Quadtree Matrix Algorithms Revisited: Basic Issues and their Resolution

D. Stott Parker Dinh Le
stott@cs.ucla.edu dinh@cs.ucla.edu
UCLA Computer Science Department
Los Angeles, CA 90024-1596 USA

With the *quadtree* representation of matrices popularized by Wise, many matrix algorithms are naturally implemented in a recursive functional style. This style is useful in exposing parallelism, identifying mappings for matrix computations onto particular computer architectures, and more generally in exposing new ways to reason about matrix algorithms.

The world has not beaten a path to quadtree-based algorithms, unfortunately. It seems there are a number of reasons that quadtree algorithms and architectures have not reached their potential in practice. Important among these are that quadtree represention are not unique, and that straightforward implementations of block-matrix algorithms fail for degenerate matrices (e.g., with singular quadrants). Proposed extensions of quadtree-based algorithms to deal with multiple representations, or degenerate matrices (such as resorting to pivoting), have not really removed the objections: they lose either numerical stability or the benefits in elegance, parallelism, and functional style that encouraged the use of quadtrees in the first place.

We survey the situation in this paper, and present new approaches for handling degenerate quadrants in quadtree algorithms: First, we rely on canonical quadtree representations. Second, we randomize the input matrix in a way that guarantees nondegeneracy of the quadrants, and also permits derandomization after the algorithm. This approach should make the quadtree representation more successful in real matrix computations.

We have been investigating alternative matrix representations and techniques which provide computational benefits on parallel processors. Since the outset, computer architecture has been strongly influenced by computational linear algebra. A number of specialized architectures (e.g., array processors) are designed primarily to run matrix computations, and with a non-recursive vector orientation in mind. We feel alternative matrix representations and functional style can lead to improved parallelism and programmability, and to interesting new specialized architectures.

The **quadtree**, a particularly interesting representation, has been popularized by David Wise [9, 10, 11, 12, 13]. Wise begins by noting that many matrix operations can be expressed in block form. For example, the following block-Gaussian Elimination step can be used recursively on square submatrices of a square matrix:

$$\begin{pmatrix} I & 0 \\ -CA^{-1} & I \end{pmatrix} \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} A & B \\ 0 & D - CA^{-1}B \end{pmatrix}.$$

Assuming that $\begin{pmatrix} A & B \\ C & D \end{pmatrix}$ is a nonsingular square matrix, and that A is nonsingular, its inverse is just $U^{-1}D^{-1}L^{-1}$, where the matrices $L\,D\,U$ are given by

$$\begin{pmatrix} I & 0 \\ CA^{-1} & I \end{pmatrix} \begin{pmatrix} A & 0 \\ 0 & D - CA^{-1}B \end{pmatrix} \begin{pmatrix} I & A^{-1}B \\ 0 & I \end{pmatrix}.$$

The determinant can also be computed as

$$det \begin{pmatrix} A & B \\ C & D \end{pmatrix} = det\, A \cdot det\,(D - CA^{-1}B).$$

Wise emphasizes the case where A, B, C, D are **quadrants** of the matrix (blocks of equal size). This yields elegant recursive algorithms that can be run effectively on some parallel computers. Wise has shown at length how many important matrix functions can be computed in a natural functional style with this representation.

At first glance it is surprising that four decades of work in computational linear algebra did not turn up the value of recursion and quadtrees before Wise's papers in the 1980's. Although block algorithms are still very important [2], they tend to be executed for *small* block sizes, and the quadtree approach still is not mentioned in new numerical analysis texts. The approach is also not mentioned in work on sparse matrices, where the quadtree representation has apparent advantages. What is the problem?

Apparently, while recursive specification of matrix operations using this block style is elegant, it is usually viewed as impractical. The block-Gaussian Elimination step above, for example, works only when the submatrix A is nonsingular, the upper left blocks of both A and $D - CA^{-1}B$ are nonsingular, the upper left blocks of both of these upper left blocks are nonsingular, etc. Although this recursive condition is known to hold for some important classes of input matrices, such as symmetric positive definite matrices, it certainly will not hold for all. There are other reasons also, but this lack of generality is a key reason why recursive definitions are not used more widely.

In implementing a number of quadtree-based matrix algorithms in ML, we got perspective on various problems confronting the approach. For example, the pivoting operation in general LDU decomposition is not really compatible with quadtrees, particularly if the matrix is sparse. It seems significantly more awkward than Wise makes clear in his discussion [11]. Also, the quadtree algorithms can be numerically unstable if not designed carefully.

The purpose of this paper is then to discuss ways that quadtree theory and functional style can be reconciled with computational linear algebra practice.

This paper begins by describing the quadtree matrix representation, and some subtle problems it raises. Several quadtree-based algorithms are then described, and the problems introduced by pivoting. We then describe a randomization technique in quadtree-based algorithms that avoids some existing problems, such as the need for pivoting.

The randomization technique is particularly interesting, since it gives real hope that block matrix computations can be performed in recursive, functional style without concern about exceptional situations like singularity. It is described in detail elsewhere [6, 5]. An overview of the technique is available in [7].

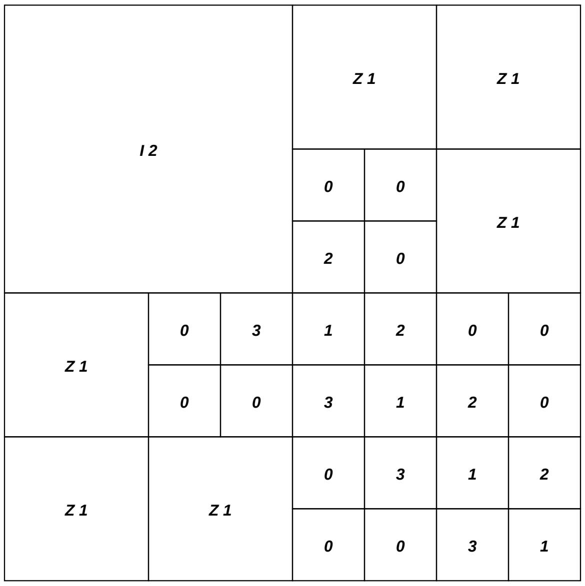
A **quadtree** representation of a matrix is restricted to be an indentity matrix, a zero matrix, a scalar, or a matrix that is composed of four equal-sized submatrices. In our ML implementation [3], quadtrees are defined polymorphically:

α **qtree** $=$
 $Zmat_q$ of int
| $Imat_q$ of int
| $Scal_q$ of α
| $Quad_q$ of α qtree $\times$ α **qtree** $\times$ α **qtree** $\times$ α **qtree**

Here α specifies an appropriate element type such as integers (**int**) or real numbers (**real**), $Imat(i)$ denotes an identity matrix of size $2^i \times 2^i$, $Zmat(i)$ denotes a $2^i \times 2^i$ zero matrix, $Scal(v)$ denotes a scalar of value v, $Quad(A11,A12,A21,A22)$ denotes a matrix of four equal-sized quadtree submatrices $A11$, $A12$, $A21$, and $A22$. For example, the matrix M_e

$Quad(Imat(2),$
 $Quad(Zmat(1),$
  $Zmat(1),$
  $Quad(Zmat(0), Zmat(0),$
   $Scal(2), Zmat(0)),$
  $Zmat(1)),$
 $Quad(Zmat(1),$
  $Quad(Zmat(0), Scal(3),$
   $Zmat(0), Zmat(0)),$
  $Zmat(1),$
  $Zmat(1)),$
 $Quad(Quad(Scal(1), Scal(2),$
   $Scal(3), Scal(1)),$
  $Quad(Zmat(0), Zmat(0),$
   $Scal(2), Zmat(0)),$
  $Quad(Zmat(0), Scal(3),$
   $Zmat(0), Zmat(0)),$
  $Quad(Scal(1), Scal(2),$
   $Scal(3), Scal(1))))$

can be represented pictorially as:



The matrix above is sparse, and the quadtree representation captures sparseness very effectively, better than commonly-used adjacency list representations for many matrix computations. The usefulness of the representation for sparse matrices could be improved, however, since it does not deal well with common sparse matrix types (such as tridiagonal or banded matrices), it does not address 'fill in' of zero entries resulting from computation in any real way, etc. Basically, quadtrees give an interesting representation of sparse matrices, but are not yet tailored to be effective in sparse matrix computations.

The process of converting a matrix, says M_n, of size $n \times n$ to a sparse representation, called M_c, is actually sort of interesting. Conversion can be naturally implemented as two stages — *padding* and *compacting*. First of all, if n is not a power of 2, then M_n is padded with, say, identity or zero matrices so that the resulting matrix M_p has *full size* $2^i \times 2^i$ where $i = \lceil \log_2 n \rceil$.

Compacting can then be applied to this matrix to render it more sparse, replacing all-zero subquadrants with $Zmat$(i), etc., and arriving ultimately at a quadtree matrix. It is not difficult to prove by induction on i that there is a unique 'maximally compact' quadtree representation for any $2^i \times 2^i$ matrix. So compacting is a normalization process that gives a unique *normal form* for full size quadtree matrices.

For example, consider the 5×5 matrix below:

1	2	0	0	0
3	1	2	0	0
0	3	1	2	0
0	0	3	1	2
0	0	0	3	1

The northwest corner can be padded with identity matrices to give the $2^3 \times 2^3$ matrix M_p:

1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	1	2	0	0	0
0	0	0	3	1	2	0	0
0	0	0	0	3	1	2	0
0	0	0	0	0	3	1	2
0	0	0	0	0	0	3	1

Compacting the matrix above would now yield the first quadtree matrix shown in this section.

A deep issue that must be dealt with here is that *most matrices have multiple quadtree representations.* This issue has many manifestations:

1. Compacting does not guarantee a unique normal form for matrices that are not full size. For this reason we have distinguished here between 'compacting' and 'normalizing' a matrix, since the latter term has the connotation of always yielding a normal form. Operating exclusively on full size matrices is clearly desirable.

2. There are many ways to pad to full size. If we wish to do padding to full size only once initially, rather than do padding before, and de-padding after, each matrix operation, then how padding is done makes a great difference. For example, in computing $(M_1 - M_2)^{-1}$, padding both M_1 and M_2 with either identity or zero matrices will fail. There seems to be no deterministic initial padding scheme that guarantees freedom from problems like this.

3. The identity, zero, and scalar subtypes are not disjoint, so multiple representations can exist even for 1×1 matrices. Some will be tempted to conflate these subtypes. However, we feel that the important roles of identity and zero matrices in matrix algebra and the advantages of strong typing outweigh the minor inconveniences of reconciling overlapping types.

4. Any matrix can have multiple semi-compacted representations. For example, the uncompacted matrix M_p above also is representable as a quadtree with either one $Imat(2)$, two $Imat(1)$, or four $Imat(0)$ submatrices. Semi-compacted representations will result from recursive block matrix computations. Maintaining normal forms throughout requires constant normalization after these computations, just as maintaining normalized values requires this after floating-point computations.

   Essentially the quadtree representation is an *expression* or *description* that *denotes* a numeric matrix. This is important, since operators on matrices (such as equality testing) must be clear about whether they are *intensional* (operate on the quadtree expressions) or *extensional* (operate on the corresponding denotations). Extensional operators should yield equivalent results on equivalent representations. Exactly when or if quadtrees are compacted to normal form is thus very important in practice.

5. Padding and compacting are related. For example, padding with identity matrices can give very different normal forms than padding with zero matrices (consider padding an identity matrix).

These issues reflect subtle aspects of the quadtree approach that must be managed carefully. For numerical analysts living in the non-recursive, mostly-extensional world view of FORTRAN, these issues will only increase reluctance to adopt quadtrees.

Quadtree-based matrix operators are easy to understand, yet present various challenges. Even matrix addition raises interesting issues, despite its trivial recursive specification:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} + \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}+B_{11} & A_{12}+B_{12} \\ A_{21}+B_{21} & A_{22}+B_{22} \end{pmatrix}.$$

This specification can be implemented with a function *matadd* taking two matrices of type $\mathbf{Mat}_q$ (matrices represented as **qtree** s). The cases can be grouped into base, zero matrix, identity matrix, and quad matrix cases:

(* Base cases *)
$matadd(Scal_q(v_1),\ Scal_q(v_2)) = Scal_q(v_1{+}v_2)$
$matadd(Imat_q(0),\ Imat_q(0)) = Scal_q(2)$
$matadd(Imat_q(0),\ Scal_q(v_2)) = Scal_q(1{+}v_2)$
$matadd(Scal_q(v_1),\ Imat_q(0)) = Scal_q(v_1{+}1)$

(* Zero matrix cases *)
$matadd(Zmat_q(\mathsf{i}),\ M_2) = M_2$
$matadd(M_1,\ Zmat_q(\mathsf{i})) = M_1$

(* Identity matrix cases *)
$matadd(Imat_q(\mathsf{i}),\ Imat_q(\mathsf{i})) =$
$\quad Quad_q(matadd(Imat_q(\mathsf{i\text{-}1}), Imat_q(\mathsf{i\text{-}1})),\ Zmat_q(\mathsf{i\text{-}1}),$
$\qquad Zmat_q(\mathsf{i\text{-}1}),\ matadd(Imat_q(\mathsf{i\text{-}1}), Imat_q(\mathsf{i\text{-}1})))$
$matadd(Imat_q(\mathsf{i}),\ Quad_q(\mathsf{B11,B12,B21,B22})) =$
$\quad Quad_q(matadd(Imat_q(\mathsf{i\text{-}1}),\ \mathsf{B11}),\ \mathsf{B12},$
$\qquad \mathsf{B21},\ matadd(Imat_q(\mathsf{i\text{-}1}),\ \mathsf{B22}))$
$matadd(Quad_q(\mathsf{A11,A12,A21,A22}),\ Imat_q(\mathsf{i})) =$
$\quad Quad_q(matadd(\mathsf{A11},\ Imat_q(\mathsf{i\text{-}1})),\ \mathsf{A12},$
$\qquad \mathsf{A21},\ matadd(\mathsf{A22},\ Imat_q(\mathsf{i\text{-}1})))$

(* Quad matrix case *)
$matadd(Quad_q(\mathsf{A11,A12,A21,A22}),$
$\qquad Quad_q(\mathsf{B11,B12,B21,B22})) =$
$\quad Quad_q(matadd(\mathsf{A11},\ \mathsf{B11}),\ \mathsf{matadd(A12,\ B12)},$
$\qquad matadd(\mathsf{A11},\ \mathsf{B11}),\ \mathsf{matadd(A12,\ B12)})$

The large number of cases stem from the overlap between the scalar, 1×1 identity, and 1×1 zero types, and the overlap between the general identity and quadtree types.

This function possesses a few other shortcomings. Despite the large number of cases, it is incomplete. It assumes the input matrices are both well-formed and of the same size, and (less obviously) it assumes the input matrices are both full-size. Finally, it does nothing to compact the resulting sum. These shortcomings are not fatal.

Deeper issues are raised by operators that impose more stringent requirements on their inputs (e.g., that they be Hermitian, positive definite, nonsingular, etc.). For example, the block LDU decomposition mentioned earlier

$$\mathrm{LDU}\left(\left(\begin{array}{cc} A & B \\ C & D \end{array}\right)\right) =$$

$$\left(\begin{array}{cc} \mathbf{I} & \mathbf{0} \\ CA^{-1} & \mathbf{I} \end{array}\right) \mathrm{LDU}\left(\left(\begin{array}{cc} A & \mathbf{0} \\ \mathbf{0} & D - CA^{-1}B \end{array}\right)\right) \left(\begin{array}{cc} \mathbf{I} & A^{-1}B \\ \mathbf{0} & \mathbf{I} \end{array}\right).$$

requires the submatrix A to be nonsingular. In general, this requirement does not hold, and LDU decomposition must be generalized to a (significantly more difficult-to-specify) LDUP decomposition [1].

In LDUP decomposition, *pivoting* is used to permute matrix entries so that the *NW*-entry is nonzero. In [3] a relatively complete implementation of pivoting on quadtrees was developed. Incorporating pivoting introduced many complexities into programming, more than suggested by Wise in [11]. For example: the datatype for quadtrees (**qtree**) was decorated with additional information information about matrix elements, giving a new type (**ptree**). A new datatype **elim** was introduced to keep track of the eliminant rows and columns, and information was kept on pivot elements and their positions. An indexing scheme was introduced to determine the pivot element, row elements, column elements and off elements. A new datatype (**pair**) was introduced to pass row and column elements to their appropriate quadrants. In order to preserve sparseness of the matrix, the implementation went to some lengths to preserve the identity and zero matrices. Various auxiliary functions (*decorate*, *rowlist*, *knuthPiv*, *piv1*, *piv2*, *piv3*, *piv4*, *piv5* and *piv6*) were needed to consider all possible cases and, thus, the code size was large (although comprehensible).

Unfortunately, pivoting not only sacrifices the conceptual benefits of block-matrix algorithms, but also sacrifices parallelism. It is a sequential approach requiring synchronization of iterations. Furthermore, pivoting gives formal backwards roundoff error bounds that are polynomial in the size of the matrix rather than in its logarithm.

Summarizing, a challenge to the whole quadtree approach is that recursive block-matrix definitions, while important in theory, often present problems in practice:

1. Since equivalent matrices can have multiple quadtree representations, a large number of cases are needed to cover this redundancy.

2. Block-matrix definitions often rely on assumptions that are impossible to guarantee in practice, such as that submatrices are nonsingular. Handling the general situation where these assumptions are dropped often leads to uglier definitions (such as LDUP decomposition) that sacrifice the benefits of quadtrees.

3. Block-matrix definitions do not capture realities of matrix computations, where high performance and good roundoff properties are primary goals. Despite their elegance, significant compiler technology is needed for recursive block-matrix definitions and quadtrees to be really competitive.

Several approaches have been proposed for addressing the requirement that submatrices be nondegenerate (recursively nonsingular). For example, symmetric positive definite matrices are known to meet this requirement. Bunch and Hopcroft [1] mention that any nonsingular matrix M can be made recursively block-nonsingular by premultiplying with its Hermitian adjoint (conjugate transpose) M^*. The resulting matrix M^*M is Hermitian and positive definite. With this approach, for example, we can compute the inverse M^{-1} by computing a generalized inverse of M:

$$M^+ \quad = \quad (M^*M)^{-1}\ M^*.$$

When M is nonsingular, $M^+ = M^{-1}$. Unfortunately this approach is relatively expensive, since it requires multiple matrix multiplications. Furthermore, it is less stable than Gaussian Elimination with pivoting, since the condition number of M^*M is always larger than that of M: $\| M^*M \|_2 = \| M \|_2^2$, where $\| \cdot \|_2$ is the spectral matrix norm, so with this norm the condition number of M^*M is the square of that for M. So although there are known techniques for avoiding degeneracy, they have poor numeric properties.

Many of the problems with quadtrees raised above stem from the fact that degenerate matrices must be handled carefully, although these matrices 'almost never arise'. The identity and zero matrices are important, but have measure zero in the space of all matrices. Similarly the matrices with singular *NW* quadrants have measure zero.

Our idea now is to exploit the fact that degenerate matrices almost never arise. (This can be viewed as an attempt to exploit a maxim of Ronald Reagan, who observed, "once you have it 90% right, who cares about the other 3%?") By injecting some randomization into the matrices, we can effectively eliminate the need for dealing with degeneracy.

The randomization technique is particularly useful in eliminating the possibility of singular quadrants. It converts a nonsingular matrix $M = \left(\begin{array}{cc} A & B \\ C & D \end{array}\right)$ to a revised matrix $M' = \left(\begin{array}{cc} A' & B' \\ C' & D' \end{array}\right)$ which has the property that its diagonal blocks are nonsingular (with probability 1). The randomized matrix M' can thus be used with block-matrix definitions that assume nonsingularity, and derandomization will then yield the correct answer when it is needed.

The technique is described in detail in [6, 5]. An overview is also available in [7]. Here we give a brief summary.

Let M be a nonsingular $n \times n$ matrix, and let V and W be random $n \times n$ matrices. Our basic idea is that, given the randomness of V and W, *we can perform LDU decomposition on the product matrix VMW without any pivoting.* Intuitively VMW is sufficiently 'random' that, with probability 1, pivoting is never needed. After LDU decomposition we can undo the effects of V and W.

Specifically, suppose M is a nonsingular real $n \times n$ matrix, let V and W be nonsingular random $n \times n$ matrices. (By a random matrix, we mean a matrix whose elements are chosen independently from a suitable distribution. We can guarantee such matrices to be nonsingular by choosing them from a nonsingular class of matrices, like orthogonal matrices. See [6].) Then:

- To solve $M\mathbf{x} = \mathbf{b}$, we can instead solve $(VMW)\mathbf{y} = V\mathbf{b}$. Afterwards, it follows that $\mathbf{x} = W\mathbf{y}$.
- To compute M^{-1}, we can instead compute $(VMW)^{-1}$, since V and W are both guaranteed to be nonsingular. Afterwards, it follows that $M^{-1} = W\,(VMW)^{-1}\,V$.
- To compute $det\,M$, we can instead compute

$$det\,(VMW)/(det\,V\;\,det\,W),$$

assuming the determinants of V and W can be computed easily.

Each of the computations here is easily implemented if LDU decomposition (without pivoting) successfully factors the preconditioned matrix VMW.

To illustrate the basic idea, consider the 2×2 matrix

$$M = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

If we arbitrarily select the random matrices

$$V = \begin{pmatrix} +.6483 & -.7614 \\ +.7614 & +.6483 \end{pmatrix}, \quad W = \begin{pmatrix} +.7279 & -.6857 \\ +.6857 & +.7279 \end{pmatrix}$$

we get the random transform

$$\begin{aligned} & VMW \\ = & \begin{pmatrix} +.6483 & -.7614 \\ +.7614 & +.6483 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} +.7279 & -.6857 \\ +.6857 & +.7279 \end{pmatrix} \\ = & \begin{pmatrix} .1097 & .9940 \\ .9940 & .1097 \end{pmatrix}. \end{aligned}$$

LDU decomposition can be applied directly to this matrix without any use of pivoting. For example, with $\mathbf{b} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$,

$$V\,\mathbf{b} \;=\; \begin{pmatrix} +.6483 & -.7614 \\ +.7614 & +.6483 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \end{pmatrix} \;=\; \begin{pmatrix} -.9870 \\ 3.468 \end{pmatrix}.$$

LU decomposition on $V\,M\,W$ followed by solving the pair of triangular systems $L\mathbf{z} = V\,\mathbf{b}$, $U\mathbf{y} = \mathbf{z}$, then gives

$$\mathbf{y} \;=\; \begin{pmatrix} 3.556 \\ -.6005 \end{pmatrix},$$

so the solution to $M\mathbf{x} = \mathbf{b}$ is

$$\mathbf{x} \;=\; W\,\mathbf{y} \;=\; \begin{pmatrix} 3.000 \\ 2.001 \end{pmatrix}$$

— correct to three digits.

To achieve the effect described above, the randomizing transform needs to accomplish two things:

1. The randomized matrix VMW must be nondegenerate (recursively nonsingular in its NWdiagonal quadrant).
2. The roundoff errors incurred in the randomization must be acceptable.

In [6] we describe a general fast unitary randomizing transform having the property that, for any nonsingular matrix M, the transform of M is nondegenerate with probability 1. Thus, with probability 1, LDU decomposition will succeed.

In [5] we further clarify properties of LDU decomposition (Gaussian Elimination), and investigate roundoff error. We show that the intermediate values $m_{ij}^{(k)}$ depend directly on the condition number of the leading $k \times k$ principle submatrix of M, clarifying the backwards error bounds mentioned earlier.

Ultimately we feel a statistical error analysis like that in [8] is important for testing any real variant of Gaussian Elimination. Backwards error analyses, useful though they are, give only partial certification of any numerical algorithm.

We believe that randomization actually has many uses quadtree algorithms:

1. Randomization can be used in padding to full size. Instead of padding with identity or zero matrices, we can pad with random matrices. This avoids the problems concerning padding mentioned earlier.
2. Generally, randomization yields dense matrices. Thus it avoids special cases (such as identity matrices) and the complexities they introduce. It trades away clever handling of sparse matrices to get simplicity.
3. Randomization eliminates degeneracy, as just shown.

All-in-all, a practical outlook on the use of quadtrees seems to lie in emphasizing randomized dense matrix computations, where block-matrix definitions work well.

We have discussed issues in using quadtrees to represent matrices in achieving practical algorithms based on recursive block-matrix definitions. Although the promise of quadtrees is enormous, this promise has not yet translated to success in practice. This paper has tried to summarize why, and to suggest ways that their promise can be realized.

We have shown that randomization can play an important role in quadtree-based algorithms. Generally, block matrix algorithms make assumptions about the blocks of the matrices being operated upon, and these assumptions may fail to hold for certain degenerate matrices. The importance of randomization stems from the fact that it can eliminate this degeneracy, with probability 1. With this in mind, quadtrees appear to have great potential in dense matrix computations.

It may also be possible to develop effective sparse matrix computations using quadtree-like representations. However these representations will most likely be more 'semantic' in nature, reflecting qualitative properties of the matrices, such as whether they are positive definite, stochastic, in some normal form, etc. This is an exciting but very nontrivial extension.

Currently we are interested in applying quadtree methods on distributed computers and array processors. See [4] for a discussion on array processors. Parallel implementations raise many interesting further issues, such as scheduling submatrix computations and communication techniques. Quadtrees offer some hope in achieving multiprocessing naturally, without excessive data movement among processors.

[1] J.R. Bunch and J. Hopcroft. Triangular factorization and inversion by fast matrix multiplication. *Math. of Computation*, 28(125):231–236, 1974.

[2] Gene H. Golub and Charles F. van Loan. *Matrix Computations (2nd. ed.)*. Johns Hopkins, 1989.

[3] Dinh Le. Solving matrix inversion using quadtrees. Technical report, UCLA Computer Science Department, 1992.

[4] Dinh Le, Milos Ercegovac, Tomas Lang, and Jaime Moreno. MAMACG: A tool for automatic mapping of matrix algorithms onto mesh array computational graphs. In Edward Lee Jose Fortes and Teresa Meng, editors, *Application Specific Array Processors*, pages 511–525. IEEE Computer Society, IEEE Computer Society Press, August 1992.

[5] D.S. Parker. Explicit Formulas for the Results of Gaussian Elimination. Technical Report CSD-950025, UCLA Computer Science Department, 1995.

[6] D.S. Parker. Random Butterfly Transformations with applications in Computational Linear Algebra. Technical Report CSD-950023, UCLA Computer Science Department, 1995.

[7] D.S. Parker and Dinh Le. How to Eliminate Pivoting from Gaussian Elimination — by Randomizing Instead. Technical Report CSD-950022, UCLA Computer Science Department, 1995.

[8] L.N. Trefethen and R.S. Schreiber. Average-case stability of gaussian elimination. *SIAM J. Matrix Anal. Appl.*, 11(3):335–360, July 1990.

[9] David S. Wise. Representing matrices as quadtrees for parallel processors. *Information Processing Letters*, pages 195–199, July 1984.

[10] David S. Wise. Representing matrices as quadtrees for parallel processors (extended abstract). *ACM SIGSAM Bulletin*, 3(18):24–25, August 1984.

[11] David S. Wise. Parallel decomposition of matrix inversion using quadtrees. In K. Hwang, S.J. Jacobs, and E.E. Swartzlander, editors, *Proc. 1986 International Conference on Parallel Processing*, pages 92–99. IEEE Computer Society Press, 1986.

[12] David S. Wise. Matrix algebra and applicative programming. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 134–153. Springer, 1987.

[13] David S. Wise and J. Franco. Costs of quadtree representation of non-dense matrices. Technical Report 229, Computer Science Department, Indiana University, October 1987.