ON SOME TRENDS IN ELLIPTIC PROBLEM SOLVERS

S. C. EISENSTAT AND M. H. SCHULTZ

TECHNICAL REPORT #194/81

FEBRUARY 1981

This work was supported in part by the U.S. Office of Naval Research under Grant N00014-76-C-0277.

1. Introduction

Elliptic boundary value problems are at the core of many systems of partial differential equations occurring in mechanics. Examples of applications include fluid dynamics, semiconductor device modelling, and structural analysis. In addition, they often appear as a result of applying an implicit difference approximation to the time derivative in a parabolic problem. Thus, it is important to have efficient and robust elliptic problem solvers. Moreover, we expect that many of the issues faced in the development of such solvers are typical of those to be faced in the development of large-scale scientific problem solvers.

The development of high-technology elliptic problem solvers must include serious consideration of algorithms, architecture, and applied mathematics, each traditionally the subject of an entire discipline. Recent and expected advances in algorithm and hardware technologies strongly suggest the possibility of both dramatically reducing the cost and dramatically increasing the scope of scientific computational modelling. But an integrated highly interdisciplinary approach will be necessary to realize this possibility.

Success in the development of high-technology elliptic problem solvers will significantly increase the effectiveness and productivity of engineers and scientists by decreasing the cost of their computing, by increasing the efficiency with which they can use the computer, and by allowing them to solve increasingly more

complex problems, which are computationally intractable on even the fastest machines available today.

In this paper we discuss some of the issues involved in the design of a high-technology elliptic problem solver. In particular, we will concentrate our attention on the design of a modular, heterogeneous multi-processor elliptic problem solver consisting of a host computer and one or more peripheral processors.

We wish to acknowledge the help of our current and former students and colleagues in the Department of Computer Science of Yale University in developing the ideas presented here. In particular, our colleague Josh Fisher played a key role in formulating the ideas in the Architecture section of this paper.

2. Architecture

We propose a high-technology elliptic problem solver consisting of a heterogeneous multi-processor computer system. This system would have a host machine (such as a minicomputer with a 32-bit word length and an operating system supporting virtual memory) and one or more highly optimized peripheral processors. An important practical advantage to this type of architecture is the fact that one does not have to design a complete hardware-software system to gain great computational power.

Commercial high-performance, programmable peripheral processors, which we will refer to hereafter as attached processors, have become quite popular for signal processing. These processors

feature tailored scientific instruction sets, direct user microprogramming, and very wide instruction words. This provides potentially very large speed-ups, but makes the processors extremely difficult to code for efficient performance. Clearly, programming tools such as optimizing FORTRAN compilers are a critical need if the use of these devices is to become commonplace. It seems quite likely that for the next five years such attached processors will provide a very attractive architecture for rapid and cost-effective scientific computation.

The value of dual-processor systems involving a general-purpose host and an attached processor for solving elliptic problems has yet to be convincingly demonstrated. Little is known about many fundamental issues. Perhaps the most critical is the identification of which key subalgorithms attached processors can do sufficiently well to insure that the cost of moving the data describing the problem does not overwhelm the gain obtained. Indeed, short of moving the entire problem to the attached processor, which may not be feasible because of programming and memory considerations, it is not a priori clear that such subalgorithms exist. If this multi-processor approach is to be successful, the host machine must be capable of supporting peripheral processors via a very fast specialized bus, rather than a slow general-purpose communications bus.

In thinking about the architecture of an elliptic problem solver, we should look beyond current commercial attached

processors. VLSI-based peripheral processors, which provide extremely fast cycle time because of high density, the potential for specification of highly concurrent calculations, very low cost, and easy design with automated design tools, may turn out to be revolutionary.

In combination, these attractions will allow us to fabricate special-purpose processors with the potential for massively reducing computation time. Furthermore, they introduce the possibility of moving many of the complexities of large scientific software systems into hardware and in this way drastically reducing the cost of scientific programming. The potential of this technology is so great that it promises to revolutionize our concept of what is a computationally tractable problem.

As Kung [6, 7] has emphasized, the key to cost-effective design and performance analysis of VLSI is the underlying high-level algorithm. In order to maximize chip density at any level of fabrication technology, it is necessary to have an architecture that involves simple and regular interconnections. Hence, underlying the chip should be an efficient parallel algorithm that has simple and regular movements of data. Moreover, to maximize the throughput of the peripheral processor, the algorithm should use pipe-lining techniques to overlap I/O with computation.

Despite its extraordinary potential, VLSI design and fabrication are still somewhat experimental. The current design process is lengthy and chip densities of the type we need are still

well in the future. We believe, however, that is not too early to begin thinking about the basic issues.

3. Algorithms

As the peripheral processors and their algorithms become computationally more effective, we run an increasing risk that the whole system will founder on communications. The turn-around or wall-clock time for a given simulation will be bounded below by communication time, the time needed to move the data between the host and the peripheral processors.

We will focus our attention in this section of the paper on some of the implications algorithms have for communications. We believe that the interplay between architecture and algorithms for such multi-processor systems, and in particular their impact on the turn-around time for large problems, is ill understood. Much theoretical modelling and experimentation remains to be done. Our intent is to illustrate some of the issues.

Generally speaking, algorithms for elliptic problems at some stage reduce to solving very large sparse linear systems. These linear algebra subproblems have two distinct computational phases, the assembly (i.e., the computation) of the linear systems to be solved and the solution of the linear systems.

This leads to many questions. What is the class of algorithms

for assembling linear systems? What are the implications of each assembly algorithm for computation and communication costs? What is the class of algorithms for solving the resulting highly structured sparse linear systems? What are the implications of each solution algorithm for computation and communication costs? What are the bandwidths for data flow between different levels of memory and processors? How much fast memory is needed to guarantee that algorithms will be compute-bound? Which subalgorithms can and should be implemented on attached processors? For which algorithms should special-purpose VLSI based processors be built?

How can we reduce turn-around time? Despite what the textbooks tell us, neither algorithms that are more efficient in arithmetic operations nor faster peripheral processors will in themselves automatically reduce turn-around time. The key is to do something about I/O. In the remainder of this section, we will illustrate four general techniques for addressing this issue:

1. Hide the I/O, i.e., overlap the I/O as much as possible with useful computation (under the assumption, of course, that the computing environment supports user-controllable parallel execution of I/O and computation).

2. Trade I/O off for incremental arithmetic computation until the two are completely overlapped.

3. Use linear algebra techniques to reduce the amount of required storage and hence the data flow.

4. Use analytic techniques to reduce the amount of required storage and hence the data flow by, for example, using adaptive high-order discretizations.

In order to illustrate these techniques, we consider the model problem of a second-order, linear, self-adjoint elliptic boundary value problem in the unit square. We discuss how each technique can be used to improve turn-around time.

For the first three techniques we consider solving the standard five-point central-difference approximation to the model problem on a uniform nxn grid [11]. To illustrate how we might overlap I/O with computation in a multi-processor system, we consider the case of a dual-processor system, a central processing unit and an I/O processor, with dual-ported fast primary memory and slow secondary memory, such as a disk. We assume that the I/O processor is user-programmable and that it can operate in parallel with the CPU.

Suppose we wish to compute the Cholesky factorization [5] of the model matrix, but that the mesh is arbitrarily large, i.e., the number of mesh points, n, in each direction is arbitrarily large, so that the band of the matrix cannot be stored in primary memory. What can we say about communication costs and turn-around time? Along with Jack Perry, we formulated and proved a rather startling result [8] about "out-of-core" Cholesky factorization algorithms, which we state informally.

> Given the ratio of the speeds of the two processors, there exists a constant M such that for all systems with at least M words of primary memory there exists a block Cholesky factorization algorithm and a two-processor schedule so that the factorization is compute-bound independent of n.

It is important to observe that, in discussing software for

multi-processor systems, one must give not only a computational algorithm in the classical sense for serial machines but also a specification for the flow of data between memories and processors and a schedule for the processors. The specific case under discussion is particularly startling in that using larger amounts of primary memory beyond the minimum M given by the preceding result actually increases the turn-around time. If we assume that the band of the matrix can fit into primary memory but is initially stored in secondary memory, we can prove the following result [8]:

> The turn-around time of the above "out-of-core" algorithm is shorter than the turn-around time required for (1) reading the band into primary memory and (2) doing an "in-core" Cholesky factorization.

Many currently available computers do not allow us to control their I/O processors explicitly, but do have a virtual memory system using a heuristic page replacement algorithm such as LRU (Least Recently Used). We have been able to prove the following result about this situation; see Perry [8] for a formal statement and proof.

> Given a fixed amount of primary memory and page size, there exists a block Cholesky factorization algorithm, independent of n, that minimizes the number of page faults.

Despite what the operating-systems experts of several commercial vendors claim, and despite the fact that automatic paging may minimize the number of times the data must be moved, given the choice between using automatic paging and scheduling the I/O, scheduling the I/O is bound to be more efficient in turn-around time. The point is that automatic paging schemes do not necessarily move the data at the right time to overlap it with useful

computation. Such considerations certainly should be of major importance in designing systems for scientific computation.

The last three general techniques for minimizing turn-around time are based on two observations: (1) Reducing storage is likely to reduce I/O. (2) There often exists a trade-off between solve-time and storage (or I/O time).

To illustrate a trade-off between solve-time and storage, we again consider the model linear system. The classical band Cholesky factorization algorithm requires $1/2\ n^4 + O(n^3)$ multiplications and n^3 storage. However, along with Andrew Sherman we developed a divide-and-conquer band elimination or minimal storage band elimination algorithm [3] that requires $5/6\ n^4 + O(n^3)$ multiplications and $(n+1)^2$ storage. Thus we can gain an order-of-magnitude improvement in the required storage at the expense of a constant-factor increase in the required work. The reduction in primary memory occupancy is particularly dramatic.

The third general technique for reducing turn-around is to use sophisticated linear algebra methods. This is a place where more classical kinds of numerical analysis play a very important role. Four of our favorite linear algebra methods are (1) sparse elimination [10]; (2) minimal storage sparse elimination [4]; (3) preconditioned conjugate gradient methods [2]; and (4) multi-grid [1]. These four methods have the following asymptotic costs for the model problem:

	Multiplications	Storage
(1) sparse elimination	$O(n^3)$	$O(n^2 \log n)$
(2) minimal storage sparse elimination	$O(n^3)$	$O(n^2)$
(3) preconditioned conjugate gradient method	$O(n^{2.5} \log n)$	$O(n^2)$
(4) multi-grid	$O(n^2)$	$O(n^2)$

These methods have the following virtues:

(1)-(2): (a) Good, general codes available.
(b) Efficient for two-dimensional problems.

(3): (a) Generally applicable with promise of good codes.
(b) Efficient for two- and three-dimensional problems.
(c) Low storage requirement.

(4): (a) Asymptotically optimally efficient.

On the other hand, these methods have the following possible shortcomings:

(1)-(2): (a) Inefficient in three dimensions.
(b) Large storage requirements, especially in three dimensions.

(3): (a) In some cases, parameters need to be estimated.

(4): (a) Difficult to implement efficiently.
(b) Difficult to implement for general problems.

What can we conclude from these asymptotic results? Methods (2), (3), and (4) have storage proportional to the number of unknowns. In theory, multi-grid looks like the clear winner. In practice, however, it is so difficult to implement and has such complex logic and overhead that it may not be efficient on any existing peripheral processor -- maybe not even on one specially designed.

The last but perhaps most powerful way to reduce time and storage is analytic: the use of adaptive high-order finite difference or finite element discretization techniques. Because the techniques are high-order, for a given accuracy the discretization involves few parameters and little storage and so can be solved quickly. Because the techniques are adaptive, we can attain high convergence rates for smooth and nonsmooth problems alike.

As an example, consider the use of piecewise bicubic polynomials with the Rayleigh-Ritz-Galerkin method [9]. Such a method requires $c\ n^2 + k\ n^q$, $2 \leq q \leq 4$, $k \ll c$ multiplications, where the first term corresponds to assembling the matrix and the second term corresponds to solving it. The value of q depends on the method used to solve the linear system. In general, for a given fixed accuracy, the linear systems for high-order discretizations are much smaller, and hence have more "information content" per nonzero entry, than the linear systems for low-order discretizations. But the computation per nonzero to assemble the matrix is much higher. Thus, we may draw several conclusions:

1. Solving the linear system for a high-order approximation is relatively easy.

2. I/O is reduced and is often unnecessary for high-order methods.

3. The assembly times are relatively high.

Clearly we should focus on optimizing the assembly phase. The first thing to investigate in the use of a peripheral processor is how well the assembly phase can be implemented. What can be done algorithmically to improve the assembly phase? Our studies with Alan Weiser [12] indicate that it is advantageous to use smooth tensor-product, piecewise-polynomial basis functions and to take advantage of all possible symmetries. For example, we get the following assembly costs per interior element:

Smoothness	Multiplications
C^1 (using only symmetry of matrix)	8102
C^1	882
C^2	558.

Moreover, smoothness has a beneficial effect on the solve phase in terms of work and storage. The following table gives the relative operation counts and storage for the Cholesky factorization of the corresponding band matrix:

Smoothness	Relative Multiplication Counts	Relative Storage
C^0	364.5	81
C^1	32.0	16
C^2	4.5	3

This shows that symmetries can play an important role in the assembly phase. If possible, it is highly advantageous to map the problem domain onto a union of rectangles and to use smooth tensor-products of piecewise polynomials.

Unfortunately, actual computational experience for realistic problem sizes giving engineering accuracy is not quite as clear-cut as we would like. The adaptive high-order methods have a very large overhead because of to their logical complexity. In practice we are trading arithmetic operations and storage for logical complexity. Moreover, it is not at clear whether we can effectively implement these methods on a multi-processor system.

When it comes to turn-around time, we do not know whether it is better to use a simple low-order method that involves a lot of data but can be implemented very effectively on a multi-processor or a sophisticated adaptive high-order method that involves little data but might not be effectively implementable. We cannot yet answer this fundamental question.

REFERENCES

[1] A. Brandt.
Multi-level adaptive solution to boundary value problems.
Mathematics of Computation 31:333-390, 1977.

[2] R. Chandra.
Conjugate Gradient Methods for Partial Differential Equations.
PhD thesis, Yale University, 1978.

[3] S. C. Eisenstat, M. H. Schultz, and A. H. Sherman.
Minimal storage band elimination.
In D. J. Kuck, D. H. Lawrie, and A. H. Sameh, editors, Proceedings of the Symposium on High Speed Computer and Algorithm Organization, pages 273-286. Academic Press, New York, 1977.

[4] S. C. Eisenstat, M. H. Schultz, and A. H. Sherman.
Software for sparse Gaussian elimination with limited core storage.
In Proceedings of the Sparse Matrix Meeting, pages 135-153. SIAM, Philadelphia, 1978.

[5] G. Forsythe and C. Moler.
Computer Solution of Linear Algebraic Systems.
Prentice-Hall, 1967.

[6] M. J. Foster and H. T. Kung.
The design of special-purpose VLSI chips.
Computer 13:26-40, 1980.

[7] C. Mead and L. Conway.
Introduction to VLSI Systems.
Addison-Wesley, 1980.

[8] J. R. Perry.
Secondary Storage Methods for Solving Banded Linear Systems.
PhD thesis, Yale University, 1981.

[9] M. H. Schultz.
Spline Analysis.
Prentice-Hall, 1973.

[10] A. H. Sherman.
On the efficient solution of sparse systems of linear and nonlinear equations.
PhD thesis, Yale University, 1975.

[11] R. S. Varga.
Matrix Iterative Analysis.
Prentice-Hall, 1962.

[12] A. Weiser, S. C. Eisenstat, and M. H. Schultz.
Solving finite element equations to moderate accuracy.
SINUM 17:908-929, 1980.