Experiments with Quadtree Representation of Matrices

S. Kamal Abdali
Tektronix Labs
P.O. Box 500, MS 50-662
Beaverton, Oregon 97077, USA

David S. Wise†
Indiana University
101 Lindley Hall
Bloomington, Indiana 47405, USA

Abstract

The quadtrees matrix representation has been recently proposed as an alternative to the conventional linear storage of matrices. If all elements of a matrix are zero, then the matrix is represented by an empty tree; otherwise it is represented by a tree consisting of four subtrees, each representing, recursively, a quadrant of the matrix. Using four-way block decomposition, algorithms on quadtrees accelerate on blocks entirely of zeroes, and thereby offer improved performance on sparse matrices. This paper reports the results of experiments done with a quadtree matrix package implemented in REDUCE to compare the performance of quadtree representation with REDUCE's built-in sequential representation of matrices. Tests on addition, multiplication, and inversion of dense, triangular, tridiagonal, and diagonal matrices (both symbolic and numeric) of sizes up to 100x100 show that the quadtree algorithms perform well in a broad range of circumstances, sometimes running orders of magnitude faster than their sequential counterparts.

CR categories and Subject Descriptors:

I.1.2 [Algebraic Manipulation Algorithms]; Algebraic algorithms; E.1 [Data Structures]: Trees; G.1.3 [Numerical Linear Algebra]: Sparse and very large systems.
General Term: Measurement.

Section 1. Introduction

Representation and manipulation of matrices has been a driving force for computer development from the beginnings of digital computation. Indeed, it is difficult to say which has had the greater effect on the other: the architecture of computers or the development of matrix algorithms. Certainly both are responsible for the present situation: that the standard algorithms manipulate matrices, or at least rows thereof, sequentially according to consecutive

† Visiting scientist at Tektronix Labs during the summer of 1987 when this work was done.

indexing, and that processors access memories, or at least the pages thereof, sequentially according to consecutive addresses. By now it has become a chicken-egg problem.

The linear storage of matrices makes it, in general, inefficient to implement algorithms exploiting the matrix theory relations based on partitioning. Consider, for example, the following algorithm for computing the product of two partitioned matrices (where the blocks *A*, *B*, *etc.*, are assumed to be of such dimensions that all matrix multiplications stipulated on the right-hand side are possible):

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \times \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix}$$

In linear storage, the blocks *A*, *B*, *etc.* are not available directly; assembling them from the elements of the whole matrices is too costly to make the above algorithm of much practical value. This is unfortunate since such algorithms abound in matrix theory. In fact, while many of such relations have been known for over a century, relatively recent adaptations of some of these have given rise to asymptotically fast algorithms (*e.g.*, [10]).

Since computer algebra systems are built over a heap model of memory, they generally use tree storage for representing and manipulating symbolic expressions. Yet, perhaps because of the perceived necessity of indexing in known algorithms, these systems invariably resort to linear storage for vectors and matrices. Little has been done, consequently, working toward heap-based algorithms that manipulate matrices and vectors as structures more abstract than a linear list. (*cf.* § 2.2 and § 2.3 of Knuth [6].) Such an effort might well uncover new algorithms better suited to the heap memories that are already in use, and to the newer requirements of decomposing problems for multiprocessor solution.

The quadtree representation of matrices has recently been proposed by the second author [11] for a heap-based, multiprocessing environment. This representation makes it possible to implement with relative ease the algorithms based on partitioning. Another attractive attribute of the quadtree representation of matrices is that it unifies computation on both dense and sparse matrices. That is, this single representation can represent both dense and non-dense matrices with relatively efficient use of space, and this single family of algorithms manipulates both sparse and non-sparse matrices with relatively conservative use of time [14]. There are better specialized structures and algorithms for extremely dense or extremely sparse matrices, but no other approach avoids a dichotomy of performance across the spectrum. Although not often a design criterion in computer algebra systems, efficient handling of both sparse and non-sparse problems is a welcome dividend of this approach.

This paper reports the first empirical exploration of the quadtree structure for matrix manipulations and its associated family of algorithms. To obtain a fair performance comparison of quadtree and traditional linear structures, we have tried to minimize the effect of programming level and style differences in the codes for the two representation. The experiments were performed on a conventional uniprocessor, using REDUCE [5], a widely distributed computer algebra system. REDUCE was selected for three reasons. First, it already contains a mature package of algorithms, which uses linear structures. Next, all of its source code is available for examination. Finally, it allows efficient insertion of user-written code for execution at par with system code. For our experiments we were able to implement a package for quadtree matrices closely following Hearn's style in REDUCE's matrix package, and, therefore, to extract meaningful performance comparisons.

The experiments do not address the significance of quadtree matrix representation for parallel processing [12]. The results do establish a comparison, however, between the traditional serial-access algorithms delivered in REDUCE and analogously coded quadtree version. In some cases the algorithms are equivalent, except for the access pattern; in others, the quadtree structure suggests a completely different algorithm. In the former cases, a direct comparison is possible between performance of the traditional and of the quadtree algorithms on matrices of various sizes and of differing density/sparsity. It is seen that, for example, the quadtree matrix inversion algorithms are up to five times slower than REDUCE's algorithms on completely dense symbolic matrices of very small sizes, but run many times faster on similarly sized sparse matrices. In the case of symbolic multiplication, quadtree algorithms are overall much faster.

The remainder of this paper is in six parts. The next section reviews the normal form representation of quadtree matrices and outlines some familiar algorithms. Section 3 describes the matrices used in experiments. Section 4 offers comparisons of run times of matrix additions and multiplications of integer as well as symbolic matrices sized from 4×4 to 100×100 of different sparsities: fully dense, triangular, tridiagonal, and diagonal; all these results are directly comparable to REDUCE's default performance. Section 5 considers one approach to matrix inversion, by simple partitioning, and presents results running times for this algorithm, which is not effective in all cases. Section 6 discusses how to construct an algorithm that will always be effective, but whose performance may approach that of the partitioning inverse. Section 7 offers conclusions.

Section 2. Quadtree Representation

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

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

Matrix algorithms are arranged so that we may perceive any nonzero scalar, s, as a diagonal matrix of arbitrary order, entirely of zeroes except for s's on the main diagonal; that is, $s = [s\delta_{i,j}]$. Thus, a domain is postulated that coalesces scalars and matrices, with every scalar-like object conforming also as a matrix of any order. Of particular interest is the scalar 0, which is at once the *unique* additive identity and multiplicative annihilator for both scalar and matrix arithmetic. It is often represented by the null pointer (nil in LISP notation) to save space from non-dense matrices [14].

A matrix (of otherwise-known order) is either a 'scalar' or it is a quadruple of four equally-ordered submatrices. So that this recursive cleaving works smoothly, we embed a matrix of order n in a $2^{\lceil \lg n \rceil} \times 2^{\lceil \lg n \rceil}$ matrix, justified to the lower, right (southeast) corner with padding to the north and west. Padding to the north and to the west is 0, minimizing space. There are two choices for padding *on* the principal diagonal: padding with 0 suffices there under additive operations, multiplication, and inversions that use pivoting; other inversions and the usual algorithms for determinants require 1 padding there to avoid singularities. (The choice is not critical because this padding can easily be switched before any algorithm at a cost logarithmic in the order of the matrix.) Either choice prescribes a *normal form* for quadtree matrices.

Similarly, we may perceive any non-zero scalar s as a homogeneous vector all of whose components are s. Thus, a vector is either a 'scalar' or it is an ordered pair of equally-sized subvectors. For the purposes of this paper we embed a vector of size n, justified downward, in a vector of size $2^{\lceil \lg n \rceil}$ with zero padding at the top.

Inferring the conventional meaning from such a matrix or vector now requires additional information (*viz.* its order), but we can proceed quite far without size information; it does becomes critical upon Input or Output and in computing eliminants [1] or determinants. One must acknowledge that the I/O conversions are non-trivial algorithms [12], but this is not serious because they also consume comparatively little processor resource and are restrained by communication bandwidth. Like floating-point number conversions, they are an irritating impediment to one who would experiment with algorithms discussed below.

The recursive definition of quaternary trees molds the recursive structure of programs that manipulate them. Moreover, the bifurcation of tree composition leads naturally to more stable algorithms. For instance, each addend in the sum over a vector of size 2^p (as a binary tree of depth p) naturally participates in no more than p binary additions; if the vector were instead stored in consecutive memory locations, the "natural" algorithm has each addend participating in up to $2^p - 1$ additions. This can be important in postponing overflow of an inexpensive representation (*e.g.* fixed integers) to a more expensive one (respectively, *bignums*) until higher in the tree; since each addend participates in at most p sums, the partial results do not accumulate to overflow quite so soon and only a few of the additions, at the top or the tree, become the expensive ones.

It is particularly surprising to uncover new variants of old, well-studied algorithms, like the folding of full-matrix search into the Pivot Step algorithm [12]. While Pease's block decomposition of the Fast Fourier Transform is not new, the two factorings of the *shuffle* and the *deal* bit-reversal permutation, each precisely following the nesting of the FFT recurrence pattern, is an insight useful in its implementation [13]. This practical representation of an array as a tree suggests—as already done in computer algebra systems—that data is more efficiently linked across a heap memory, than allocated sequentially.

Size	Dense	Triangular	Tridiagonal	Diagonal
4	10	33	33	83
	83	33	50	33
5	50	66	83	50
	116	66	100	83
6	50	83	66	83
	150	100	116	83
8	100	50	83	100
	166	133	83	83
10	133	150	116	100
	216	183	150	83
12	150	183	133	166
	300	200	183	50
20	416	416	350	366
	766	450	284	116
30	900	800	700	733
	1616	866	400	150
40	1516	1383	1233	1233
	2783	1516	516	200
50	2383	2166	1933	1916
	4400	2316	683	266
60	3434	3050	2716	2733
	6250	3250	733	283
80	6067	5483	4884	4833
	11200	5650	1050	383
100	9417	8517	7650	60334
	18200	8834	1350	7550

Table 1. Addition of symbolic matrices

Section 3. Matrices Used in Experiments

The experiment consisted of computing matrix sums, products, and inverses. Both sym-bolic and numerical matrices were used. The matrices of four patterns of sparseness were util-ized: fully dense, triangular, tridiagonal, and diagonal. Zero entries were used only for the par governed by the sparseness pattern. For example, in triangular matrices, the entries in the par above the principal diagonal were zero, and the remaining entries were all non-zero.

In symbolic matrices, each non-zero entry was a distinct symbol—to be exact, an operato expression whose arguments were the row and column indices of the entry. In numerica matrices, each non-zero entry was a random integer. Lest the arithmetic complexities of float ing point or *bignum* computations cloud the results, only small random integers of absolut value between 1 and 30 were used.

The experiments were attempted on matrices of sizes ranging from 4×4 to 100×100. Eacl column in the tables to be discussed below was generated by a single program. In several cases the programs could not run to completion: sometimes they prematurely ended by running ou of available memory; sometimes we just interrupted them because they had consumed to much time. For example, the inverse computation of dense symbolic matrices could be don only for sizes 4×4 and 5×5, but the multiplication of triangular symbolic matrices could b completed for sizes up to 80×80.

The algorithm for matrix addition and subtraction [11] decomposes naturally into four quadrant additions, separate and independent processes. Whenever either addend is 0, their sum is efficiently represented as a shared reference to the root of the other addend, without need for any further traversal.

Matrix multiplication decomposes into four sums and eight products. Whenever a factor is either 0 or 1, the product is directly available, either as 0, or as a shared reference to the other factor. The former case occurs particularly often within sparse factors, and annihilates the recursion not only of quadrant multiplication, but also of the addition of quadrant-products that follows.

Analytic measures of both sparse and dense quadtree representations have been presented elsewhere [14]. These results have guided the selection of the test cases presented here. We acknowledge that completely dense matrices and diagonal matrices are extreme cases, but they bound the range of performance. Triangular matrices and tridiagonal matrices are a coarse, but feeble, attempt to cover the middle of the spectrum; it is easy to generate reliable data for these cases, but not for other cases that are more typical. Indeed, characterization of a "typically" non-dense matrix seems to be an open problem; no one knows what pattern is typical of real data.

Tables 1 and 2 summarize the timing results for addition of symbolic and numerical (integer) matrices, respectively, for the four sparseness patterns: dense, triangular, tridiagonal, and diagonal. Of the two entries shown for any size and pattern combination, the upper one is the time taken by REDUCE's built-in matrix package, and the lower one is the time taken by the quadtree package. All times shown in the tables are produced by REDUCE's timing functions, measuring CPU time in milliseconds, exclusive of garbage collection. The values in each column come from a single experiment run under Tektronix's Franz Lisp version of REDUCE 3.3 on a SUN 3/160 with eight megabytes of memory.

Tables 3 and 4 show similar results from symbolic and numerical multiplication.

The experiments indicate that quadtree addition is at most twice as slow as linear addition for fully dense symbolic matrices of sizes larger than 40, but up to 8 times faster for diagonal matrices of the same size range. Even more startling is the performance of quadtree multiplication. For fully dense symbolic matrices, quadtree catches up at about size 13, and is already twice as fast by size 30. For very sparse matrices, it is remarkably faster.

It is interesting to note that the times for triangular matrices grow at about the same rate as dense matrices, because the problem is dominated by the dense subquadrants of the problem.

Space was not measured. Accurate analytic results for these special cases are available [13, 14], although the analysis therein does not consider the extended costs for precise representation of large numeric entries. In order to avoid time corruption from large numbers, in fact, these data were constrained so that every entry in the "filled" portion of an operand-matrix was an integer between one and thirty.

Size	Dense	Triangular	Tridiagonal	Diagonal
4	83	83	66	50
	133	33	100	66
5	166	133	100	83
	216	150	150	66
6	116	150	133	100
	216	150	183	50
8	283	266	233	133
	300	266	200	83
10	466	400	383	216
	616	383	366	116
12	650	650	516	350
	800	500	416	166
20	1933	1800	2066	1100
	2316	1216	933	250
30	4166	4150	4866	3000
	5083	2500	1383	400
40	7400	7750	8866	6000
	8783	4283	2016	583
50	11450	12183	13683	9900
	14160	6683	2533	766
60	16684	17883	20400	15200
	19583	9250	3033	900
80	29617	32300	37550	32900
	34617	16150	4233	1366
100	46283	50884	60050	51517
	54983	25200	5416	1700

Table 2. Addition of integer matrices

Section 5. Inversion by Partitioning

REDUCE's default matrix inversion procedure is based on Bareiss's method [2]. (Inversion based on Cramer's rule can be invoked by flipping a special switch.) Bareiss's method, as he presents it, is more suitable for linearly stored matrices than for quadtree matrices. While a quadtree adaptation of his method is possible by using some ideas given in [1], it is rather complicated, and its computational advantage is quite doubtful. For our experiment, we just chose to use the simple relation expressing the inverse of a matrix in terms of its four partitions (see *e.g.*, [3]).

While very simple to implement in the quadtree representation, inversion by partitioning obviously fails to compute the inverse of a non-singular matrix any of whose northwest (principal) subquadrants is singular. Furthermore, at each level a composite matrix (Schur product) must also be non-singular. For the abstract symbolic matrices considered in our experiments, this situation cannot arise; we were certainly fortunate that none of our numerical test matrices encountered any problem with this inversion procedure either. Not only do we question how often such singularities occur in real data, but also we wonder whether more realistic sparseness in data hampers the algorithm (by proliferating singular quadrants) or whether it actually steers the refinement (by making it easier to identify and to avoid quadrants that, themselves, are singular or whose partitioning always results in four singular quadrants.)

The timing results for symbolic and numerical matrices are shown in Tables 5 and 6, respectively. The particular inversion method we used is inferior for fully dense symbolic matrices. Since the inversion of dense symbolic matrices of even sizes as small as 10 is infeasible anyway without using a great deal of abbreviations for expressing intermediate expressions, this test case is not of much practical significance. But in sparse cases, symbolic as well as numerical, the partitioning method with quadtrees is unignorably efficient.

Section 6. Hybrid Inversion by Pivoting

It is unfair to test Bareiss's total algorithm against the partial algorithm for matrix inversion by partitioning; Bareiss's is strictly more powerful than partitioning. Therefore, we are unable draw any strong conclusion from the comparison.

Size	Dense	Triangular	Tridiagonal	Diagonal
4	133	100	50	66
	200	100	116	66
5	216	100	83	66
	516	150	266	100
6	366	100	100	66
	650	216	283	83
8	1050	233	166	83
	1416	416	316	100
10	2400	416	200	150
	3000	700	583	100
12	4667	800	283	216
	5000	1133	633	166
20	32700	4150	716	650
	24617	4633	1200	216
30	157466	17583	1916	1716
	82867	14967	1883	283
40	485233	50634	4000	3716
	207800	35083	2550	400
50		116550	7183	6817
		68784	3300	433
60		232784	11783	11400
		117633	3867	566
80		699534	25850	25633
		290084	5266	700
100			49366	48534
			6784	883

Table 3. Multiplication of symbolic matrices

There remains a need for a total algorithm for matrix inversion based on block decomposition. One has already been proposed for numeric problems [12], specifically for stable performance on floating-point number representations. It is based on pivoting, actually Crout's formulation of Gaussian elimination as presented by Knuth [6], and has the unique feature of providing full pivoting (full search of uneliminated elements) at no significant cost for accessing memory—especially in sparse matrices. It can be extended to symbolic/exact arithmetic by replacing the local maximizaton (intended for floating point stability) with a local (non-zero) minimizaton (to scale accumulated denominators).

Size	Dense	Triangular	Tridiagonal	Diagonal
4	150	83	66	66
	183	116	150	66
5	166	133	100	66
	533	250	283	116
6	300	166	150	100
	650	300	300	116
8	550	316	216	250
	1283	466	400	133
10	1066	533	450	233
	2600	783	700	183
12	4766	833	633	366
	4250	1250	916	183
20	6817	3000	2450	1383
	18034	4333	1816	350
30	21683	8150	5833	4033
	59984	12684	2866	516
40	49500	17167	11933	8417
	139000	27750	3966	700
50	94816	30767	21166	15150
	271767	52533	5433	916
60	161866	50084	32083	23850
	464233	87333	6367	1133
80	376167	108800	64550	53183
	1091166	199183	8817	1666
100	727000	200417	111333	92500
	2128817	380484	11333	2050

Table 4. Multiplication of integer matrices

While this algorithm has an elegant quadrant decomposition, it does not exhibit the divide-and-conquer behavior of inversion by partitioning, as we usually expect of tree decompositions. As published, this algorithm still requires *n* successive pivot steps to invert a matrix of order *n*. We would like to reduce the number or the order of the full pivots.

A hint of the desired improvement has already appeared [13], but the goal is briefly described here. Rather than pivoting on elementary elements, better performance will be obtained from pivoting on whole blocks. Thus, if we could find a non singular subquadrant of order near $\sqrt{n}$, then only $\sqrt{n}$ such pivots would be necessary.

A useful way to find these subquadrants is to compute determinants, where that can be easily done. Not only do non-zero determinants identify candidate pivot blocks, but also their magnitudes can be used to choose among several candidates, scaling exact arithmetic or sustaining floating point stability. If the determinant computation becomes too difficult, then we treat the quadrant as if it were singular: don't pivot there! Those matrices whose determinants we can afford to compute are called *affordable*. We are, therefore, interested in affordably non-singular quadrants.

Quadrants of order 1 and 2 are certainly affordable (as Bareiss and others have noticed.) Matrices of order 4 are affordable if one of their quadrants is zero, but more generally Sylvester's identity can be used to construct a divide-and-conquer (partial) algorithm for determinants, allowing other larger matrices to be affordable. This computation becomes less affordable just as *some* of their non-trivial quadrants (of order more than 4) are affordably non-singular. In those cases, it might appear that this effort will be lost when all larger quadrants are non-singular, but even when the bottom-up computation breaks down, many non-trivial pivot blocks will, nevertheless, have been identified.

Size	Dense	Triangular	Tridiagonal	Diagonal
4	7933	933	2516	383
	19800	300	3200	83
5	76517	1733	6300	566
	468233	950	8766	83
6		4050	16416	883
		1733	23283	116
8		20650	94050	2083
		5483	196417	116
10		102800	531950	3966
		21400	1275433	100
12		514083		6866
		55916		150
20				35917
				183
30				142484
				266
40				392434
				333
50				878766
				416
60				1709416
				483

Table 5. Inversion of symbolic matrices

In this way, the recursive computation (decoration [12]) using determinants offers a hybrid algorithm which can pivot on smaller quadrants or invert on an entire partition, depending upon which quadrants turn out to be affordably non-singular. Moreover the decomposition of Sylvester's identity makes it attractive for a multiprocessing environment.

Section 7. Conclusions

The commonly used linear storage for matrices is not suitable for implementing algorithms based on partitioning. This representation thus deprives us of exploiting many matrix theory relations which can be the basis of divide-and-conquer algorithms. If the underlying programming system uses the heap model of memory, then quadtrees offer an alternative data structure to use for representing matrices, leading to very natural, straightforward implementation of algorithms based on partitioned matrices. We have done an empirical investigation of quadtree matrices and some associated algorithms, and are very impressed with their performance compared to the conventional matrix implementations.

The quadtree structure has the appealing feature that it takes advantage of matrix sparseness without the need of any special programming. Entire blocks of zeros often do not require any storage space. Moreover, during computations, little time is spent in program steps related to those blocks. In the case of dense matrices, the quadtree structure actually requires more space than in the linear storage due to the overhead involved in storing the non-leaf nodes the tree. But even for dense matrices, it is often possible for the quadtree structure to be mc efficient with respect to computing time, because it may allow the use of an algorithm based the divide-and-conquer strategy. Our results on symbolic matrix multiplication prove tl point.

Size	Dense	Triangular	Tridiagonal	Diagonal
4	216 633	116 150	200 300	100 50
5	400 1317	266 350	300 516	183 100
6	666 4217	383 466	434 716	233 50
8	2767 17133	1050 1066	983 1416	383 100
10	6717 53133	2134 2066	2700 2284	717 150
12	13517 142467	3834 3516	5383 5933	1283 166
20	102866 1887967	20533 21550	39850 38900	4350 200
30		79400 150700	199866 296483	15967 300
40		198733 435633	476017 542283	38433 416
50		458016 1157217	1070984 1191984	82050 583
60		955417 3504084	2082250 2404150	157983 700
80				444733 916
100				1051284 1316

Table 6. Inversion of integer matrices

There is another important, less obvious computational advantage in certain manipulatic with quadtree matrices. For conventional matrices, each element in a matrix product, for exa ple, is computed by accumulating terms one by one into a sum. In quadtree matrix multipli tion, the sum is developed over a binary tree. This benefits each type of arithmetic used in co puter algebra systems: For integer addition, this is likely to postpone the invocation of *bign* calculations, if any, until at higher levels in the tree. For floating-point addition, computatic are likely to be more stable. For symbolic addition, the lookup and gathering of common sy bols is likely to be more efficient. This a partial explanation of the consistently good perf mance of quadtree multiplications in our experiments.

The earliest applications of parallel computing have been in the area of matrix compu tions. For conventional matrices, the parallelism is gained by 'array processing' in which t number of processors needed is comparable to the matrix size. In contrast, the parallelism quadtree matrix computations arises most naturally from the four-way recursive decompositi of the tree. It thus seems that quadtree matrices can offer parallel computing opportunit even in the environments where the number of processors is small and fixed.

Very large matrices derived from practical applications are usually sparse. For example, it would be rare indeed for each component of an electrical circuit to be connected to a large number of other components. Circuits usually tend to contain a small number of loosely connected components each of which is strongly connected or has a linear (ladder) structure. The corresponding matrices are overall sparse, with their non-zero entries organized into dense blocks and bands. The quadtree representation can be expected to do well with such matrices.

Even for a very sparse large matrix, it is quite possible, of course, that the blocks of zeros do not neatly align with large subquadrants. The quadtree representation in this case can be made more efficient by suitably permuting the original matrix. Algorithms to do this will be important in practice.

The quadtree structure often requires different matrix manipulation algorithms from those suitable for linearly stored matrices. (This is similar to the situation of different requirements for parallel and sequential solutions for the same problem.) We have, for example, discussed in some detail the problem of matrix inversion where entirely different algorithms are desirable to deal with the two representations. Due to the importance of matrix computations, a lot of research has been devoted to designing efficient algorithms for the conventional matrix representation. This is especially true in the field of numerical linear algebra. Development of optimal algorithms for manipulating quadtree matrices seems to be an interesting area for future research.

References

1. S. K. Abdali & D. D. Saunders. Transitive closure and related semiring properties via eliminants. *Theoretical Computer Science* 40, 2,3 (1985), 257–274.
2. E. H. Bareiss. Sylvester's identity and multistep integer-preserving Gaussian elimination. *Math. Comp.* 22}, 103 (July, 1968), 565–578.
3. V. N. Faddeeva. *Computational Methods of Linear Algebra*, Dover, New York (1959).
4. F. R. Gantmacher. *The Theory of Matrices* 1, Chelsea, New York (1960).
5. A. C. Hearn. REDUCE User's Manual, Version 3.3. Rand Publication CP78, The Rand Corp., Santa Monica, CA (July, 1987).
6. D. E. Knuth. *The Art of Computer Programming, I, Fundamental Algorithms*, 2nd Ed., Addison-Wesley, Reading, MA (1975).
7. A. C. McKellar & E. G. Coffman, Jr. Organizing matrices and matrix operations for paged memory systems. *Comm. ACM* 12, 3 (March, 1969), 153–165.
8. T. Sasaki & H. Murao. Efficient Gaussian elimination method for symbolic determinants and linear systems. In P. S. Wang (Ed.), *Proc. 1981 ACM Symp. on Symbolic and Algebraic Computation*, ACM Order No. 505810 (August, 1981), 155–159.
9. M. K. Sridhar. A new algorithm for parallel solutions of linear equations. *Inf. Proc. Lett.* 24, (April, 1987), 407–412.

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

11. D. S. Wise. Representing matrices as quadtrees for parallel processors (extended abstract). *ACM SIGSAM Bulletin* 18, 3 (August, 1984), 24–25.

12. D. S. Wise. 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,* IEEE Computer Society Press, Washington, 1986, pp. 92–99.

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

14. D. S. Wise & J. Franco. Costs of quadtree representation of non-dense matrices. Technical Report No. 229, Computer Science Department, Indiana University (October, 1987).

TR 241

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

358

P. Gianni (Ed.)

(International Symposium ISSAC '88
(1988 : Rome, Italy)

Symbolic and Algebraic Computation

International Symposium ISSAC '88
Rome, Italy, July 4–8, 1988
Proceedings