ADI ON THE CONNECTION MACHINE

JAMES STRZELEC* AND STEWART A. LEVIN†

Submitted to Electronic Transactions of Numerical Analysis

Abstract. Alternating directions (ADI) methods are used in a number of fields for the practical approximate solution of implicit finite-difference problems. They typically substitute a tensor product of banded matrices in place of a block-banded system, simplifying the solution method and reducing the solution time. In this paper, we examine the challenges such methods pose for parallel implementation on fine-grained parallel machines such as the Connection Machine (CM). We work exclusively with the high level Fortran 90 language available on the CM as well as a number of other of today's high performance computers. For the CM we present a highly parallel algorithm for ADI. The distinguishing characteristic of our algorithm is the preskewing of the input matrix along a single axis. In so doing, we distribute both matrix axes across all processors, with the goal of maximizing parallelism. We will explain in this paper how we implement the method, what improvement we achieve over the serial version of the code, and how it connects to other parallel alternating direction algorithms.

Key Words. Linear algebra, tridiagonal linear systems, cyclic reduction, vector computers

1. Introduction. We study in this paper computational aspects of alternating direction implicit (ADI) methods, used, for instance, in the solution of the Navier-Stokes equations [10, 8, 4, 6, 9, 5, 2, 1] and imaging of seismic reflection data in oil and gas exploration [3]. ADI makes for an interesting case study because it is a very simple algorithm, yet its recursive flavor makes it challenging to take advantage of the Connection Machine's massive parallelism.

1.1. Scalar ADI. For the study, we chose a factored form of ADI in which it is implemented as forward and backward passes of first order recursion. This form arises from prefactoring implicit tridiagonal systems into LU form, for example. The serial Fortran 77 code for our ADI representative algorithm is quite straight-forward:

```
INTEGER M,N, I,J, I1,J1, I2,J2
REAL X(M,N),A,B
DO I = 1,M
  DO J1 = 2,N
    X(I,J1) = A*X(I,J1) + B*X(I,J1-1)
  ENDDO
  DO J2 = N-1,1,-1
    X(I,J2) = A*X(I,J2) + B*X(I,J2+1)
  ENDDO
ENDDO
DO J = 1,N
  DO I1 = 2,M
    X(I1,J) = A*X(I1,J) + B*X(I1-1,J)
  ENDDO
  DO I2 = M-1,1,-1
    X(I2,J) = A*X(I2,J) + B*X(I2+1,J)
```

* Department of Scientific Computing/Computational Mathematics, Stanford University, Stanford, California 94305. (strzelec@sccm.stanford.edu)

† Mobil Exploration and Producing Technical Center, P.O. Box 650232, Dallas, Texas 75265-0232. (salevin@dal.mobil.com)

```
        ENDDO
      ENDDO
```

The first outer DO loop (over I) sweeps over rows of the matrix X, while the second outer loop (over J) sweeps over the columns. The nested DO loops sweep forward (over I1, J1) and backward (over I2, J2) along the rows and columns in each respective main DO loop. We see that the ADI algorithm, as it stands, is not partial to either the row dimension or the column dimension, at least when considering arithmetic. In practice, one of the sweeps will normally take longer than the other because of memory access patterns. This can be seen if we consider that, in Fortran 77, the matrix X is stored in column-major order. The column sweeps access data sequentially, which is generally more efficient than accessing data in large strides as in the row sweeps, especially on a cache memory system[1].

1.2. Parallel ADI. On vector machines, such as the Cray series, the above code is modified by interchange of the outer and inner loops, either manually or automatically, and vectorized over the new inner DO loops. These loops are "embarrassingly parallel" and vectorize very efficiently. While there still can be delays due to memory access patterns, for example bank conflicts, these are not generally severe and can be overcome with modest effort. In effect, the vector version uses the shared memory system to avoid physically transposing the matrix for each pass.

On distributed-memory parallel computers, such as the Connection Machine, the vector version of the ADI algorithm has difficulties. Data motion between memory and processors may require interprocessor communication, imposing costs much higher than for the shared memory vector machines. If we arrange for the compiler to place complete columns of the matrix in each processor's memory in order to avoid interprocessor communication during column sweeps, then the orthogonal row sweeps keep only one processor active at a time. To reclaim the lost parallelism we would have to transpose the complete array every time we switch directions, an operation involving lots of costly nonlocal communication.

A better choice, though still suboptimal, is to have the compiler break the matrix into blocks instead of columns or rows. This is the default layout on the Connection Machine. For example, if we are attempting to perform ADI on a square matrix of dimension 500, with 16 available processors, the run-time system will pad the data and distribute it to the processors as illustrated in Figure 1. For the vectorized ADI algorithm, this setup produces far from ideal performance, since only $\sqrt{k}$ processors are busy while completing a row or column of the sweep, where k represents the number of processors available on the machine.

For MIMD machines, in which each processor runs an independent serial (or vector) code, the classical solution [12, 7] to this problem is to skew the distribution of the blocks of the matrix as indicated in Figure 2. This block-cyclic distribution keeps all processors busy during both row and column sweeps, using the full parallelism of the machine, assuming the column or row length of the matrix is large enough. Data movement is restricted to circular shifts at block boundaries, a generally efficient (and safe) communication mode.

Unfortunately, such a layout is not an option at either the Fortran or the run-time system level of the Connection Machine and is only available on the CM-5 model using its explicit MIMD mode, something we want to avoid[2].

2. Algorithm.

[1] To combat this effect, we have employed judicious loop unrolling and interchange to optimize

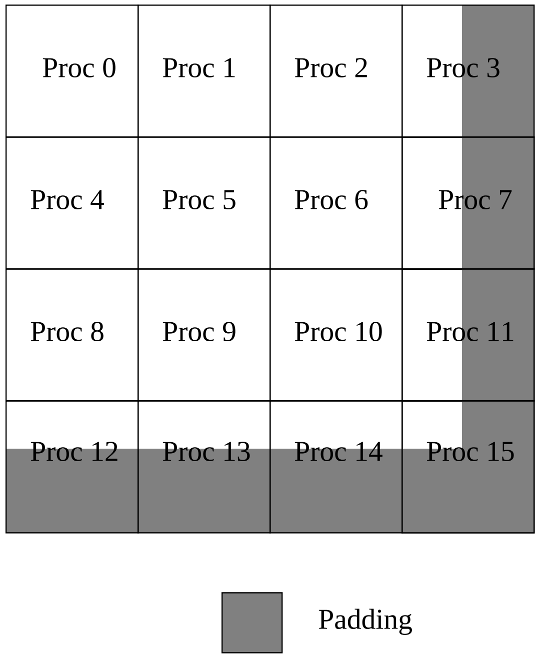

FIG. 1. *Default block distributing matrix X across processors.*

2.1. A SIMD skew trick. To avoid the complications of the block-skewed method, we instead linearly skew the matrix `X` along one dimension. (We restrict our matrix to be square for simplicity.) The matrix `X` initially looks like:

$$\begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} & x_{1,5} \\ x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} & x_{2,5} \\ x_{3,1} & x_{3,2} & x_{3,3} & x_{3,4} & x_{3,5} \\ x_{4,1} & x_{4,2} & x_{4,3} & x_{4,4} & x_{4,5} \\ x_{5,1} & x_{5,2} & x_{5,3} & x_{5,4} & x_{5,5} \end{pmatrix}$$

and we issue the Fortran 90 command:

```
X = CSHIFT(X,DIM=2,SHIFT=-[0:4])    .
```

This replaces `X` with the skewed matrix:

$$\begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} & x_{1,5} \\ x_{2,5} & x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} \\ x_{3,4} & x_{3,5} & x_{3,1} & x_{3,2} & x_{3,3} \\ x_{4,3} & x_{4,4} & x_{4,5} & x_{4,1} & x_{4,2} \\ x_{5,2} & x_{5,3} & x_{5,4} & x_{5,5} & x_{5,1} \end{pmatrix} .$$

In conjunction with this particular skewing of the matrix, we declare `X` with layout `(:serial,:news)`. On the CM, this determines that the columns reside in-processor.

our serial ADI timings.

[2] It is actually possible to do this in CM-Fortran by recasting the array as a four-dimensional object. The programming is basically the same, and just as messy, as the MIMD algorithm.

0	1	2	3	4	5	6	7
7	0	1	2	3	4	5	6
6	7	0	1	2	3	4	5
5	6	7	0	1	2	3	4
4	5	6	7	0	1	2	3
3	4	5	6	7	0	1	2
2	3	4	5	6	7	0	1
1	2	3	4	5	6	7	0

FIG. 2. *Block-cyclic layout for 8 processors.*

That is, processor 0 would now contain the vector:

$$\begin{pmatrix} x_{1,1} \\ x_{2,5} \\ x_{3,4} \\ x_{4,3} \\ x_{5,2} \end{pmatrix}$$

This SIMD version of skewed data distribution quite naturally follows from the MIMD method. The link is the concept of *virtual processors*. In this view of parallel computing, each individual element on a parallel axis is associated with a parallel virtual processor. The underlying run-time system handles the distribution of virtual processors to physical processors. What we are doing is applying the block skew layout to a virtual processor array, with the block size being 1x1.

2.2. Row sweep. To perform the row sweeps with the skewed matrix, we extract what were the columns of the original matrix, in order to sweep through the rows in parallel. Therefore, we extract the first column, and place it in a temporary vector `Z1`, which has been declared as `(:news)`:

$$\mathtt{Z1} = \begin{pmatrix} x_{1,1} & x_{2,1} & x_{3,1} & x_{4,1} & x_{5,1} \end{pmatrix} .$$

Likewise, `Z2(:news)` holds the second column:

$$\mathtt{Z2} = \begin{pmatrix} x_{5,2} & x_{1,2} & x_{2,2} & x_{3,2} & x_{4,2} \end{pmatrix} .$$

At this point the simple Fortran 90 command

```
Z2 = A*Z2 + B*CSHIFT(Z1,DIM=1,SHIFT=-1)
```

allows us to take one step in the row sweep (heading to the right) along all `N` rows. The updated `Z2` is incorporated into `X` before continuing. We next switch the roles of `Z1` and `Z2` for the next step in the sweep. By this renaming, the necessary data already resides in Z2, so we need only extract data from X into Z1. Taking advantage of this flip-flop technique, we minimize the number of times we need to retrieve and store the temporary vectors. Clearly, this process can be continued until we have swept along the rows, traveling to the right side of the matrix, and returning to the left. We emphasize that loading the appropriate data into Z1 and Z2 is basically 'dropping' the data down from the diagonals into a temporary vector, all with local in-processor data motion. Still, this maneuver requires gather/scatters (involving in-processor indirect addressing), and we rely on the Connection Machine compiler and run-time system to optimize this process as best it can.

2.3. Column sweep. We apply the same technique to the column sweep, except now extracting the data from the matrix is easier, at least conceptually. Loading the first element of each original column, we have:

$$\mathtt{Z1} = \begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} & x_{1,5} \end{pmatrix}$$

and

$$\mathtt{Z2} = \begin{pmatrix} x_{2,5} & x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} \end{pmatrix} \quad .$$

The Fortran 90 statement

```
Z2 = A*Z2 + B*CSHIFT(Z1,DIM=1,SHIFT=-1)
```

again performs the required updating, this time of the second row in the column sweep. Because of differing memory access patterns, the gather/scatters are presumably more efficient for one of the directions (row or column sweep). Judging from our experiments, however, this does not seem to be the case.

3. Results.

3.1. Comparison of serial and CM-5 code. To establish a baseline for comparison, we ran an optimized version of the serial ADI algorithm on the SPARC processor which serves as the interface for the CM-5. Figure 3 compares these run times to those produced by our skewed-matrix CM-5 algorithm. Note the (expected) quadratic curve for the serial code, as compared with the approximately linear shape of the CM-5 run times. As the number of startup operations (code blocks and calls to run-time communication routines) is linear with the matrix axis length but the amount of arithmetic and communication grows quadratically, this tells us that the computations on the CM-5 are dominated by overhead for this range of problem sizes. We then compared the linear-skew method to non-skewed ADI with explicit matrix transposition between passes to reduce calculations to purely in-processor sweeps. See Figures 4 and 5. These results show that for smaller problem sizes the explicit transpose took less wall clock time than the skewed method, while for larger sizes the linear-skewed method was slightly better. The large difference between CM-busy time and elapsed time reflects the CM-5 setup overhead associated with the interleaving of communication and computation at each row and column update step.

For Figure 6 we ran the same skew-vs-transpose test on an earlier model of the Connection Machine, the CM-200. This computer is, unlike the CM-5, a fine-grained parallel machine. Therefore we expect that our fine-grained SIMD algorithm should perform especially well on it. The CM-Fortran timings bear this out. We found it necessary, however, to call an explicit library routine to implement in-processor indirect addressing.

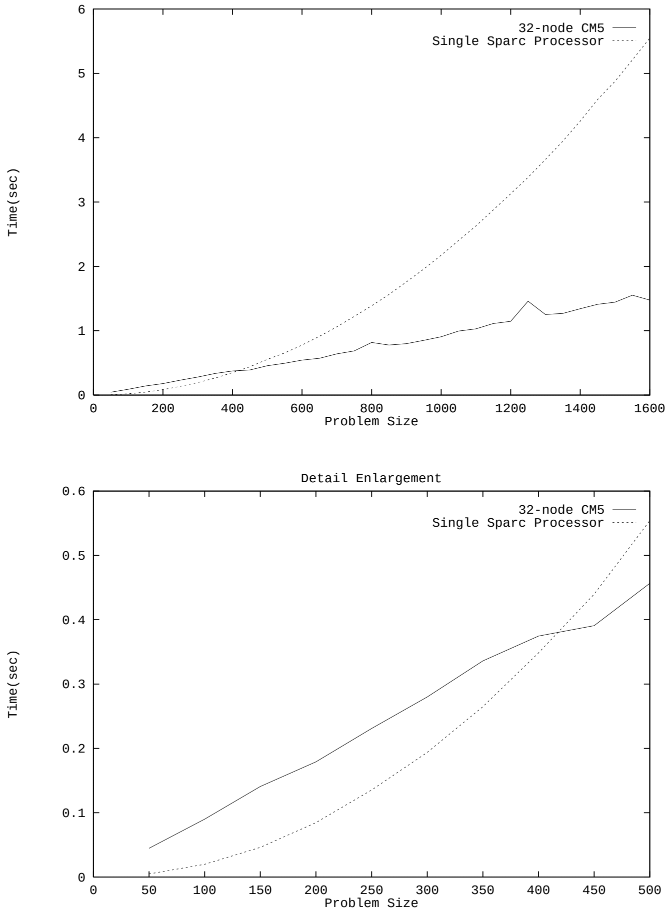

FIG. 3. *Plot of time taken to complete one full ADI step (i.e. sweep right and left along the rows, then sweep down and up along the columns). The matrix X is square.*

3.2. Adding a third dimension. When dealing with recursive algorithms having limited parallelism, it can be beneficial to introduce an extra "embarassingly-parallel" dimension to the problem by computing multiple results simultaneously. For fluid flow simulation, this permits examination of a suite of initial flow patterns for stability analysis. For imaging of seismic exploration data, one can correct or refine the subsurface velocity model during the computations.

To see how this affects our performance, we declared `X` as a three-dimensional array, with the third dimension representing a fixed number of copies of our original array. By inserting inner loops over this third dimension, we were able to cut across the grain of either the row sweeps or column sweeps, allowing the CM-5 to parallelize our previous code more efficiently. For example, for the first step in the row sweep heading right, we now are able to update multiple copies of the second column. Having duplicated our original problem along a third axis as described above, we obtain the per-matrix speed-up plotted in Figure 7 for a problem size of 50. The dramatic improvement with the number of copies and the leveling off of the performance curve

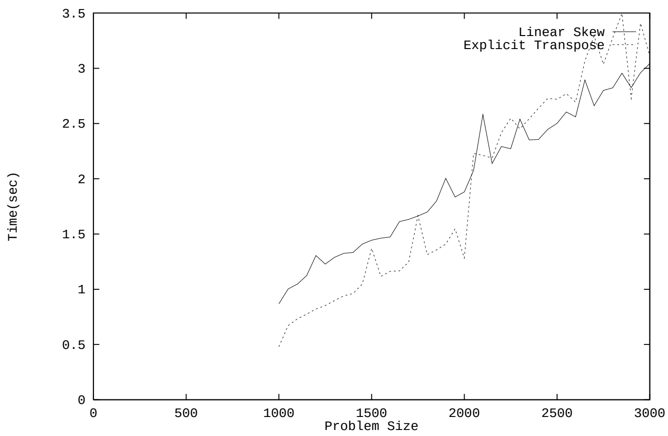

FIG. 4. *Wall clock time versus problem size.*

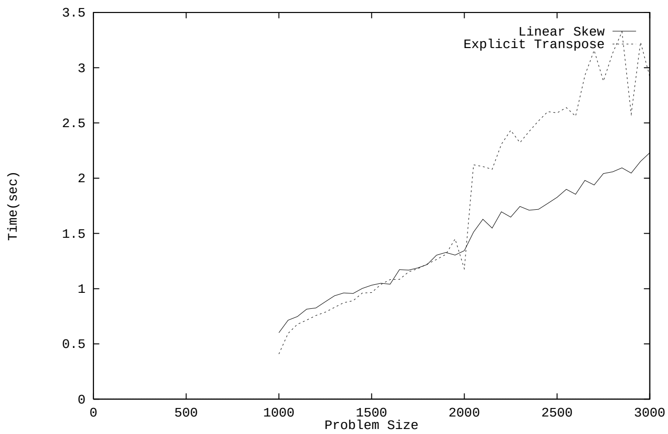

FIG. 5. *CM-busy time versus problem size.*

at around a thousand copies both reaffirm that overhead is the main contributor to run times for smaller numbers of copies. (If overhead were not a dominating factor, the theoretical performance ought to match that of a single ADI calculation for a problem size m times larger.)

4. Conclusions. On paper, our linear-skewed ADI algorithm allows the Fortran-90 programmer to cleanly access the parallelism of the Connection Machine and restrict communication to efficient, nearest-neighbor circular shifts. In practice, the direct method of repeatedly transposing the matrix between sweeps outperformed our approach on the CM-5 for all but the largest problem sizes. The basic reason is overhead. ADI has only O(1) flops per matrix element in each sweep, and hence places a premium on efficiency. On the CM-5, the run time is dominated by the number of times computation and communication are synchronized. In the explicit transpose method, synchronization occurs only twice per sweep. In the linear-skewed method, it occurs during every individual step of the ADI recursion and hence grows linearly

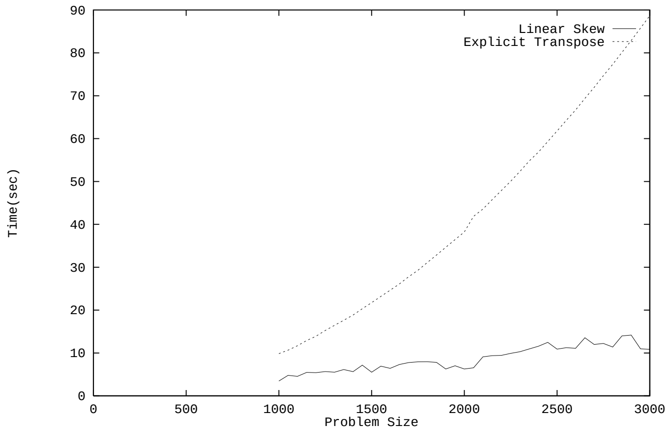

FIG. 6. *8K CM-200 elapsed time versus problem size. (Courtesy Julian Cabrera, Western Geophysical.)*

with the matrix axis length. In both cases, performance is measured in disappointing fractions of a percent of peak arithmetic efficiency. Indeed the cross-over point where ordinary scalar ADI was faster was a dimension of about $n = 300$.

In two niches, however, our alternative algorithm is superior. First, on the truly fine-grained massively parallel CM-200, the linear-skew method is an order of magnitude faster than repeated matrix transposition for large problems sizes. Second, when multiple right-hand sides are being solved the CM-5 overhead that prevents linear-skew from being effective for a single right-hand side remains constant while the amount of computation grows quadratically. This increases the overall efficiency of linear-skewed ADI by an order of magnitude or more.

REFERENCES

[1] E. Baum, *Final Technical Report - Turbulent Wakes*, Contract No. N00014-72-C-0448, TRW Syst. Group, TRW Inc., Dec. 31, 1973.

[2] E. Baum, E. Ndefo, *A Temporal ADI Technique*, AIAA Computational Fluid Dynamics Conference, July 1973, pp. 133-140.

[3] J. F. Claerbout, *Imaging the Earth's Interior*, 1985, Blackwell Scientific, Palo Alto.

[4] J. Douglas, Jr., *On the Numerical Integration of* $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{\partial u}{\partial t}$ *by Implicit Methods*, J. Soc. Ind. & Appl. Math., vol. 3, no. 1, Mar. 1955, pp. 42-65.

[5] J. Douglas, Jr., J. E. Gunn, *A General Formulation of the Alternating Direction Methods*, Part I, Parabolic and Hyperbolic Problems, Numer. Math., Bd. 6, 1964, pp. 428-453.

[6] A. R. Gourlay, A. R. Mitchell, *Alternating Direction Methods for Hyperbolic Systems*, Numer. Math., Bd. 8, Heft 2, Apr. 4, 1966, pp. 137-149.

[7] S. L. Johnsson, Y. Saad, M. H. Schultz, *Alternating Direction Methods on Multiprocessors*, SIAM J. Sci. Stat. Comput., 8(1987), pp. 686-700.

[8] D. W. Peaceman, H. H. Rachford, Jr., *The Numerical Solution of Parabolic and Elliptic Differential Equations*, J. Soc. Ind. & Appl. Math., vol. 3, no. 1, Mar. 1955, pp. 28-41.

[9] V. I. Polezhaev, *Numerical Solution of the System of Two-Dimensional Unsteady Navier-Stokes Equations for a Compressible Gas in a Closed Region*, Fluid Dyn.(2), Mar.-Apr. 1967, pp. 70-74.

[10] D. H. Rudy, D. J. Morris, *Time-Asymptotic Solution of the Navier-Stokes Equations for the Shear Flows Using An Alternating-Direction Implicit Method*, NASA Technical Note, TN D-8217(1976).

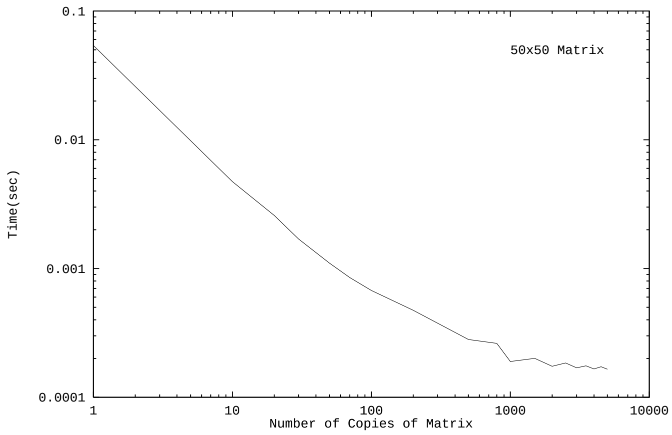

FIG. 7. *Plot of time taken to do one ADI iteration (with 50 by 50 problem being solved) versus number of copies (size of third dimension of X). Note: both axes are logarithmic scales.*

[11] *CM Fortran Release Notes*, Version 2.1, November 1993, Thinking Machines Corporation.

[12] Y. Saad, M. H. Schultz, *Alternating Direction Methods on Multiprocessors: an Extended Abstract*, Research Report 381, Dept. Computer Science, Yale University, New Haven, CT, 1985.