J. Functional Programming **9** (6): 605–624, November 1999. Printed in the United Kingdom

Using randomization to make recursive matrix algorithms practical

DINH LÊ and D. STOTT PARKER

Computer Science Department, University of California, Los Angeles, CA 90095-1596, USA
(*e-mail:* stott@cs.ucla.edu)

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, determinant computation, block LDU decomposition and Cholesky and QR factorization. Until now, such algorithms have been seen as impractical, since they require leading submatrices of the input matrix to be invertible (which is rarely guaranteed). We show how to randomize an input matrix 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. By 'homogenizing' the input, randomization provides a way to avoid degeneracy in numerical problems that permits simple recursive quadtree algorithms to solve these problems.

Capsule Review

Block recursive algorithms for matrix operations hold a fascination for functional programmers and numerically-oriented computional scientists alike. Their recursive structure is very naturally expressible in functional languages, this expression lends itself straightforwardly to decomposition on parallel computers, and in certain cases the asymptotic complexity of the algorithms beats traditional schemes. Practical application has been hampered by matrices with zero entries, or other kinds of 'degeneracy'. Recursive algorithms for inversion, for example, can break down on matrices with zeroes on the diagonal. Lê and Parker show that a novel application of randomization can go a long way towards solving these problems. The randomized algorithms are still easily expressible in functional style, they have performance similar to (or even better than) traditional algorithms with pivoting, and their level of numerical accuracy should be acceptable for many applications.

1 Introduction

We have investigated 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.

https://doi.org/10.1017/S0956796899003470 Published online by Cambridge University Press

1.1 Recursive block decomposition algorithms

The basic idea here is that when a matrix is decomposed into smaller blocks, many useful functions of the matrix can be computed recursively. A natural question is whether recursive programming can play a practical role in numerical computation, although today most numerical algorithms are programmed iteratively.

A central example is LDU factorization of a square matrix. the block-Gaussian elimination step

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

can be used to give the block LDU decomposition

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

assuming A^{-1} is defined. When applied recursively, decompositions like this give natural (and often asymptotically faster) algorithms for matrix transpose, sum, product, inverse, generalized inverse, Schur complement ($D - CA^{-1}B$), determinant, norm, unitary transforms, SVD, and factorizations like QR and Cholesky (Aho *et al.*, 1974; Golub and van Loan, 1989).

1.2 Problems facing quadtree algorithms

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 (1985, 1986, 1987, 1992). The quadtree representation has many strengths: it is elegant, can be mapped naturally to various memory and machine architectures, and also can often handle sparse matrices in a uniform manner.

Unfortunately, general-purpose implementations of quadtree algorithms can be very complicated. For instance, when pivoting is included the complexity of Wise's matrix inversion algorithm (in terms of the number of cases and code size) seems much higher than that which is suggested by Wise (1986). The quadtree structure seems ill-suited to extensions like pivoting.

Also, the quadtree representation raises many challenges for serious implementors of matrix computations. First, naive quadtree algorithms for standard computations like matrix multiplication and Gaussian elimination tend not to be competitive, in terms of speed, memory requirements, and input restrictions, with their iterative counterparts such as those in LAPACK (Anderson *et al.*, 1995) and ScaLAPACK (Choi *et al.*, 1996) for uni- and multi-processors, respectively. Although their memory overhead may not be significant on many machines, and the quadtree algorithms

may be asymptotically faster, implementors must weigh practical considerations carefully.

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 invertible (Golub and van Loan, 1989).

1.3 The role of randomization

Recently, a *randomization technique* developed by Parker (1995*a, b, c*), Parket and Lê (1995*a*) and Parker and Pierce (1995) 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 U^*MV, where U and V belong to a particular class of random matrices. If M is invertible, then the submatrices of the transformed matrix are guaranteed to be invertible. For example, with the LDU factorization above the leading principal submatrix A is guaranteed to be invertible after randomization.

The randomization technique makes it possible to avoid complications like pivoting that have obstructed adoption of quadtree algorithms in the past. Randomization is not a panacea, but it opens a door to new 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 (Strassen, 1969).

These qualities translate to important benefits. First, simplicity and expressiveness implies ease of programming. To anyone who has tried to express matrix computations in existing multiprocessor programming environments, this is a *great* benefit. For example, the ScaLAPACK Gaussian elimination code exceeds 2500 lines! 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.

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, QR and Cholesky decomposition – are then presented. In these problems the problem of degeneracy (matrices whose first quadrant is noninvertible) is highlighted.

Next, the randomization technique developed by Parker (1995*a, b, c*) 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 (van Loan, 1992). 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. We discuss some of these issues in more detail elsewhere (Parker and Lê, 1995*b*). 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:

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

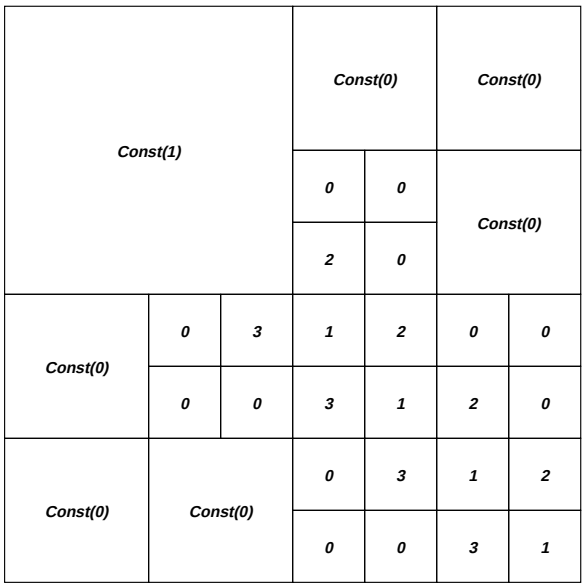
where $Const(v)$ is a diagonal matrix of size $2^\ell \times 2^\ell$ whose diagonal entries are all v, $Quad(A_{11},A_{12},A_{21},A_{22})$ is a matrix of four equal-sized submatrices A_{11}, A_{12}, A_{21}, and A_{22}. For example, the matrix M_e below

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

https://doi.org/10.1017/S0956796899003470 Published online by Cambridge University Press

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

can be represented pictorially as:



In this paper we consider only two constructors, *Const* and *Quad*, but of course there are alternative representations. This is a sparse representation; Wise (1992) 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, 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.

For example, consider the 5×5 matrix below:

https://doi.org/10.1017/S0956796899003470 Published online by Cambridge University Press

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, 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.

The definitions here are complicated by the fact that matrices may have multiple quadtree representations (Parker and Lê, 1995*b*). For example, the identity matrix can be implemented as a single constant matrix (whose diagonal elements are all

https://doi.org/10.1017/S0956796899003470 Published online by Cambridge University Press

1), or as a dense matrix of 0's and 1's. A disadvantage of multiple representations is that function definitions must cover more cases. This is illustrated by matrix multiplication in the following subsection, which covers four cases instad of only the two *Const* × *Const* and *Quad* × *Quad* required by dense matrices. An advantage of multiple representations is that sparse matrices are neatly representable (and usable directly with dense matrices), and that treating sparseness specially has performance benefits. Another advantage is that all quadtrees can be viewed as having the same full size, so function definitions can ignore the possibility that one input matrix is smaller than another.

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:

```
datatype  α qtree  =
    Const of α
  | Quad of α qtree × α qtree × α qtree × α qtree
```

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 quadtree submatrices A_{11}, A_{12}, A_{21}, A_{22}.

3.2 Matrix multiplication

Let $M_1, M_2 \in$ **Mat** where **Mat** be the set of matrices represented with **qtree**. The function `matmul` returns the product $M_1 \times M_2$ of M_1 and M_2 (`matadd` is similar):

`matmul`: **Mat** × **Mat** → **Mat**

`matmul`($Const(v)$, $Const(w)$) = $Const(v \times w)$

`matmul`($Const(v)$, $Quad(A_{11},A_{12},A_{21},A_{22})$) =
  $Quad$(`matmul`($Const(v)$,A_{11}), `matmul`($Const(v)$,A_{12}),
    `matmul`($Const(v)$,A_{21}), `matmul`($Const(v)$,A_{22}))

`matmul`($Quad(A_{11},A_{12},A_{21},A_{22})$, $Const(v)$) =
  $Quad$(`matmul`(A_{11},$Const(v)$), `matmul`(A_{12},$Const(v)$),
    `matmul`(A_{21},$Const(v)$), `matmul`(A_{22},$Const(v)$))

`matmul`($Quad(A_{11},A_{12},A_{21},A_{22})$, $Quad(B_{11},B_{12},B_{21},B_{22})$) =
  $Quad$(`matadd`(`matmul`(A_{11}, B_{11}), `matmul`(A_{12}, B_{21})),
    `matadd`(`matmul`(A_{11}, B_{12}), `matmul`(A_{12}, B_{22})),
    `matadd`(`matmul`(A_{21}, B_{11}), `matmul`(A_{22}, B_{21})),
    `matadd`(`matmul`(A_{21}, B_{12}), `matmul`(A_{22}, B_{22})))

The final case breaks the two matrices into four regions

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

https://doi.org/10.1017/S0956796899003470 Published online by Cambridge University Press

before multiplying them recursively as

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

It is significant that we can simply replace this standard computation with the asymptotically faster computation (Strassen, 1969)

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\ M_3 &= (A_{11} + A_{12})B_{22} \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M'_2 &= A_{11}(B_{12} - B_{22}) \\ M'_3 &= (A_{21} + A_{22})B_{11} \\ M'_4 &= (A_{11} - A_{21})(B_{11} + B_{12}) \end{aligned} \qquad \begin{aligned} C_{11} &= M_1 + M_2 - M_3 + M_4 \\ C_{12} &= M'_2 + M_3 \\ C_{21} &= M'_3 + M_4 \\ C_{22} &= M_1 + M'_2 - M'_3 - M'_4. \end{aligned}$$

Let $T_s(n)$ and $T_r(n)$ represent the *time* required by the Strassen and recursive block multiplication algorithms, respectively. Their recurrence relations are $T_s(n) = 7T_s(n/2) + 18(n/2)^2$, and $T_r(n) = 8T_r(n/2) + 4(n/2)^2$. Assuming that when $n = 1$ the time required is 1, these have solutions $T_r(n) = 2n^3 - n^2$, $T_s(n) = 7n^{lg(7)} - 6n^2$, where $lg(7) \simeq 2.81$.

The space $S(n)$ required to hold intermediate results by a naive implementation of Strassen's algorithm satisfies $S(n) = S(n/2) + 7(n/2)^2 = 7/3\,n^2 - 7/3$. (This assumes that each subproduct's local variable space is deallocated on completion, as is usual in implementations of recursion.) Furthermore, this space will be reduced by the factor 4/7 if M_2, M_3, and M_4 are deallocated after their lifetimes (and scavenged by M'_2, M'_3, and M'_4). With or without this compiler optimization, the space overhead is reasonable.

3.3 Matrix inversion

A recursive decomposition for matrix inversion is (Faddeev and Faddeeva, 1963)

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} - A^{-1}BY & -A^{-1}BZ \\ Y & Z \end{pmatrix},$$

where $Z = (D - CA^{-1}B)^{-1}$ and $Y = -ZCA^{-1}$. This may be implemented in ML in the following way, using only six multiplications and two recursive inversions:

`matinv`: **Mat** → **Mat**

```
matinv(Const(v)) = Const(1/v)
matinv(Quad(A, B, C, D)) =
    let
        val A⁻¹ = matinv(A)
        val CA⁻¹ = matmul(C, A⁻¹)
        val Z = matinv(matsub(D, matmul(CA⁻¹, B)))
        val Y = matneg(matmul(Z, CA⁻¹))
        val A⁻¹B = matmul(A⁻¹, B)
```

https://doi.org/10.1017/S0956796899003470 Published online by Cambridge University Press

```
        val W = matsub(A^-1, matmul(A^-1 B, Y))
        val X = matneg(matmul(A^-1 B, Z))
    in
        Quad(W,X,Y,Z)
    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 has complexity $M(n)$, the complexity for the inversion algorithm is $T(n) = 2T(n/2) + 6M(n/2) + 4(n/2)^2 = 14n^{lg(7)} - 17n^{lg(6)} + 4n^2$.

The space required by the program above is $S(n) = 2S(n/2) + 10(n/2)^2 = 5n^2 - 5n$, which again is reasonable, but can be improved by an optimizing compiler.

3.4 Gaussian elimination

Like the block LU approach described by Golub and van Loan (1989), a recursive Gaussian elimination algorithm (without pivoting) follows from the decomposition

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} L & \mathbf{0} \\ X & \mathbf{1} \end{pmatrix} \begin{pmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & A' \end{pmatrix} \begin{pmatrix} U & Y \\ \mathbf{0} & \mathbf{1} \end{pmatrix} = \begin{pmatrix} L & \mathbf{0} \\ X & L' \end{pmatrix} \begin{pmatrix} U & Y \\ \mathbf{0} & U' \end{pmatrix}$$

if $\mathbf{1}$ is the identity, $A = LU$, and $D - XY = A' = L'U'$. This may be implemented:

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

```
GE(Const(v)) = (Const(1), Const(v))
GE(Quad(A, B, C, D)) =
    let
        val (L, U) = GE(A)
        val X = matmul(C, matinv(U))
        val Y = matmul(matinv(L), B)
        val A' = matsub(D, matmul(X, Y))
        val (L', U') = GE(A')
    in
        (Quad(L,Const(0),X,L'),
         Quad(U,Y,Const(0),U'))
    end
```

Both `matinv` and GE succeed if, and only if, we can recursively guarantee the the input matrix and its first quadrant are both invertible. Obviously this is necessary. It is sufficient because when both $Quad(A, B, C, D)$ and A are invertible, the result of Gaussian elimination $(D - CA^{-1}B)$ also will be invertible.

3.5 Some other matrix algorithms

Other suitable block decomposition algorithms using the quadtree representation include the Cholesky and QR factorizations, and the FFT.

https://doi.org/10.1017/S0956796899003470 Published online by Cambridge University Press

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 $M = LL^T$.

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

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

$$\sqrt{M} = \begin{pmatrix} \sqrt{A} & \mathbf{0} \\ Y & \sqrt{D - YY^T} \end{pmatrix} \qquad \text{where } Y = C(\sqrt{A}^T)^{-1}.$$

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, obtainable via Cholesky factorization:

$$Q = MR^{-1}, \qquad R = (\sqrt{M})^T.$$

Discrete Fourier transform, and tensors

Generally speaking, quadtrees are effective at representing transformations that are naturally expressed with tensor (Kronecker product) notation. For example, when n is a power of 2, $\omega_n = exp(-2\pi\sqrt{-1}/n)$, and $1 \leqslant i, j \leqslant n$, the Fast Fourier Transform (FFT) of size n factors the $n \times n$ matrix $F_n = (\omega_n^{(i-1)(j-1)})$ into the form

$$F_n = B_n \ (I_2 \otimes F_{n/2}) \ \sigma_n{}^{-1}, \qquad B_n = \left(\begin{array}{c|c} I_{n/2} & D_{n/2} \\ \hline I_{n/2} & -D_{n/2} \end{array}\right)$$

where B_n is a 'butterfly matrix' with $D_n = diag(\omega_n^{i-1})$, and σ_n is the 'perfect shuffle' permutation matrix (van Loan, 1992). Here '$\otimes$' 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

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

Quadtrees easily handle this tensor product. An ML implementation is as follows:

`tensor`: **Mat** $\times$ **Mat** $\rightarrow$ **Mat**

`tensor`$(Const(x), Const(y)) = Const(x \times y)$

`tensor`$(Const(x), Quad(Y_{11}, Y_{12}, Y_{21}, Y_{22})) =$

https://doi.org/10.1017/S0956796899003470 Published online by Cambridge University Press

$$\begin{aligned}&\mathit{Quad}(\texttt{tensor}(\mathit{Const}(x),\ Y_{11}),\ \texttt{tensor}(\mathit{Const}(x),\ Y_{12}),\\ &\qquad \texttt{tensor}(\mathit{Const}(x),\ Y_{21}),\ \texttt{tensor}(\mathit{Const}(x),\ Y_{22}))\\ \texttt{tensor}(&\mathit{Quad}(X_{11},X_{12},X_{21},X_{22}),\ Y) =\\ &\qquad \mathit{Quad}(\texttt{tensor}(X_{11},Y),\ \texttt{tensor}(X_{12},Y),\\ &\qquad \texttt{tensor}(X_{21},Y),\ \texttt{tensor}(X_{22},Y))\ .\end{aligned}$$

3.6 At issue: guaranteeing nondegeneracy

Some important quadtree algorithms work if and only if we can guarantee *nondegeneracy*: every recursively input quadtree and its first quadrant are both invertible. For example, when the input matrix is symmetric positive definite this guarantee will be met (Golub and van Loan, 1989). In general, of course, it is not.

There are several old tricks that work in the important case where M is invertible but nothing specific is known about its submatrices. Bunch and Hopcroft (1974) mention two. First, the rows of such M can always be permuted so that the invertibility 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. Secondly, any invertible matrix M can be made nondegenerate 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 invertible, $M^+ = M^{-1}$. Unfortunately, this method needs extra multiplications.

This adjoint approach is also risky because it amplifies numerical ill-conditioning. The quality of many computations involving a matrix A depends upon its condition number $\kappa(A) = \|A\|\|A^{-1}\|$, where $\|A\|$ is a norm reflecting the magnitude of all entries in A (Golub and van Loan, 1989). A popular choice is the spectral norm $\|A\|_2 = \max\{\sqrt{\lambda} \mid \lambda \text{ is an eigenvalue of } A^*A\}$. A matrix A is called *ill-conditioned* when $\kappa(A)$ is very large, and *well-conditioned* when it is small. A rule of thumb (Golub and van Loan, 1989) is that when $\kappa(A) \approx 10^q$, floating-point arithmetic computations involving A often lose at least q significant digits. Because $\|M^*M\|_2 = \|M\|_2^2$, $\kappa(M^*M) = \kappa(M)^2$. Thus, operating on M^*M instead of on M generally loses twice as many significant digits.

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 led 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

https://doi.org/10.1017/S0956796899003470 Published online by Cambridge University Press

measure zero in the space of all matrices. Our idea is to exploit their insignificance through the fact that randomization of an invertible matrix can eliminate degeneracy (noninvertible blocks).

4.1 The randomization technique

Suppose M is an invertible real $n \times n$ matrix, and let U and V be invertible random $n \times n$ matrices. Then:

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

Intuitively U^* and V 'homogenize' M enough so that $\widetilde{M}$ is not degenerate. For example,

$$U^* = \begin{pmatrix} +0.6483 & -0.7614 \\ +0.7614 & +0.6483 \end{pmatrix}, \quad V = \begin{pmatrix} +0.7279 & -0.6857 \\ +0.6857 & +0.7279 \end{pmatrix}$$

randomize the degenerate problem

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

into the nondegenerate problem

$$\begin{pmatrix} 0.1097 & 0.9940 \\ 0.9940 & 0.1097 \end{pmatrix} \mathbf{y} = \begin{pmatrix} -0.9870 \\ 3.468 \end{pmatrix}$$

where $\mathbf{x} = V\mathbf{y}$. Solving the randomized problem gives the solution (correct to 3 digits)

$$\mathbf{y} = \begin{pmatrix} 3.556 \\ -0.6005 \end{pmatrix}, \quad \mathbf{x} = V\,\mathbf{y} = \begin{pmatrix} 3.000 \\ 2.001 \end{pmatrix}.$$

Theorem 1 (Parker (1995*a*)
If $M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$ is a $n \times n$ invertible matrix, and R, S are invertible diagonal independent random complex-valued $(n/2) \times (n/2)$ matrices, then in the randomized matrix

$$\widetilde{M} = U^*MV = \begin{pmatrix} I & R \\ I & -R \end{pmatrix} \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} I & I \\ S & -S \end{pmatrix}$$

each quadrant is invertible, with probability 1.

Here, by an independent random complex-valued matrix, we mean a matrix whose elements are chosen independently from a suitable distribution. When R and S are of the form $\exp(\mathrm{i}\,\Theta)$ and Θ is a diagonal random real matrix, they are invertible, diagonal, and unitary. Furthermore, U and V are unitary butterfly matrices.

The proof proceeds by showing that the quadrants are invertible with probability 1 because their determinants are nonzero with probability 1. Note that the random

https://doi.org/10.1017/S0956796899003470 Published online by Cambridge University Press

butterfly transformations U^* and V on M yield

$$\widetilde{M} \quad = \quad \begin{pmatrix} A+BS+RC+RDS & A-BS+RC-RDS \\ A+BS-RC-RDS & A-BS-RC+RDS \end{pmatrix}.$$

Each entry in $\widetilde{M}$ is a polynomial of form $a+b\,s+c\,r+d\,rs$ in some random variable r of R and s of S. The determinant of any quadrant of $\widetilde{M}$ is thus a polynomial in the variables of R and S. Using the Binet-Cauchy theorem (Marcus and Minc, 1964), we can find an explicit form for this determinant polynomial, and show that it cannot be identically zero if M is invertible. So, the determinant is zero only if all of its random variables take on values that make the polynomial zero. Since these variables are continuous (complex-valued), the probability such values arise simultaneously is 0; and with probability 1, the value of the determinant polynomial will be nonzero. Thus with probability 1 each quadrant is invertible.

Technically, the phrase "with probability 1" is inaccurate here, because we are using floating point computation. If floating point numbers have t-bit precision, and the probability that two randomly-chosen floating point numbers are equal is 2^{-t}, the phrase should be replaced by "with probability $1 - O(2^{-t})$". In what follows, "with probability 1" should be read this way.

A similar approach works for many kinds of random matrices (Parker, 1995*b, c*; Parker and Pierce, 1995). With non-sparse random matrices U, V, the product U^*MV is generally nondegenerate. The challenge lies mainly in finding randomizing transforms that are fast (e.g. sparse or factorable).

Theorem 2 (Parker and Pierce, 1995)
If M is a $n \times n$ invertible matrix, and R, S are invertible diagonal independent random complex-valued $n \times n$ matrices, then in the *Random Fast Fourier Transform (RFFT)*

$$\widetilde{M} \quad = \quad U^*\,M\,V \quad = \quad (R\,F_n)^*\,M\,(S\,F_n) \quad = \quad F_n{}^*\,R^*\,M\,S\,F_n$$

each leading principal submatrix (submatrix with row and column indices $1,\ldots,k$) is invertible, with probability 1.

The proof rests on F_n being a Vandermonde matrix (Marcus and Minc, 1964), and its submatrices with column indices $1,\ldots,k$ also being Vandermonde, hence having nonzero determinant. Expansion formulas for determinants show again that the determinant of any leading principal square block of $\widetilde{M}$ is a polynomial in the random variables of R and S, and because M is invertible this polynomial is nonzero with probability 1.

To be effective in quadtree LU decomposition or matrix inversion, randomization should accomplish two goals:

1. The randomized matrix $\widetilde{M}$ must be nondegenerate.
2. Roundoff error incurred in the randomized problem must be acceptable.

Theorem 2 allows us to show the RFFT accomplishes the first goal. Note that in any block decomposition Consider any leading principal submatrix S of $\widetilde{M}$, $S = \begin{pmatrix} W & X \\ Y & Z \end{pmatrix}$, where only W and Z need be square and X and Y can be

https://doi.org/10.1017/S0956796899003470 Published online by Cambridge University Press

rectangular. Theorem 2 shows both W and S will be invertible. Furthermore, every diagonal square block produced by Gaussian elimination is the *Schur complement* $(Z - Y W^{-1} X)$ of some such S, which is invertible since W and S are. The RFFT does as much randomization in one step as a sequence of random butterfly transforms.

As to the second goal, the randomized algorithms do generate greater roundoff errors than their traditional counterparts with pivoting. However, the errors appear acceptable with well-conditioned input matrices as we show below. See also Parker (1995*a, c*). Ultimately, a statistical error analysis like that in Trefethen and Schreiber (1990) is important to evaluate any real variant of Gaussian elimination. In fact, although Gaussian elimination with partial pivoting has been believed 'stable in practice' for several decades, recently papers have appeared pointing out that this belief is incorrect for commonly-encountered matrices, including important orthogonal matrices; see Parker (1995*b*).

Furthermore, *iterative improvement* can be used to reduce error in the quadtree solution. After solving $M\mathbf{x} = \mathbf{b}$, iterative improvement repeatedly recycles the error $(\mathbf{b} - M\mathbf{x})$ through the initial LU factorization to get an update for $\mathbf{x}$. Each iteration requires only $O(n^2)$ time to compute, and generally performing k iterations gives $\min(t, k(t-q))$ correct digits in $\mathbf{x}$ if $\kappa(M) \approx 10^q$ (Golub and van Loan, 1989).

4.2 Randomized matrix inversion

Recall Faddeev's matrix inversion scheme mentioned earlier:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} - A^{-1}BY & -A^{-1}BZ \\ Y & Z \end{pmatrix},$$

where $Z = (D - CA^{-1}B)^{-1}$ and $Y = -ZCA^{-1}$. When Strassen's multiplication algorithm is used, this algorithm has complexity $O(n^{2.81})$.

For comparison with LAPACK[1], the Faddeev routine was rewritten in C. The timings and errors of Faddeev matrix inversion and LAPACK's matrix inversion using 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 running Linux with 32Mb RAM and 64Mb swap space.

The point of using matrices with normally-distributed entries is that they are well-conditioned (Edelman, 1988) 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 $T_f(n)$ be the time required by Faddeev inversion and $T_g(n)$ the time required by GE inversion. Then Table 1 indicates $T_g(n)/T_g(n/2) \approx 8.0$; whereas

[1] "LAPACK is a library of Fortran 77 subroutines for solving the most commonly occurring 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." (Anderson *et al.*, 1995)

[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 inverse A^{-1} can then be formed as $U^{-1}L^{-1}P^{-1}$.

https://doi.org/10.1017/S0956796899003470 Published online by Cambridge University Press

Table 1. *Timings from Faddeev and LAPACK matrix inversion on normally distributed random matrices. Timings for the RFFT alone show its overhead*

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

Table 2. *Errors from LAPACK, plain Faddeev, and randomized Faddeev matrix inversion on normally distributed random matrices. For each method, the average, high, and low numbers of significant digits are shown*

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	1_{D3}	3_{D-13}	13.9	15	12	13.0	14	11	12.9	14	11
128	7_{D3}	2_{D-12}	13.0	15	11	11.7	13	9	10.8	13	8
256	2_{D4}	3_{D-12}	12.7	15	10	10.2	13	7	9.0	11	6
512	4_{D4}	3_{D-11}	12.1	14	9	8.6	12	5	7.6	10	4

$T_f(128)/T_f(64) \approx 7.62$, $T_f(256)/T_f(128) \approx 7.36$, and $T_f(512)/T_f(256) \approx 7.27$ — converging to the scaling factor 7 we would expect ($n^{lg(7)} = 7^{lg(n)}$). Thus we expect $T_f(1024)$ to take approximately $7.27T_f(512) = 3850$ seconds and $T_g(1024)$ to take about $8.0T_g(512) = 5700$ seconds. Table 1 again shows the low overhead of the $O(n^2 \log n)$ RFFT.

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 speedup 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.

4.3 Randomized Gaussian elimination

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

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} L & \mathbf{0} \\ X & \mathbf{1} \end{pmatrix} \begin{pmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & A' \end{pmatrix} \begin{pmatrix} U & Y \\ \mathbf{0} & \mathbf{1} \end{pmatrix}.$$

https://doi.org/10.1017/S0956796899003470 Published online by Cambridge University Press

When the input is randomized, the matrix A is invertible with probability 1. 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 block recursive matrix multiplication, instead of Strassen matrix multiplication, is used.

Inversion of lower triangular and upper triangular matrices can be achieved with:

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

$$\begin{pmatrix} A & B \\ \mathbf{0} & D \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & -A^{-1}BD^{-1} \\ \mathbf{0} & D^{-1} \end{pmatrix}.$$

Assuming $T_m(n)$ is Strassen multiplication complexity, the complexity for either upper or lower triangular inversion is $T_i(n) = 2T_i(n/2) + 2T_m(n) = 14/5\, n^{lg(7)} - 6n^2 + 21/5\, n$. Similarly, the products

$$\begin{pmatrix} L & \mathbf{0} \\ X & L' \end{pmatrix}\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} LA & LB \\ XA+L'C & XB+L'D \end{pmatrix}$$

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}\begin{pmatrix} U & Y \\ \mathbf{0} & U' \end{pmatrix} = \begin{pmatrix} AU & AY+BU' \\ CU & CY+DU' \end{pmatrix}$$

execute in time $T(n) = 6T_m(n) + 2(n/2)^2 = 6n^{lg(7)} - 17/2\, n^2$.

The complexity of both $X \leftarrow CU^{-1}$ and $Y \leftarrow L^{-1}B$ is thus $T(n) = T_i(n) + T_m(n) = 49/5\, n^{lg(7)} - 12\, n^2 + 21/5\, n$. Tables 4 and 5 show 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 4 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 approximately $8.0 \times T_l(1024) = 15000$ seconds and $T_s(2048)$ to take approximately $7.5 \times T_s(1024) = 12000$ seconds. Table 4 again shows that the RFFT overhead is relatively low.

Analogous to Table 2, Table 5 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 Bailey *et al.* (1991) a general Strassen multiplication is derived to solve LU factorization on a Cray-YMP. Also, in Balle and Hansen (1994), 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

Table 3. *Timings from recursive GE without pivoting vs. block GE without pivoting*

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 4. *Timings from recursive LU without pivoting using Strassen multiplication vs. LAPACK's LU with partial pivoting on normally distributed random matrices*

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

the recursive block decomposition and quadtree approaches practical for a broader class of invertible 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 recursive 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

Table 5. *Errors from plain and randomized recursive LU without pivoting using Strassen multiplication vs. LAPACK's LU with partial pivoting, on normally distributed random matrices. For each method, the average, high, and low numbers of significant digits are shown*

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	5_{D3}	1_{D-12}	13.5	15	11	11.8	15	9	11.3	13	8
256	1_{D4}	5_{D-12}	13.0	15	10	11.0	14	7	10.4	12	7
512	8_{D4}	1_{D-11}	12.5	15	9	9.5	14	6	9.3	12	6
1024	8_{D4}	4_{D-11}	12.1	15	9	8.6	13	4	8.2	11	4

https://doi.org/10.1017/S0956796899003470 Published online by Cambridge University Press

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 (van Loan, 1992) 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 block submatrices that fail to be invertible) can render the quadtree approach useless.

We showed how the randomization technique developed by Parker (1995*a, b, c*, Parker and Lê (1995*a*) and Parker and Pierce (1995) 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:

- simplicity and expressiveness;
- access to faster divide-and-conquer algorithms like Strassen's algorithm.

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 (Edelman, 1988)).

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 (Parker and Lê, 1995*b*):

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. Randomization 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 Parker (1995*c*), and appears to yield somewhat better numerical accuracy than the implementations in the previous section. Further improvements in accuracy result from scaling, tricks for avoiding cancellation error, and iterative improvement (Golub and van Loan, 1989).
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 (Parker and Lê, 1995*b*).
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 elsewhere (Lê *et al.*, 1995).

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.

Acknowledgements

We are grateful to the two anonymous referees, Brad Pierce, Jack Carlyle, Tony Chan and Miloš Ercegovac, for suggestions that significantly improved this paper.

References

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1974) *The Design and Analysis of Computer Algorithms.* Addison-Wesley.

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S. and Sorenson, D. (1995) *LAPACK User's Guide. (2nd ed).* SIAM.

Bailey, D. H., Lee, K. and Simon, H. D. (1991) Using Strassen's algorithm to accelerate the solution of linear systems. *J. Supercomput.*, **4**(4), 357–371.

Balle, S. M. and Hansen, P. C. (1994) *A Strassen-type matrix inversion algorithm*, pp. 22–30. Advances in Parallel Algorithms. IOS Press.

Bunch, J. R. and Hopcroft, J. (1974) Triangular factorization and inversion by fast matrix multiplication. *Math. Comp.*, **28**, 231–236.

Choi, J., Demmel, J., Dhillon, I., Dongarra, J., Ostrouchov, S., Petitet, A., Stanley, K., Walker, D. and Whaley, R. C. (1996) ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers – Design Issues and Performance. *Computer Physics Communications*, **97**(1–2).

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

Faddeev, D. K. and Faddeeva, V. N. (1963) *Computational Methods of Linear Algebra.* W. H. Freeman.

Golub, G. H. and van Loan, C. F. (1989) *Matrix Computations (2nd. ed.).* Johns Hopkins.

Lê, D., Parker, D. S. and Pierce, B. (1995) Input randomization and automatic derivation of systolic arrays for matrix computations. Technical Report CSD-950038, UCLA Computer Science Department.

Marcus, M. and Minc, H. (1964) *A Survey of Matrix Theory and Matrix Inequalities.* Dover.

Parker, D. S. (1995*a*) A Randomizing Butterfly Transformation Useful in Block Matrix Computations. *Technical Report CSD-950024*, UCLA Computer Science Department.

Parker, D. S. (1995*b*) Explicit Formulas for the Results of Gaussian Elimination. *Technical Report CSD-950025*, UCLA Computer Science Department.

Parker, D. S. (1995*c*) Random Butterfly Transformations with Applications in Computational Linear Algebra. *Technical Report CSD-950023*, UCLA Computer Science Department.

Parker, D. S. and Lê, D. (1995*a*) How to Eliminate Pivoting from Gaussian Elimination – by Randomizing Instead. *Technical Report CSD-950022*, UCLA Computer Science Department.

Parker, D. S. and Lê, D. (1995*b*) Quadtree Matrix Algorithms Revisited: Basic Issues and their Resolution. *Technical Report CSD-950028*, UCLA Computer Science Department.

Parker, D. S. and Pierce, B. (1995) The randomizing FFT: an alternative to pivoting in Gaussian elimination. *Technical Report CSD-950037*, UCLA Computer Science Department.

Strassen, V. (1969) Gaussian elimination is not optimal. *Numer. Math.*, **13**, 354–356.

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

van Loan, C. F. (1992) *Computational Frameworks for the Fast Fourier Transform.* SIAM.

Wise, D. S. (1985) Representing matrices as quadtrees for parallel processors. *Information Processing Letters*, **20**(4), 195–199.

Wise, D. S. (1986) Parallel decomposition of matrix inversion using quadtrees. In: Hwang, K., Jacobs, S. J. and Swartzlander, E. E. (eds.), *Proc. 1986 International Conference on Parallel Processing*, pp. 92–99. IEEE Press.

Wise, D. S. (1987) Matrix algebra and applicative programming. In: Kahn, G. (ed.), *Functional Programming Languages and Computer Architecture: Lecture Notes in Computer Science 274*, pp. 134–153. Springer-Verlag.

Wise, D. S. (1992) Matrix algorithms using quadtrees. In: Hains, G. and Mullin, L. M. R. (eds.), *ATABLE-92: 2nd International Workshop on Array Structures*, pp. 11–26. University of Montreal.

The references by Parker et al. are all available at:

`http://www.cs.ucla.edu/~stott/ge/`