Parallel Decomposition of Matrix Inversion using Quadtrees

by

David S. Wise

Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 192

Parallel Decomposition of Matrix Inversion using Quadtrees

by

David S. Wise

May, 1986

This material is based on work supported by the National Science Foundation under grant number DCR 84-05241.

To appear in the Proceedings of the 1986 International Conference on Parallel Processing.

Parallel Decomposition of Matrix Inversion using Quadtrees

David S. Wise
Computer Science Department
Indiana University
Bloomington, IN 47405-4101

CR categories and Subject Descriptors:
G.1.3 [**Numerical Linear Algebra**]: Matrix inversion, Sparse and very large matrices; E.1 [**Data Structures**]: Trees; C.1.2 [**Multiple Data Stream Architectures (Multiprocessors)**]: Array and vector processors, Parallel processors; D.1.1 [**Applicative (Functional) Programming Techniques**].
General Term: Algorithms.
Additional Key Words and Phrases: Quadtrees, quaternary trees, Gaussian elimination, Pivot Step.

Abstract

The quadtree representation of matrices is explored, particularly as it admits a parallel matrix inversion algorithm. A version of Gaussian Elimination (full matrix pivoting) is described as an applicative program which minimizes process dispatch by folding the pivot search into the preceding pivot operation. The tree structure incorporates incremental decomposition (for arbitrary, but small, numbers of processors), aids in load balancing, and provides a uniform representation for both scalars and sparse matrices that eliminates all compatibility/bounds checking within the important algorithms. Like other algorithms particularly suited to larger problems (where parallelism pays off), it may be used at the higher level in a hybrid strategy, for example, over pipelined vector-processing on smaller, conventionally represented submatrices.

Section 1. Introduction

Consider a scenario of parallel or multi- processing with realistic constraints. A machine with p processors is available to implement a matrix algebra package for sparse matrices of size, say, $n \times n$. Restrictions are that $7 < p \ll n$ and that the cost to dispatch/recover a processor is significantly greater than the cost to perform simple arithmetic.

Those restrictions [12] preclude some popular solutions wherein processes are dispatched whenever a processor might be (wished) available and often on processes that are so simple as to be trivial. For an algorithm to be useful under these restrictions, it must admit isolation of *substantial* subprocesses, sufficiently high in the computation tree (of the the chosen algorithm) in order to assure that the p processors can be loaded using as few process dispatches as possible while balancing the load and avoiding duplication of effort. To do that requires identification of independent, arbitrarily large processes as high in that tree as possible—particularly for the cases where p is indeed very small.

What is really needed first is an algorithm, presenting such a tree with the suggested decomposition properties. This paper presents such an algorithm, matrix inversion via classic Gaussian elimination, over a new data structure—the quadtree representation for matrices—in a purely applicative style [3, 5]. This formulation lends itself to satisfying the restrictions set forth above, regardless of the particular values of n and p.

Applicative programs are necessarily presented as expressions without assignment statements or control statements except for (pure) function application. Such programs implicitly solve the problem of decomposition into independent processes [4] because each subexpression within the program is necessarily independent; therefore, the syntax tree may be strongly associated with the tree for process decomposition. Intermediate binding still allows intermediate results to be shared—rather than recomputed. While these matrix results are interesting and useful as they stand, an implicit goal of this paper is to encourage further study of algorithms for parallel computation through the philosophy enforced by applicative (or functional) programming style.

Gaussian elimination is hardly new, so what can be said that is really novel? Certainly, no improvement to well-studied asymptotic behavior will be offered. Three results, however, are offered that, together, promise a thoroughly practical algorithm under the envisioned constraints, whether or not their ultimate realization follows the discipline of applicative programming.

First, the relatively new idea of quadtrees for representing matrices [10, 11] is developed further, in a way that unifies our approach both to matrix/scalar algebra and to sparse/dense matrix manipulation. All scalars, x, also represent diagonal matrices, blurring the distinction between scalar and matrix, between sparse representation and dense representation. We shall see that one family of algorithms handles all pathologies.

Second is a version of matrix-inversion via Gaussian Elimination, through Pivot Step [6] with the pivot element selected as the largest candidate (in magnitude) from the entire matrix before each pivoting. Known to be most stable, this algorithm is also shown to present a pattern of parallelism. Each pivot step naturally decomposes by quadrants which, most interestingly, is a pattern suitable for the search tree for identifying the next (largest in magnitude) pivot candidate. There is, therefore, no question whether a *search* for the next pivot element should be implemented using parallelism or on a cheaply-dispatched uniprocessor.

0190-3918/86/0000/0092 $01.00 © 1986 IEEE

While such a search surely could use parallel processing, eliminating the explicit search phase by folding it into each (sparse) pivot saves search time, and better amortizes the overhead to dispatch each pivoting process.

Finally, an interesting relationship between padding and processor allocation is proposed to balance the load across independent processes. The quadtree matrix representation appears to be suited only to representation of $2^m \times 2^m$ matrices. When the size of a matrix is not a power of two, some padding is necessary which only wastes space proportional to m. What is interesting is that, having embedded an $n \times n$ matrix in (the lower right of) a $2^m \times 2^m$ one, processor allocation can make profitable use of the value of $(2^m - n)$ in partitioning the p processors among subproblems. The argument is cast in terms of familiar matrix operations.

The remainder of this paper is in four parts. The quadtree representation is introduced first, including discussions of its restriction to vectors and generalizations to higher dimensions. The second section explores a Gaussian elimination algorithm under the new parallelism, as outlined above. The third offers a strategy for allocation of processor resources to discount the padding that might be necessary to fill out the quadtree representation. The final section offers some conclusions and hopes for further work.

Section 2. Matrices

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

Matrix algorithms will be arranged so that we may (without loss) perceive any scalar, x, as a diagonal matrix of arbitrary size, entirely of zeroes except for x's on the main diagonal; that is, $x = [x\delta_{i,j}]$. Thus, a domain is postulated that coalesces scalars and matrices, with every scalar-like object conforming also as a matrix of any size. Of particular interest are 0 and 1, which are at once the *unique* additive and multiplicative identities, respectively, for scalar/matrix arithmetic. Similarly, the scalar x as a binary tree is interpreted as a vector of arbitrary length, each of whose components is x (much like Daisy's [7] notation $\langle x^* \rangle$.) Inferring the conventional meaning from such a matrix now requires additional information (*viz.* its size), but we can proceed quite far without size information; it only becomes critical upon Input or Output.

Lest it appear that this coalescing of hitherto disjoint types hides too much, it is useful to draw a analogy from ordinary computation on floating-point numbers (FPNs), where details of internal representation are also suppressed. The point is that the way that quadtree-matrices (and FPNs) are commonly represented outside the machine has little to do with their internal representation. There are, in fact, conventional styles for writing matrices on paper—in row-major order (and for writing FPNs in scientific notation), but these may differ wildly

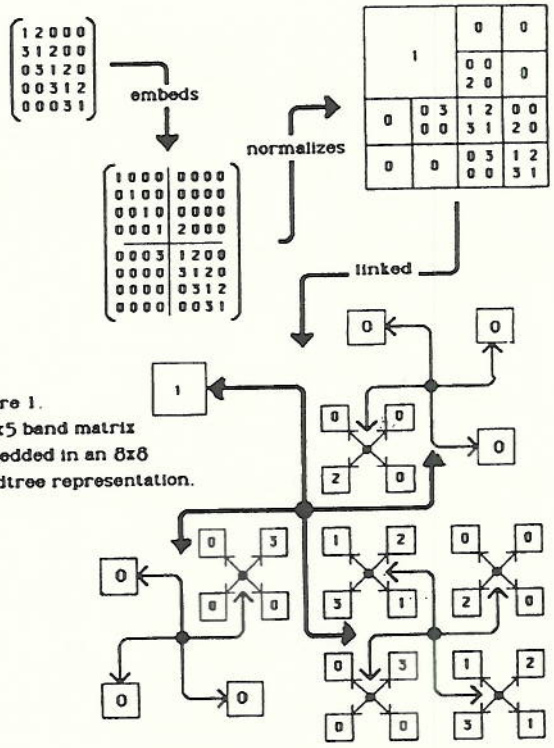

Figure 1.
A 5x5 band matrix embedded in an 8x8 quadtree representation.

from the way that they (and FPNs) are to be represented within the machine. Although the algorithms for translating between such representations are elegant, they are so complicated that they are surely not the first thing that should be shown to those unfamiliar with the nature of these internal representations. Thus, an unfortunate barrier rises in the path before those who would tinker on them: one must first write the I/O translators, which are among the least comprehensible, least efficient, and least exercised programs over the structure.

A matrix (of otherwise-known size) is either a vague 'scalar' or it is a quadruple of four equally-sized submatrices. So that this recursive cleaving works smoothly, we embed a matrix of size $n \times n$ in a $2^{\lceil \lg n \rceil} \times 2^{\lceil \lg n \rceil}$ matrix, justified at the lower, right (southeast) corner with zero padding to the north and west, except for nonzeroes—preferably ones—padded along the northwest diagonal (to avoid introducing an unnecessary singularity; see Figure 1.) The matrix is justified to the southeast, rather than the northwest, so that its eliminant [2] is properly defined.

There is also a *normal form* convention. Under this quad representation, no submatrix will ever be composed of four 'scalar' quadrants, of whom the northeast and southwest are zero, and whose northwest and southeast coincide. Such a matrix would be represented, instead, by the latter 'scalar,' standing alone. Thus, the two important identity/annihilator matrices are represented *uniquely* by 1 and 0. If we require that the northwest padding, as in the previous paragraph, is necessarily one, then a *canonical form* results.

Elsewhere [10] I observed how algorithms for matrix addition, transpose, and multiplication follow the desirable pattern of decomposition, generally into 4^m or 8^m independent processes that can be dispatched high in the computation tree, up to the capacity of the execution environment. Of note is the role of 1 and, especially, of 0 as a constituent quadrant to such an operation. When any addend's quadrant is 0, the effort for matrix addition immediately simplifies by 25% because it is, therefore, unnecessary to descend and to traverse the corresponding quadrant of the other addend; all we need is a borrowed reference to it as one quadrant of the result. A factor's quad of 1 reduces Strassen's decomposition [9] of Gaussian multiplication from eight recursive multiplications to six; not only does a 0 quad similarly annihilate two recursive multiplications, but also it avoids two of the four subsequent additions, as well.

These properties are particularly valuable for matrices with regular patterns of non-zero entries, especially those that are sparse or in diagonal form. No special code is necessary to accelerate conventional operations on them (but one can wish for hardware that accelerates specialized tests like the ubiquitous tests for 0 submatrices.) It is, however, necessary to maintain the normal form so that, like rational numbers being always "reduced to lowest terms", matrices are reduced to their corresponding scalars whenever zero southwest\northeast quadrants and coincident northwest/southeast quadrants permit.

Another advantage shows up upon deeper study of several matrix manipulation programs over quadtrees: algorithms written to accept canonical-form operands need not be sensitive to the usual compatibility requirements. That is, ordinary quadtree algorithms for various operations will work regardless of the depth of their tree/operands; when operands are of different depth, the shorter paths—ending at a scalar—will be interpreted as all-conforming, diagonal matrices. Although their meaning may be questionable, the algorithms will run to completion instead of crashing with array-bounds-violations. In terms of domain theory, we have raised these operations to a higher point in their respective function-domain (given them more meaning by defining results where others fail because of incompatibility), while simplifying the code (by eliminating all the incompatibility tests and signaling thereof).

A "header" above each matrix quadtree might usefully contain two values needed for output translation: the length of the diagonal padding, and the exponent, m, for a $2^m \times 2^m$ matrix. The value of m also suffices for run-time compatibility checking. Another bit there indicates whether the quadtree is to be interpreted as transposed, recursively interchanging southwest\northeast quadrants upon any access. Thus, not only does quadtree representation allow us to transpose an entire matrix in constant time—at the cost of building a new header—but also it allows row and column traversal at equally high efficiency, at the cost of symmetric-order traversal [6] of the appropriately projected binary tree.

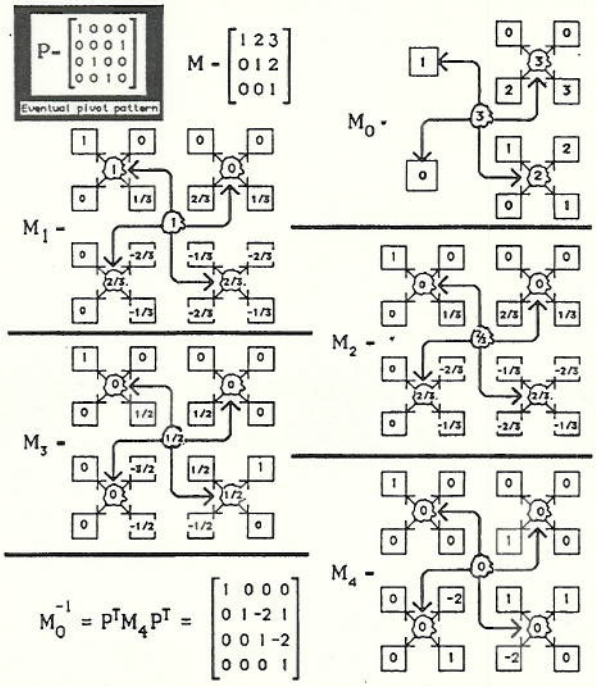

Figure 2. Knuth's Problem 2.2.6-18 [6, p. 556], worked.

Section 3. Pivot Step and Inversion

This section describes an algorithm for matrix inversion, extended from Knuth's [6] for an entirely different data structure (also sparse). His terminology is used because it is readily available. Figure 2 outlines this algorithm applied to his problem, and should be compared against the solution that he provides. The algorithm is Gaussian elimination with full pivoting (although Knuth only selects pivots rowwise.) Much of the description applies to a single Pivot Step, but its most interesting property relates one such step to the next. It is easily simplified to one that only finds the root of the linear equation, $Ax = y$, for Matrix A and Vector y.

Let a non-singular matrix, M, be represented with a quadtree as described in the previous section. Each non-terminal node, however, is to be decorated with additional information: the magnitude of the largest element in that quadrant, and its local horizontal and vertical coordinates. (It will be necessary to qualify the decoration further to exclude any element in an already-pivoted row or column from decorations. At this point, however, assume that no pivoting has yet occurred.) These coordinates need not be traditional indices—though it is easy to think of them that way. A cheaper implementation is just a pair of bits at each node selecting one subtree; the catenation of these subtrees identifies a path through each quadtree to the appropriately largest scalar.

Let us consider an algorithm to invert M, a matrix represented as a canonical-form quadtree, padded along its northwest diagonal as described in the previous section. The first step in computing the inverse of M, therefore, is to traverse its tree representation in postorder, installing these decorations at all internal nodes. (Sibling subtrees, of course, may be traversed in parallel.) Call the decorated matrix M_0. Decorations appear in florets in Figure 2; to save space, however, only the local maxima (no local coordinates) are shown.

Also needed are two trivial binary trees, each initially the scalar 0 indicating a boolean vector of all zeroes, and one trivial quadtree, P_0, also initially 0, similarly. For each i, Quadtree P_i, indicates the exact position of the first i pivot elements—none so far; it will be filled in to become a permutation matrix, P_{2^m}, by the time the fully pivoted matrix, M_{2^m}, is computed. The boolean vectors indicate which rows/columns have already been pivoted; they will be filled with 1's until they both indicate that all rows/columns have been eliminated. The two vectors, therefore, are merely row (column) projections from the corresponding P_i, but in a format useful for directing subsequent decorations.

Having established the initial values of M_i and P_i, for the next of 2^m pivot steps, we discover that the decoration at the root of this tree identifies the next pivot element. We presume here that M_i is not a scalar. If it were (and thereby identified as the pivot scalar), then its reciprocal is the pivoted matrix.

Otherwise, the M_i may be decomposed into quadrants, each distinguished by the decoration. One is *pivotquad*, distinguished because the pivot element lies within. Another is *rowquad*, named because it lies horizontally from *pivotquad*. The third is *columnquad*, because it lies above or below *pivotquad*. The last is *offquad*, so called because it does not coordinate on *pivotquad*, but lies diagonally from it.

The description that follows discusses the transformations on the four quadrants in reverse order from that just above. It turns out that *offquad*'s transformation is simplest, but it does depend on intermediate results derived from processing the other three quadrants. So many additional results are needed from handling *pivotquad*, moreover, that its description will be considerably eased by working backwards in order to justify them.

An important property of functions is that one invocation may return several results of differing types. This feature has been lost in many programming languages (perhaps because their designs presume that a computer has but one accumulator,) but it is critical to applicative style and allows one function invocation to return all these intermediate results.

Knuth's transformation of the matrix,

$$\begin{pmatrix} & \vdots & & \vdots & \\ \cdots & a & \cdots & b & \cdots \\ & \vdots & & \vdots & \\ \cdots & c & \cdots & d & \cdots \\ & \vdots & & \vdots & \end{pmatrix}$$

where a is the pivot element, b is any element in the pivot row, c is any element in the pivot column, and d is any off pivot element coordinating on b and c, is

$$\begin{pmatrix} & \vdots & & \vdots & \\ \cdots & \frac{1}{a} & \cdots & \frac{b}{a} & \cdots \\ & \vdots & & \vdots & \\ \cdots & \frac{-c}{a} & \cdots & d-\frac{bc}{a} & \cdots \\ & \vdots & & \vdots & \end{pmatrix}$$

respectively. This is the transformation to be made, though it is to be done here recursively and (likely) in parallel.

The transformation of *offquad* requires the values of d contained therein, and two vectors of values b/a and c coordinating on each d, occurring in the pivot row and column, respectively. Since these two vectors, represented as binary trees in normal form, occur in *rowquad* and *columnquad* they will have been extracted as intermediate results while processing those quadrants. If either of these vectors is zero, however, then the transformation collapses to the identity function; all values of $bc/a = 0$ and not even the internal decorations in *offquad* change, as neither the newly eliminated pivot row nor pivot column cross it. (This is *the* savings of sparse representation for Pivot Step.)

When *offquad* is a scalar, d—even if it is a normalized representation of a larger matrix (notably if $d = 0$) and b/a and c are vectors—then the correct transformation is the decorated form of d's difference with their outer product, $d - bc/a$. In order to decorate the difference, the appropriate halves of the boolean vectors identifying rows and columns eliminated from M_i are necessary parameters. (If d is scalar and these vectors are non-trivial, then d must be expanded into $\begin{pmatrix} d & 0 \\ 0 & d \end{pmatrix}$ and decomposed, as below.) When $d - bc/a$ is a scalar, either zero or its absolute value becomes its decoration, depending on whether it lies in an already pivoted row/column, or not.

When *offquad* is decomposed into four quadrants, then each of these is treated as an *offquad*, with the vector parameters cleaved in half to provide the four sets of vector arguments, and the four results decorated and normalized into the transformed quadtree.

The treatment of *rowquad* and *culumnquad* are similar, so only that of the former is presented here; the latter's is nearly dual to what follows. The treatment of *rowquad* requires the inverse of the pivot element, a, and a vector that is the portion of the pivot column that lies in *pivotquad*, but for the pivot element, itself. It is sufficient to represent both as a copy of the pivot-column vector with $1/a$ in place of the pivot element (which will have been extracted during the treatment of *pivotquad*). Also needed are a relative index locating the pivot row and halves from the boolean vectors indicating which rows/columns crossing *rowquad* have already been eliminated.

Results are the transformed *rowquad* and half of the pivot row (as a binary tree) extracted from *rowquad* for use in handling *offquad*. That vector is also needed for handling most of *rowquad*, itself because, unless *rowquad* is trivial, it must be decomposed into two sub*rowquad*s and two sub*offquad*s, the latter of which coordinates on that half of the pivot row.

Thus, the transformation of *rowquad* focuses first on the pivot row. When the row index indicates that *rowquad*– call it b— is entirely within the pivot row, then the residue column vector is just $1/a$, and the needed results (matrix and row-vector) are each the product, b/a. Decorated as a matrix, the local maximum is 0 because this row is being eliminated.

If *rowquad* is to be decomposed into submatrices, then the local index will indicate whether the pivot row crosses the upper or lower half. Accordingly, the pivot column and boolean vectors (identifying already eliminated rows/columns) are split, and those two containing the pivot row are treated (as *rowquad*s) first. The pieces of the pivot row, extracted thereby, are joined at a binary node to become the vector-result of treating *rowquad, and* are passed as arguments with the other two quadrants for treatment as *offquad*s. Then these four quadrants are assembled and decorated into the matrix-result of this treatment.

Finally, we consider the treatment of *pivotquad*, upon which all the other three quadrants' treatments depend; it must occur first and yield as a partial result the transformed, decorated version of *pivotquad*; and as intermediate results: the row and column indices of the pivot element, and vector copies of the pivot column (with $1/a$ in place of the pivot element) and pivot row (with $-1/a$ in place of the pivot element). Moreover, while locating the pivot element, updated versions for P_i, the permutation matrix (which will only change in the quadrant corresponding to *pivotquad*), and for its projections, the two boolean vectors of eliminated rows/columns should be constructed. That's eight results, four of which are intermediate—not to be included directly in an answer, but to be used in treating sibling quadrants.

Three observations complete this description. First, the treatment of *pivotquad* is the same as the treatment of the whole matrix, M_i; the only difference is the unneeded, extra four results, beyond M_{i+1}, P_{i+1}, and the two boolean vectors projected therefrom. Secondly, the arguments to each Pivot Step (and pivoting successive *pivotquad*s) are M_i, P_i, and the two boolean projections from P_i (and the corresponding quadrants/halves therefrom.) The last three arguments are used as seeds for updated results, and the last two also help to place 0 decorations.

Finally, if M_i (correspondingly, *pivotquad*) is a scalar, a, then all eight results are trivial: $M_{i+1} = 1/a$ and is decorated as 0 (now that both it's row and column have been eliminated); $P_{i+1} = 1$ and both its projections are 1, also; the pivot column is $1/a$, and the pivot row is $-1/a$; and both relative indices are 1.

When M_i is not scalar, it decomposes into four quadrants, one of each type considered above. Algorithms for three of them (*rowquad, columnquad, offquad*) have been discussed above, and the fourth (*pivotquad*) is to be treated recursive as a Pivot Step, with basis stated the preceding paragraph.

The **parallelism** in this algorithm manifests itself in the interdependence of these recursive decompositions. For instance, treatment of successive candidates for *pivotquad* must precede transformation of all other quadrants; the depth of this recursion is at most m. After each is completed, its associated *rowquad* and *columnquad* may be dispatched simultaneously, and, of these, half must be transformed before the other half. Again, the depth of recursion is at most m. Thereafter, however, all *offquad*s at all levels of the quadtree may be treated *simultaneously.* They generate most of the effort in a Pivot Step, but and thier transformations are mutually independent.

It should be clear that a pivot step can change lots of decorations from those in M_i; is there, then, sufficient information to restore them? Yes, because decorations will only change where scalar values have changed, or because a local maximum is disqualified because it resides in either the current pivot row or current pivot column. Such decorations have already been visited by this algorithm! (This point is most important when inverting sparse matrices, where little traversal is necessary.) We need only arrange that, as each interior node in the quadtree (that becomes the pivoted matrix) is reassembled, it must be redecorated with the appropriate maximum and local coordinates. Therefore, the position of each scalar encountered is resolved against the boolean vector indicating already-eliminated rows and columns; if it is to be excluded, treat its magnitude as zero *for the purposes of finding the maximum local magnitude.* If all four local maxima are zero, then the subtree is decorated with zero (and the local coordinates may be left undefined.)

As the four new quadrants are reassembled, it is necessary to find the maximum of their decorations and its two-bit coordinates, according to which of the four quadrants it came from. Internal zero decorations do not propagate, because some decoration must be positive if the original matrix was non-singular.

That completes the description of a single pivot step, $M_i \mapsto M_{i+1}$. It only remains to observe that the results of one step, including the pivoted, decorated matrix, the two vectors (binary trees) of eliminated rows and columns, and the building permutation matrix are passed from one step directly along to the next. *There is no need to search for the next pivot element,* because it has already been located. Moreover, it has been located by parallel processes already dispatched for the pivot step, itself, in parallel. Thus, there is no dispatch/recovery overhead for the parallel search!

Finally, observe that the desired inverse, M^{-1} is readily available after permutations:

$$M^{-1} = P_{2^m}^T \times M_{2^m} \times P_{2^m}^T.$$

In fact, the code in the appendix builds up P_i^T, rather than P_i, anticipating this transpose..

This entire algorithm proceeds on non-singular matrices without any counters; even the outer control over of 2^m successive pivot steps may be set up as a loop until decoration becomes zero. In some sense, then, it is more abstract, and more useful, than algorithms that depend heavily on size declarations and bounds tests.

Section 4. Subprocess Balancing

Suppose an $n \times n$ matrix is embedded in a $2^m \times 2^m$ matrix, where $k = 2^m - n$ and $0 \leq k < 2^{m-1}$ as in Figure 3. Section 2 suggested that the values of k and m might need to be available to the system at run-time, even though they remain unnecessary to (in particular) the abstract multiplicative operators. This section proposes another use for this same information: load balancing among the processors.

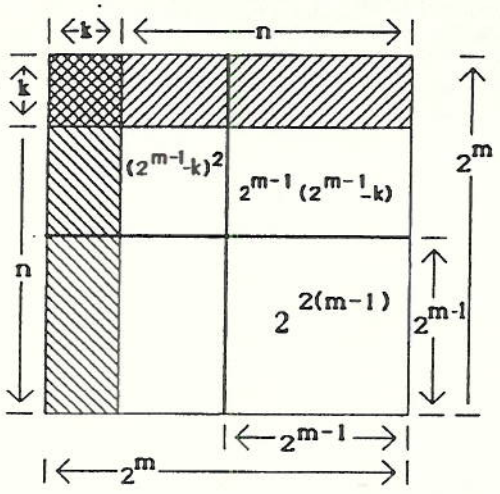

Figure 3. Areas within quadrants, excluding padding of k rows/columns.

The following discussion does not depend on denseness or sparseness of matrices. It does presume that the distribution of non-zero entries is uniform; if more patterns are known, then further inferences might be possible. The values of m and k indicate the size of a matrix and what proportion of it is trivial. From them we can determine what portion of each of the four quadrants is serious, *i.e.* likely to cause serious effort for the processors, and we can use this information to distribute the p processors among the four quadrants.

Consider matrix addition, for instance. The quad recursion pattern is simple; each quadrant requires addition effort in proportion to its "serious" area (Figure 3). The serious area of such a matrix is $(2^m - k)^2$. It is divided up among its four quadrants in the following proportions:

Quadrant	Relative share of area
Northwest	$\frac{2^{m-1} - k)^2}{(2^m - k)^2}$
Northeast	$\frac{2^{m-1}(2^{m-1} - k)}{(2^m - k)^2}$
Southwest	$\frac{2^{m-1}(2^{m-1} - k)^2}{(2^m - k)^2}$
Southeast	$\frac{2^{2(m-1)}}{(2^m - k)^2}.$

It is unlikely that these proportions have integer products with p. If only a fraction of a processor is available to a quadrant, a good solution is to combine quadrants on shared processors in a way that the individual requests for a processor sum to an integer. A likely grouping is to set the northwest, northeast, and southwest sums (the partial quadrants) on one processor, and to set the southeast sum alone on another.

Therefore, if one had p processors to add such matrices, one could use this information to distribute them to four quadrant process in these proportions in a top-down pattern. As mentioned in the introduction, it is important that processor allocation be done as high in the data structure/computation tree as possible, so that the overhead of process dispatch/recovery not be paid repeatedly.

In this way it is likely that larger quadrants (southeast) gets more processors. When a quadrant receives but one processor to compute its result, it operates as a uniprocessor; if it receives more than one, it can apply the same idea to divide up its processor resource once again, and so forth. The first three quadrants make the most interesting further processor allocations; the southeast quadrant is presumed to be uniform and so its share will just be divided in even fourths.

These same proportions could apply to the Pivot Step algorithm above. Unlike addition, however, we saw that there was some serial behavior to Pivot Step. Thus, (using terminology from before) *rowquad* and *columnquad* may be

dispatched together, but they are only two of four quadrants. Nevertheless, their computational effort may also be approximated by the relative size of two areas (diagonal from each other), in the table above. Although we may not know which processors until run time, we may select the proper proportion then and split the processor resources in that manner.

Gaussian matrix multiplication (under Strassen's formulation [9]) easily decomposes into eight products, which are pairwise summed to form a four-quadrant answer. Excluding the effect of scalars (*i.e.* a quadrant of 1 or 0 avoids a quadrant multiplication) and asserting that the $O(n^3)$ algorithm does require a processor resource proportional to $(2^m - k)^3$, we determine the proportion of this resource that each of the eight products needs. This was done by setting two matrices, as in Figure 3, as multipliers and extrapolating the effort to multiply each of eight sub-products from the areas of the sub-multipliers (quadrants.)

The eight ratios are best described by combining them pairwise, as their associated products are to be added, to yield the proportion of effort invested in building each of the four sums that become the matrix product. The four ratios coincide with those already derived above! Furthermore, proportions associated with each of the eight products may be obtained by multiplying each of these four ratios by $2^{m-1}/(2^m - k)$ and by $(2^{m-1} - k)/(2^m - k)$, respectively. Thus, the four-quadrant ratios are exactly as before, and the eight-quadrant ratios are uniform extensions from these.

Such process allocation could be determined statically at compile time [8] when the language requires matrices to be of declared, constant size and uniform sparseness. The algorithms proposed here do not require bounds declarations, but if they were available, it would be possible to avoid much communication with, and system saturation of, a dynamic scheduler.

Section 5. Conclusions

Because we assumed that the number of processors, p, is small compared to the size of the matrix, n, we need to cleave matrix manipulations into a *few* subprocesses of *balanced* size, so that the resource p can be allocated deliberately. If one cleaving does not consume all the processes, the pieces may be further split. Avoiding repeated dispatch and recovery, these algorithms have the virtue of splitting at the base of the quadtree—at the root of the problem.

Although these algorithms are described as if only scalars could be leaves of the quadtree, that arrangement is not necessary. It is perfectly possible that sizable matrices dwell at the leaves, matrices that might be represented in traditional row-major order, and manipulated using traditional iterative and pipelining algorithms, programmed in FORTRAN *et fils*. (Pipelines would need an extra input stream of bits, identifying candidates for updated local maximum, and each could then compute updated vector decorations during a vector update.) The size of such leaf matrices should be chosen to balance the efficiencies of existing style and existing machinery against the obvious need for multiprocessing techniques to accelerate large matrix computation.

Derived from a purely applicative approach, they better suit a computing environment with many processors, with memory banked at varying distances from the different processors, and with contention for access to shared resources as a real constraint on efficiency. The quadtree approach shows us how to represent matrices so that useful pieces may be localized where some processor can make computational headway on the problem without excessive interference from all its brethren. It has long been known [4] that applicative style solves the problem of decomposing an extant algorithm; the problem addressed here is the discovery of good algorithms that cleave into usefully sized pieces.

One last issue raised by this work relates to all sparse matrix techniques. I have seen only vague definitions of a what makes matrices *sparse*, possibly because sparse representations are so different from one–another, and maybe because sparseness has been purely an operational concept. Quadtrees, as we have seen, offer a reasonable representation for sparse matrices that is consistent with what we would use for dense matrices; based on that observation, *alone*, I suggest a **measure of sparseness** for matrices: the ratio between its average path length in its quadtree representation (from root to leaf) and the logarithm of its size. Ratios closer to zero indicate sparser matrices..

Quadtree representation of matrices was motivated by studies in applicative programming and as part of an effort to study its impact on Matrix Algebra for MIMD multiprocessors. The results of the effort are more than satisfactory: not only does the technique apply to a well-worked problem, but also it yields new insight (through a distribution of searching across Pivot Step) on optimal hardware/software solutions. Thus, there we have more support for using applicative programming (and its algebra) for programming parallel architectures and a suggestion that those architectures provide a multi-banked heap.

There is already interest using quadtrees as *the* uniform representation for matrices in a large computer algebra system [1], a sophisticated piece of software running on fairly conventional hardware. They also have an important role to play within conventional languages used on very sophisticated hardware, an experiment that remains to be done.

Acknowledgement: Research reported herein was sponsored, in part, by the National Science Foundation under Grant Number DCR 84-05241. I would like to thank John Lash and Kamal Abdali for ideas and encouragement that helped these ideas mature.

References

1. S. K. Abdali. Personal communication (1985).

2. S. K. Abdali. & D. D. Saunders. Transitive closure and related semiring properties via eliminants. *Theoretical Computer Science* **40**, 2,3 (1985), 257-274.

3. J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM* **21**, 8 (August, 1978), 613-641.

4. D. P. Friedman & D. S. Wise. Aspects of applicative programming for parallel processing. *IEEE Trans. Comput.* **C-27**, 4 (April, 1978), 289-296. Preliminary version appeared as: The impact of applicative programming on multiprocessing. *Proc. 1976 International Conference on Parallel Processing*, 263-272.

5. S. D. Johnson. *Synthesis of Digital Designs from Recursion Equations*, M.I.T. Press, Cambridge, MA (1984).

6. D. E. Knuth. *The Art of Computer Programming, I, Fundamental Algorithms*, 2nd Ed., Addison-Wesley, Reading, MA (1975), 299-318 + 401, 556.

7. A.T. Kohlstaedt. Daisy 1.0 Reference Manual. Tech. Rept. 119, Computer Science Dept., Indiana University (November, 1981).

8. V. Sarkar & J. Hennessy. Compile-time partitioning and scheduling of parallel programs. it Proc. SIGPLAN 86 Symp. on Compiler Construction, SIGPLAN Notices **21**, to appear.

9. V. Strassen. Gaussian elimination is not optimal. *Numer. Math.* **13**, 4 (August 19, 1969), 354-356.

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

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

12. M. F. Young. A functional language and modular arithmetic for scientific computing. In Jean-Pierre Jouannaud (ed.), *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science* **201**, Berlin, Springer (1985), 305-318.

Appendix

The following examples, all of which evaluate to 1, are useful as an introduction to the Daisy [7] code that follows. They exemplify a new style of applicative programming that depends on functional combination and *data recursion* [5] to specify multiple and interdependent results **without cognizance of a necessary sequence of evaluation.** All primitives used here, however, have been in Daisy from its birth. The forms

```
let:[  identifierStruc  binding  result]
rec:[  identifierStruc  binding  result]
```

return *result* computed in an environment enhanced with *identifierStruc* bound to *binding*. Evaluation is lazy, and the list structure of *binding* must match the structure of *identifierStruc*, wherein bound identifiers become bound according to their position within that list. Functional combination [4] is indicated by a list structure as a function, to the left of the colon, the "apply" operation. In the layout below, one may perceive that the constituent components of that combination is applied vertically to the (transposed) argument matrix. Thus, the intermediate results of all the functional combinations below is that of ⟨10⟨3 1⟩⟩.

```
let:[   [sum     [quotient        remainder]]
        <add     <div             rem      >>:<
        <6       <        10      *        >>
        <4       <        3       *        >> >
    remainder]

divide = ^\[a b].<div:<a b> rem:<a b>>
let:[   [sum     [quotient        remainder]]
        <add             divide           >:<
        <6               10               >
        <4               3                > >
    remainder]

rec:[   [sum     [quotient        remainder]]
        <add             divide           >:<
        <6               sum              >
        <4               3                > >
    remainder]
```

The skeleton of the Daisy code for Pivot Step follows. It presumes that quadtree-matrix arguments/results are decorated at non-terminal nodes. This code has been cut back to remove all provision for normalized (sparse matrix) representation and for permuting the quadrants. Normalized matrices may require expansion of scalar arguments into a quadruple (two of which are zero; two of which are the scalar) upon function entry, and collapse of such quadruple patterns to the scalar upon exit. Without provision for permuting, this code will pivot only on the northwest-most (upper left) scalar entry; the expanded code tests two bits of "decoration" (*ibit* and *jbit*), and provides permutations on arguments and results to translate to/from this northwest orientation.

Notice that the instances of functional combination in *PIVOT*, *ROW*, *COLUMN*, and *OFF* are

$$[PIVOT\ ROW\ COLUMN\ OFF],$$

$$[ROW\ OFF\ ROW\ OFF],$$

$$[COLUMN\ OFF\ COLUMN\ OFF],$$

$$[OFF\ OFF\ OFF\ OFF],$$

respectively, reflecting the recursive decomposition of each kind of quadrant. It is also important that *OFF* immediately tests whether either vector argument (the projection from the pivot row or pivot column) is zero, and acts as an identity function in that instance. Also, this code builds the transpose of the permutation matrix (P_i^T from the paper) directly.

```
PIVOT = ^λ[decoratedMtx elimrow-col PermutT]. if:<
        Scalar?:decoratedMtx    let:[ inverse  reciprocal:decoratedMtx
            < inverse [TRUE TRUE] 1 <negate:inverse inverse> [1 1]> ]
        let:[  [[max ibit jbit] ! mtx]
               decoratedMtx
        let:[[ [epivot [] [] [eright ebot]]      [permutHEAD ! permutTAIL] ]
             <       spread4:elimrow-col                PermutT          >
        rec:[
            [   [i [eleft etop] permutPIVOT [pleft ptop] [ipos jpos]     ]
                                [ii pright]     [iii pbot]      iv       ]
                <PIVOT          ROW             COLUMN          OFF>:    <
                                      mtx
                <epivot         <eright etop>   <eleft ebot>    <eright ebot>>
                <permutHEAD     ptop            pleft           <pright pbot>>
                <[]             ipos            jpos            []  >    >
        let:[   [elimrow-col pivotrow-col]
                <<<eleft eright> <etop ebot>>   <<pleft pright> <ptop pbot>> >
        <decorate:<elimrow-col   <i ii iii iv> >
            elimrow-col         <permutPIVOT ! permutTAIL>
            pivotrow-col        <twice:ipos twice:jpos> >       ]]]]>


ROW = ^λ[decoratedMtx elimrow-col pivcol index]. if:<
        one?:index let:[ BoverA
                         decorateproduct:<elimrow-col pivcol decoratedMtx>
                         <BoverA BoverA> ]
        let:[   [iresidue ibit]         divide:<index 2>
            rec:[ [ [i left]    [ii right]      iii         iv          ]
                    <ROW        ROW             OFF         OFF         >:<
                            tail:decoratedMtx
                            spread4:elimrow-col
                                                            let:[ [top bot] pivcol
                    <top        top             <bot left> <bot right> >]
                    <iresidue   iresidue        iresidue    iresidue    > >
                < decorate:<elimrow-col <i ii iii iv>>  <left right> > ]]>

COLUMN = ^λ[decoratedMtx elimrow-col prow index]. if:<
        one?:index
            <decorateproduct:<elimrow-col prow decoratedMtx> decoratedMtx>
        let:[   [jresidue jbit]         divide:<index 2>
            rec:[ [ [i top]     ii          [iii bot]   iv          ]
                    <COLUMN     OFF         COLUMN      OFF         >:   <
                            tail:decoratedMtx
                            spread4:elimrow-col
                                                        let:[[left right] prow
                    <left       <top right> left        <bot right>>]
                    <jresidue   jresidue    jresidue    jresidue    >    >
                < decorate:<elimrow-col <i ii iii iv>>      <top  bot  > > ]]>

OFF = ^λ[decoratedMtx elimrow-col prow-col index]. if:<
        anyzero?:prow-col       decoratedMtx
        one?:index decoratedifferencE: <elimrow-col decoratedMtx
                                        decorateproduct:<elimrow-col prow-col> >
        decorate:<elimrow-col
                    <OFF        OFF         OFF         OFF         >:    <
                            tail:decoratedMtx
                            spread4:elimrow-col
                            spread4:prow-col
                    <half:index half:index  half:index  half:index>     > >>
```

SN 0190-3918
EE Computer Society Order Number 724
brary of Congress Number 79-640377
EE Catalog Number 86CH2355-6
BN 0-8186-0724-6

PROCEEDINGS

OF THE

1986 INTERNATIONAL CONFERENCE

ON

PARALLEL PROCESSING

August 19–22, 1986

Editors
Kai Hwang
Steven M. Jacobs
Earl E. Swartzlander

Co-Sponsored by

Department of Electrical Engineering
PENN STATE UNIVERSITY
University Park, Pennsylvania

and the

IEEE Computer Society
In Cooperation with the

Association for Computing Machinery

IEEE COMPUTER SOCIETY

IEEE
THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

COMPUTER SOCIETY PRESS

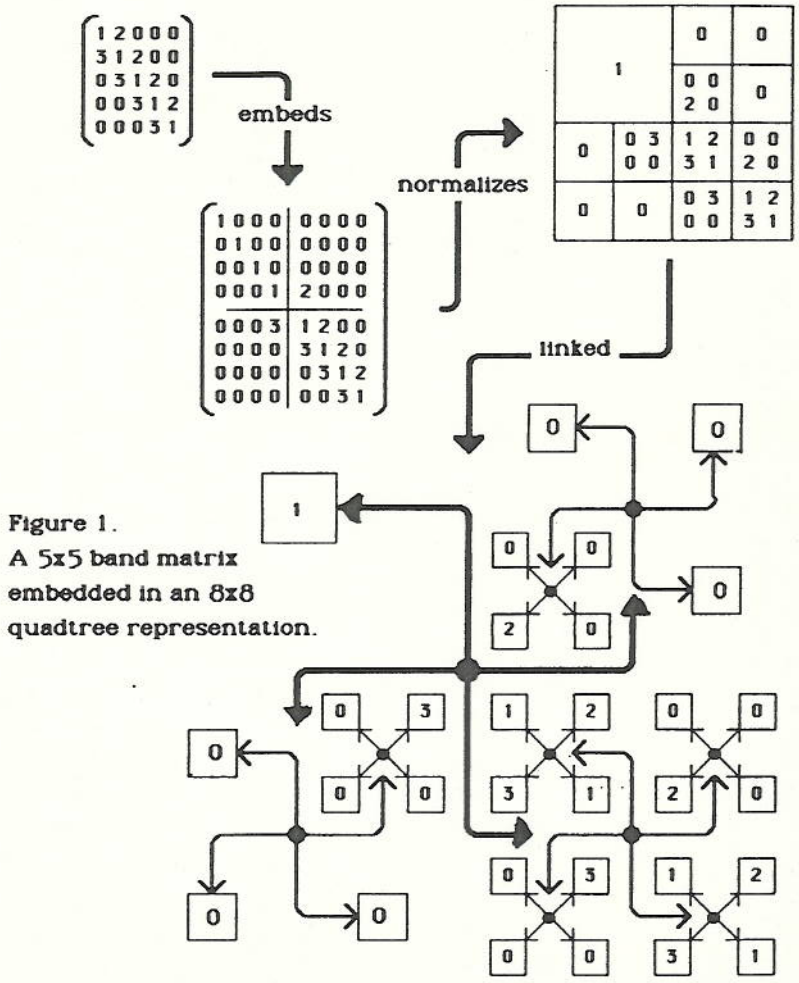

Figure 1.
A 5x5 band matrix
embedded in an 8x8
quadtree representation.

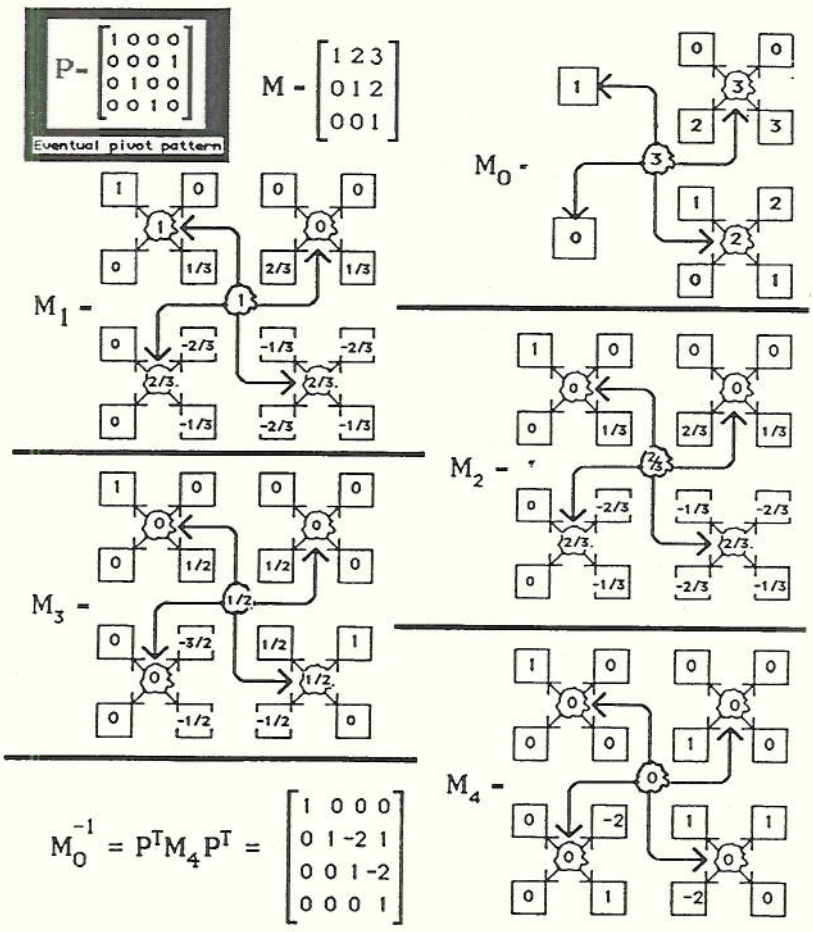

Figure 2. Knuth's Problem 2.2.6-18 [6, p. 556], worked.

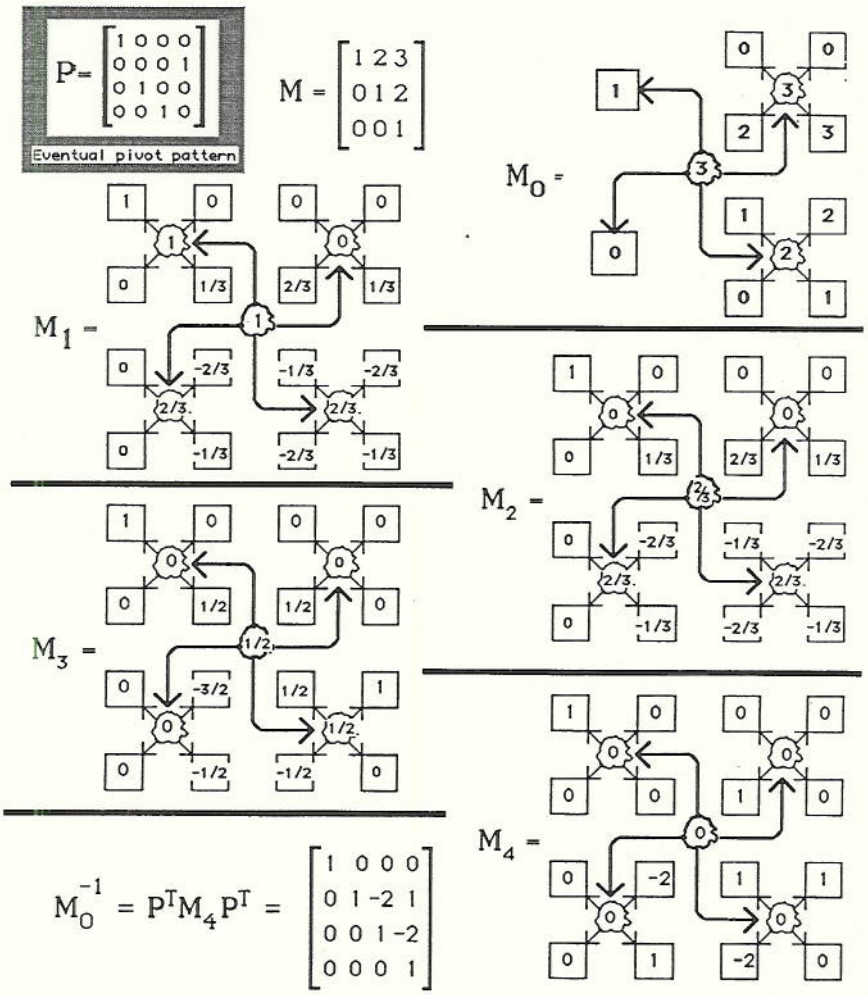

Figure 2. Knuth's Problem 2.2.6-18 [6, p. 556], worked.

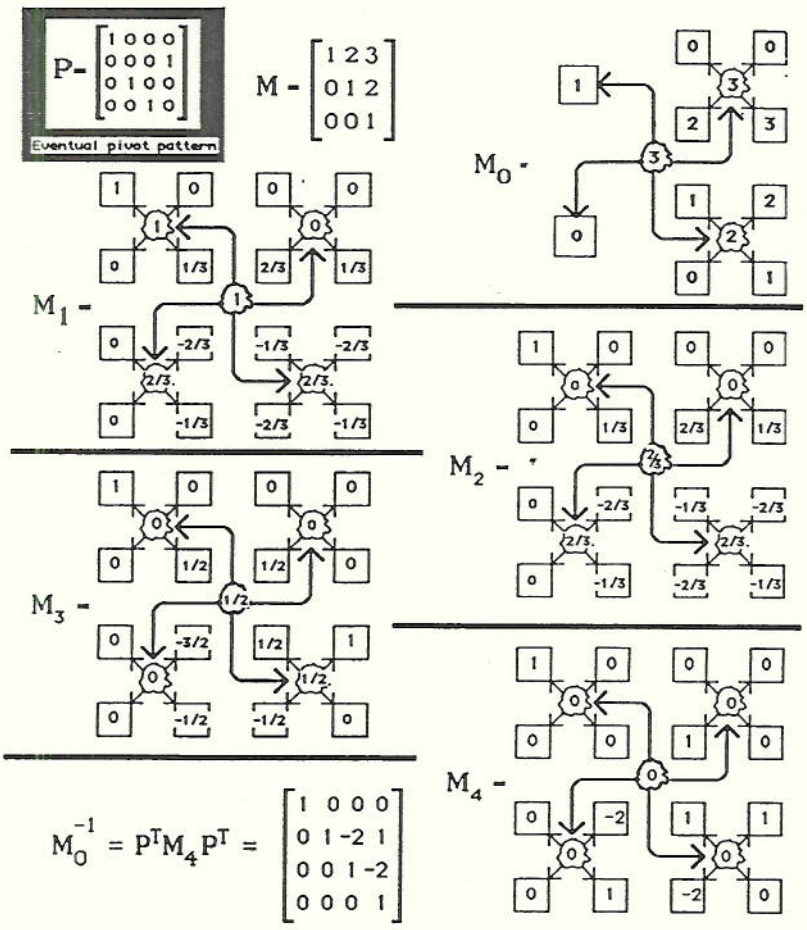

Figure 2. Knuth's Problem 2.2.6-18 [6, p. 556], worked.

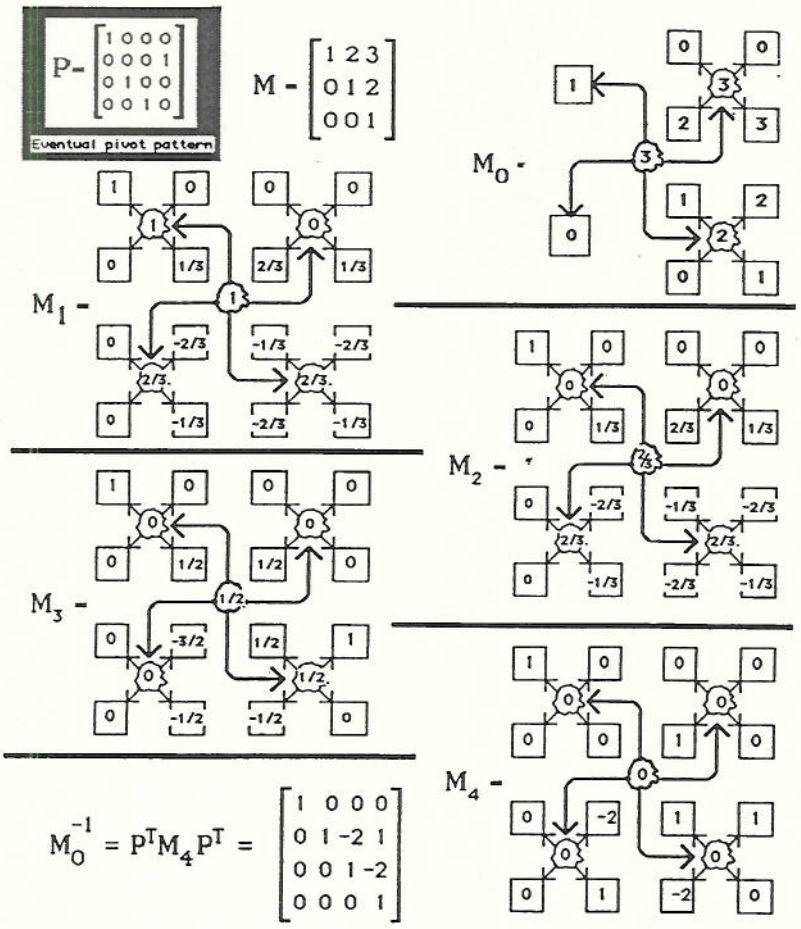

Figure 2. Knuth's Problem 2.2.6-18 [6, p. 556], worked.

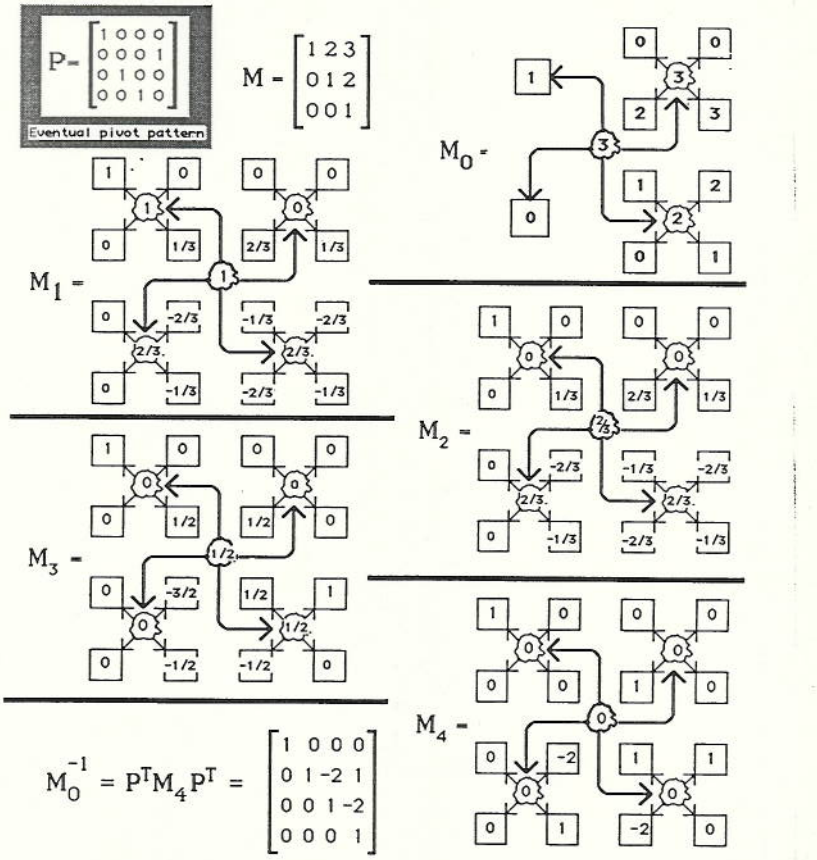

Figure 2. Knuth's Problem 2.2.6-18 [6, p. 556], worked.