The Implicit Pipeline Method

John B. Pormann*
NSF/ERC for Emerging Cardiovascular Technologies
Duke University, Durham, NC, 27708-0296

John A. Board, Jr.
Department of Electrical and Computer Engineering
Duke University, Durham, NC, 27708-0291

Donald J. Rose
Department of Computer Science
Duke University, Durham, NC, 27708-0129

Abstract

We present a novel scheme for the solution of linear differential equation systems on parallel computers. The Implicit Pipeline (ImP) method uses an implicit time-integration scheme coupled with an iterative linear solver to solve the resulting differential algebraic system. The ImP method then allows for two independent mechanisms for parallelism: pipelining of the solution of several time-steps simultaneously, and pipelining of the successive linear iterations in the solution of each individual time-step. Since pipelining allows for a highly structured communication pattern, it is possible to achieve good parallel performance on large processor sets. Performance results from a Cray T3E are given.

1. Introduction

Many problems in science and engineering can be posed as a system of coupled differential equations. These equations relate the rate of change of a given quantity to the current state of the system. The systems which we will be concerned with are first order, linear differential equations:

$$\left.\begin{array}{c} \frac{d[Q(t)\Phi(t)]}{dt} + A(t)\Phi(t) = F(t) \\ \Phi(t_0) = \Phi_0 \end{array}\right\} \qquad (1)$$

where $\Phi(t)$ is the vector of functions which defines the system's state at each point in time (the desired solution), Φ_0 is the initial state of the system at some time t_0, $Q(t)$ is a matrix reflecting the interaction of the rates of change, $A(t)$ is a matrix relating the effects of the state itself, and $F(t)$ is the driving function. Equation 1 may also be used to investigate partial differential equation systems if we assume that $Q(t)$, $A(t)$, $F(t)$, and $\Phi(t)$ all reflect the desired spatial discretization. Since we are primarily interested in such systems, we will assume that the $Q(t)$ and $A(t)$ matrices are sparse.

While there is a great wealth of knowledge on numerical solution techniques for systems of the form in Equation 1, relatively few of the techniques can be made to run well on massively parallel computing platforms. In some cases, the predominately sequential nature of an algorithm may preclude it from an effective parallel implementation. In other nominally parallel algorithms, the message passing overhead required to synchronize the processors and communicate new information among them may severely limit the efficiency and scalability of the code. Often, this is seen when the amount of computational work done, compared to the amount of communication, is relatively small. The Implicit Pipeline (ImP) Method has been designed to exploit the "parallelizable" nature of an implicit method while also providing a mechanism to reduce the communications overhead that may limit its efficacy.

2. Crank-Nicholson method

There are many ways of solving differential equations like that posed in Equation 1. The basic idea is to "march" forward from the initial value at t_0 to a point in time slightly ahead, say $(t_0 + h)$, where h is called the "time-step size" or simply the "time-step." That solution may then be used

*This work was supported in part by the Duke-North Carolina NSF Engineering Research Center for Emerging Cardiovascular Technologies (Grant CDR8622201), and in part by the North Carolina Supercomputing Center and Cray Research, a Silicon Graphics Company.

to "march" forward to $(t_0 + 2h)$, to $(t_0 + 3h)$, etc. The ImP Method uses a Crank-Nicholson method to step forward in time. A parameter of the method, θ, can be varied from 0 to 1. When $\theta = 1$, we have the Implicit or Backward Euler method, and when $\theta = \frac{1}{2}$, we have the Trapezoid Rule. Using a time-step size of h, Equation 1 becomes:

$$\frac{Q(t+h)\Phi(t+h) - Q(t)\Phi(t)}{h} + \theta A(t+h)\Phi(t+h) + (1-\theta)A(t)\Phi(t) = \theta F(t+h) + (1-\theta)F(t)$$

Rearranging terms, we see that we have reduced the solution of the differential equation system to repeated solutions of linear systems of the form:

$$B(t+h)\Phi(t+h) = C(t+h)$$

where:

$$\begin{aligned} B(t+h) &= \frac{1}{h}Q(t+h) + \theta A(t+h) \\ C(t+h) &= (1-\theta)F(t+h) + \theta F(t+h) \\ &\quad -(1-\theta)A(t) + \frac{1}{h}Q(t)\Phi(t) \end{aligned}$$

It is important to note that the linear systems representing each time-step are not independent since $C(t+h)$ requires the solution of $\Phi(t)$. On a sequential computer, this is of no consequence since we will naturally work our way up from the initial condition at t_0 to time-step $(t_0 + h)$, then to $(t_0 + 2h)$, etc. The ImP Method will exploit this interrelation to form a parallel-in-time approach to solving the differential equation system.

3. Pipelined SOR

In [1], Bonomo and Dyksen present a method for solving banded linear systems on a parallel machine (see also [10]). The method is algorithmically equivalent to the SOR algorithm (see for example [12], [6] and [7]), however, the authors use a clever trick to allow for parallelism between successive iterations. In solving the linear system, $B\Phi = C$ (ie. solving the system for a single time-step), the PiSOR iteration is computed according to:

$$\Phi_i^k = \frac{\omega}{B_{i,i}}(C_i^t - \sum_{j=i-W}^{i-1} B_{i,j}\Phi_j^k - \sum_{j=i+1}^{i+W} B_{i,j}\Phi_j^{(k-1)}) + (1-\omega)\Phi_i^{(k-1)}$$

where N is the size of the system, W is the bandwidth of the matrix, i is the component number, k is the iteration number, $B_{i,j}$ is the matrix entry at (i, j), and ω is a parameter of the method, $\omega \in (0, 2)$. This differs from the usual SOR method in that the summations over the rows of B are truncated according to the bandwidth. Once all of the components of Φ have been updated in this manner, the accuracy of the solution is checked to determine if more iterations are necessary. When some tolerance is reached, we say that Φ is solved and we use this value to compute C for the next time-step.

We can now see that once computations on $\Phi_{(i+W)}^{(k-1)}$ have been finished, work on Φ_i^k may begin. On a sequential computer, there is no particular advantage to this; we would simply be solving the system in a permuted form. If, however, we assign each processor in a parallel machine a different iteration number and allow them to compute successive iterations independently, then we can compute components of several iterates simultaneously. In order to insure independence of the successive iterations, we must require that iteration k has completed up through component $\Phi_{(i+W)}^k$ before the next processor begins the computation of component $\Phi_i^{(k+1)}$, and the iterations must remain separated by at least W components at all times. Eventually, all processing elements (PEs) are working simultaneously but on different iterations. Each PE becomes part of a "bucket brigade" of information: it receives data from its upstream neighbor, processes it, and sends the new component information off to its downstream neighbor. In this manner, we have "pipelined" the SOR iterations (PiSOR).

Note that there will be some amount of communication required for each component of each iteration. Thus, each PE would have to send N separate messages, each of which were on the order of one or two floating point values in size. Since there is a start-up cost associated with each message, such a large number of small messages can cause a significant decrease in performance. As a mechanism to reduce this network traffic, Bonomo and Dyksen suggest concatenating successive messages according to a "pipeline spacing" parameter. After computing δ components on a given PE, we then send one, larger message to the next PE. The initial method, then, can be viewed as having $\delta = 1$. As δ increases, however, the PEs have to wait longer before starting their iteration and must remain farther separated throughout the iteration; PE 2 must wait for PE 1 to complete component $\Phi_{W+\delta}^k$ before it can start $\Phi_1^{(k+1)}$. This increase in separation directly impacts on the number of PEs that can be used effectively. From [1], the maximum number of PEs that can be utilized on a matrix of size N and bandwidth, W, is:

$$P_{\max} = \left\lfloor \frac{N}{W+\delta} \right\rfloor$$

There is an obvious trade-off here. If δ is small, then more processors may be used. If δ is large, then fewer processors may be used, but they should be utilized more efficiently. This allows for some "fine tuning" of the algorithm to a

given computer architecture. On a true MPP machine with fast network connections, the message overheads may not be so significant, and δ can be set fairly small to allow use of many PEs. On a network of workstations, where communications is very costly, δ can be set higher to reduce the number of messages, but this also reduces the usable number of PEs.

4. Pipelining the time-steps

We can view the sequence of linear systems in a different manner by writing the matrix equation for, say, a "window" of three time-steps. For Backward Euler, we have:

$$\begin{bmatrix} B(1) & 0 & 0 \\ -\frac{1}{h}Q(1) & B(2) & 0 \\ 0 & -\frac{1}{h}Q(2) & B(3) \end{bmatrix} \begin{bmatrix} \Phi(1) \\ \Phi(2) \\ \Phi(3) \end{bmatrix} = \begin{bmatrix} C(1) \\ F(2) \\ F(3) \end{bmatrix}$$

where $C(1) = F(1) + \frac{1}{h}Q(0)\Phi(0)$. This can be written in a "big matrix" shorthand: $\mathcal{B}\Phi = \mathcal{C}$ Solving this equation would produce the solution of all components of all four time-steps.

Rather than solving $\mathcal{B}\Phi = \mathcal{C}$ directly, we instead perform a two level iteration: computing several "inner" iterations on each block-row (ie. time-step) and then doing "outer" iterates over the larger $\mathcal{B}$ matrix until some convergence criteria is met. For example, we could perform a single SOR iteration on the first block-row, giving us $\Phi^1(1)$. While this estimate is inaccurate, we can use it in the second block-row (which is also the second time-step) and calculate an iteration there, giving us $\Phi^1(2)$. Since the value of $\Phi^1(1)$ is inaccurate, the value of $C(2)$ that we are solving against is also in error. Thus, we are really solving the system $B(2)\Phi(2) = \hat{C}(2)$. Not only is this the "wrong" system to solve, but we only compute a single iteration on it and thus the value of $\Phi^1(2)$ is highly inaccurate. However, we continue for $\Phi^1(3)$, obtaining progressively worse approximations to the true solutions. This sweep through all the time-steps constitutes the first outer iteration. When we have completed the computations on the final time-step, we would start again at the top, calculating a second iteration on the first block-row, giving us $\Phi^2(1)$. The convergence properties of the SOR method indicate that this second iteration will be more accurate than the previous one. Thus, when we use $\Phi^2(1)$ in the computations for the second block row, it will enable us to compute a more accurate estimate for $\Phi^2(2)$. This, in turn, will allow for more accurate computations on $\Phi^2(3)$. This would be the second outer iteration. We continue performing outer iterations until $\Phi^k(1)$ meets some convergence criteria. Note that we do not force convergence across the entire $\mathcal{B}\Phi = \mathcal{C}$ system, just the first time-step.

When time-step t_1 has converged, the window is slid forward and time-steps t_2, t_3, and t_4 are computed. When t_2 converges, the window is shifted forward to include time-steps 3, 4, and 5. This process is continued until all of the time-steps are completed. Note, however, that this sliding window approach means that different size time-windows are not algorithmically equivalent

In order to slow the propagation of error through the pipeline, we can perform multiple inner iterations on each block row. Thus, instead of doing a single SOR sweep through the first time-step (resulting in a poor estimate of its true value), we perform 10 SOR sweeps thereby giving us a more accurate answer. When we move to the second time-step, this more accurate answer reduces the error in the approximate system, $B(2)\Phi(2) = \hat{C}(2)$, and after performing 10 SOR sweeps there, the error in the $\Phi(2)$ estimate will be reduced.

Solving the system in this way can offer us significant potential for parallelization. As we mentioned earlier, the big $\mathcal{B}$ matrices above are block lower bidiagonal. Thus, if we are working on t_1, t_2, and t_3, the iterations on t_1 are not affected by any of the computations on t_2 or t_3. Similarly, the computations on t_2 are not affected by the computations on t_3. If we assign a different processor to each time-step (or equivalently, to each block-row), all PEs can work in parallel. The chain of PEs again becomes a "bucket brigade" of information: each PE receives data from its upstream neighbor, processes it, and sends the new time-step information off to its downstream neighbor. Hence, we have "pipelined" the solution of the time-steps (PiTS).

The PiTS Method is similar to the work of La Scala in [8] and [9]. However, his papers dealt solely with non-linear equations. Also, the pipelining of the time-steps was done in a different manner than presented above. In La Scala's case, PE1 always computed the first iteration of every time-step, PE2 always computed the second iteration of every time-step, etc. The PiTS Method also has similarities to various waveform relaxation techniques (see [3] and [4]).

5. Experimental results

We have implemented the Implicit Pipeline Method, ImP, on a parallel platform in order to test and examine both the PiSOR and the PiTS methods. The two methods are also combined in a novel, hybrid approach which allows additional potential for parallel speed-up for larger processor sets. This hybrid method is a simple combination of the two methods: multiple time-steps are processed simultaneously according to the PiTS algorithm, while inside each time-step we perform multiple iterations simultaneously as in the PiSOR algorithm. The ImP code was tested and run with a number of different system configurations. We will refer to the number of pipelined time-steps as the number of "stages" in the configuration and the number of PEs working on each individual time-step as the number of "threads."

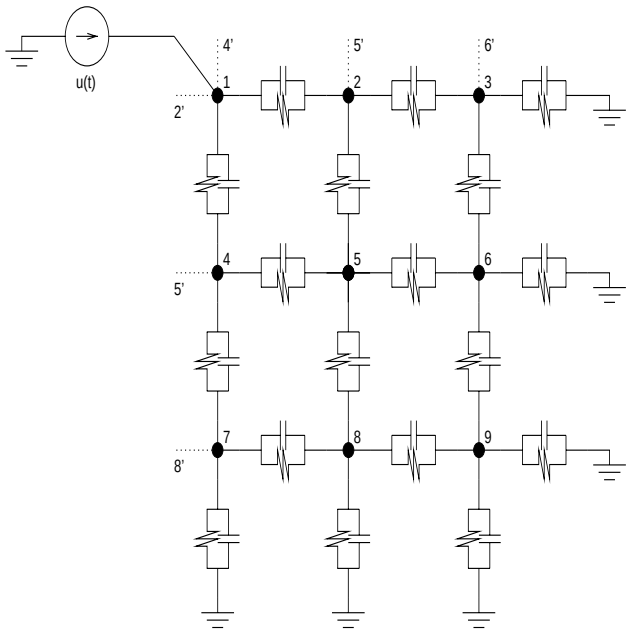

Figure 1. A small RC test problem. There is mirror symmetry along the top and left edges.

Stages	Threads	Speed-Up
1	1	0.966
1	2	1.817
1	4	3.497
1	8	6.403
1	16	10.129
1	32	11.029
2	1	1.711
2	2	3.141
2	4	5.763
2	8	9.724
2	16	13.953
4	1	2.994
4	2	5.274
4	4	9.603
4	8	15.552
8	1	4.391
8	2	7.770
8	4	13.013
16	1	5.298
16	2	8.851

Table 1. "Best case" speed-up seen by each pipeline configuration.

An "$m \times n$" pipeline will refer to m stages of n threads (for a total of mn PEs being used). All of the simulations were run on a Cray T3E at the North Carolina Supercomputing Center.

The test system for the following cases is a 2-D connection of resistors and capacitors. A small system is shown in Figure 1. The actual domain used in the simulations was 32×32 nodes. The interconnection of each interior node with its four neighbors gives both the Q and A matrices (and hence the B matrix as well) a pentadiagonal structure with a bandwidth of 32. The boundary conditions are set to simulate a larger, mirror-symmetric region. Thus the top edge of Figure 1 has a no flux boundary condition imposed by reflecting the voltages at nodes $4, 5, 6$ to positions $4', 5', 6'$. Similarly, on the left edge, we reflect the voltages at nodes $2, 5, 8$ to locations $2', 5', 8'$. There is a single current source at the upper-left corner node which is used to provide a time-dependent stimulus to the system. For the following cases, a unit-step current was applied to the system. From [12], we calculated the SOR parameter as $\omega \approx 1.8264$.

The Implicit Pipeline Method was tested with a variety of input parameters to compare the attainable speed-up for different pipeline configurations. The number of stages and threads as well as the number of inner iterations and pipeline spacing parameter were all varied. In all, 147 different parameter sets were tested. The code was run at least 3 times for each parameter set to insure that the data were statistically valid. The run-times never deviated by more than ± 15 % from the average.

By "speed-up," we are referring to relative speed-up, the run-time of the parallel code versus the run-time of the sequential code. The "best case" speed-ups are reproduced in Table 1. In this case, the displayed speed-up is the best average speed-up obtained by the given number of stages and threads, for any combination of pipeline spacing and inner iteration parameters.

These data are also plotted in Figure 2 and show some of the expected results. In the single-stage configurations, increasing the number of threads (hence increasing the number of PEs) up through 16 increased the speed-up of the code. Similarly, in single-thread configurations, increasing the number of stages (hence increasing the number of PEs) also increased the run-time speed-up, although less significantly. Only when the $P_{\max}$ limitation was reached in the single-stage configurations was there a significant decrease in efficacy with additional processors. In the smaller multi-threaded configurations, we also saw that there is little effect from the pipeline spacing parameter, δ, as there is ample room for the PEs to space themselves out. The multi-stage configurations show some potential for speed-up but they were not directly competitive with the multi-threaded configurations. The hybrid configurations showed more speed-up as more processors were added and they

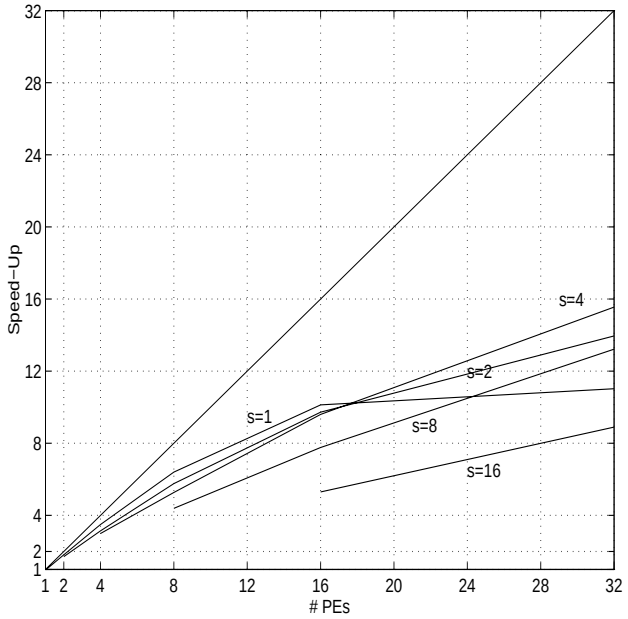

Figure 2. ImP speed-up (s is the number of time-steps in the window).

eventually eclipsed the straight PiSOR configurations.

As can be seen, the 1-stage pipelines, essentially performing a straight PiSOR algorithm solving 1 time-step at a time, exhibit the best speed-up of all configurations up through 16 processors. However, beyond 16 processors, the number of PEs exceed the maximum set by $P_{\max}$. For the given block sizes, $\delta = 16, 32, 64$, $P_{\max} = 21, 16, 10$ respectively. Thus, for 1 stage of 32 threads, the PiSOR algorithm requires that 9 to 21 of the processors sit idle while waiting for previous iterations to complete. In the 2×16 configuration, which uses the same number of processors, only 16 PEs interact in the PiSOR algorithm, hence only when $\delta = 64$ are PEs sitting idle. Similarly, the 4×8 pipeline has only 8 PEs interacting in the PiSOR algorithm (hence it is far from the $P_{\max}$ limit), and it also shows improved performance relative to the 1×32 configuration. In fact, the best performance achieved on 32 PEs was done by the 4×8 configuration.

6. Conclusions

The Implicit Pipeline Method shows promise for obtaining good parallel speed-up with larger processor sets for discretized partial differential equations. The results show that for smaller processor sets, the PiSOR algorithm produces good speed-up until the algorithmic limitation on processor usage is met. Once $P_{\max}$ is hit, the algorithm shows negligible speed-up for larger processor sets. While the PiTS method does not show exceptionally good speed-up by itself, it can be used to extend the efficiency of the parallel solution of differential equations. The hybrid algorithm, which allows for pipelining-the-iterations and pipelining-the-time-steps, adds another mechanism for obtaining parallel speed-up once the $P_{\max}$ limitation of PiSOR is reached. These results could improve further if the system size is scaled according to the size of the parallel machine (i.e. so-called "scaled speed-up").

References

[1] JP Bonomo and WR Dyksen. Pipelined successive overrelaxation. In Carey GF, editor, *Parallel Supercomputing: Methods, Algorithms and Applications*. John Wiley and Sons, New York, 1989.

[2] JJ Dongarra, IS Duff, DC Sorensen, and HA van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, 1991.

[3] DJ Erdman. *The Newton Waveform Relaxation Approach to the Solution of Differential Algebraic Systems for Circuit Simulation*. PhD thesis, Duke University, April 1989.

[4] DJ Erdman and DJ Rose. Newton waveform relaxation techniques for tightly coupled systems. *IEEE Trans on CAD*, 11(5), May 1992.

[5] GH Golub and JM Ortega. *Scientific Computing and Differential Equations*. Academic Press, Inc., New York, 1981.

[6] GH Golub and CF van Loan. *Matrix Computations*. Johns Hopkins, 1989.

[7] D Kincaid and W Cheney. *Numerical Analysis*. Brooks/Cole Publishing Co., Pacific Grove, 1991.

[8] M LaScala and A Bose. Relaxation/newton methods for concurrent time step solution of differential-algebraic equations in power system dynamic simulations. *IEEE Trans on Circuits and Systems I*, 40(5), May 1993.

[9] M LaScala, R Sbrizzai, and F Torelli. A pipelined-in-time parallel algorithm for transient stability analysis. *IEEE Trans on Power Systems*, 6(2), May 1991.

[10] NR Patel and HF Jordan. Parallelized point rowwise sor method. *Parallel Computing*, 1, 1984.

[11] RS Varga. *Matrix Iterative Analysis*. Prentice Hall, Englewood Cliffs, 1962.

[12] DM Young. *Iterative Solution of Large Linear Systems*. Academic Press, Inc., New York, 1971.