Pergamon

Computers Math. Applic. Vol. 27, No. 9/10, pp. 67–75, 1994

0898-1221/94 $7.00 + 0.00

0898-1221(94)E0050-T

Multidimensional Dynamic Programming on Massively Parallel Computers

E. ANGEL AND P. LEONG
Computer Science Department, University of New Mexico
Albuquerque, NM 87131, U.S.A.

Abstract—Massively parallel computers have become widely available for the solution of a wide range of numerical problems. However, methods that work well on single processor architectures are often not the best algorithms when multiple processors are available. In this paper, we present a dynamic programming approach to the numerical solution of elliptic partial differential equations. This method, based upon the earlier work of Angel and others, indicates that reconsideration of direct (noniterative) approaches to the numerical solution of partial differential equations can lead to competitive algorithms.

1. INTRODUCTION

Numerical solution of partial differential equations have provided a fertile testing ground for a variety of computational algorithms. The majority of techniques are based upon discretizing the original continuous problem, resulting in a large set of coupled equations. For the linear elliptic equations we will consider here, the discrete problem requires the solution of a large set of linear equations.

The sparseness of these equations has led to the investigation of a number of iterative approaches from the simple Gauss-Seidel method to more rapidly converging methods such as successive over-relaxation and alternating direction methods [1]. The structure of the algebraic equations has been exploited in a number of direct methods ranging from block elimination techniques to Fast Fourier Transform methods [2].

Dynamic Programming and Invariant Imbedding were proposed during the 1960's as numerical methods for the solution of the discrete approximations to partial differential equations [2]. Both approaches led to direct (noniterative) two-sweep methods. For standard elliptic problems over rectangular regions, these methods were equivalent to other numerical approaches and were less efficient than a number of iterative techniques.

Over the last few years, a number of types of massively parallel computers have become available. Such computers can have up to thousands of individual processors, each of which can do an independent computation. Solution of numerical problems on these machines often requires new or restructured algorithms to make best use of the available processors. Our contention is that the direct methods such as dynamic programming possess some important advantages for the numerical solution of partial differential equations on massively parallel machines.

In the next section, we will review the dynamic programming approach. The invariant imbedding approach leads to an equivalent set of equations, and thus, will not be considered any further here. In the next section, we will describe a particular massively parallel architecture, the distributed memory message passing architecture used in a number of commercial MIMD machines, particularly hypercubes such as those of nCUBE and Intel. Then, we will show how to modify

This work was supported by the Applied Mathematical Sciences Program, U.S. Department of Energy, Office of Energy Research through Sandia National Laboratories.

the dynamic programming algorithm for such machines. Finally, we will present some numerical results.

2. DYNAMIC PROGRAMMING

The dynamic programming approach to the numerical solution of partial differential equations starts with the equivalence between the partial differential equation and a particular optimization problem. For example, the solution of the potential equation

$$u_{xx} + u_{yy} = g(x, y),$$

over some region R with fixed boundary conditions is the same as the minimization of functional

$$J = \iint (u_x^2 + u_y^2 - 2gu)\, dy\, dx,$$

over the same region subject to the same boundary conditions.

The discrete version of the problem over a rectangular region can be obtained by replacing the function $u(x, y)$ by the set of approximate values on an $N \times M$ rectangular grid. We denote these values by $\{u_{ij}\}$, $i = 0, \ldots, N$, $j = 0, \ldots, M$. The boundary conditions are the values $\{u_{i0}\}$, $\{u_{0j}\}$, $\{u_{iM}\}$, $\{u_{Nj}\}$. We can pose the problem in vector form through the notation

$$\begin{aligned}
\mathbf{u}_i &= [u_{ij}], \\
\mathbf{g}_i &= [g_{ij}], \\
\mathbf{r}_i &= [r_{ij}] = \begin{cases} u_{i0}, & j = 0, \\ u_{iM}, & j = M-1, \\ 0, & \text{otherwise}, \end{cases} \\
\mathbf{Q} &= [q_{ij}] = \begin{cases} 2, & i = j, \\ -1, & |i-j| = 1, \\ 0, & \text{otherwise}, \end{cases} \\
s_i &= u_{i0}^2 + u_{iM}^2.
\end{aligned}$$

With these definitions, a vector version of the problem is to minimize the functional

$$J = \sum_{i=1}^{N} \left[\mathbf{u}_i^T \mathbf{Q} \mathbf{u}_i - 2\mathbf{u}_i^T \mathbf{r}_i + s_i - 2\mathbf{g}_i^T \mathbf{r}_i + (\mathbf{u}_i - \mathbf{u}_{i-1})^T (\mathbf{u}_i - \mathbf{u}_{i-1}) \right],$$

where $\mathbf{u}_0$ and $\mathbf{u}_N$ are known.

The dynamic programming approach to the problem starts with the sequence of optimization problems

$$f_k(\mathbf{v}) = \min_{\mathbf{u}_k, \ldots, \mathbf{u}_{N-1}} \sum_{i=k}^{N} \left[\mathbf{u}_i^T \mathbf{Q} \mathbf{u}_i - 2\mathbf{u}_i^T \mathbf{r}_i + s_i - 2\mathbf{g}_i^T \mathbf{r}_i + (\mathbf{u}_i - \mathbf{u}_{i-1})^T (\mathbf{u}_i - \mathbf{u}_{i-1}) \right],$$

where

$$\mathbf{v} = \mathbf{u}_{k-1}.$$

The solution of our original problem is given by $f(\mathbf{u}_0)$. Applying Bellman's principle of optimality, we find

$$f_k(\mathbf{v}) = \min_{\mathbf{u}_k} \left[\mathbf{u}_k^T \mathbf{Q} \mathbf{u}_k - 2\mathbf{u}_k^T \mathbf{r}_k + s_k - 2\mathbf{u}_k^T \mathbf{g}_k + (\mathbf{u}_k - \mathbf{v})^T (\mathbf{u}_k - \mathbf{v}) + f_{k+1}(\mathbf{u}_k) \right].$$

It is easy to show by induction that $f_k(\mathbf{v})$ must be quadratic in $\mathbf{v}$:

$$f_k(\mathbf{v}) = \mathbf{v}^T \mathbf{A}_k \mathbf{v} - 2\mathbf{b}_k^T \mathbf{v} + c_k.$$

Substitution of this form into $f(\mathbf{v})$ and optimization over $\mathbf{u}_k$ leads to a backward set of recurrences for $\{\mathbf{A}_k\}$ and $\{\mathbf{b}_k\}$,

$$\mathbf{A}_k = \mathbf{I} - [\mathbf{I} + \mathbf{Q} + \mathbf{A}_{k+1}]^{-1},$$
$$\mathbf{b}_k = [\mathbf{I} - \mathbf{A}_k](\mathbf{r}_k + \mathbf{g}_k + \mathbf{b}_{k+1}),$$

subject to the initial conditions

$$\mathbf{A}_N = \mathbf{I},$$
$$\mathbf{b}_N = \mathbf{u}_N.$$

The recurrence for $\{c_k\}$ is unnecessary if we are interested only in $\{\mathbf{u}_k\}$ and not the optimal value of J. Finally the $\{\mathbf{u}_k\}$ can be found through the forward recurrence

$$\mathbf{u}_k = [\mathbf{I} - \mathbf{A}_k]\mathbf{u}_{k-1} + \mathbf{b}_k,$$

starting with $\mathbf{u}_0$.

These equations can be shown to always have a solution and to be numerically stable [2]. The solution involves two sweeps but no iteration. The solution for $\{\mathbf{A}_k\}$ and $\{\mathbf{b}_k\}$ must be saved to solve for $\{\mathbf{u}_k\}$ on the second sweep. For $N = M$, the work is $O(N^4)$ due to the matrix inversions, all other steps being at worst $O(N^3)$. This set of equations can be shown to be equivalent to a block Gaussian elimination technique applied to the original set of equations. Note, also, that we have made no attempt to exploit the constant coefficient nature of the original problem. In this case, Fourier methods can be used to reduce the effort to $O(N^2 \log N)$ [3]. Nevertheless, at best the dynamic programming method (and the equivalent invariant imbedding method) are equal to other direct methods. In most applications on a single processor, iterative methods have been more efficient. However, when we try to move either a direct or iterative algorithm to a parallel architecture, the relative efficiency of the methods changes.

3. DISTRIBUTED MEMORY ARCHITECTURES

There are a variety of parallel architectures presently available. The two major categories of massively parallel machines are single instruction-multiple data (SIMD) and multiple instruction-multiple data (MIMD). In a SIMD machine, the same instruction is executed on each processor. Usually each processor is fairly simple (fine grained parallelism), and solving a problem such as a partial differential equation would usually proceed by assigning a single discrete variable to a either a physical or virtual processor and using an iterative method. In SIMD machines, each processor can communicate efficiently only with its neighbors. Consequently, iterative approaches map very well to such machines. However, the number of processors required for a reasonable discretization often exceeds the number of available processors.

MIMD machines use standard processors at each node (medium grain parallelism) and can provide a large amount of memory. In shared memory machines, any memory cell is accessible to any processor, while in a distributed memory machine, each processor has its own memory. Synchronization tends to be a problem with shared memory machines, while distributed memory machines have a problem with passing messages between processors to convey information. We will consider distributed memory hypercube architectures. Such machines have 2^k processors arranged as a k-dimensional hypercube as in Figure 1. Each processor is connected directly to its k neighbors, thus limiting the total number of connections but allowing any processor to communicate with any other processor with messages passing through at most k interprocessor links.

In such an architecture, there is no memory contention since each processor has its own memory. However, great care must be taken to avoid overflowing the communication channels and to avoid

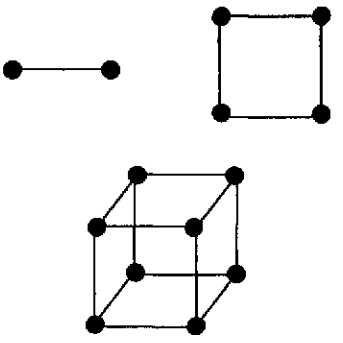

Figure 1. Hypercubes of 2, 4, and 8 nodes.

deadlocks where processors are waiting for information from other processors. However, for the problems considered here, there are simple strategies for avoiding such difficulties. In addition, most algorithms are organized so that processors communicate only with their neighbors, thus reducing communication overhead.

4. DISTRIBUTED ALGORITHMS

The usual method of solving partial differential equations on a parallel architecture is to divide the domain into a number of regions equal to the number of processors and assign each region to a processor. In order to understand the difference between direct and iterative solutions on multiprocessor architectures, we will briefly consider a simple iterative method and then our dynamic programming algorithm.

4.1. Iterative Methods

For the potential problem, the discretization yields the linear

$$u_{i+1,j} + u_{i-1,j} - 4u_{ij} + u_{i-1,j} + u_{i,j-1} = h^2 g_{ij},$$

at each of the $(N-1)(M-1)$ unknown grid points, where h is the spacing between points. The Gauss-Seidel method is an iterative method that solves the equations

$$u_{ij}^{(k+1)} = \frac{1}{4}\left(u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j-1}^{(k)} - 4h^2 g_{ij}\right).$$

It is fairly simple to show the method converges linearly and converges regardless of the initial guess of the solution $\left\{u_{ij}^{(0)}\right\}$.

Implementing these equations on a massively parallel machine is fairly straightforward. Suppose we have 2^n processors, arranged in a hypercube. We can divide our region into 2^n subregions and assign each to a processor. By gray coding the numbers of the processors, we can assure that adjacent regions are assigned to adjacent processors. Figure 2 illustrates the subdivision for $n = 3$. The processor numbers in the regions are shown in both decimal and binary to illustrate that adjacency of both processors and subregions.

Each processor can do independent Gauss-Seidel iterations for its subregion [4]. The problem is how to handle the boundaries between processors. A simple solution is to have each processor solve a problem which includes extra rows and columns corresponding to the edges of adjacent processors. Consider a simple one-dimensional example (with $g = 0$) using two processors. We wish to solve

$$u_i^{(k+1)} = \frac{1}{2}\left(u_{i+1}^{(k)} + u_{i-1}^{(k)}\right)$$

for $i = 1, \ldots, N-1$, where u_0 and u_N are the known boundary conditions. We can assign some (usually half) of the points $(u_0, \ldots, u_p)$ to the first processor and the rest $(u_{p+1}, \ldots, u_N)$ to the

0 000	1 001	3 011	2 010
4 100	5 101	7 111	6 110

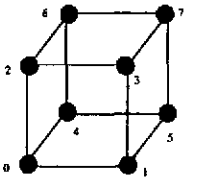

Figure 2. Domain decomposition and processor assignment.

second. The first processor iterates on its points (starting with u_1) as does the second with its points. However, the first processor has a problem with u_p as it has no value for u_{p+1}. Likewise the second processor needs u_p. The solution is for the first processor to have an extra point, u_{p+1}, and for the second to have an extra point, u_p. These extra points are sometimes called *ghost cells*. After each iteration, the two processors swap the values next to the ghost cells, e.g., processor zero sends $u_p^{(k+1)}$ to processor one so as to update these extra points. We can thus accomplish a true Gauss-Seidel iteration for the entire region with a minimal amount of communication between processors. The extension to two dimensions requires each processor to maintain rows and/or columns of ghost cells for each of its neighboring processors.

In practice, the rate of convergence of the Gauss-Seidel method is too slow. Other methods such as successive over-relaxation, alternating direction, and conjugate gradient have the same basic structure and better convergence rates. However, achieving the desired convergence properties is problem dependent. In the case of conjugate gradient, a preconditioning matrix is employed that optimizes the condition number of the matrix that determines the rate of convergence [5].

4.2. Distributed Dynamic Programming Algorithm

Direct methods have an obvious appeal as they avoid the problem of parameter selection and slow convergence due to suboptimal parameter selection. On the other hand, direct methods can require a great deal of computation and may not map well to multiprocessor architectures. We will show that we can avoid these problems with the dynamic programming method.

The basis of the dynamic programming algorithm is to divide the region of interest into a number of subregions equal to the number of available processors. In the simplest variant of our method there are no ghost cells, but adjacent processors share common boundaries. For example, in the one-dimensional case with two processors, we assign the points $(u_0, \ldots, u_p)$ to the first processor and $(u_p, \ldots, u_N)$ to the second. Note that the point u_p appears on both processors. A two-dimensional example is shown in Figure 3 where the dotted lines indicate artificial boundaries that are repeated on adjacent processors.

We treat each subregion as a completely independent region and use the direct dynamic programming to solve for the solution in the subregion. The obvious problem is that except at the edges of the full region, the processors do not have the correct boundary conditions, i.e., most boundaries for a subregion are in reality interior points of the whole region. In the one-dimensional example above, u_p looks like a boundary point to the two processors as do all points along the dotted line in Figure 3.

The actual domain decomposition we employ is to divide a rectangular region into strips as in Figure 4. For a given number of processors, this decomposition minimizes one of the dimensions of the rectangle. The advantage of such a situation is that the most computational complex operation, the matrix inversion, will be done on matrices of the lowest possible dimension.

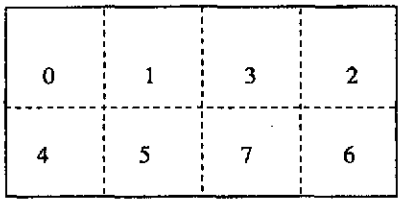

Figure 3. Domain decomposition with 8 processors.

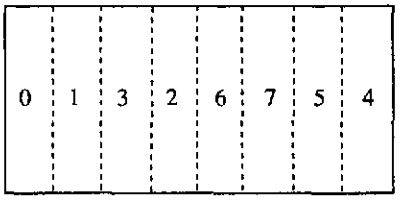

Figure 4. Domain decomposition into strips with 8 processors.

Figure 4 also shows the processor numbers for the case of 8 available processors. The processor numbers are gray coded. Processors that are used for adjacent subregions are adjacent processors on the cube, thus minimizing the communication paths on the hypercube.

We assume the points along the artificial boundaries are known. Initially, we can set them to any approximation such as might be obtained by some simple interpolation using the full set of boundary conditions, or we can simply set them to zero. We then solve the problem in each subregion subject to the incorrect boundary conditions on the artificial boundaries.

We next try to improve our guess of the artificial boundary conditions. The improved boundary conditions can be found by treating each artificial boundary as a sequence of unknowns that must solve a potential problem with the just obtained values on each side as boundary conditions. For example, in our one-dimensional example with two regions, u_p is the boundary. At the end of an iteration, we have values for u_{p-1} and u_{p+1}. Although these values are on different processors, a simple sending of the value on one to the other allows the receiving processor to solve for an improved u_p by requiring it satisfy the potential equation

$$u_p = \frac{1}{2}\Big(u_{p+1} + u_{p-1}\Big).$$

In two dimensions, the processor that receives the values from its neighbors solves a one-dimensional potential problem. An even simpler approach is to simply do a Gauss-Seidel step to update the boundary values. Thus, if u_{ij} is a point along the artificial boundary after iteration k, we replace it by

$$u_{ij} = \frac{1}{4}\Big(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}\Big).$$

As we shall see, even this simple scheme gives linear convergence.

Hence, we alternate between solving the potential problem with approximate boundary conditions by dynamic programming and trying to obtain better estimates for the values along the artificial boundaries. If enough storage is available on each processor, iterations after the first require significantly less work than the first. This follows from the the observation that the matrix inversions are the most expensive part of our algorithm, but the recurrences involving inversions are independent of the boundary conditions. Thus, the matrix computations need only be carried out on the first iteration if the results can be stored for further iterations.

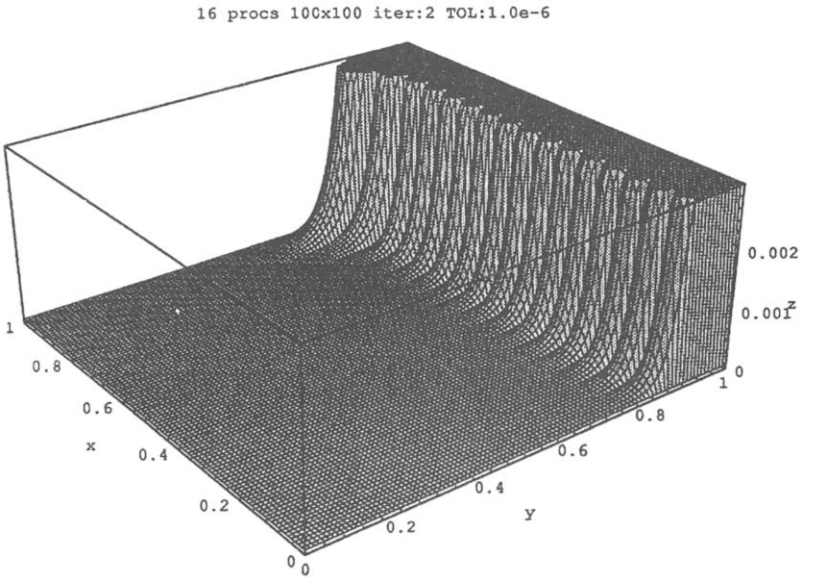

Figure 5.

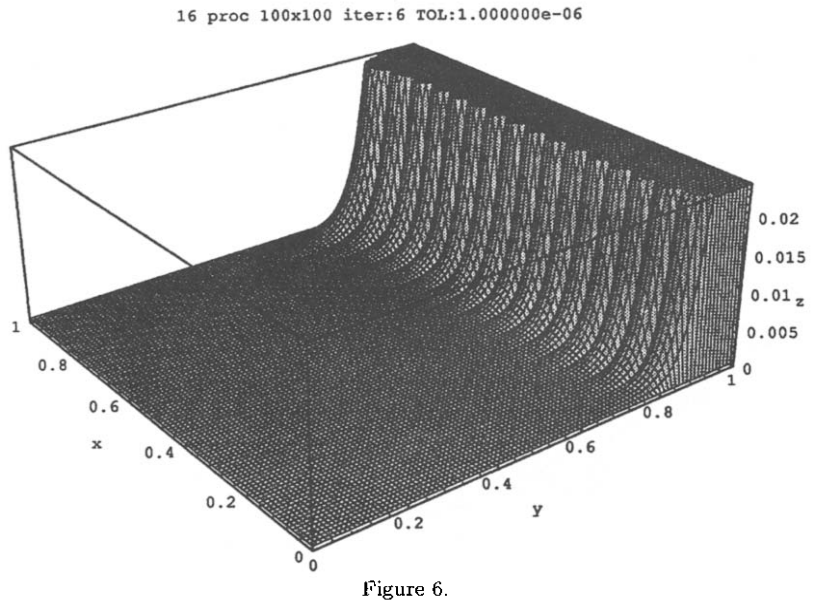

Figure 6.

5. RESULTS

We carried out the process using 16 nodes of an nCUBE-2 processor. The problem size was a 100×100 grid. The problem was divided up into 16 strips similar to the division in Figure 4 with the artificial boundaries at $j = 6, 12, \ldots, 84, 90$. The boundary conditions were zero on three sides and one on the fourth. The values along the artificial boundaries were set to zero. Note that each processor has to solve a problem of size 5×99 except for the last processor which has a problem of size 8×99. Except for the last processor, all matrices are 5×5.

Figures 5–8 illustrate the solution after 2, 6, 45 and 100 iterations. Further iterations reduce the error as can be seen in Figure 9. We can also see the convergence is linear. The first iteration requires more work, as it is the only iteration that computes the matrix inverses. The linear

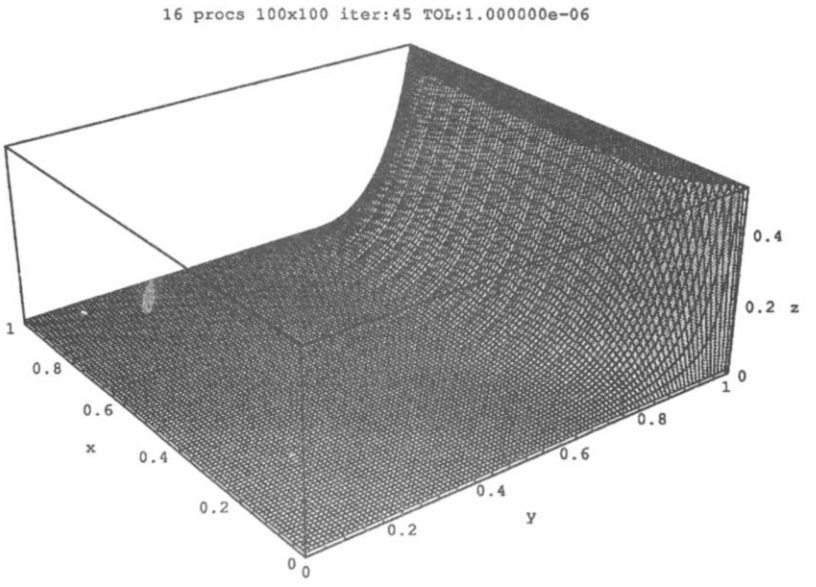

Figure 7.

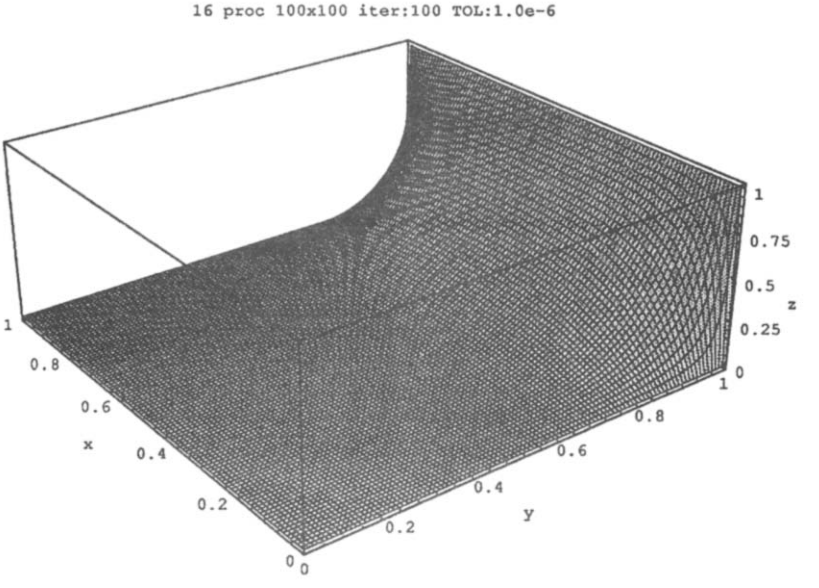

Figure 8.

convergence makes sense, as we can interpret the updating of the artificial boundary conditions as either a gradient step on the original functional or as a partial Gauss-Seidel step on an approximate solution to the potential problem.

6. DISCUSSION

Although we have done only preliminary investigations of the dynamic programming method, our results are very encouraging. Using only a small subcube of the nCUBE, we can solve a 100×100 problem almost trivially. Note that if we used the full 1024 processors, the solution of a 5000×5000 problem can be accomplished with inversion of only 5×5 matrices. If we invert only slightly bigger matrices, say 20×20, the method can be used for some very large problems.

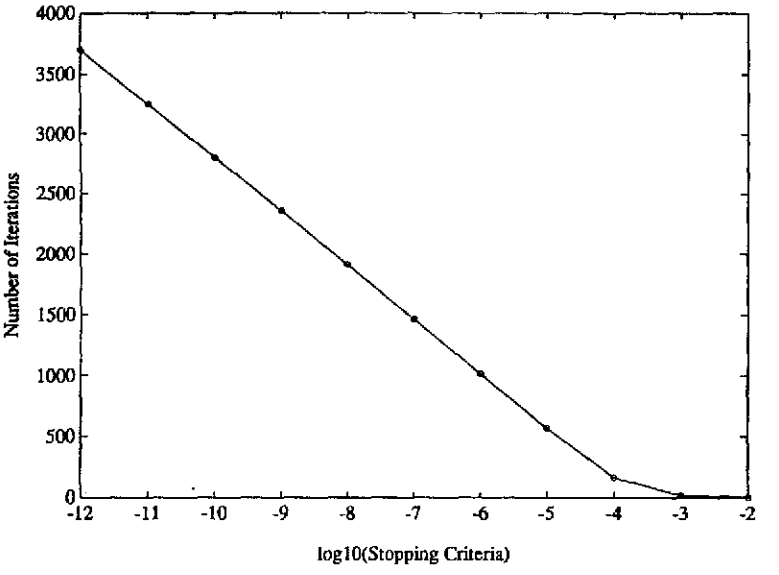

Figure 9. Number of iterations vs. stopping criteria for looping.

We have not compared the method with other methods such as preconditioned conjugate gradient methods or other direct methods such as the invariant imbedding method of Womble [6]. Nor have we done a detailed convergence analysis of our scheme. These efforts will be forthcoming.

Of more interest are a number of extensions. We believe faster convergence can be obtained by a couple of techniques. First, we can solve the potential problem along the artificial boundary rather than simply taking a Gauss-Seidel step. We can also overlap the regions. While this step requires either more processors or that each processor solve a larger problem, the larger the overlap, the greater the amount of information shared between adjacent processors. We believe this overlap will speed convergence at, hopefully, only a slight increase in computing cost per iteration.

More generally, the ability to do a regional decomposition with dynamic programming leads to many other problems. For example, problems over irregular regions easily can be mapped to multiprocessor architectures. Other large dynamic programming problems without an obvious geometric interpretation may also be solvable by our iterative approach.

REFERENCES

1. L.A. Hageman and D.M. Young, *Applied Iterative Methods*, Academic Press, New York, (1981).
2. E. Angel and R. Bellman, *Dynamic Programming and Partial Differential Equations*, Academic Press, New York, (1972).
3. E. Angel, Dynamic programming to fast transforms, *J. Math. Anal. Appl.* **119**, 82–89 (1986).
4. D.P. Bertsekas and J.N. Tsitsiklis, *Parallel and Distributed Computing: Numerical Methods*, Prentice-Hall, Englewood Cliffs, NJ, (1989).
5. T.F. Chan and C.C.J. Kuo, Parallel elliptic preconditioners: Fourier analysis and performance on the connection machine, *Computer Physics Communications* **53**, 237–252 (1989).
6. D.E. Womble, R.C. Allen and L.S. Baca, Invariant imbedding and the method of lines for parallel computers, *Parallel Computing* **11**, 263–273 (1989).