processes remain alive, the performance seems to scale. Only using six to eight processors (of 100), it can beat a linear-list sparse matrix package tuned to a uniprocessor on a sufficiently dense matrix. Problems that are too dense, however, are still Cray-fodder. In tests on hypermatrices, this algorithm running on four processors beat the linear-list package when hypermatrix entries were 8×8 or larger; on sixteen processors when the entries were 6×6 or larger. Therefore, this algorithm seems to be targeted for a middle ground—not too sparse and not too dense—or for an architecture more friendly to it.

6 Conclusion

In a practical sense, we have not demonstrated that this representation is successful, but we have confirmed the nature of problems that can use it. And we have shown that it responds very well to parallelism, in some respects (annihilation of processes) better than was anticipated.

However, regardless of its status among practitioners of matrix algebra, its role before this audience is as a representative of a promising family of matrix algorithms. It is presented to you as an algorithm for functional programming; I have coded it and its kin in pure Scheme and Haskell. The challenge is that your favorite array language should be able to express it, as well.

Whatever code results in any of these languages should address the goals that motivated it. Eventually a compiler—if not the programmer—must be able to extract parallelism from unannotated source code that might be written today. This means heavy use of algebra and mapping functions, where all functional languages can shine; most are rich in the former, but few yet in the latter.

The intent of this algorithm is to admit block operations as “first-class” citizens in matrix arithmetic. To that end, the tree representation has been convenient to limit the population of “blocks” and also to facilitate divide-and-conquer algorithms that ameliorate identification and scheduling of independent processes. It also provides an address (the root) where complicated structures carry summarizing decoration, so that the bandwidth to share a tree *and* its decoration collapses. An extreme case of this compression is the definition of additive/multiplicative identities/annihilators as trivial constructors, allowing

their algebra to be realized for free. Likewise, full pivoting was proposed because it, too, requires no additional scheduling.

Finally, this is a uniform algorithm: there is no difference between sparse and dense matrices or anything in between. Similarly, there is no distinction between hypermatrices, integers, floats, “bignums”, or symbolic data. Be sure to use your favorite polymorphic type-checker.

Of course, these advantages imposed some constraints. In order to facilitate sharing and locality, subtrees should be retained intact as much as possible for as long as possible. Repeated row/column permutations, taken for granted under usual matrix representations, must be avoided because they twist the tree terribly; a shared subtree of an unpermuted matrix bears no similarity to its arbitrary permutation. Of course, no in-place updating is specified, although linear logic might allow it to be compiled in later.

Above all, block decomposition algorithms must descend the structure to perform most of their computation in parallel on many substructures, rather than serially from the root. Not only is this critical to absorbing logarithmic access times (Table 1), but also it provides locality where, for instance, subblocks are stored locally—either on the same distributed processor or adjacently in cached or paged [8] memory.

My presumption that memory is a heap creates a large storage-management problem, which a separate project at Indiana is addressing. We are already building our second edition of a self-managing heap designed for general use on multiprocessors like the Butterfly [16]. It would accelerate the performance reported in the last section more than twice.

Significant work remains to be done. For instance, there has been no effort to order the matrix before applying this decomposition. Certainly a bad ordering can spread the data across blocks so that only scalar operations are possible; can a good ordering be found, one that clusters the matrix into subblocks of the proper size (powers of 2) at the proper indices (multiples thereof)? Perhaps padding of zeroes, hitherto done only at the edges, should be injected internally, to align blocks more favorably. Furthermore, a problem might be reordered part-way through elimination; not only would this increase the likelihood of block operations on resumption, but each reordering could remove level(s) from the tree, accelerating all transactions that follow.

A novel representation and a novel approach to familiar problems has been presented. It is encumbered by all sorts of limitations inherent in today’s architectures and languages, which were refined for another representation and another approach. If we find a way to write cogent parallel programs for the new approach, then future computer designers will be able to appreciate what is required for its proper implementation. So, if array-language designers and implementors can deliver convenient, expressive tools to these algorithms, then both these new languages and new algorithms will survive. Otherwise, we all will be stuck with conventional architecture, conventional languages, conventional algorithms, and conventional performance, because that’s all conventional com-

putists deserve.

References

- [1] S. K. Abdali & D. S. Wise. Experiments with quadtree representation of matrices *Proceedings 1988 Intl. Symp. on Symbolic and Algebraic Computation, Lecture Notes in Computer Science* **358** Berlin, Springer (1989), 96–108.
- [2] P. Beckman. Static measures of quadtree prerepresentation of the Harwell-Boeing sparse matrix collection. Tech. Rept. 324, Computer Science Dept., Indiana University (Jan. 1991).
- [3] P. Beckman. *Parallel LU Decomposition for Sparse Matrices using Quadtrees on a Shared-Heap Multiprocessor*. Ph.D. dissertation, Indiana University (in preparation).
- [4] I. S. Duff, A. M. Erisman, & J. K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford (1989).
- [5] J. W. Demmel & N. J. Higham. Stability of block algorithms with fast Level 3 BLAS. LAPACK Working Note 22, Computer Science Dept., The Univ. of Tennessee (revised: July 1991). *ACM Trans. Math. Software* (to appear).
- [6] I. S. Duff & R. G. Grimes & J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software* **15**, 1 (March, 1989), 1–14.
- [7] V. N. Faddeeva *Computational Methods of Linear Algebra*. Dover Publications, New York (1959). Translated from Russian, originally published Moscow (1950).
- [8] P. C. Fischer & R. L. Probert. Storage reorganization techniques for matrix computation in a paging environment. *Comm. ACM* **22**, 7 (July 1979), 405–415.
- [9] G. H. Golub & C. F. Van Loan. *Matrix Computations* 2nd edition. The Johns Hopkins University Press, Baltimore (1989).
- [10] P. Hudak. Arrays, non-determinism, side-effects, and parallelism: a functional perspective (extended abstract). In J. Fasel and R. Keller (eds.) *Graph Reduction. Lecture Notes in Computer Science* **279**, New York, Springer (1987), 312–327.
- [11] P. Hudak & P. Wadler (eds.) Report on the Programming Language Haskell, a Non-strict, Purely Functional Language, Version 1.2. *ACM SIG-PLAN Notices* **27**, 5 (May, 1992).

- [12] H. Samet. The quadtree and related hierarchical data structures. *Comput. Surveys* **16**, 2 (June, 1984), 187–260.
- [13] G. W. Stewart *Introduction to Matrix Computations*. Academic Press, New York (1973), 154.
- [14] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.* **13**, 4 (Aug. 1969), 354–356.
- [15] D. S. Wise. Representing matrices as quadtrees for parallel processors (extended abstract). *ACM SIGSAM Bulletin* **18**, 3 (August 1984), 24–25.
- [16] D. S. Wise. Design for a multiprocessing heap with on-board reference counting. In J.-P. Jouannaud (ed.), *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science* **201**, Berlin, Springer (1985), 289–304.
- [17] D. S. Wise. Parallel decomposition of matrix inversion using quadtrees. *Proc. 1986 International Conference on Parallel Processing* (IEEE Cat. No. 86CH2355-6), 92–99.
- [18] D. S. Wise. Matrix algebra and applicative programming. In Kahn, G. (Ed.), *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science* **274**, Springer, Berlin, 1987, 134–153.
- [19] D. S. Wise & J. Franco. Costs of quadtree representation of non-dense matrices. *J. Parallel Distrib. Comput.* **9**, 3 (July 1990), 282–296.
- [20] D. S. Wise. Undulant-block pivoting and $(L + U)$, D' decomposition. Tech. Rept. 328, Computer Science Dept., Indiana University (Sept. 1991).