Practical Recursive Block Decomposition Matrix Algorithms, and Quadtree Algorithms, via Randomization

Dinh Lê D. Stott Parker
dinh@cs.ucla.edu stott@cs.ucla.edu
UCLA Computer Science Department
Los Angeles, CA 90095-1596

Abstract

Recursive block decomposition algorithms (also known as *quadtree algorithms* when the blocks are all square) have been proposed to solve well-known problems such as matrix addition, multiplication, inversion, block LDU decomposition, discrete Fourier transform, and Cholesky factorization.

Until now, such algorithms have been seen as impractical, since they require leading submatrices of the input matrix to be nonsingular (which is rarely guaranteed). This requirement essentially has rendered functional programming impractical for matrix computations.

We show how to use *randomization* to guarantee that submatrices meet these requirements, and to make recursive block decomposition methods practical on well-conditioned input matrices. The resulting algorithms are elegant, and we show the recursive programs can perform well for both dense and sparse matrices, although with randomization dense computations seem most practical. Essentially, by 'homogenizing' the input, randomization provides a way to avoid degeneracy in numerical problems and permit simple functional programs to solve these problems.

1 Introduction

We have been investigating alternative computation schemes for large-scale matrix computations. A natural functional programming approach called *recursive block decomposition* (or *quadtree decomposition* when the blocks are all square) operates via divide-and-conquer recursion.

1.1 Recursive Block Decomposition Algorithms

For example, the following block-Gaussian elimination step can be used recursively:

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

Then assuming $\begin{pmatrix} A & B \\ C & D \end{pmatrix}$ is a nonsingular square matrix, and that A is nonsingular, its block LDU decomposition is:

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

Thus the determinant can be computed as:

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

Recursive block decomposition algorithms are often used in texts for natural presentation of basic operations like matrix sum, product, inverse, transforms like the Discrete Fourier Transform, and factorizations like Cholesky and QR [1, 9]. Our objective is to study what it would take to make these algorithms used more widely in practice.

1.2 The Quadtree Structure

In this paper we are particularly concerned about the case where A, B, C, D are **quadrants** of the matrix (blocks of equal size). In this case, the recursive style is particularly elegant, and naturally leads to the representation of square matrices as recursive quadtrees.

The heritage of the quadtree decomposition dates back to the very roots of matrix theory, in the quaternion theory of Hamilton from the mid-nineteenth century. Recently the use of quadtrees to represent matrices was popularized by David Wise [21, 22, 23, 24, 25]. The quadtree representation has many strengths: it is elegant, suitable for placement among the processors in a parallel computer, and also can handle sparse matrices in a uniform manner.

Unfortunately, the quadtree representation raises problems for serious implementors of matrix computations. First, naive implementations of quadtree algorithms for standard computations like matrix multiplication and Gaussian elimination are far from being competitive, in terms of speed, memory requirements, usability, and scalability, with their iterative counterparts like those in LAPACK [2] and ScaLAPACK [6] for uni- and multi-processors, respectively. Improved compilation technology is a minimum that is needed if naively-expressed quadtree algorithms are to be competitive.

Also, practical implementations of quadtree algorithms can be very complicated. For instance, in [10], one of us implemented Wise's quadtree matrix inversion algorithm. When pivoting is included, the complexity of the algorithm (in terms of the number of cases and code size) seems to be much higher than what is suggested in [23]. The quadtree structure seems ill-suited to extensions like pivoting.

Perhaps the most serious problem is that quadtree algorithms cannot always be used for all input matrices. In the block Gaussian elimination and LDU factorization above, for example, the leading principal submatrix A must be invertible. If this fails to hold, and it often does even when the whole matrix is invertible, the algorithm cannot be used. Block Gaussian elimination, for another, can be used only when pivoting is not needed, i.e., the leading principal submatrices of A are nonsingular [9, p.102].

1.3 The Role of Randomization

Recently, a *randomization technique* developed in [12, 13, 14, 15, 17] has been applied to recursive block decomposition matrix algorithms. The basic idea (which will be described below in more detail) is to transform an input matrix M to a 'randomized' matrix UMV, where U and V belong to a particular class of random matrices. If M is nonsingular, then the submatrices of the transformed matrix are guaranteed to be nonsingular. For example, with the LDU factorization above the leading principal submatrix A is guaranteed to be nonsingular after randomization.

The randomization technique makes it possible to avoid complications like pivoting that have obstructed adoption of quadtree algorithms in the past. Although there remain many challenges confronting the use of these algorithms, randomization opens a door to new parallel algorithms and matrix computation techniques.

1.4 Objectives

Our purpose is to study the use of functional programming in matrix computations (via quadtree and recursive block decomposition) using the randomization technique. We argue that randomization preserves the following qualities that make the functional style attractive in the first place:

1. simplicity and expressiveness;
2. consistency with fast divide-and-conquer algorithms, such as Strassen's matrix multiplication algorithm [18];
3. ease of translation to parallel code.

These qualities translate to important benefits. First, simplicity and expressiveness implies ease of programming. Second, utilization of fast divide-and-conquer algorithms permit better asymptotic complexities than possible with iterative algorithms, at the cost of potentially higher error growth. Strassen's algorithm, once considered impractical, is seeing increasing use with large matrices in real applications. Third, ease of translation to parallel code implies scalability on multiprocessors. To anyone who has tried to express matrix computations in existing parallel programming environments, this is a *great* benefit. For example, the ScaLAPACK Gaussian elimination code exceeds 2500 lines!

This paper begins by describing matrix computations with quadtrees. Matrix addition and multiplication are then shown in full detail. High level descriptions of other matrix algorithms — including matrix inversion, Gaussian elimination, discrete Fourier transformation, and Cholesky decomposition — are then presented. The problem of degeneracy in matrix inversion and Gaussian elimination is highlighted.

Next, the randomization technique developed in [12, 13, 14] is described and used to avoid this potential degeneracy. Experience with implementations of randomized matrix inversion and Gaussian elimination is then presented. Their performance and error properties are analyzed, showing effectiveness of the randomization technique.

The quadtree representation is certainly not the only possible representation for matrix decompositions. For example, in transformations related to the Fourier Transform, tensor (Kronecker) product decompositions can be more natural [20]. However, the quadtree is an important special case, and can even be used to implement tensor products. This paper argues the quadtree position from the new perspective of randomization.

2 Quadtrees

Wise's quadtree representations of matrices are very interesting, and raise issues going far beyond the scope of this paper. In [16] we discuss some of these issues in more detail. Here, we review essential aspects.

2.1 Quadtree Representations of Matrices

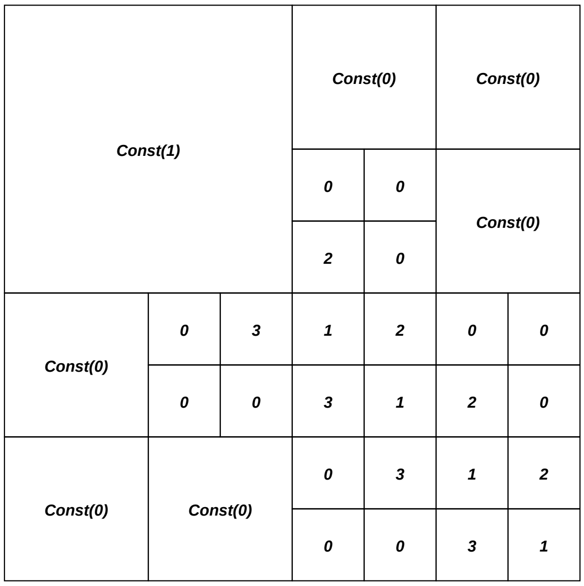
Here let us define a **quadtree** representation of a matrix to be either a constant diagonal matrix or a matrix that is composed of four equal-sized square submatrices. Let M be an $n \times n$ matrix, where n is a power of two. The canonical representation of M in quadtrees can be defined recursively:

$$M = \begin{cases} Const(v), \\ Quad(A_{11}, A_{12}, A_{21}, A_{22}) \end{cases}$$

where $Const(v)$ is a diagonal matrix of size $2^\ell \times 2^\ell$ whose diagonal entries are all v, $Quad(A_{11}, A_{12}, A_{21}, A_{22})$ is a matrix of four equal-sized submatrices A_{11}, A_{12}, A_{21}, and A_{22}. For example, the matrix M_e below

$$\begin{array}{l}
Quad(\,Const(1), \\
\quad Quad(\,Const(0), \\
\qquad Const(0), \\
\qquad Quad(\,Const(0),\ Const(0), \\
\qquad\quad Const(2),\ Const(0)), \\
\qquad Const(0)), \\
\quad Quad(\,Const(0), \\
\qquad Quad(\,Const(0),\ Const(3), \\
\qquad\quad Const(0),\ Const(0)), \\
\qquad Const(0), \\
\qquad Const(0)), \\
\quad Quad(\,Quad(\,Const(1),\ Const(2), \\
\qquad\quad Const(3),\ Const(1)), \\
\qquad Quad(\,Const(0),\ Const(0), \\
\qquad\quad Const(2),\ Const(0)), \\
\qquad Quad(\,Const(0),\ Const(3), \\
\qquad\quad Const(0),\ Const(0)), \\
\qquad Quad(\,Const(1),\ Const(2), \\
\qquad\quad Const(3),\ Const(1))))
\end{array}$$

can be represented pictorially as:



This is a sparse representation; Wise [25] has studied the space requirements of quadtree representations for a variety of sparse matrices.

2.2 Conversion To Quadtree Representations

The process of converting a $n \times n$ matrix M_n to a sparse representation M_c can be naturally implemented in 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* $2^\ell \times 2^\ell$ where $\ell = \lceil \log_2 n \rceil$.

Compacting can then be applied to this matrix to render it more sparse, replacing all-zero subquadrants with $Const(0)$, etc., and arriving ultimately at a quadtree matrix. It is not difficult to prove by induction on ℓ that there is a unique 'maximally compact' quadtree representation for any $2^\ell \times 2^\ell$ 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 an identity matrix to give the (dense) $2^3 \times 2^3$ quadtree 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 yield the sparse quadtree matrix M_e shown at the beginning of this section.

3 Recursive Block Decomposition Algorithms

This section reviews implementations of well-known block decomposition algorithms: matrix multiplication and inversion, Gaussian elimination, block LDU decomposition, discrete Fourier Transform, QR factorization, and Cholesky factorization. To ground the discussion, matrix multiplication is shown in detail. The rest of the algorithms are described at a high level of abstraction, but translation to detailed implementations is immediate. We use the ML programming language for implementation here, as it reflects the recursion naturally, but it could be replaced by virtually any recursive language.

The implementations here are complicated by the fact that matrices may have multiple quadtree representations [16]. For example, the identity matrix can be implemented as a single constant matrix (whose diagonal elements are all 1), or as a dense matrix of 0's and 1's. In this paper we consider only two cases (*Const* and *Quad*), but of course there are alternative representations. The benefit of multiple representations is that sparse matrices are neatly representable (and usable directly with dense matrices), and that treating sparseness specially has performance benefits.

3.1 Quadtree Data Structure

The quadtree data structure described in Section 2 can be defined in ML as a polymorphic datatype **qtree** that takes one argument α as follows:

$$
\begin{aligned}
&\mathsf{datatype}\quad \alpha\ \mathbf{qtree}\ =\\
&\quad\quad \mathit{Const}\ \mathsf{of}\ \alpha\\
&\quad\ \mid \mathit{Quad}\ \mathsf{of}\ \alpha\ \mathsf{qtree} \times \alpha\ \mathbf{qtree} \times \alpha\ \mathbf{qtree} \times \alpha\ \mathbf{qtree}
\end{aligned}
$$

Here α specifies an appropriate element type, such as **real**. $Const(v)$ denotes a $2^\ell \times 2^\ell$ diagonal matrix whose diagonal elements are all v, and $Quad(A_{11},A_{12},A_{21},A_{22})$ denotes a matrix of four equal-sized quadtree submatrices A_{11}, A_{12}, A_{21}, A_{22}. We will assume in all the programs that follow that input quadtrees are well-formed (submatrices are always of the same size).

3.2 Matrix Multiplication

Let $M_1, M_2 \in \mathbf{Mat}$ where **Mat** be the set of matrices represented with **qtree**. The function matmul takes quadtrees M_1 and M_2, and returns $M_1 \times M_2$:

$$\mathsf{matmul}\colon\ \mathbf{Mat} \times \mathbf{Mat} \to \mathbf{Mat}$$

$$
\begin{aligned}
&\mathsf{matmul}(\mathit{Const}(v),\ \mathit{Const}(w)) = \mathit{Const}(v \times w)\\
&\mathsf{matmul}(\mathit{Const}(v),\ \mathit{Quad}(A_{11},A_{12},A_{21},A_{22})) =\\
&\qquad \mathit{Quad}(\mathsf{matmul}(\mathit{Const}(v),A_{11}),\ \mathsf{matmul}(\mathit{Const}(v),A_{12}),\\
&\qquad\qquad \mathsf{matmul}(\mathit{Const}(v),A_{21}),\ \mathsf{matmul}(\mathit{Const}(v),A_{22}))\\
&\mathsf{matmul}(\mathit{Quad}(A_{11},A_{12},A_{21},A_{22}),\ \mathit{Const}(v)) =\\
&\qquad \mathit{Quad}(\mathsf{matmul}(A_{11},\mathit{Const}(v)),\ \mathsf{matmul}(A_{12},\mathit{Const}(v)),\\
&\qquad\qquad \mathsf{matmul}(A_{21},\mathit{Const}(v)),\ \mathsf{matmul}(A_{22},\mathit{Const}(v)))\\
&\mathsf{matmul}(\mathit{Quad}(A_{11},A_{12},A_{21},A_{22}),\ \mathit{Quad}(B_{11},B_{12},B_{21},B_{22})) =\\
&\qquad \mathit{Quad}(\mathsf{matadd}(\mathsf{matmul}(A_{11},\ B_{11}),\ \mathsf{matmul}(A_{12},\ B_{21})),\\
&\qquad\qquad \mathsf{matadd}(\mathsf{matmul}(A_{11},\ B_{12}),\ \mathsf{matmul}(A_{12},\ B_{22})),\\
&\qquad\qquad \mathsf{matadd}(\mathsf{matmul}(A_{21},\ B_{11}),\ \mathsf{matmul}(A_{22},\ B_{21})),\\
&\qquad\qquad \mathsf{matadd}(\mathsf{matmul}(A_{21},\ B_{12}),\ \mathsf{matmul}(A_{22},\ B_{22})))
\end{aligned}
$$

with similar but slightly simpler code for matadd.

The final case breaks the two matrices into four regions before multiplying them recursively. That is,

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

where

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

It is very significant that we can trivially substitute this standard computation with Strassen's method [18], simply by using the following:

$$\begin{aligned} M_1 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\ M_2 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_3 &= (A_{11} - A_{21})(B_{11} + B_{12}) \\ M_4 &= (A_{11} + A_{12})B_{22} & C_{11} &= M_1 + M_2 - M_4 + M_6 \\ M_5 &= A_{11}(B_{12} - B_{22}) & C_{12} &= M_4 + M_5 \\ M_6 &= A_{22}(B_{21} - B_{11}) & C_{21} &= M_6 + M_7 \\ M_7 &= (A_{21} + A_{22})B_{11} & C_{22} &= M_2 - M_3 + M_5 - M_7. \end{aligned}$$

Let $T_s(n)$ and $T_t(n)$ represent the timing functions for the Strassen and traditional multiplication algorithms, respectively. Then the coresponding recurrence relations are $T_s(n) = 7T_s(n/2) + 18(n/2)^2$, and $T_t(n) = 8T_t(n/2) + 4(n/2)^2$. As explained in [1, p.231], these yield asymptotic complexities of $O(n^{2.81})$ and $O(n^3)$ respectively.

3.3 Matrix Inversion

Faddeev & Faddeeva [8, pp.161–163] provide a recursive method of solving matrix inversion as follows:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} - A^{-1}BM & -A^{-1}BN \\ M & N \end{pmatrix},$$

where $N = (D - CA^{-1}B)^{-1}$ and $M = -NCA^{-1}$. This may be implemented in ML in the following way, using only seven multiplicative operations:

matinv: **Mat** $\rightarrow$ **Mat**

```
matinv(Const(v)) = Const(1/v)
matinv(Quad(A, B, C, D)) =
    let
        val A^-1 = matinv(A)
        val A^-1 B = matmul(A^-1, B)
        val CA^-1 = matmul(C, A^-1)
        val CA^-1 B = matmul(CA^-1, B)
        val N = matinv(matsub(D, CA^-1 B))
        val M = matneg(matmul(N, CA^-1))
        val L = matneg(matmul(A^-1 B, N))
        val K = matsub(A^-1, matmul(A^-1 B, M))
    in
        Quad(K,L,M,N)
    end
```

This algorithm requires both A and $(D - CA^{-1}B)$ to be invertible (recursively). Assuming this is guaranteed, and Strassen's matrix multiplication algorithm is used, then the complexity for the combined Faddeev/Strassen algorithm is $T(n) = 2T(n/2) + 6(n/2)^{2.81} + 4(n/2)^2 = O(n^{2.81})$.

3.4 Gaussian elimination

Like the block LU approach described in [9, pp.100–101], a recursive Gaussian elimination algorithm (without pivoting) follows from the LU decomposition equation

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \mathbf{0} \\ L_{21} & I_{n/2} \end{pmatrix} \begin{pmatrix} I_{n/2} & \mathbf{0} \\ \mathbf{0} & \tilde{A} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ \mathbf{0} & I_{n/2} \end{pmatrix}$$

and can be implemented as follows:

GE: **Mat** $\rightarrow$ **Mat** $\times$ **Mat**

```
GE(Const(v)) = (Const(1), Const(v) )
GE(Quad(A11, A12, A21, A22)) =
    let
        val (L11, U11) = GE(A11)
        val U12 = matmul(matinv(L11), A12)
        val L21 = matmul(A21, matinv(U11))
        val (L22, U22) = GE(matsub(A22, matmul(L21 U12)))
    in
        (Quad(L11, Const(0), L21, L22),
         Quad(U11, U12, Const(0), U22))
    end
```

The GE algorithm requires the submatrix A to be invertible, which is not guaranteed in general.

3.5 Discrete Fourier Transform

Let $n = 2^p$ for some integer $p > 0$, $\omega_n = exp(-2\pi i/n)$ be a principal n^{th} root of unity, $\mathbf{x} = [x_0, \ldots, x_{n-1}]^T$, and F_n be the $n \times n$ matrix $F_n = (\omega_n^{ij})$, $0 \le i, j \le n-1$.

The vector $F(\mathbf{x}) = F\mathbf{x}$ with i^{th} component $\sum_{j=0}^{n-1} \omega_n^{ij} x_j$ is called the **Discrete Fourier transform** of $\mathbf{x}$ [20]. The DFT can be computed efficiently with the following decomposition:

$$\begin{aligned} F_n &= B_n \left(\begin{array}{c|c} F_{n/2} & 0 \\ \hline 0 & F_{n/2} \end{array}\right) {\sigma_n}^{-1} \\ &= \left(\begin{array}{c|c} (\omega_n^{i(2j)}) & (\omega_n^{i(2j+1)}) \\ \hline (\omega_n^{(i+n/2)(2j)}) & (\omega_n^{(i+n/2)(2j+1)}) \end{array}\right) {\sigma_n}^{-1} \end{aligned}$$

where the 'butterfly' matrix B_n is defined by

$$B_n = \left(\begin{array}{c|c} I_{n/2} & D_{n/2} \\ \hline I_{n/2} & -D_{n/2} \end{array}\right)$$

$$D_{n/2} = \begin{pmatrix} 1 & & & & \\ & \omega_n & & & \\ & & {\omega_n}^2 & & \\ & & & \ddots & \\ & & & & {\omega_n}^{n/2-1} \end{pmatrix}$$

and the 'perfect unshuffle' ${\sigma_n}^{-1}$ is a permutation matrix:

$${\sigma_n}^{-1} = \left(\begin{array}{cccccccc} 1 & & & & & & & \\ & & 1 & & & & & \\ & & & & \ddots & & & \\ & & & & & & 1 & \\ \hline & 1 & & & & & & \\ & & & 1 & & & & \\ & & & & & \ddots & & \\ & & & & & & & 1 \end{array}\right),$$

Quadtrees thus give a natural divide-and-conquer algorithm — the FFT.

Generally speaking, quadtrees are effective at representing transformations that are naturally expressed with tensor (Kronecker product) notation [20]. For example, the DFT can be expressed

$$F_n \;=\; B_n\ (I_2 \otimes F_{n/2})\ {\sigma_n}^{-1}$$

where '$\otimes$' is the **tensor** or **Kronecker product**, which for $m \times m$ and $n \times n$ matrices A and B is the $(mn) \times (mn)$ matrix defined by

$$A \otimes B \;=\; \left(\begin{array}{c|c|c|c} a_{11}B & a_{12}B & \cdots & a_{1m}B \\ \hline a_{21}B & a_{22}B & \cdots & a_{2m}B \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline a_{m1}B & a_{m2}B & \cdots & a_{mm}B \end{array}\right).$$

Quadtrees easily implement the tensor product recursively. An ML implementation is as follows:

tensor: $\mathbf{Mat} \times \mathbf{Mat} \to \mathbf{Mat}$

tensor($Const(v)$, $Const(w)$) = $Const(v \times w)$
tensor($Const(v)$, $Quad(A_{11}, A_{12}, A_{21}, A_{22})$) =
 $Quad$(tensor($Const(v)$, A_{11}), tensor($Const(v)$, A_{12}),
 tensor($Const(v)$, A_{21}), tensor($Const(v)$, A_{22}))
tensor($Quad(A_{11}, A_{12}, A_{21}, A_{22})$, B) =
 $Quad$(tensor(A_{11}, B), tensor(A_{12}, B),
 tensor(A_{21}, B), tensor(A_{22}, B))

3.6 Some Other Matrix Algorithms

Other suitable block decomposition algorithms using the quadtree representation include determinant (shown earlier), matrix transpose, and Cholesky and QR factorizations:

Cholesky Factorization

If a square matrix M is symmetric positive definite, then there exists a unique lower triangular matrix L with positive diagonal entries such that $M = LL^T$.

$$\begin{aligned} M \;&=\; \begin{pmatrix} A & B^T \\ B & D \end{pmatrix} = \begin{pmatrix} XX^T & XY^T \\ YX^T & YY^T + ZZ^T \end{pmatrix} \\ &=\; \begin{pmatrix} X & \mathbf{0} \\ Y & Z \end{pmatrix} \begin{pmatrix} X^T & Y^T \\ \mathbf{0} & Z^T \end{pmatrix}. \end{aligned}$$

If '$\sqrt{}$' denotes Cholesky factorization, then

$$\sqrt{M} \;=\; \begin{pmatrix} \sqrt{A} & \mathbf{0} \\ Y & \sqrt{D - YY^T} \end{pmatrix}$$

where $Y = B(\sqrt{A}^T)^{-1}$.

QR Factorization

If M is a symmetric positive definite matrix, then M can be written uniquely in the form $M = QR$, where Q is orthogonal and R is upper triangular with positive diagonal elements. Specifically, Q and R can be obtained using the Cholesky factorization as follows:

$$\begin{aligned} R \;&=\; (\sqrt{M})^T \\ Q \;&=\; MR^{-1}. \end{aligned}$$

3.7 At Issue: Guaranteeing Nondegeneracy

Many of the algorithms above work only with the guarantee that submatrices are nondegenerate (recursively nonsingular). When the input matrix M is symmetric positive definite this guarantee will be met, for example. In general, of course, it is not.

There are several old tricks that work in the important case where M is nonsingular but nothing specific is known about its submatrices. Bunch and Hopcroft [5] mention two: they point out that the rows of such M can always be permuted so that the nonsingularity requirement needed for Gaussian elimination is met. Unfortunately there is no way of knowing the permutation *a priori* other than, for example, performing Gaussian elimination. Bunch and Hopcroft also 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:

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

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. Since most matrix computations depend in one way or another on the matrix condition number, the increase in rounding error resulting from this transformation can be enormous.

4 Randomized Matrix Algorithms

We have discussed issues in using quadtrees to achieve 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.

Many of the problems with quadtrees raised above stem from the fact that invertible matrices can have noninvertible submatrices. Although such matrices arise often in practice, they are 'insignificant' in theory, in the sense that they have measure zero in the space of all matrices. Our idea now is to exploit their insignificance through the fact that randomization of an invertible matrix can eliminate any degenerate submatrices.

The randomization technique is particularly useful in eliminating the possibility of singular quadrants. It randomizes $M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$ to $M' = \begin{pmatrix} A' & B' \\ C' & D' \end{pmatrix}$, and if M is nonsingular then all blocks of M' are nonsingular (with probability 1). In particular, $(A')^{-1}$ exists. The randomized matrix M' can thus be used with block-matrix definitions that assume nonsingularity, and derandomization is possible whenever necessary.

4.1 The Randomization Technique

The randomization technique is described in detail in [12, 13, 14, 17]. An overview is also available in [15]. Here we give a brief summary.

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

Specifically, suppose M is a nonsingular real $n \times n$ matrix, let U and V 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 [14].) Then:

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

  $$det\,(UMV)/(det\,U \;\; det\,V)$$

  when $det\,U$ and $det\,V$ can be found easily.

Each of the computations here is easily implemented if Gaussian elimination (without pivoting) successfully factors the preconditioned matrix UMV.

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

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

we get the random transform

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

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

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

In other words, we can perform the randomization

$$\boxed{\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \mathbf{x} \;=\; \begin{pmatrix} 2 \\ 3 \end{pmatrix}}$$

$$\downarrow$$

$$\boxed{\begin{pmatrix} .1097 & .9940 \\ .9940 & .1097 \end{pmatrix} \mathbf{y} \;=\; \begin{pmatrix} -.9870 \\ 3.468 \end{pmatrix}}$$

where $\mathbf{x} = V\mathbf{y}$, and solve the second system directly. Gaussian elimination on $U\,M\,V$ followed by solving the pair of triangular systems $L\mathbf{z} = U\,\mathbf{b}$, $U\mathbf{y} = \mathbf{z}$, then gives

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

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

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

— correct to three digits.

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

1. The randomized matrix UMV must be recursively nonsingular in its first (NW) quadrant.
2. The roundoff errors incurred in the randomization must be acceptable.

In [14] we describe a general fast unitary randomizing transform having the property that, for any nonsingular matrix M, the transform of M satisfies the first condition with probability 1. Thus, with probability 1, Gaussian elimination will succeed. In [12] we consider related transforms that succeed for quadtree decompositions. Furthermore, in these papers and in [13] we study analytic properties of Gaussian elimination, and investigate its roundoff error. We show that the intermediate values $m_{ij}^{(k)}$ depend formally on the condition number of the leading $k \times k$ principle submatrix of M.

The randomized algorithms do generate greater roundoff errors than their traditional counterparts with pivoting. However, the errors appear acceptable with well-conditioned input matrices [12, 14]. Below we give some results for implementations of several algorithms. Ultimately a statistical error analysis like that in [19] is needed to test any real variant of Gaussian elimination. In fact, although Gaussian elimination with partial pivoting has been believed 'stable in practice' for several decades, recently a number of papers have appeared pointing out that this belief is incorrect for commonly-encountered matrices, including important orthogonal matrices; see [13].

4.2 Randomized Matrix Inversion

Recall Faddeev's matrix inversion scheme mentioned earlier:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} - A^{-1}BM & -A^{-1}BN \\ M & N \end{pmatrix},$$

where $N = (D - CA^{-1}B)^{-1}$ and $M = -NCA^{-1}$. The randomization technique will guarantee that A and $(D - CA^{-1}B)$ are invertible. Thus, if Strassen's matrix multiplication algorithm is used, this algorithm has complexity $O(n^{2.81})$.

In order to be competitive with LAPACK[1], the Faddeev routine was rewritten in C. The timings and errors of Faddeev matrix inversion and LAPACK's matrix inversion using

[1] "LAPACK is a library of Fortran 77 subroutines for solving the most commonly occuring problems in numerical linear algebra. It has been designed to be efficient on a wide range of modern performance computers. The name LAPACK is an acronym for Linear Algebra PACKage." [2, p.3]

partial pivoting[2] on normally distributed random matrices are shown in Tables 1 and 2. All results in this section were obtained on a 80Mhz Intel 486 PC clone running Linux with 32Mb RAM and 64Mb swap space.

n	64	128	256	512
RFFT	1.4	4.1	9.3	25.8
Faddeev	1.3	9.9	72.9	530.1
LAPACK	1.3	10.8	87.7	711.7

Table 1: Timings from executing Faddeev and LAPACK matrix inversion on normally distributed random matrices

The point of using matrices with normally-distributed entries is that they are well-conditioned [7] and almost certainly have no degenerate submatrices. Thus unrandomized algorithms will succeed, and they can be compared directly with our randomized version to get an appreciation for the timing overhead and error properties of randomization itself.

Let $T_f(n)$ denote the timing function of the Faddeev inversion and $T_g(n)$ denote the timing function of the GE inversion. Then Table 1 indicates that $T_g(n)/T_g(n/2) \approx 8.0$; whereas $T_f(128)/T_f(64) \approx 7.62$, $T_f(256)/T_f(128) \approx 7.36$, and $T_f(512)/T_f(256) \approx 7.27$ — converging to the scaling factor 7. Thus we expect $T_f(1024)$ to take approximately $7.27T_f(512) = 3850$ seconds and $T_g(1024)$ to take approximately $8.0T_g(512) = 5700$ seconds. The RFFT complexity is $O(n^2 \log n)$ and Table 1 shows its low timing impact on the overall computation of $T_g(n)$.

n	κ	err	$\bar{x}_g$	H_g	L_g	$\bar{x}_p$	H_p	L_p	$\bar{x}_r$	H_r	L_r
64	.14D4	.25D−12	13.9	15	12	13.0	14	11	12.9	14	11
128	.73D4	.20D−11	13.0	15	11	11.7	13	9	10.8	13	8
256	.15D5	.30D−11	12.7	15	10	10.2	13	7	9.0	11	6
512	.42D5	.32D−10	12.1	14	9	8.6	12	5	7.6	10	4

Table 2: Errors from executing LAPACK, plain Faddeev, and randomized Faddeev matrix inversion on normally distributed random matrices

Table 2 shows the average errors incurred by executing LAPACK, plain and randomized Faddeev matrix inversion on fifteen (0,1)-normally distributed random matrices with double-precision real computation. Here κ denotes the matrix condition number, err the matrix relative lower bound, and $\bar{x}/H/L$ the average/average highest/average lowest digits of accuracy over all runs for LAPACK, plain Faddeev, and randomized Faddeev, respectively. This table shows that computational speed-up is countered by error growth. As the rank n increases from 64 to 512 the average digits of accuracy decrease from 13.9 to 12.1, 13.0 to 8.6, and 12.9 to 7.6 for the LAPACK, plain and randomized Faddeev inversions, respectively.

[2] Using Gaussian elimination with partial pivoting, the matrix A can be decomposed into PLU where P is the pivoting permutation, L is unit lower triangular, and U is upper triangular. The inversion A^{-1} is performed by straightforwardly computing $U^{-1}L^{-1}P^{-1}$.

4.3 Randomized Gaussian Elimination

Recall the recursive Gaussian elimination algorithm (without pivoting) developed earlier from the LU decomposition equation

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \mathbf{0} \\ L_{21} & I_{n/2} \end{pmatrix} \begin{pmatrix} I_{n/2} & \mathbf{0} \\ \mathbf{0} & \tilde{A} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ \mathbf{0} & I_{n/2} \end{pmatrix}.$$

When the input is randomized, the matrix A_{11} is almost certainly invertible. For comparison with LAPACK, the recursive GE routine shown earlier in ML was rewritten in C. Experimentation (Table 3) shows that the recursive method is about 10% slower than block LU without pivoting if plain matrix multiplication, instead of Strassen matrix multiplication, is used.

n	**128**	**256**	**512**	**1024**
Recursive GE	3.5	28.5	232.9	1887.3
Block GE	3.2	26.3	209.4	1712.9

Table 3: Timings from executing recursive GE without pivoting vs. block GE without pivoting

The computations $U_{12} \leftarrow L_{11}^{-1}A_{12}$ and $L_{21} \leftarrow A_{21}U_{11}^{-1}$ required by the GE recursion are mutually independent, so they can be executed in parallel. Table 4 shows an estimate of 30% speed-up if these two mutually independent computations are run in parallel (as obtained by omitting the simpler of the two in our tests).

n	**128**	**256**	**512**	**1024**
one-processor	3.5	28.5	232.9	1887.3
two-processor	2.7	21.7	174.2	1393.5

Table 4: Timings from executing recursive GE without pivoting: one vs. two processors

The inversion of lower triangular and upper triangular matrices can now be determined recursively by:

$$\begin{aligned} \begin{pmatrix} A_{11} & \mathbf{0} \\ A_{21} & A_{22} \end{pmatrix}^{-1} &= \begin{pmatrix} A_{11}^{-1} & \mathbf{0} \\ -A_{22}^{-1}A_{21}A_{11}^{-1} & A_{22}^{-1} \end{pmatrix} \\ \begin{pmatrix} A_{11} & A_{12} \\ \mathbf{0} & A_{22} \end{pmatrix}^{-1} &= \begin{pmatrix} A_{11}^{-1} & -A_{11}^{-1}A_{12}A_{22}^{-1} \\ \mathbf{0} & A_{22}^{-1} \end{pmatrix}. \end{aligned}$$

Assuming Strassen multiplication is used, then the complexity for both upper and lower unit triangular matrix inversions is $T_i(n) = 2T_i(n/2) + 2n^{2.81} = O(n^{2.81})$. Similarly, the products

$$\begin{aligned} \begin{pmatrix} L_{11} & \mathbf{0} \\ L_{21} & L_{22} \end{pmatrix}\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} &= \begin{pmatrix} L_{11}A_{11} & L_{11}A_{12} \\ L_{21}A_{11}+L_{22}A_{21} & L_{21}A_{12}+L_{22}A_{22} \end{pmatrix} \\ \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}\begin{pmatrix} U_{11} & U_{12} \\ \mathbf{0} & U_{22} \end{pmatrix} &= \begin{pmatrix} A_{11}U_{11} & A_{11}U_{12}+A_{12}U_{22} \\ A_{21}U_{11} & A_{21}U_{12}+A_{22}U_{22} \end{pmatrix} \end{aligned}$$

can be computed in time $T_m(n) = 6n^{2.81} + 2(n/2)^2$, assuming that Strassen multiplication is used. Thus, the complexity for the computations $U_{12} \leftarrow L_{11}^{-1}A_{12}$ and $L_{21} \leftarrow A_{21}U_{11}^{-1}$ is $T(n) = T_i(n) + T_m(n) = O(n^{2.81})$. Tables 5 and 6 show

n	128	256	512	1024
RFFT	4.1	9.3	25.8	68.3
Strassen LU	3.4	27.4	212.3	1600.1
LAPACK's LU	3.6	29.3	231.9	1887.4

Table 5: Timings from executing recursive LU without pivoting using Strassen multiplication vs. LAPACK's LU with partial pivoting on normally distributed random matrices

the timings and errors for (0,1)-normally distributed random matrices for recursive LU without pivoting using Strassen multiplication versus LAPACK's LU with partial pivoting.

Let $T_s(n)$ denote the timing function of recursive LU without pivoting using Strassen multiplication and $T_l(n)$ the timing function of LAPACK's LU with partial pivoting. In Table 5 one can determine that $T_l(n)/T_l(n/2) \approx 8.0$; whereas with Strassen multiplication $T_s(256)/T_s(128) = 8.1$, $T_s(512)/T_s(256) = 7.75$, and $T_s(1024)/T_s(512) = 7.53$, converging to the scaling factor 7. Thus, we expect $T_l(2048)$ to take approximinately $8.0 \times T_l(1024) = 15000$ seconds and $T_s(2048)$ to take approximately $7.5 \times T_s(1024) = 12000$ seconds. The complexity for the RFFT is $O(n^2 \log n)$ and Table 5 again shows only low timing impact on the overall computation of $T_s(n)$.

n	κ	err	$\bar{x}_l$	H_l	L_l	$\bar{x}_p$	H_p	L_p	$\bar{x}_r$	H_r	L_r
128	.49D4	.14D−11	13.5	15	11	11.8	15	9	11.3	13	8
256	.14D5	.49D−11	13.0	15	10	11.0	14	7	10.4	12	7
512	.78D5	.13D−10	12.5	15	9	9.5	14	6	9.3	12	6
1024	.80D5	.35D−10	12.1	15	9	8.6	13	4	8.2	11	4

Table 6: Errors from executing plain and randomized recursive LU without pivoting using Strassen multiplication vs. LAPACK's LU with partial pivoting, on normally distributed random matrices

Analogous to Table 2, Table 6 compares LAPACK's LU with partial pivoting and plain and randomized recursive LU without pivoting using Strassen multiplication. Fifteen runs were made on (0,1)-normally distributed random input matrices with double-precision real computations. Again, these numbers show that computational speed-up is countered by error growth. As the rank n increases from 128 to 1024 the average digits of accuracy decrease from 13.5 to 12.1, 11.8 to 8.6, and 11.3 to 8.2 for LAPACK, plain and randomized recursive LU decomposition, respectively.

We should stress that the quadtree/Faddeev/Strassen approach to matrix inversion is not new. For example, in [3] a general Strassen multiplication is derived to solve LU factorization on a Cray-YMP. Also, in [4], a more thorough analysis on the stability of a Strassen-type matrix inversion algorithm is described. Our contribution is to point out that randomization makes the recursive block decomposition and quadtree approaches practical for a broader class of nonsingular matrices than has been thought possible in the past.

5 Conclusion

The elegance of functional programming often cannot be exploited in practice because 'real' problems exhibit quirks and peculiarities that frustrate general recursive definitions. This is particularly true in matrix computations, where mild degeneracy can prevent a problem from being solved with a natural functional style.

Matrix computations represent an important application area in which functional programming can excel. As matrix computations grow in size, FLOPS fall in cost, and word lengths get longer, interest in new matrix algorithms will inevitably increase. Although Strassen's algorithm was initially dismissed as impractical, it has recently seen growing use because it runs faster for large n than other algorithms (thanks to its $O(n^{2.81})$ complexity).

This paper has explored the use of a new randomization technique in making recursive block decomposition and quadtree algorithms practical. We have reviewed the potentials and pitfalls that the quadtree representation of matrices provides in implementing recursive block decomposition algorithms. The quadtree representation is not the only representation possible for matrix decompositions, but it is very natural, and for example we have shown that even tensor product decompositions [20] can be implemented directly with quadtrees.

In spite of the fact that many important matrix computations can be expressed naturally with quadtree decompositions, they have not been adopted in mainstream implementations. A key reason for quadtrees not gaining popularity in practice appears to be that degenerate input matrices (matrices with singular block submatrices) can render the functional approach useless.

We showed how the randomization technique developed in [12, 13, 14, 15, 17] can avoid degeneracy. We have successfully incorporated the randomization technique to avoid degeneracy in solving block decomposition matrix inversion and Gaussian elimination with quadtrees. In particular, this gives a choice between two basic approaches to Gaussian elimination:

Gaussian elimination		*Degeneracy handling*	
Standard	$O(n^3)$	Partial pivoting	$O(n^2)$
Quadtree	$O(n^{2.81})$	RFFT	$O(n^2 \log n)$

Our initial discussion argued three qualities of recursive block decomposition (especially quadtree) algorithms:

1. simplicity and expressiveness;
2. consistency (interoperability) with known low-complexity divide-and-conquer algorithms, such as Strassen's algorithm and the FFT;
3. ease of translation to parallel code.

We have demonstrated the first two. Block decomposition codes using quadtrees are simple and expressive and generally have better computational complexity than their iterative counterparts, at the cost of somewhat higher error growth on well-conditioned input matrices (normally-distributed matrices are known to be well-conditioned [7]).

Because recursive block decomposition algorithms naturally divide computations in ways that reflect the architecture of many multiprocessors, particularly hypercube processors, ease of translation of these algorithms to parallel programs and efficient data movement among processors appear to be tractable. However, this remains to be demonstrated.

The ML programs shown earlier can do much more sophisticated handling of sparse matrices; for example, multiplication by identity matrices can be handled specially. Generally, however, randomization voids special handling of sparse matrices (such as identity matrices) and the tradeoffs they present. Randomization yields dense matrices, and trades away clever handling for simplicity.

Actually we believe that randomization has multiple uses in quadtree algorithms [16]:

1. Randomization eliminates degeneracy. Recursive block decomposition algorithms make assumptions about the submatrices being operated upon, and these assumptions fail to hold for degenerate matrices. The importance of randomization stems from the fact that it can eliminate this degeneracy, with probability 1.

2. When the quadtree/randomization approach is adopted, fast algorithms like Strassen's matrix multiplication and Faddeev matrix inversion can also be naturally adopted. This is especially important for computations involving very large matrices, where the improvement in complexity is significant.

3. Randomization can also be performed recursively. That is, randomization can be interleaved with the recursive computations described above. This was actually done in [12], and appears to yield somewhat better numerical accuracy than the implementations in the previous section. Further improvements in accuracy result from scaling (equilibration) and other numerical tricks for avoiding cancellation error.

4. Randomization can also be used in padding to full size. Instead of padding with identity or zero matrices, we can pad with random matrices. This avoids several problems concerning padding [16].

5. Because randomization eliminates conditional computations such as pivoting, and as shown here permits modular evaluation of matrix expressions by recursive decomposition, it seems a natural tool for developing systolic systems. We discuss this further in [11].

All in all, a practical direction for the use of quadtrees and randomization seems to lie in dense matrix computations, particularly large-scale computations, where block-matrix definitions work well.

Acknowledgement

We are grateful to Brad Pierce for many helpful discussions, and to Jack Carlyle, Tony Chan, and Milos Ercegovac for a number of suggestions that have improved this paper.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Computer Science and Information Processing. Addison-Wesley, 1974.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorenson. *LAPACK User's Guide*. Siam, second edition, 1995.

[3] D.H. Bailey, K. Lee, and H.D. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *Journal of Supercomputing*, 4(4):357–71, Jan. 1991.

[4] S.M. Balle and P.C. Hansen. *A Strassen-type matrix inversion algorithm*. Advances in Parallel Algorithms. Amsterdam, Netherlands: IOS Press, 1994.

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

[6] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. LAPACK Working Note 95 — ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers — Design Issues and Performance. Technical report, University of Tennessee, 1995.

[7] A. Edelman. Eigenvalues and condition numbers of random matrices. *SIAM J. Matrix Anal. Appl.*, 9(4):543–560, 1988.

[8] D.K. Faddeev and V.N. Faddeeva. *Computational Methods of Linear Algebra*. San Francisco, W.H. Freeman, 1963.

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

[10] Dinh Lê. Solving Matrix Inversion Using Quadtree. Technical report, University of California, Los Angeles, 1992.

[11] Dinh Lê, D.S. Parker, and Brad Pierce. Input randomization and automatic derivation of systolic arrays for matrix computations. Technical Report CSD-950038, UCLA Computer Science Department, 1995. (http://www.cs.ucla.edu/csd-lanai/fweb/stott/ge/).

[12] D.S. Parker. A Randomizing Butterfly Transformation Useful in Block Matrix Computations. Technical Report CSD-950024, UCLA Computer Science Department, 1995. (http://www.cs.ucla.edu/csd-lanai/fweb/stott/ge/).

[13] D.S. Parker. Explicit Formulas for the Results of Gaussian Elimination. Technical Report CSD-950025, UCLA Computer Science Department, 1995. (http://www.cs.ucla.edu/csd-lanai/fweb/stott/ge/).

[14] D.S. Parker. Random Butterfly Transformations with Applications in Computational Linear Algebra. Technical Report CSD-950023, UCLA Computer Science Department, 1995. (http://www.cs.ucla.edu/csd-lanai/fweb/stott/ge/).

[15] D.S. Parker and Dinh Lê. How to Eliminate Pivoting from Gaussian Elimination — by Randomizing Instead. Technical Report CSD-950022, UCLA Computer Science Department, 1995. (http://www.cs.ucla.edu/csd-lanai/fweb/stott/ge/).

[16] D.S. Parker and Dinh Lê. Quadtree Matrix Algorithms Revisited: Basic Issues and their Resolution. Technical Report CSD-950028, UCLA Computer Science Department, 1995. (http://www.cs.ucla.edu/csd-lanai/fweb/stott/ge/).

[17] D.S. Parker and Brad Pierce. The randomizing FFT: an alternative to pivoting in Gaussian elimination. Technical Report CSD-950037, UCLA Computer Science Department, 1995. (http://www.cs.ucla.edu/csd-lanai/fweb/stott/ge/).

[18] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

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

[20] Charles F. van Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM, 1992.

[21] David S. Wise. Representing Matrices as Quadtrees for Parallel Processors. *Information Processing Letters*, pages 195–199, July 1984.

[22] David S. Wise. Representing Matrices as Quadtrees for Parallel Processors (extended abstract). *ACM SIGSAM Bulletin*, 3(18):24–25, August 1984.

[23] 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, Washington, 1986.

[24] 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-Verlag, Berlin, 1987.

[25] David S. Wise and J. Franco. Costs of Quadtree Representation of Non-dense Matrices. Technical Report 229, Computer Science Department, Indiana University, October 1987.