SIAM J. SCI. STAT. COMPUT.
Vol. 12, No. 6, pp. 1480–1485, November 1991

015

SOLUTION OF LINEAR SYSTEMS OF ORDINARY DIFFERENTIAL EQUATIONS ON AN INTEL HYPERCUBE*

L. LUSTMAN†, B. NETA†, AND C. P. KATTI‡

Abstract. In this paper there is developed and tested a parallel scheme for the solution of linear systems of ordinary initial value problems based on the box scheme and a modified recursive doubling technique. The box scheme may be replaced by any stable integrator. The algorithm can be modified to solve boundary value problems. Software for both problems is available upon request.

Key words. initial value problems, parallel processing, hypercube, box scheme, recursive doubling

AMS(MOS) subject classifications. 65L, 65W

1. Introduction.

We consider the solution of linear problems on a hypercube. By a hypercube we intend "a distributed memory MIMD computer with communication between processors ... via a network having the topology of a p-dimensional cube, with the vertices considered as processors and the edges as communication links" [4]. See also Fox [1], [2] and Fox and Otto [3]. Our method of solution is based on the box scheme to discretize the system of initial value problems:

$$y' = Ay + f(x),$$
$$y(a) = y_0,$$

where y and f are n-dimensional vectors and A is an $n \times n$ matrix. We obtain, in parallel, fundamental solutions on subintervals. The resulting system of equations is solved by a modified version of the recursive doubling technique (see Stone, [7]). Another technique that parallelizes the solution by subinterval decomposition has been proposed by Skeel [6].

In the next section, the general problem is stated and some information on the INTEL Hypercube is given. The algorithm for initial value problems is described in §3, and the efficiency of the algorithm is discussed in §4, where we detail the numerical experiments performed with our algorithm. In the last section we present our conclusions.

2. The general problem.

The numerical solution of ordinary differential systems is an intrinsically sequential procedure: given the data at a point x (or at several points $x, x-h, \cdots, x-Kh$), one advances to the following point $x+h$. In order to parallelize this procedure, we make the basic remark that for a linear system $y' = A(x)y$ on an interval $[a, b]$, the solution at the right endpoint is a linear function of the values at the left endpoint:

$$y(b) = Y_{[a,b]}y(a).$$

*Received by the editors September 24, 1990; accepted for publication (in revised form) July 2, 1991. This research was conducted for the Office of Naval Research and was funded by the Naval Postgraduate School. Preliminary results were obtained by the last two authors, whose research was partially supported by the National Science Foundation, through grants INT-8519159 and INT-8613396.

†Naval Postgraduate School, Department of Mathematics, Monterey, California 93943.

‡Jawaharlal Nehru University, School of Computer and Systems Sciences, New Delhi 110067, India.

Here $Y_{[a,b]}$ is the value at $x = b$ of Y, the fundamental solution on the interval, which is defined by:

$$Y' = A(x)Y, \qquad Y(a) = I, \quad \text{the identity matrix.}$$

To solve a problem on the interval $[x_{\min}, x_{\max}]$, we propose to assign several contiguous subintervals:

$$[x_0, x_1], [x_1, x_2], \cdots, [x_{N-1}, x_N],$$

with $x_0 = x_{\min}$, $x_N = x_{\max}$ to the N processors, and let each of them compute in parallel the corresponding fundamental solution. This is a task that may require a large number of sequential steps, for the numerical evaluation of $Y_{[x_i, x_{i+1}]}$. After these quantities are ready, one may obtain $y(x_i)$ from the initial data $y(x_{\min})$ by matrix multiplication, as obviously

$$Y_{[a,b]} Y_{[b,c]} = Y_{[a,c]}.$$

Let us remark at the outset that this elementary procedure may be extended to inhomogeneous equations, with the following initial data:

$$y' = A(x)y + f(x),$$
$$y(x_{\min}) = \text{given},$$

or two-point boundary data:

$$y' = A(x)y + f(x), \qquad x_{\min} \le x \le x_{\max},$$
$$B_1\ y(x_{\min}) + B_2\ y(x_{\max}) = \text{given}.$$

Such extensions necessitate only the linearity of the equations and initial or boundary conditions. We shall discuss these general algorithms, as well as the steps necessary to obtain computational efficiency.

In order to address efficiency matters, we must also briefly present the machine on which the algorithm is run. The iPSC/2 Intel Hypercube is a MIMD (multiple instruction–multiple data) machine, consisting of several processors in hypercube connection. Each such processor—also called a node—executes its own program, on data in its own memory. The nodes are controlled by another processor—the host—which loads the programs into the nodes and starts them running. Host and nodes communicate by message passing; these messages are strings of arbitrary length, with an arbitrary "message type" (an integer), which may be sent from any processor to any other processor. At any moment a processor may send a message, find whether messages of a certain type are pending, or receive messages. The communication may be performed synchronously, i.e., the processing stops until a message is sent or received, or asynchronously, where processing and communication overlap.

It is seen, therefore, that an algorithm is optimal on such a machine if it may be set as several parallel processes, each working on its own data, with a minimum of interprocess communication. We shall see that our ordinary differential equation solvers fit very well the Intel architecture.

3. The algorithm for the initial value problem.

Step 1. Using N processors to solve the linear inhomogeneous system with initial conditions:

$$y' = A(x)y + f(x),$$
$$y(x_{\min}) = g,$$

divide the required interval into N subintervals:

$$[x_0, x_1], [x_1, x_2], \cdots, [x_{N-1}, x_N], \quad \text{with } x_0 = x_{\min}, \quad x_N = x_{\max}.$$

The algorithm will produce numerical approximations for $y(x_j)$, $j = 1, \cdots, N$.

Step 2. Do in parallel:

Processor j, working on the interval $[x_{j-1}, x_j]$ solves numerically the following two systems:

$$Y_j' = A(x)Y_j,$$
$$Y_j(x_{j-1}) = I, \quad \text{the identity matrix,}$$

and

$$\phi_j' = A(x)\phi_j + f(x),$$
$$\phi_j(x_{j-1}) = 0.$$

In our program this is done using the box scheme (see, e.g., Neta and Katti, [5]). The matrix Y_j is the fundamental solution on the subinterval, whereas ϕ_j incorporates the inhomogeneous effect of the forcing function f. When this step is completed, one may recursively compute $y(x_j)$ from:

$$\begin{aligned} y(x_1) &= Y_1(x_1)g + \phi_1(x_1), \\ y(x_2) &= Y_2(x_2)y(x_1) + \phi_2(x_2), \\ &\vdots \\ y(x_N) &= Y_N(x_N)y(x_{N-1}) + \phi_N(x_N). \end{aligned}$$

The last step of the algorithm is an efficient performance of the recursion above, assuming that $N = 2^m$, as usual on a hypercube.

Step 3. (This is a modification of the recursive doubling due to Stone [7])

(3a) For $1 \le j \le N$, initialize:

$$y_j = \phi_j(x_j), \qquad M_j = Y_j(x_j).$$

Also initialize: $y_1 = g + M_1 y_1$, $k = 1$.

(3b) For all $j > k$ compute:

$$y_j^* = y_j + M_j y_{j-k}, \qquad M_j^* = M_j M_{j-k}.$$

(3c) For all $j > k$ replace M, y by M^*, y^*:

$$y_j = y_j^*, \qquad M_j = M_j^*.$$

(3d) Set $k = 2k$. If $k < N$, repeat steps (3b)–(3c) above. Otherwise the algorithm ends with y_j the numerical approximations to the solutions at x_j.

4. The efficiency of the algorithm. We begin our discussion with an investigation of the communication overhead.

In Step 3 there will be interprocessor communication, as processor j obtains data from processor $j - k$. It is obvious that the algorithm requires only one additional buffer per processor, to hold M^* and y^*—under the assumption that the matrix

multiplications are performed in the order shown. It is also possible to perform steps (3b)–(3c) in parallel, but then care must be exercised to avoid data corruption by message passing.

One option is to use just one buffer, and accept data only when ready. We shall call this the "send on request" scheme. The other option is to broadcast data as soon as it is ready. This we shall call the "multiple buffer" scheme. Yet another possibility is to use as temporary buffers the memory provided by the hypercube communication technology. For example, processor 1 sends data to processor 5, in a message with message type 1 (the identity of the sender). Processor 5, executing Step (3b) with $k = 1$, expects data from processor 4, so it will accept only a message with message type 4. The data from processor 1 are left in the communication buffers, to be read when processor 5 reaches the stage $k = 4$. This version, the "message type" scheme, is clearly the simplest to program.

We have implemented all three versions mentioned above. As expected, the "send on request" program has a higher communication overhead, because about twice as many messages are passed as in the other schemes. The multiple buffer scheme and the message type scheme essentially run at the same speed, although the messages arrive in a different order. The test problems show that the message type scheme is preferable, unless the data to be transferred are so bulky as to slow down communication. This certainly does not happen in this program, which transfers matrices of moderate size. Moreover, as the size of the problem (i.e., the dimension of the vector y) increases, more and more work will be done on actually solving the differential equations, and the communication overhead will be less significant.

An idea about the magnitude of the communication overhead may be obtained from the data in the following tables, which summarize several numerical experiments in solving the following system:

$$\begin{aligned} y'_i &= y_i + x y_{i+1} + f_i, \qquad 0 \le i < 10, \\ y'_{10} &= y_{10} + f_{10}, \end{aligned}$$

where f_i is adjusted so that the exact solution is:

$$y = (1, e^x, e^{-x}, e^{2x}, e^{-2x}, e^{3x}, e^{-3x}, x, \sin(x), \cos(x)).$$

The first table shows the total time spent by each processor in solving the problem, as obtained from the `mclock` system call.

Most of the work is done in computing the fundamental solutions, and communication is a relatively small quantity. Even the "send on request" scheme, which has a large number of messages transmitted, does not strongly influence the run times, which seem nearly constant on the various processors.

Another efficiency measure, critical for comparing single processor and multiprocessor versions of the same mathematical procedure, is the total running time needed for the complete solution.

We can roughly estimate this quantity as follows: let the unknown vector y be of dimension n, and assume that the numerical solution involves s steps (of size h) to reach $x_{\max}$ from $x_{\min}$. A single processor algorithm will need a time proportional to

$$sn,$$

as it evaluates n right-hand sides s times (we assume that most of the computational work is spent on obtaining the right-hand sides of the differential equations, and

TABLE 1
System of order 10, *"send on request." Total busy time in msec.*

Processor :	1	2	3	4	5	6	7	8
No. of steps per processor								
10	871	856	847	827	873	852	834	825
20	1641	1630	1621	1600	1637	1617	1608	1597
40	3183	3163	3162	3139	3188	3159	3153	3138
80	6265	6244	6244	6221	6270	6234	6240	6220

ignore matrix-vector or matrix-matrix multiplications). Our parallel algorithm, using N processors, will have a running time of:

$$\frac{s}{N}n^2,$$

because each processor executes only s/N steps; the quantity computed is the fundamental solution, an $n \times n$ matrix. Thus, it appears that there will be a gain only if $n < N$, i.e., the order of the differential system is less than the number of processors.

Even if there is no obvious gain in parallelization if all one needs is the solution of one differential problem, the algorithm proposed may become efficient when used as the first step of an inverse problem, or distributed parameter problem. In such a case, the same system is solved repeatedly with different initial conditions (say); then, after obtaining the quantities M_j, ϕ_j in the processors, one may use Step 3 of the algorithm to obtain sets of values y_j from sets of initial conditions.

The discussion and numerical experiment data of this section have been concerned only with initial value problems. It is also valid for the parallel solution of boundary value problems.

5. Conclusion. We have presented a parallel algorithm for solving ordinary initial value problems. We have shown that this algorithm is easy to program, and that machine-dependent optimization is readily achievable. Moreover, the algorithm is very flexible: as the equations are solved independently on each subinterval, it is possible to use different subinterval sizes, or different solution strategies in each subinterval, in order to control the error or balance the work among processors. The algorithm can be modified slightly to solve boundary value problems.

We have identified certain classes of practical mathematical procedures, for which our methods will be useful; these include various forms of inverse problems.

The basic limitation of our algorithm is that it applies only to linear problems. We are currently working on a method of parallelizing the solution of general, nonlinear ordinary differential systems.

Acknowledgments. The authors gratefully acknowledge comments and corrections by the editor and referees.

REFERENCES

[1] G. C. FOX, *Concurrent processing for scientific calculations*, Proc. COMPCON 1984, 1984, pp. 70–73.

[2] G. C. Fox, *Are concurrent processors general purpose computers?*, IEEE Trans. Nucl. Sci., NS-32 (1984), pp. 182–186.

[3] G. C. Fox and S. W. Otto, *Algorithm for concurrent processors*, Physics Today, 37 (1984), pp. 50–59.

[4] H. B. Keller and P. Nelson, *Hypercube implementations of parallel shooting*, Appl. Math. Comp., 31 (1986), pp. 574–603.

[5] B. Neta and C. P. Katti, *Solution of linear initial value problems on a hypercube*, Tech. Report NPS-53-89-001, Naval Postgraduate School, Monterey, CA, 1988.

[6] R. D. Skeel, *Waveform iteration and the shifted Picard splitting*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 756–776.

[7] H. S. Stone, *An efficient parallel algorithm for the solution of a tridiagonal linear system of equations*, J. Assoc. Comput. Mach., 20 (1973), pp. 27–38.