NASA Contractor Report 178337

ICASE REPORT NO. 87-43

ICASE

IMPLEMENTATION OF AN ADI METHOD ON PARALLEL COMPUTERS

Raad A. Fatoohi

Chester E. Grosch

Contract No. NAS1-18107
July 1987

(NASA-CR-178337) IMPLEMENTATION OF AN ADI METHOD ON PARALLEL COMPUTERS Final Report (NASA) 28 p Avail: NTIS HC A03/MF A01 CSCL 09B

N87-27446

Unclas

G3/62 0092570

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

NASA

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

IMPLEMENTATION OF AN ADI METHOD ON PARALLEL COMPUTERS

Raad A. Fatoohi *and* *Chester E. Grosch*

Old Dominion University

Norfolk, VA 23508

and

Institute for Computer Applications in Science and Engineering

NASA Langley Research Center, Hampton, VA 23665

ABSTRACT

In this paper we discuss the implementation of an ADI method for solving the diffusion equation on three parallel/vector computers. The computers were chosen so as to encompass a variety of architectures. They are: the MPP, an SIMD machine with 16K bit serial processors; FLEX/32, an MIMD machine with 20 processors; and CRAY/2, an MIMD machine with four vector processors. The Gaussian elimination algorithm is used to solve a set of tridiagonal systems on the Flex/32 and Cray/2 while the cyclic elimination algorithm is used to solve these systems on the MPP. The implementation of the method is discussed in relation to these architectures and measures of the performance on each machine are given. Simple performance models are used to describe the performance. These models highlight the bottlenecks and limiting factors for this algorithm on these architectures. Finally conclusions are presented.

This research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18107 while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665.

1. Introduction

It appears that future supercomputers will be multiprocessors; in fact some have already appeared. There are a number of important, unresolved questions concerning these multiprocessor computers. Among these issues are: should they consist of a few, rather powerful processors or many, very much less powerful processors, or something in between? Should the new computers be SIMD or MIMD? There is a natural expectation that the multiprocessors with a few, powerful processors will have an MIMD architecture and that the others will have SIMD architectures. Another issue is the communication among the processors. How the memory is connected to the processors and how these processors are connected to each other? Should the interconnection scheme be a lattice, a bus, a switch, or something else? It is sterile at this point to argue what is the "best" combination of number of processors and power per processor and what is the "best" interconnection scheme. Rather, carrying out experiments with existing multiprocessor computers would appear to be of greater value.

Despite the fact that the most powerful existing parallel/vector computers can perform at peak rates of several hundred MFLOPS, the average processing rates of many codes are in the range of 20 to 30 MFLOPS [3]. In part, this can be explained by invoking Amdahl's law,

$$S = \frac{1}{1 + f\,(R^{-1} - 1)},$$

where S is the speedup, f is the fraction of the code that can be parallelized/vectorized, and R is the parallel/vector to scaler speed ratio. Calculations with this formula show that nearly all the code must be parallelized/vectorized in order to achieve a substantial speedup. It is also clear that increasing R will only give a very modest improvement for a fixed f.

In this paper we describe the implementation of an ADI method for the solution of the diffusion equation on three different parallel architectures. These architectures are: the MPP, an SIMD machine with 16K serial 1-bit processors; Flex/32, an MIMD machine with 20 processors based on 32 bit NSC 32032 microprocessors; and Cray/2, an MIMD machine with four powerful vector processors. The method and two algorithms for solving a set of tridiagonal systems are described in section 2. The implementation of the method on these architectures and simple performance models are described in sections 3 through 5. Finally, section 6 contains a comparison of performance at the problem solving level and some concluding remarks.

2. The numerical method

Consider the diffusion equation,

$$\frac{\partial U}{\partial t} = \nabla^2 U, \tag{2.1}$$

to be solved in $0 < t \le T$ and the square region R: $0 \le x \le 1$, $0 \le y \le 1$ with boundary values at $U(0,y)$, $U(1,y)$, $U(x,0)$, $U(x,1)$. Consider replacing R by a net whose mesh points are denoted by $x_i = (i-1)\Delta x$, $y_j = (j-1)\Delta y$ where $i = 1, 2, \ldots, N+1$; $j = 1, 2, \ldots, M+1$. The numerical method used to approximate equation (2.1) is based on an Alternating Direction Implicit (ADI) method for solving parabolic equations [1]. This method consists of two half steps to advance the solution one full step in time. Let Δt be the full time step and apply the forward difference operator to equation (2.1) for the time derivative, giving

$$U_{i,j}^{n+1/2} - U_{i,j}^{n} = \frac{\Delta t}{2}(\delta_x^2\, U_{i,j}^{n+1/2} + \delta_y^2\, U_{i,j}^{n}), \tag{2.2}$$

$$U_{i,j}^{n+1} - U_{i,j}^{n+1/2} = \frac{\Delta t}{2}(\delta_x^2\, U_{i,j}^{n+1/2} + \delta_y^2\, U_{i,j}^{n+1}), \tag{2.3}$$

where only one of the space derivatives is evaluated at the advanced time level; this restriction is imposed in order to produce a set of tridiagonal equations. Applying the centered second difference operator in equations (2.2) and (2.3) for the space derivatives, we get

$$U_{i,j}^{n+1/2} - U_{i,j}^{n} = \frac{\Delta t}{2}\left[\frac{U_{i+1,j}^{n+1/2} - 2U_{i,j}^{n+1/2} + U_{i-1,j}^{n+1/2}}{(\Delta x)^2} + \frac{U_{i,j+1}^{n} - 2U_{i,j}^{n} + U_{i,j-1}^{n}}{(\Delta y)^2}\right], \tag{2.4}$$

$$U_{i,j}^{n+1} - U_{i,j}^{n+1/2} = \frac{\Delta t}{2}\left[\frac{U_{i+1,j}^{n+1/2} - 2U_{i,j}^{n+1/2} + U_{i-1,j}^{n+1/2}}{(\Delta x)^2} + \frac{U_{i,j+1}^{n+1} - 2U_{i,j}^{n+1} + U_{i,j-1}^{n+1}}{(\Delta y)^2}\right]. \tag{2.5}$$

Equations (2.4) and (2.5) can be rewritten as:

$$\alpha\, U_{i-1,j}^{n+1/2} - (1+2\alpha)\, U_{i,j}^{n+1/2} + \alpha\, U_{i+1,j}^{n+1/2} = F_{i,j}, \tag{2.6}$$

$$\beta\, U_{i,j-1}^{n+1} - (1+2\beta)\, U_{i,j}^{n+1} + \beta\, U_{i,j+1}^{n+1} = G_{i,j}, \tag{2.7}$$

where

$$F_{i,j} = -\beta\, U_{i,j-1}^{n} - (1-2\beta)\, U_{i,j}^{n} - \beta\, U_{i,j+1}^{n}, \tag{2.8}$$

$$G_{i,j} = -\alpha\, U^{n+1/2}_{i-1,j} - (1-2\alpha)\, U^{n+1/2}_{i,j} - \alpha\, U^{n+1/2}_{i+1,j}, \tag{2.9}$$

$$\alpha = \Delta t \,/\, 2(\Delta x)^2, \tag{2.10}$$

$$\beta = \Delta t \,/\, 2(\Delta y)^2, \tag{2.11}$$

for $i = 1, 2, \ldots, N+1$; $j = 1, 2, \ldots, M+1$. Equation (2.6) represents a set of $M+1$ independent tridiagonal systems (one for each vertical line of the net) each of size $N+1$. Similarly, equation (2.7) represents a set of $N+1$ independent tridiagonal systems (one for each horizontal line of the net) each of size $M+1$. Equation (2.6) is solved for the set $\{U^{n+1/2}_{i,j}\}$ using the values of $U^{n}_{i,j}$ while equation (2.7) is solved for the set $\{U^{n+1}_{i,j}\}$ using the computed values of $U^{n+1/2}_{i,j}$. In order to solve these two sets of equations, the following boundary conditions are incorporated:

$$F_{1,j} \equiv U^{n+1/2}_{1,j} = U(0,y_j), \quad F_{N+1,j} \equiv U^{n+1/2}_{N+1,j} = U(1,y_j), \qquad \text{for } j = 1, 2, \ldots, M+1, \text{ and}$$

$$G_{i,1} \equiv U^{n+1}_{i,1} = U(x_i,0), \quad G_{i,M+1} \equiv U^{n+1}_{i,M+1} = U(x_i,1), \qquad \text{for } i = 1, 2, \ldots, N+1.$$

In brief, the ADI method is implemented in two steps. In the first step, the set $\{F_{i,j}\}$ is computed, using equation (2.8), followed by solving $M-1$ tridiagonal systems (excluding the boundary systems), using equation (2.6). With $U^{n+1/2}$ known, the set $\{G_{i,j}\}$ is computed, using equation (2.9), followed by solving $N-1$ tridiagonal systems (excluding the boundary systems), using equation (2.7). These two steps are needed to advance the solution one full time step.

The main issue in implementing the ADI method on a parallel computer is choosing an efficient algorithm for the solution of tridiagonal systems. The selection of the algorithm depends on the amount of hardware parallelism available on the computer, storage requirements, and some other factors. Two algorithms are considered here: Gaussian elimination and cyclic elimination. Although these algorithms are described in the literature (see [8] for details), they are briefly described here for the sake of completeness.

Consider solving a single tridiagonal system of n equations,

$$a_i\, x_{i-1} + b_i\, x_i + c_i\, x_{i+1} = d_i, \tag{2.12}$$

for $i = 2, 3, \ldots, n-1$, and the boundary conditions,

$$x_1 = d_1, \tag{2.13}$$

$$x_n = d_n. \tag{2.14}$$

The Gaussian elimination algorithm, based on an LU decomposition of the tridiagonal matrix, has two stages: the forward elimination and the back substitution. In the forward elimination stage two auxiliary vectors, w and g, are computed as follows:

$$w_1 = 0.0, \tag{2.15a}$$

$$w_i = c_i \,/\, (b_i - a_i\, w_{i-1}), \qquad i = 2, 3, \ldots, n-1; \tag{2.15b}$$

$$g_1 = d_1, \tag{2.16a}$$

$$g_i = (d_i - a_i\, g_{i-1}) \,/\, (b_i - a_i\, w_{i-1}), \qquad i = 2, 3, \ldots, n-1. \tag{2.16b}$$

The values of x_i are obtained in the back substitution stage as follows:

$$x_n = d_n, \tag{2.17a}$$

$$x_i = g_i - w_i\, x_{i+1}, \qquad i = n-1, n-2, \ldots, 1. \tag{2.17b}$$

Gaussian elimination is an inherently serial algorithm because of the recurrence relations in both stages of the algorithm. However, if one is faced with solving a set of independent tridiagonal systems, then Gaussian elimination will be the best algorithm to use on a parallel computer. This means that all systems of the set are solved in parallel. In this case we obtain both the minimum number of arithmetic operations and the maximum parallelism.

The cyclic elimination algorithm, also called odd-even elimination [7] or parallel cyclic reduction [8], is a variant of the cyclic reduction algorithm [8] applying the reduction procedure to all of the equations and eliminating the back substitution phase of the algorithm. It can be described as follows. Assume that $n = 2^r$ where r is an integer and

$$x_i = 0, \qquad \text{for } i \le 0 \text{ and } i > n. \tag{2.18}$$

Solving equation (2.12) for x_i and the corresponding equations for x_{i-1} and x_{i+1}, we have:

$$x_{i-1} = (d_{i-1}/b_{i-1}) - (a_{i-1}/b_{i-1})\, x_{i-2} - (c_{i-1}/b_{i-1})\, x_i, \tag{2.19}$$

$$x_i \;= (\, d_i \,/\, b_i \,) - (\, a_i \,/\, b_i \,)\, x_{i-1} - (\, c_i \,/\, b_i \,)\, x_{i+1}, \tag{2.20}$$

$$x_{i+1} = (d_{i+1}/b_{i+1}) - (a_{i+1}/b_{i+1})\, x_i \quad - (c_{i+1}/b_{i+1})\, x_{i+2}. \tag{2.21}$$

Substitute for x_{i-1} and x_{i+1} in equation (2.19), to get

$$a_i^{(1)}\, x_{i-2} + b_i^{(1)}\, x_i + c_i^{(1)}\, x_{i+2} = d_i^{(1)}, \qquad i = 2, 3, \ldots, n-1, \tag{2.22}$$

where

$$a_i^{(1)} = -\,(a_i \,/\, b_i)\,(a_{i-1} \,/\, b_{i-1}), \tag{2.23}$$

$$b_i^{(1)} = 1 - (a_i \,/\, b_i)\,(c_{i-1} \,/\, b_{i-1}) - (a_{i+1} \,/\, b_{i+1})\,(c_i \,/\, b_i), \tag{2.24}$$

$$c_i^{(1)} = -\,(c_i \,/\, b_i)\,(c_{i+1} \,/\, b_{i+1}), \tag{2.25}$$

$$d_i^{(1)} = (d_i \,/\, b_i) - (a_i \,/\, b_i)\,(d_{i-1} \,/\, b_{i-1}) - (c_i \,/\, b_i)\,(d_{i+1} \,/\, b_{i+1}). \tag{2.26}$$

The above process eliminates the odd variables in the even equations and the even variables in the odd equations by performing elementary row operations. The resulting system is again tridiagonal of the same form as equation (2.12) but with different coefficients (a_i, b_i, c_i) and forcing terms (d_i). This process can be repeated for r steps ($r = \log_2 n$) until one set of equations remains. These equations are

$$a_i^{(r)}\, x_{i-2^r} + b_i^{(r)}\, x_i + c_i^{(r)}\, x_{i+2^r} = d_i^{(r)}, \qquad i = 2, 3, \ldots, n-1. \tag{2.27}$$

The terms x_{i-2^r} and x_{i+2^r} of equation (2.27) are really zero because they refer to values outside the range $1 \le i \le n$ and by equation (2.18) are zero. Thus equation (2.27) becomes

$$x_i = d_i^{(r)} \,/\, b_i^{(r)}, \qquad \text{for } i = 2, 3, \ldots, n-1. \tag{2.28}$$

The cyclic elimination procedure therefore requires the computation of new sets of coefficients and forcing terms, for levels $k = 1, 2, \ldots, r$, from

$$a_i^{(k)} = -\,A_i\, A_{i-K}, \tag{2.29}$$

$$b_i^{(k)} = 1 - A_i\, C_{i-K} - A_{i+K}\, C_i, \tag{2.30}$$

$$c_i^{(k)} = -\,C_i\, C_{i+K}, \tag{2.31}$$

$$d_i^{(k)} = D_i - A_i\, D_{i-K} - C_i\, D_{i+K}, \tag{2.32}$$

where

$$A_i \equiv \frac{a_i^{(k-1)}}{b_i^{(k-1)}}, \quad C_i \equiv \frac{c_i^{(k-1)}}{b_i^{(k-1)}}, \quad D_i \equiv \frac{d_i^{(k-1)}}{b_i^{(k-1)}}, \quad K \equiv 2^{k-1}, \quad i = 2, 3, \ldots, n-1, \tag{2.33}$$

followed by computing the values of x_i, using equation (2.28). Note that for $k = 1$ equations (2.29) to (2.32) are equivalent to equations (2.23) to (2.26) with $a_i^{(0)} = a_i$, $b_i^{(0)} = b_i$, $c_i^{(0)} = c_i$, and $d_i^{(0)} = d_i$.

The ADI procedure to solve equation (2.1) can be applied for any set of boundary and initial conditions. A test problem is developed by setting U to zero everywhere at $t = 0$ on the interior and

$$U(0,y) = 1 + 0.25(y - 3y^2), \quad U(1,y) = 1 + 0.25(2y - 3y^2 - 3),$$

$$U(x,0) = 1 + 0.25(3x^2 - 6x), \quad U(x,1) = 1 + 0.25(3x^2 - 5x - 2),$$

for all time. The steady state solution to this problem is,

$$U = 1 + \frac{1}{4}\left[3(x^2 - y^2) + xy - 6x + y\right].$$

3. Implementation on the MPP

The Massively Parallel Processor (MPP) is a large-scale SIMD processor developed by Goodyear Aerospace Co. for NASA Goddard Space Flight Center [2]. The MPP is a back-end processor for a VAX-11/780 host, which supports its program development and I/O needs.

The MPP has three major units: the Array Unit (ARU), Array Control Unit (ACU), and staging memory. The ARU is a square array of 128×128 bit-serial Processing Elements (PE's). Each PE has a local 1024 bit random access memory and is connected to its four nearest neighbors with programmable edge connections. Arithmetic in each PE is performed in bit serial fashion using a serial-by-bit adder. The ACU supervises the PE array processing, performs scaler arithmetic, and shifts data across the PE array. The staging memory is a large scale data buffer that stores, permutes, and transfers data between external devices and the array. The MPP has a cycle time of 100 nsec.

The MPP's high level language is MPP Pascal [9]. It is a machine-dependent language which has evolved directly from the language Parallel Pascal defined by Reeves [10]. Parallel Pascal is an extended version of serial Pascal with a convenient syntax for specifying array operations. These extensions provide a parallel array data type for variables to be stored in the array memory and operations on these parallel arrays. One of these operations that is used extensively in this work is shifting. The shift function shifts data by the amount specified for each dimension and shifts zeros in at the edges of the array. It has the form *shift* (a, k_1, k_2) where a is the parallel array to be shifted and k_1 and k_2 specify the amount of shift in the first and second dimensions of a.

The ADI method, described in section 2, was implemented on the MPP for a 128×128 mesh point problem. In each processor, data corresponding to one mesh point is stored. The main program as well as the ADI procedure were written in MPP Pascal. The main program, run on the VAX, handles input and output, initialization of the computational domain, and calling the ADI procedure. The ADI procedure, which was executed entirely on the array, computes the coefficients, forcing terms, and solves two sets of 128 tridiagonal systems. The tridiagonal systems are solved by the cyclic elimination algorithm, described in section 2, for all rows and all columns. This is done in parallel on the array with a tridiagonal system of 128 equations being solved on each row or column. In this case the vectors x_i, a_i, b_i, c_i, d_i of equation (2.12) become the matrices $x_{i,j}$, $a_{i,j}$, $b_{i,j}$, $c_{i,j}$, $d_{i,j}$. After solving each set of the tridiagonal systems, all points of the domain are updated except the boundary points. This is implemented by masking out the boundary columns (or the boundary rows) of the array.

The ADI procedure is reasonably efficient and requires only 5 parallel arrays of floating point numbers. The procedure contains mostly matrix operations with a few scaler operations (for computing the coefficients) and has no vector operations. However, the cyclic elimination algorithm has some hidden defects. For each level of the elimination process, a set of data is shifted off the array and an equal set of zeros is shifted onto the array. Since all of the processors are executing the same instruction at every cycle, some of these processors may not be doing useful work; here they are either multiplying by zero or adding a zero. This is a problem with many algorithms on SIMD machines.

Table I contains the execution time and the processing rate of the ADI procedure for the 128×128 problem. The processing rate is determined by counting only the arithmetic operations (addition, multiplication, and division). Data transfer operations, vital as they are in this work, are not counted as floating point operations. However, all the arithmetic operations including those operations which do not contribute to the solution are counted.

In order to develop a model for the cost of implementing the ADI method on the MPP, the arithmetic operations as well as the data transfer operations of the method are counted and the cost of each of these operations on the MPP is measured. Table II contains the operation counts for one pass of the ADI method using the cyclic elimination algorithm. These operations are also required for the second pass of the method and the total number of operations required for both passes will be twice the number that is given in Table II. Table III contains the measured values of the elementary operations on the MPP. These values were obtained by measuring the execution time of each operation using a loop of length 1000.

The execution time of the ADI method on the MPP, T, is computed as follows:

$$T = T_{comp} + T_{comm},$$

$$T_{comp} = t_c\,(N_a\,C_a + N_m\,C_m + N_d\,C_d),$$

$$T_{comm} = t_c\,(N_{sh}\,C_{sh} + N_{st}\,C_{st}),$$

where

T_{comp} : Computation cost,

T_{comm} : Communication cost,

t_c : Machine cycle time,

N_a : Number of additions,

N_m : Number of multiplications,

N_d : Number of divisions,

N_{sh} : Number of shift operations,

N_{st} : Number of steps involved in all shift operations,

C_a : Cost of each add operation,

C_m : Cost of each multiply operation,

C_d : Cost of each divide operation,

C_{sh} : Startup cost of shift operation,

C_{st} : Cost of shifting one step within shift operation.

The values of N_a, N_m, N_d, N_{sh}, and N_{st} are obtained from Table II while the values of C_a, C_m, C_d, C_{sh}, and C_{st} are obtained from Table III. Table IV contains the estimated times based on this model. The computation cost contributes about 75% of the total cost and the communication cost contributes about 25% of the total cost. The scaler operations have very little impact on the performance of the method since they are inexpensive and may overlap with the array operations. It is also found that the cost of solving the tridiagonal systems represents more than 95% of the total cost of the method.

The MPP has two sets of primitives for implementing the elementary arithmetic operations; IBM and VAX format primitives. The IBM format primitives are more efficient than the VAX format primitives but were unavailable for MPP Pascal programs at the time this research was conducted. The peak performance rates of the arithmetic operations are computed based on IBM format primitives. These rates for operations involving 32 bit 128×128 floating point matrices are 430 MFLOPS for addition, 216 MFLOPS for multiplication, and 165 MFLOPS for division. If we use these peak rates and the operation counts from Table II, the maximum possible rate of the ADI method will be 236 MFLOPS. Therefore, the measured processing rate is about 57% of the maximum possible rate. The peak performance rate is not achieved because of the data transfer costs and the inefficiency of the VAX format primitives. Previous work [5] showed that the efficiency of these primitives can have a major impact on the performance of the machine.

4. Implementation on the Flex/32

The Flex/32 is an MIMD shared memory multiprocessor based on 32 bit National Semiconductor 32032 processor [4]. The results presented here were obtained using the 20 processor machine that is now installed at NASA Langley Research Center.

The machine has ten local buses; each connects two processors. These local buses are connected together and to the common memory by a common bus. The 2.25 Mbytes of the common memory is accessible to all processors. Each of processors 1, 2, and 3 contains 4 Mbytes of local memory. All other processors contain 1 Mbyte each. Each processor has a cycle time of 100 nsec.

The UNIX operating system is resident in processors 1 and 2. These processors are also used for software development and for loading and booting the other processors. Processors 3 through 20 run the Multicomputing Multitasking Operating System and are available for parallel processing.

The Flex/32 system software has a special concurrent version of Fortran 77. Concurrent Fortran comprises the standard Fortran 77 language and extensions that support concurrent processing. Among the constructs available for implementing parallel programs are: "shared", to identify variables that are shared between processors; "process", to define and start the execution of a process on a specified processor; "lock", to lock a shared variable if it is not locked by any other process; and "unlock", to release a locked variable.

The ADI method, described in section 2, was implemented on the Flex/32 using p processors with $p = 1, 2, 4, 8$, and 16 and for problems of sizes $n \times n$ mesh points with $n = 64, 128$, and 256. The main program as well as the ADI subroutine were written in Concurrent Fortran.

In order to implement the method in parallel, the domain is decomposed first vertically into n by n / p strips, for the first pass of the method, and then horizontally into n / p by n strips, for the second pass of the method. In the first pass, a set of n / p tridiagonal systems (each of n equations) corresponding to the vertical lines of the net are solved for each strip using the Gaussian elimination algorithm, described in section 2. In the second pass, likewise, a set of n / p tridiagonal systems (each of n equations) corresponding to the horizontal lines of the net are solved for each strip using the same algorithm. In our implementation each strip is given to a process. In addition to initialization of the domain and input and output operations, the main program creates and starts the execution of the processes on specified processors with each process assigned to a separate processor. The processes are spawned only once at the beginning of the program and are used for both passes of the method. The data corresponding to the domain is stored in the common memory. Also, the values of the boundary points, computed in the main program, are stored in the common memory. The forcing terms and the temporary matrices $w_{i,j}$ and $g_{i,j}$, equations (2.15) and (2.16), for each strip are computed and stored in the local memory of each processor.

The ADI method for each strip was implemented by: computing the coefficients and forcing terms of the tridiagonal systems and solving these systems for all columns first and then for all rows of that strip. Upon completing a pass, each process signals to the other processes by incrementing a counter. All of the processes will wait until each of them has finished the current pass. A lock is assigned to the counter to ensure this sequence of the events.

The measured execution times and the processing rates for problems of sizes 64×64, 128×128, and 256×256 are listed in Table V. These results were obtained using a timer with a 20 msec resolution. The speedup and efficiency as functions of the number of processors of these problems are listed in Table VI. The efficiency of the method ranges from 35%, for the 64×64 problem using 16 processors, to 95%, for the 256×256 problem using two processors.

The execution time of the ADI method on p processors of the Flex/32, T_p, can be modeled as follows:

$$T_p = T_{cmp} + T_{ovr},$$

where T_{cmp} is the computation time and T_{ovr} is the overhead time. Since the load is distributed evenly between the processors with no extra computations, the computation time on p processors can be computed by

$$T_{cmp} = T_1 \,/\, p,$$

where T_1 is the computation time using a single processor.

The overhead time is

$$T_{ovr} = T_{spn} + T_{cma} + T_{syn},$$

where

T_{spn} is the spawning time of p processors,

T_{cma} is the total common memory access time,

T_{syn} is the total synchronization time.

These times can be estimated as follows:

$$T_{spn} = p\ t_{spn},$$

$$T_{cma} = n^2\ k_{cma}\ t_{cma}\ /\ p,$$

$$T_{syn} = p\ k_{lck}\ t_{lck},$$

where

p	the number of processors,
n	the number of mesh points in each dimension,
t_{spn}	the time to spawn one process; a reasonable value is 13 msec,
t_{cma}	the time to access an element of a matrix in the common memory; a reasonable value is 4 µsec,
t_{lck}	the time to lock and unlock a variable in the common memory; a reasonable value is 47 µsec,
k_{cma}	the number of times each strip is referenced ($k_{cma} = 6$),
k_{lck}	the number of times the common variable is locked and unlocked for each process ($k_{lck} = 2$).

Table VII contains the estimated values of the computation time and the overhead times of the ADI method. The synchronization time is insignificant because the routines that provide the locking mechanism are very efficient. The spawning time dominates the overhead time for large number of processors. This resulted in the degradation on the performance of the method for large number of processors. However, the overhead time is dominated by the common memory access time for small number of processors. The total estimated times are less than the measured times (by less than 3%) because of the exclusion of the impact of the memory contention in the analysis. This contention results from having more than one processor trying to access the common memory at the same time. A memory contention factor which is a function of p may be included in T_{cma}. This factor has estimated values ranged from 1.1 for 2 processors to 3.5 for 16 processors. Finally, it was realized that the common memory access operations may overlap with the spawning operations during the first pass of the method.

5. Implementation on the Cray/2

The Cray/2 is an MIMD supercomputer with four Central Processing Units (CPU), a foreground processor which controls I/O and a central memory. The central memory has 256 million 64 bit words organized in four quadrants of 32 banks each. Each CPU has access to one quadrant during each clock cycle. Each CPU has an internal structure very similar to Cray/1 with the addition of 16K words of local memory available for storage of vector and scaler data. The clock cycle is 4.1 nsec.

The ADI method, described in section 2, was implemented on one processor of the Cray/2 for domains of sizes ranging from 64×64 to 1024×1024 mesh points. The code, in each case, was written and run through the Cray/2 vectorizing Fortran compiler, CFT/2 [6]. Each set of the tridiagonal systems are solved by the Gaussian elimination algorithm, described in section 2, for all systems of the set in parallel. This means that each element of w_i (equations (2.15)) is computed for all the tridiagonal systems before moving to the next element. This process is repeated in computing the elements of g_i (equations (2.16)) and x_i (equations (2.17)). The implementation requires that the vectors w_i, g_i, x_i be changed to the matrices $w_{i,j}$, $g_{i,j}$, $x_{i,j}$. When these are done, all statements of the code vectorize fully and the recursion problem of the algorithm is eliminated.

The ADI method requires that the data is first referenced by vertical lines and then by horizontal lines. The CFT/2 compiler stores arrays by incrementing the leftmost index first (column major order). This means that referencing the data by vertical lines causes no problems because the increment between data elements is unity. However, in referencing the data by horizontal lines the increment between the elements will be equal to the number of variables in each column. In our implementation, this number ranges between 64 and 1024. This could cause a major problem on the Cray/2. The machine has 128 memory banks and consecutive elements of an array are stored in consecutive banks. Each bank has a cycle time of 57 clock periods. For a memory stride of 128, for example, words are drawn from the same bank and each word must wait the entire 57 clock periods to be moved from main memory, even if there is no competition for resources from other processors. As a result, strides that are divisible by 128 or any large power of two result in major performance reductions. This problem is overcome by storing the data as though it had a column length one greater than its actual length. Thus for a stride of 129 data elements are stored in sequential banks with a stride of one (stride of 129 mod 128 banks = 1).

The number of memory access operations can be reduced by using the local memory whenever that is possible. Once an array is stored in the local memory, none of its elements can be changed; all the elements are treated the same. The forcing terms $F_{i,j}$ and $G_{i,j}$, equations (2.8) and (2.9), are stored in the local memory.

Table VIII contains the execution time and the processing rate of the ADI method when the domain size is varied from 64×64 through 1024×1024 mesh points. There is up to 20% offset on the results depending on the memory traffic and the number of the active processes on the system. The processing rate is computed by counting the additions, multiplications, and divisions only. Division is counted as a single arithmetic operation. The number of arithmetic operations for one pass of the ADI method, using the Gaussian elimination algorithm for solving the tridiagonal systems, are listed in Table IX. The total number of each operation will be twice the number that is given in Table IX.

The time required to perform floating point operations, T_f, and memory access operations, T_m, on vectors of length L_{vcr} can be modeled as follows:

$$T_f = (\left\lceil \frac{L_{vcr}}{64} \right\rceil L_f + L_{vcr})\, N_{vcr}\, CP, \tag{5.1}$$

$$T_m = (\left\lceil \frac{L_{vcr}}{64} \right\rceil L_m + L_{vcr})\, N_{vcr}\, CP, \tag{5.2}$$

where

$\lceil x \rceil$: Next integer greater than or equal to x,

N_{vcr} : Number of vectors,

L_f : Length of a floating point functional unit = 23 *CP*s,

L_m : Length of data path between main memory and the registers = 56 *CP*s,

CP : Clock Period = 4.1 nsec.

Table X contains the results of applying equations (5.1) and (5.2) for the ADI method. The number of floating point operations is obtained from Table IX while the number of main memory access operations is 12 for each pass of the method. The multiplication time includes the time required for division where it is assumed that each division takes three times the multiplication time. The cost of scaler operations are not included in the model. The memory access time represents about 47% of the total estimated time. The estimated time for memory access does not take into account the competition for memory banks from other processors which causes a lower data transfer rate. The total estimated time exceeds the measured time for large domains. This is because for large problems more overlapping between different operations is expected and the impact of scaler operations is reduced. For these reasons, the performance rate of the method, Table VIII, increases for large problems.

Because the ADI method has more multiplications (including divisions) than additions, the time to implement the method can be considered as a summation of two portions; a portion with one operational pipeline and a portion with two pipelines. By using Table IX and assuming that the peak performance rate of one pipeline is 244 MFLOPS, ignoring the vector startup times, the maximum processing rate of the ADI method will be 357 MFLOPS. Therefore, the measured processing rate for the 128×128 problem is about 20% of the peak performance rate. If the startup time of the floating point units is included, the measured processing rate for the 128×128 problem will be about 28% of the peak processing rate of 262 MFLOPS. The major problems are accessing the main memory and the scaler portion of the code.

6. Comparisons and Concluding Remarks

The processing rate and execution time (Tables I, V, and VIII) are used to compare the performance of the ADI method on the three parallel architectures. Since two different algorithms are used to solve the tridiagonal systems, the number of arithmetic operations involved in these algorithms should be taken into consideration. The total number of arithmetic operations required per each interior point of the 128×128 domain for one pass of the method is 97 using the cyclic elimination algorithm (Table II) while it is 15 for the Gaussian elimination algorithm (Table IX). This is a factor of about 6.5 between these algorithms which makes cyclic elimination the less efficient algorithm. This explains why for the 128×128 problem the Cray/2 outperformed the MPP (by a factor of about 3.7) with respect to the execution time while the MPP outperformed the Cray/2 (by a factor of about 1.8) with respect to the processing rate.

The relative performance rate of the method is another way of measuring the performance. This is a measure of how well the algorithm has been mapped onto the architecture including the impact of the software used in the implementation. These relative performance rates for the 128×128 problem are 57% on the MPP, 66% on 16 processors of the Flex/32, and 28% on the Cray/2. On the MPP, the main problems are the communication cost of the algorithm and the inefficiency of the primitives used in MPP Pascal. On the Flex/32, the main problems are the spawning cost and the common memory access cost. On the Cray/2, the main problems are accessing the main memory and the scaler portion of the code.

Another measure of performance is the number of machine cycles required to solve the problem. This measure reduces the impact of technology on the performance of the machine. For the 128×128 problem, the method requires about 237 million cycles on the MPP; 1579 million cycles on the Cray/2; and 7400 million cycles on the Flex/32.

The performance model on the MPP was fairly accurate on predicting the execution times of the method based on the measured values of the elementary operations. The performance model on the Flex/32 showed that the spawning cost and the common memory access cost dominated the overhead cost; although it was hard to estimate the impact of memory accesses since these operations overlap with other operations. The performance model on the Cray/2 showed the costs of individual operations and their impacts on the overall performance of the machine.

In summary we have found that the ADI method can be mapped onto different parallel architectures. Two different algorithms were used to solve the tridiagonal systems and these algorithms were chosen to exploit the parallelism available on the architectures. The machine's hardware was the main consideration in selecting the algorithm while the software influenced the performance.

References

[1] Ames, W. F. , "Numerical Methods for Partial Differential Equations," 2nd ed., Academic Press, New York, 1977, pp. 251-255.

[2] Batcher, K. E., "Design of a Massively Parallel Processor," IEEE Trans. Computers, Vol. C-29, Sept. 1980, pp. 836-840.

[3] Dongarra, J. J., "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment," Tech. Memo. 23, Argonne National Lab., Argonne, Ill., 1985.

[4] Flex/32 Multicomputer System Overview, Flexible Computer Co., 1986.

[5] Fatoohi, R. A. and Grosch, C. E., "Solving the Cauchy-Riemann Equations on Parallel Computers," ICASE Report No. 87-34, NASA Langley Research Center, Hampton, VA., 1987.

[6] Fortran (CFT2) Reference Manual, Cray Research Inc. Publication SR-2007.

[7] Heller, D., "A Survey of Parallel Algorithms in Numerical Linear Algebra," SIAM Review, Vol. 20, No. 4, 1978, pp. 740-777.

[8] Hockney, R. W. and Jesshope, C. R., "Parallel Computers: Architecture, Programming and Algorithms," Adam Hilger, Ltd., Bristol, 1981.

[9] MPP Pascal Programmer's Guide, NASA Goddard Space Flight Center, Sept. 1986.

[10] Reeves, A. P., "Parallel Pascal: An Extended Pascal for Parallel Computers," J. Parallel & Distributed Computing, Vol. 1, 1984, pp. 64-80.

Problem size (mesh points)	Execution time (msec)	Processing rate (MFLOPS)
128×128	23.698	134

Table I. Execution time and processing rate for the ADI procedure on the MPP.

Operation	Add	Multiply	Divide	Shift	Steps shifted
1. Compute $F_{i,j}$, eq. (2.8)	2	3	-	2	2
2. Compute $A_{i,j}$, $C_{i,j}$, $D_{i,j}$, eqs. (2.33)	-	-	$3\,r$	-	-
3. Compute $a_{i,j}^{(r)}$, eq. (2.29)	-	r	-	r	$2^r - 1$
4. Compute $b_{i,j}^{(r)}$, eq. (2.30)	$2\,r$	$2\,r$	-	$2\,r$	$2\,(2^r - 1)$
5. Compute $c_{i,j}^{(r)}$, eq. (2.31)	-	r	-	r	$2^r - 1$
6. Compute $d_{i,j}^{(r)}$, eq. (2.32)	$2\,r$	$2\,r$	-	$2\,r$	$2\,(2^r - 1)$
7. Compute $x_{i,j}$, eq. (2.28)	-	-	1	-	-
Total operations	$4\,r + 2$	$6\,r + 3$	$3\,r + 1$	$6\,r + 2$	$6\,(2^r - 1) + 2$
Total operations for $r = 7$	30	45	22	44	764

Table II. Operation counts for one pass of the ADI method using the cyclic elimination algorithm for solving the tridiagonal systems, where $i = 1, 2, \ldots, n, j = 1, 2, \ldots, n,$ and $r = \log_2 n$.

Add	Multiply	Divide	One step shift	k step shift
824	877	1130	166	134 + 32 k

Table III. Measured execution times (in machine cycles) of the elementary operations on the MPP.

Computation time	Communication time	Total estimated time	Measured time
17.809	6.069	23.878	23.698

Table IV. Estimated and measured times (in milliseconds) of the ADI method on the MPP.

Problem size (mesh points)	Execution time (msec)	Processing rate (MFLOPS)
64×64	340	0.36
128×128	740	0.66
256×256	2320	0.85

Table V. Execution times and processing rates for the ADI method using 16 processors of the Flex/32.

Number of	64×64 points		128×128 points		256×256 points	
processors	speedup	efficiency	speedup	efficiency	speedup	efficiency
1	1.000	1.000	1.000	1.000	1.000	1.000
2	1.827	0.913	1.874	0.937	1.900	0.950
4	3.393	0.848	3.660	0.915	3.722	0.931
8	5.278	0.660	6.807	0.851	7.271	0.909
16	5.588	0.349	10.486	0.655	13.414	0.838

Table VI. Speedup and efficiency on the Flex/32.

No. of procs.	Computation time	Spawning time	Memory time	Total estimated time	Measured time
			64×64 points		
1	-	-	-	-	1900
2	950	26	49	1025	1040
4	475	52	25	552	560
8	238	104	12	354	360
16	119	208	6	333	340
			128×128 points		
1	-	-	-	-	7760
2	3880	26	197	4103	4140
4	1940	52	98	2090	2120
8	970	104	49	1123	1140
16	485	208	25	718	740
			256×256 points		
1	-	-	-	-	31120
2	15560	26	786	16372	16380
4	7780	52	393	8225	8360
8	3890	104	197	4191	4280
16	1945	208	98	2251	2320

Table VII. Estimated and measured times (in milliseconds) on the Flex/32.

Problem size (mesh points)	Execution time (msec)	Processing rate (MFLOPS)
64×64	1.684	69
128×128	6.474	74
256×256	24.270	80
512×512	94.037	83
1024×1024	364.248	86

Table VIII. Execution times and processing rates for the ADI method on one processor of the Cray/2.

Operation	Add	Multiply	Divide
1. Compute $F_{i,j}$, eq. (2.8)	2	3	-
2. Compute $w_{i,j}$, eqs. (2.15)	1	1	1
3. Compute $g_{i,j}$, eqs. (2.16)	2	2	1
4. Compute $x_{i,j}$, eqs. (2.17)	1	1	-
Total operations	6	7	2

Table IX. Operation counts for one pass of the ADI method using the Gaussian elimination algorithm for solving the tridiagonal systems.

Problem size (mesh points)	Memory access time	Addition time	Multiplication time	Total estimated time	Measured time
64×64	0.720	0.259	0.562	1.541	1.684
128×128	2.951	1.066	2.310	6.327	6.474
256×256	11.947	4.324	9.368	25.639	24.270
512×512	48.076	17.414	37.730	103.220	94.037
1024×1024	192.883	69.893	151.430	414.206	364.248

Table X. Estimated and measured execution times (in milliseconds) of the method on one processor of the Cray/2.

Standard Bibliographic Page

1. Report No. NASA CR-178337 ICASE Report No. 87-43	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle IMPLEMENTATION OF AN ADI METHOD ON PARALLEL COMPUTERS		5. Report Date July 1987
		6. Performing Organization Code
7. Author(s) Raad A. Fatoohi and Chester E. Grosch		8. Performing Organization Report No. 87-43
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225		10. Work Unit No. 505-90-21-01
		11. Contract or Grant No. NAS1-18107
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		13. Type of Report and Period Covered Contractor Report
		14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor: J. C. South — Submitted to J. Sci. Comput.

Final Report

16. Abstract

In this paper we discuss the implementation of an ADI method for solving the diffusion equation on three parallel/vector computers. The computers were chosen so as to encompass a variety of architectures. They are: the MPP, an SIMD machine with 16K bit serial processors; FLEX/32, an MIMd machine with 20 processors; and CRAY/2, an MIMD machine with four vector processors. The Gaussian elimination algorithm is used to solve a set of tridiagonal systems on the FLEX/32 and CRAY/2 while the cyclic elimination algorithm is used to solve these systems on the MPP. The implementation of the method is discussed in relation to these architectures and measures of the performance on each machine are given. Simple performance models are used to describe the performance. These models highlight the bottlenecks and limiting factors for this algorithm on these architectures. Finally conclusions are presented.

17. Key Words (Suggested by Authors(s)) ADI method, MIMD machine, parallel algorithms, performance analysis, SIMD machine		18. Distribution Statement 62 - Computer Systems 64 - Numerical Analysis Unclassified - unlimited	
19. Security Classif.(of this report) Unclassified	20. Security Classif.(of this page) Unclassified	21. No. of Pages 26	22. Price A03