OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

Representing Matrices as Quadtrees
for Parallel Processors

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

Computer Science Department
Oregon State University
Corvallis, Oregon 97331

1984-3

Representing Matrices as Quadtrees for Parallel Processors

David S. Wise
Computer Science Department
Indiana University
101 Lindley Hall
Bloomington, IN 47405

1. Introduction

Implementation of matrix algebra is a favorite application for motivating various parallel-processing architectures. This paper approaches this familiar area by reconsidering the structure of these algorithms--more specifically the structure of matrices, themselves--in order to arrive at algorithms suitable to multiprocessing, in general. The result is a homogeneous tree data structure implementing matrices, that is at once amenable for parallel processing, on-the-fly algorithm selection, and sparse matrix representation.

An algorithm that is amenable to parallel processing will execute efficiently with a high degree of parallelism in any of several possible multiprocessing environments; it should admit high-level decomposition into a few mutually independent processes. Successive decomposition is necessary in order to extend the degree of parallelism to take advantage of whatever population of processors is available. Such decomposition should occur at a high level in the the program structure in order that each process be large or, more accurately, long-running so to better amortize the overhead of its dispatch and recovery. Finally, each such decomposition should cleave one process into but a few others, in order that parallelism expand in a controlled fashion to fit within an unknown run-time processor resource; too many processes create undesirable

process-swap overhead. It is better to have seven nested options of splitting a process in half, than to have just one choice between 1 and 128 active processes.

I mean by "on-the-fly algorithm selection" a second-order algorithm that selects between two first-order algorithms at run time, responding to the nature of data. Such a choice is often made within floating point library functions that are sensitive to convergence of alternative expansions over different ranges. Sparse matrices [5] are matrices that have a plethora of zero elements, represented in a way to avoid storage space and computation cycles for those elements.

This short description uses two-dimensional matrices and quaternary trees for exposition. Consistently, scalars are isolated nodes, vectors are binary trees, and n-dimensional arrays are 2^n-ary trees.

2. Heaps and Multiprocessing

Heaps, memories organized exclusively with explicit links within a data structure, are common in uniprocessor language environments, like those of LISP and PASCAL. They are not usually considered fundamental there, however, being absent in many useful languages and having been only recently implemented in hardware. In older languages the quadtree structure would likely have been implemented as a complete quaternary tree [5] and stored sequentially in linear memory.

In spite of the space and access overhead associated with heaps, they are better suited to multiprocessing [4]. A homogeneous heap may be distributed across several independently accessed banks of memory connected to allow many processors to access different banks simultane-

ously. Because memory addresses are transparent to the user, a single data structure may be distributed across all the banks. When a structure has been built from random nodes, two processors accessing different parts of it are not likely to be addressing into the same bank. In contrast, parallel access to related, sequentially-allocated data generates access clusters much like those into some scatter-tables.

Ordinary conventions for manipulating linked structures, moreover, alleviate conventional problems of interprocess contention. If we adopt the convention that structures are built, shared, dereferenced, but never altered (as in single-assignment or purely applicative languages), then there is never any need for fetch/store synchronization among processors; the unique store to an address implicitly precedes all fetches from there. If sharing of infrastructure is prohibited (i.e. trees are required), then two processes dispatched on different substructures will never contend for access to the same memory in the identical pattern, as they would if they were traversing the same structure.

The conventional model of a processor changes somewhat. Because some sort of interconnect-switch is interposed between physical processors and memory banks, finite delay in memory response time (depending on the number of banks) must be expected; this time can be absorbed by cacheing several active processes on each physical processor (extending the number of active processes beyond the number of physical processors) so that some may run while others await their pipelined memory responses. The burden of storage management, moreover, can be almost entirely removed from processors in this architecture; a banked and switched memory can remotely maintain its own reference counts without

overhead of additional address space.

As in all multiprocessor architectures, the overhead for process dispatch and recovery remains. This is the analog of overhead for subroutine initiation and termination on a uniprocessor and, in the absence of interprocess communication at other times, no more bothersome.

3. Definitions and Simple Operations

Inspired by the quadtree structure as used in computer graphics [7], let us structure a two-dimensional matrix in one of three ways. Either it is composed only of zero elements, or it is a non-zero scalar (equivalent to a one-by-one matrix), or it is composed of four equal-sized quadrants, each (recursively) a matrix. Let the NIL pointer refer to the zero-matrix (of any size), and let the quadrants be accessed by four non-NIL pointers.[1] By convention the upper-left quadrant will always be square, sized by the largest power of two less than the size of the matrix; thus the quadrants are all about half the size of the original matrix. Typically, the right and lower edges will be padded with NIL.[2]

If a matrix is sparse, then significant numbers of blocks within the matrix (away from the edges) will also be represented by NIL pointers. Even when these blocks are mostly two-by two, we still can

---

[1] It might also be useful to provide another easily recognized pointer (say, IDE) to the square identity matrix, $[\delta_{i,j}]$, of any size. Thus both of the ring's identities are easily detectable at run time, when various operations can be accelerated by application of appropriate axioms. In that case, a scalar should be both non-zero and non-unitary.

[2] and the lower, right corner will be IDE, preventing zero padding from introducing false singularities.

prune the equivalent of nearly a full level from the ordinary quadtree structure without special reordering of the data. (In contrast, an entirely zero substructure nearly never occurs under row-major representation; such a matrix is singular.) In addition to the space efficiency, ring theory provides axioms that accelerate operations on zero blocks, yielding a time savings whenever NIL is uncovered during a matrix operation.

Accessing an individually indexed element from this structure is, at worst, a tree walk from root to leaf, requiring time logarithmic in the size of the matrix (the height of the tree). At best, a NIL pointer is uncovered along that path, forcing the result immediately to zero. The walk algorithm is quite simple, discharging one bit from the binary expansion of the two indices at each step. Those two bits indicate in which of the four quadrants the element lies.

Transposing a matrix is just as straightforward. The matrix is recopied recursively with upper-right and lower-left quadrants exchanged. When a header is provided for each user-defined matrix (e.g. containing bounds for compatibility checks), an additional bit there can indicate whether the matrix has been transposed. Matrix transpose then takes constant time, simply by building a new header and sharing the infrastructure. That bit indicates exchange of the upper-right and the lower-left pointer-accessing functions for that matrix.

Matrix addition is also recursive. The basis cases are a NIL addend (additive identity) or scalar addition. Otherwise, addition decomposes into four independent additions, suitable for multiprocessing. Pointers to these four sums are assembled as a new sum-matrix.

Addition, as well as recopying transpose, is amenable to parallelism, regardless of the size of the matrix.

4. Matrix Multiplication

Though not too typical of large matrix problems, much theoretical work is available on the problem of efficient matrix multiplication. Treated here at length, this problem illustrates the flexibility of quadtree representation by admitting superlative algorithms over homogeneous data.

Consider the problem of multiplying two compatible matrices represented as quadtrees. Multiplication of NIL and scalar matrices is trivial. Traditional Gaussian multiplication on non-trivial quadtrees can be effected by eight recursive multiplications of quadrants followed by four additions (of the sort discussed above).

While the expanded formula for each element is identical to that of accumulated dot-products, the order of addition is altered. The association pattern of the additions follows a complete binary tree rather than a left-linear tree, a matter of concern to a numerical mathematician aware that floating-point addition is not associative. (In fact, the complete tree addition is more accurate than the linear tree addition.)

In 1969, Volker Strassen [3,6,8] showed how to multiply such quadrants using only seven (parallel) multiplications, but eighteen additions (parallel in three waves of 10 before, and 4, then 4 after the multiplications.[3]) It is obvious that this is no improvement over

[3] S. Winograd improved this to fifteen additions [3], at the cost of

Gaussian for small matrices (e.g. 2x2) but the improvement for large matrices is dramatic. The dominating exponent in the time formula, 3 for Gaussian multiplication, asymptotically decreases to 2.81 = lg(7) as factors enlarge. This algorithm, however, has not found its way into popular use; Knuth [6] attributes this fact to additional bookkeeping, apparent in memory architecture tuned to row/column traversal.

If the matrix operands are already represented in quadtree form, however, Strassen's algorithm becomes much more useful, because one can readily alternate between Gaussian and Strassen's algorithm at different levels in either recursion; no additional access overhead is accrued in switching between the two. (Stability remains a problem; extra floating-point additions erode accuracy [2].) It follows from the inefficiency of Strassen's for small matrices that the most practical parallel algorithm will be a hybrid of Gaussian multiplication (near the leaves of the quadtree) and Strassen's (nearer the root); the quadtree representation offers ready alternation between these two, while offering a homogeneous representation for other matrix operations.

Analysis of square, dense matrices shows that the hybrid algorithm should perform Strassen's multiplication step on submatrices 32x32 and larger, and perform Gaussian multiplication on submatrices 16x16 and smaller. One exercise, counting all scalar arithmetic operations, shows that Gaussian should be performed for matrices 12x12 and smaller. Another, counting process/subroutine dispatches, indicates that Gaussian should be done for matrices 17x17 and smaller. (These results are

---

parallelism and some stability [2]. Parallel addition can be done in waves of 4, 2, 2, followed by the 7 multiplications, and then addition again in waves of 3, 2, 2.

analytic, subject to verification on implementations.) Thus, a header cell giving the size of the matrix is most useful to determine which algorithm should first be used on the eight quadrants of a non-trivial matrix multiplication problem.

The presence of easily-detected sparse matrices, however, adds another wrinkle. When just one of the eight quadrants in a multiplication is all zeros (NIL), then a Gaussian multiplication step requires only six (non-trivial) recursive multiplications and but two (non-trivial) additions. Strassen's algorithm does not accelerate so nicely in the presence of NIL, because of additions within each factor.

This last point leaves us with the following hybrid algorithm for multiplying compatible matrices whose size is known:

1. If either of the matrices is NIL, then their product is NIL. If both are scalar, use scalar multiplication.
2. If the factors are 16x16 or smaller, use pure Gaussian multiplication.
3. If the factors are 32x32 or larger, fetch up their eight quadrants.
4. If any of the eight quadrants is NIL, do the multiplication using a Gaussian recurrence at this step, but this very algorithm for subsequent recursions (on quadrants.)
5. Otherwise use Strassen's recurrence at this step, but use this algorithm for subsequent recursions (on the sums of quadrants.)

The analysis of this algorithm, certainly amenable to parallel processing, is an open problem.[4]

---

[4]If IDE is used, it would be necessary to include testing for it at Steps 1 and 4. At Step 4 a Gaussian multiplication step requires only six (non-trivial) recursive multiplications and four additions.

5. Conclusions

I have also considered other matrix operations which are not described in detail here. Matrix inversion is related to multiplication [3]; determinants, eliminants [1], and pivot-step [5] all seem to led themselves to quad representation and to parallelism. Of particular note is the last, where intermediate results are triples: the pivoted matrix, the extracted pivot-row vector, and the extracted pivot-column vector--all structured as trees. Parallelism arises from simultaneous processing of the two quadrants that coordinate on the pivot element; the pivot quadrant is done (recursively) before these two, and the off-pivot quadrant is processed afterwards.

Hybrid storage representations may be built over this scheme; linearly-allocated and pipeline-processed vectors may be substituted for "scalars" above. If matrix multiplication were critical in an application, the analysis cited above suggests that matrices of size 16x16 and smaller should be so packed and pipelined, rather than linked as small quadtrees.

At the beginning of this investigation I asked myself why all of linear algebra is implemented using row-major representations. The only answers that occurred to me (and others) is that the theory was developed this way, and that the linearity of memory addresses makes it worthwhile to structure matrices linearly for conventional processors. It is possible, however, that the first answer has created the second.

Therefore, I suggest that linear algebra be reformulated under the rigors of quadtree representation of matrices. Many results are already available through proofs using block-decomposition; these blocks should

be constrained to be sized by powers of two, and that this proof technique be forced in preference to row-decomposition. Where vectors are needed, they must be treated as binary trees.

Surely, the same theory will result, perhaps some proofs will be more elegant, and possibly some new theorems will appear. I expect the real justification to be the discovery of new algorithms (e.g. [1] and [8]) that arise from viewing some old, important, problems from block decomposition. These algorithms will be most amenable to multiprocessing.

At the same time, analysis and performance evaluation of these and similar matrix algorithms might suggest that quaternary trees become the default representation for multiprocessor software packages and hardware accelerators. The historical, numerical motivation for pipelined processing would then require multiprocessing heap processors; computational mathematics would suddenly need something most similar [4] to a multiprocessing LISP machine!

6. References

1. K. Abdali. & D.D. Saunders. Transitive closure and related semiring properties via eliminants. Theoretical Computer Science 30 (August, 1984).

2. D. Bini & C. Lotti. Stability of fast algorithms for matrix multiplication. Numer. Math. 36, 1 (December, 1980), 63-72.

3. A.V. Aho, J.E. Hopcroft, and J.D. Ullman. The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA (1974), Chapter 6.

4. D.P. Friedman & D.S. Wise. Aspects of applicative programming for parallel processing. IEEE Trans. Comput. C-27, 4 (April, 1978), 289-296.

5. D.E. Knuth. The Art of Computer Programming, I, Fundamental Algorithms, 2nd Ed., Addison-Wesley, Reading, MA (1975), 299-304 + 401.

6. D.E. Knuth. The Art of Computer Programming, II, Seminumerical Algorithms, 2nd Ed., Addison-Wesley, Reading, MA (1981), 481-482.

7. H. Samet. Connected component labeling using quadtrees. J. ACM 28, 3 (July, 1981), 487-501.

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