



Research Institute for Advanced Computer Science
NASA Ames Research Center

SOME FAST ELLIPTIC SOLVERS ON PARALLEL ARCHITECTURES AND THEIR COMPLEXITIES

E. GALLOPOULOS
Y. SAAD

April, 1989

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 89.16

NASA Cooperative Agreement Number NCC 2-387

(NASA-CR-188840) SOME FAST ELLIPTIC SOLVERS
ON PARALLEL ARCHITECTURES AND THEIR
COMPLEXITIES (Research Inst. for Advanced
Computer Science) 32 p

CSCL 09B

N92-11692

Unclass
0043026

63/62

1N-62
DATE OVERVIEW
43026
P-32

SOME FAST ELLIPTIC SOLVERS ON PARALLEL ARCHITECTURES AND THEIR COMPLEXITIES

E. GALLOPOULOS

Y. SAAD

April, 1989

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 89.16

NASA Cooperative Agreement Number NCC 2-387

PAGE 1 - ~~UNCLASSIFIED~~ ~~CONFIDENTIAL~~

SOME FAST ELLIPTIC SOLVERS ON PARALLEL ARCHITECTURES AND THEIR COMPLEXITIES

E. GALLOPOULOS

*Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801*

Y. SAAD

RIACS

Mail Stop 230-5

*NASA Ames Research Center
Moffet Field, California 94035*

ABSTRACT

The discretization of separable elliptic partial differential equations leads to linear systems with special block tridiagonal matrices. Several methods are known to solve these systems, the most general of which is the Block Cyclic Reduction (BCR) algorithm which handles equations with nonconstant coefficients. A method was recently proposed to parallelize and vectorize BCR. In this paper we discuss the mapping of BCR on distributed memory architectures and compare its complexity with that of other approaches including the Alternating-Direction method. We also describe a fast parallel solver, based on an explicit formula for the solution, which has parallel computational complexity lower than that of parallel BCR.

Keywords: Block cyclic reduction algorithm, parallel processing, partial fractions, hypercube computers, computational complexity, alternating direction algorithm.

1. Introduction. In this paper we are concerned with efficient parallel methods for solving block tridiagonal systems that arise from the discretization of the general separable elliptic equation

$$(1.1) \quad a(x) \frac{\partial^2 u}{\partial x^2} + b(x) \frac{\partial u}{\partial x} + c(x)u \\ + d(y) \frac{\partial^2 u}{\partial y^2} + e(y) \frac{\partial u}{\partial y} + g(y)u = f(x, y).$$

These systems are of great importance particularly because their solution may be required in the inner loop of an iterative procedure, in preconditioning more complex systems, or in the context of time-stepping techniques. When (1.1) is of Poisson type in one direction and is defined on a domain which allows separation of variables to be used ([27]), there exist special fast methods which for N unknowns achieve sequential complexity of $O(N \log N)$ ([4,5,11,26]) and parallel computational complexity of $O(\log N)$ ([6,18,25]).

We concentrate on methods which in principle do not rely on fast transforms and can thus be used to handle discrete equations as general as (1.1). We describe their mapping on parallel architectures and investigate their computational and communication complexities. The methods we discuss are block cyclic reduction (BCR), alternating direction implicit procedure (ADI), and a new explicit elliptic solver (EES).

Parallel BCR was recently introduced in [7] and [25]. Its implementation and performance were discussed for the case of the Alliant FX/8 shared memory vector multiprocessor in the former and for the Cray-1 in the latter. A very brief discussion of the mapping of the algorithm on hypercubes and multicluster shared memory architectures can be found in [6,23] and [9] respectively. We outline the algorithm and its parallelization in Section 2 and discuss its mapping on distributed memory architectures, particularly on hypercubes, in Section 2.1. Several mapping strategies are considered, depending on the size of the problem and the number of processors and their parallel arithmetic and communication complexities are discussed. In Section 2.2 we describe an implementation of BCR for massively parallel architectures. If scalar cyclic reduction is used to solve the tridiagonal systems then the parallel computational complexity of BCR is $O(\log n \log m)$, for a block tridiagonal system of n blocks each of dimension m . ($N = n \times m$). This is inferior to the $O(\log nm)$ complexity associated with FFT based methods. In Section 3 we introduce an $O(\log nm)$ parallel algorithm which we call Explicit Elliptic Solver (EES). This algorithm affords the same generality as BCR and is very simple to implement. Nevertheless, it may be impractical due to its requirement of a very large number of processors. In Section 4 we briefly discuss the use of these methods for the most general separable problem. In Section 5 we examine ADI methods. Although, strictly speaking, these are iterative techniques, their sequential complexity to achieve a level of accuracy that is comparable with discretization errors, is of the order of

$O(n^2 \log^2 n)$ when $m = n$. For an n^2 processor hypercube connected system, their parallel complexity is of order $O(\log^3 n)$. Moreover, a nonnegligible advantage is that they are far easier to implement than the BCR schemes. Finally in Section 6 we provide some concluding remarks.

2. Block Cyclic Reduction. Consider the more restricted form of Eq. (1.1)

$$(2.1) \quad a(x) \frac{\partial^2 u}{\partial x^2} + b(x) \frac{\partial u}{\partial x} + c(x)u + \frac{\partial^2 u}{\partial y^2} = f(x, y)$$

with Dirichlet boundary conditions on a rectangle. If we discretize with a 5 point stencil on a naturally ordered $n \times m$ grid we obtain the block tridiagonal system:

$$(2.2) \quad \begin{pmatrix} A & -I & & & \\ -I & A & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & A & -I \\ & & & \ddots & \ddots & \ddots \\ & & & & -I & A & -I \\ & & & & & -I & A \end{pmatrix} \begin{pmatrix} v_1 \\ \vdots \\ v_j \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} f_1 \\ \vdots \\ f_j \\ \vdots \\ f_n \end{pmatrix}$$

A and I are tridiagonal and unit matrices respectively, of order m . The vectors v_j and f_j are of order m .

The BCR algorithm for solving systems in the above form was introduced as early as 1965 by Hockney ([11]). The basic idea of the first step is to combine every block-row of even number with the two adjacent block rows so as to eliminate the odd (block) variables. The process is repeated until only one block variable remains. A back-substitution is then applied to compute all the unknowns. In its direct application the method is unstable, but a stabilized version was introduced by Buneman ([3]) at a cost of increased complexity. The first descriptions of the method assumed that the number of blocks $n = 2^k - 1$ ([4]). The extension of the method to any n is due to Sweet ([26]). For a short history of BCR we refer to [21]. The sequential complexity of the method is $O(nm \log n)$.

Let us rewrite (2.2) as

$$(2.3) \quad \mathcal{A}v = f$$

and assume, to simplify the notation, that $n = 2^k - 1$. In the r -th reduction step of BCR, $r = 1, \dots, k-1$, the current $2^{k-r+1} - 1$ right-hand-side vectors are combined into $2^{k-r} - 1$ ones, producing a system of the form

$$\mathcal{A}^{(r)}v^{(r)} = f^{(r)}$$

in which $A^{(r)}$ is a block-tridiagonal matrix of block dimension $2^{k-r+1} - 1$ whose diagonal blocks are all equal to a matrix $A^{(r)}$ and whose co-diagonal blocks are equal to $-I$. The matrix $A^{(r)}$ can be expressed as a Chebyshev polynomial of degree 2^{r-1} in A , which we write as $A^{(r)} = p_{2^{r-1}}(A)$. In Buneman's version of the algorithm, the explicit computation of the right-hand-sides $f^{(r)}$ of the reduced systems is avoided by introducing auxiliary vectors p and q which are defined through the solution to a system of the form

$$A^{(r)} X_r = Y_r$$

where $Y_r \in \mathbb{R}^{m \times (2^{k-r+1} - 1)}$. Since the roots $\lambda_i^{(r)}$ of $p_{2^{r-1}}$ are known, $A^{(r)}$ can be written in product form, where each factor is a tridiagonal matrix of the form $A - \lambda_i^{(r)} I$. A similar strategy is used for the back-substitution phase. For completeness we next describe the Buneman algorithm.

ALGORITHM: BCR, BUNEMAN'S VERSION

A. *Initialize*: $p_i^{(0)} = 0, q_j^{(0)} = f_j, j = 1, \dots, n$ and $h = 1, r = 0$.

B. *Forward solution*:

1. Form the matrix Y_r with columns $q_{2jh}^{(r)} + p_{(2j-1)h}^{(r)} + p_{(2j+1)h}^{(r)}$,
 $j = 1, \dots, (n+1)/2h - 1$
2. Solve the (multi)-linear system $A^{(r)} X_r = Y_r$
3. Update the vectors p and q according to

$$(2.4) \quad p_{2jh}^{(r+1)} = p_{2jh}^{(r)} + X_r e_j, \quad j = 1, \dots, 2^{k-r-1} - 1$$

$$(2.5) \quad q_{2jh}^{(r+1)} = 2p_{2jh}^{(r+1)} + q_{(2j-1)h}^{(r)}, \quad j = 1, \dots, 2^{k-r-1} - 1$$

4. If $h < n$ then $h = 2h, r = r + 1$; go to 1.

C. *Solve for u* : $A^{(r)} u = q_1^{(r)}$ and set $v_h = p_h + u$.

D. *Backward substitution*: while $h \geq 1$ do

1. $h = h/2$
2. Form the matrix Y_r with column vectors $q_{jh}^{(r)} + v_{(j-1)h} + v_{(j+1)h}$,
 $j = 1, 3, 5, \dots, n/h$.
3. Solve the (multi)-linear system $A^{(r)} U_r = Y_r$
4. Update the solution vectors $v_{jh}, j = 1, 3, \dots$, by $V_r = P_r + U_r$,
 where V_r (resp. P_r) is the matrix with vector columns v_{jh} (resp. p_{jh}).

In (2.4), the vector $X_r e_j$ is the j -th column of the matrix X_r . As was mentioned above, since the roots of $p_{2^{r-1}}$ are known the systems in B.2 can

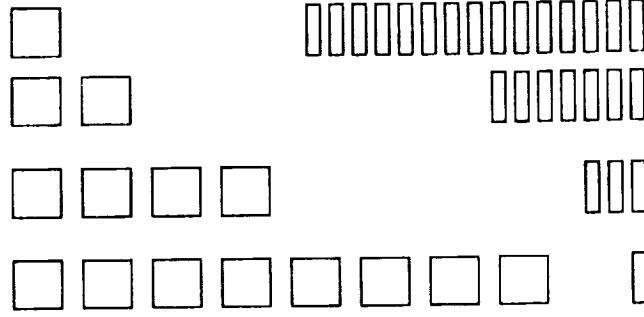


FIG. 1. Serial Block Cyclic Reduction for $n = 31$.

be written

$$(2.6) \quad \prod_{i=1}^{2^{r-1}} (A - \lambda_{i-1}^{(r)} I) [x_1 | \dots | x_{2^{k-r-1}}] = [y_1 | \dots | y_{2^{k-r-1}}].$$

Clearly, as r increases the effectiveness with which a parallel or vector computer can handle (2.6) decreases rapidly since the number of right hand sides available decays geometrically. Figure 1 depicts the rapid increase in the sequential factors (the blocks on the left) in contrast to the equally rapid decrease in the number of independent systems (the vectors on the right) for $n = 31$. The parallel version of BCR is based on expressing the matrix function $[p_{2^{r-1}}(A)]^{-1}$ as a partial fraction, i.e. as a linear combination of the 2^{r-1} components $(A - \lambda_{i-1}^{(r)} I)^{-1}$.

$$(2.7) \quad [x_1 | \dots | x_{2^{k-r-1}}] = \sum_{i=1}^{2^{r-1}} \alpha_i^{(r)} (A - \lambda_{i-1}^{(r)} I)^{-1} [y_1 | \dots | y_{2^{k-r-1}}].$$

The coefficients $\alpha_i^{(r)}$ are equal to $1/p'_{2^{r-1}}(\lambda_{i-1}^{(r)})$ and can be derived analytically. The cases of Neumann and periodic boundary conditions can also be handled similarly. Figure 2 depicts the parallel reduction for $n = 31$. When the number of blocks n is not equal to $2^k - 1$, additional systems of the form

$$(2.8) \quad \prod_{i=1}^h (A - \lambda_{i-1} I) x = \prod_{j=1}^l (A - \mu_{j-1} I) y$$

must be solved at each step. In this case the rational matrix polynomial

$$(2.9) \quad \prod_{j=1}^l (A - \mu_{j-1} I) \cdot \left[\prod_{i=1}^k (A - \lambda_{i-1} I) \right]^{-1}$$

is also reduced into a sum of partial fractions ([8]).

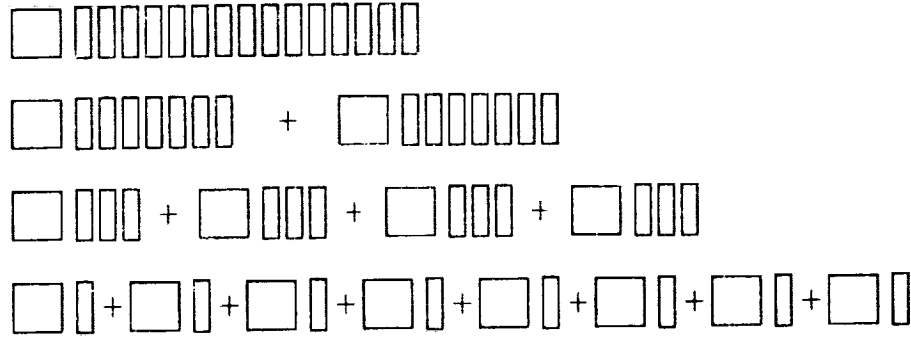


FIG. 2. Parallel Block Cyclic Reduction for $n = 31$.

The same technique can be applied for general separable equations following the algorithm in [22], the main difference being that the roots of the polynomial are not known beforehand in an analytic form. Standard methods can be used for the numerical computation of these roots, followed by the technique described earlier. We note that partial fraction expansions for BCR were also independently advocated in [25]. They have also been used in a similar context by Swayne [24] who was not, however, motivated by parallel computing. Their use in a different context for parallel processing was advocated by H. T. Kung [16].

2.1. Block Cyclic Reduction on hypercubes. Distributed memory machines based on hypercube networks represent an excellent compromise between fully connected arrays and grid arrays and have recently been developed into several commercial products. A p -cube network or p -dimensional hypercube, consists of $P = 2^p$ identical nodes that are interconnected to each other in such a way that each node has p neighbors. The rigorous description of the interconnection involves a binary numbering of the nodes: two nodes are connected if and only if their binary numbers differ in one and only one bit. Thus, for $p = 3$, a p -cube can be represented as an ordinary cube in three dimensions where the vertices are the $8 = 2^3$ nodes of the 3-cube. For $p = 4$ one can represent the hypercube network as shown in Figure 3.

In this section we consider several mappings of the BCR algorithm on hypercubes. Sections 2.1.1 and 2.1.2 deal with the limited processor case, in that the number of processors P is assumed to be smaller than m in the former and n in the latter. Section 2.2 deals with the case $P \geq N$.

An important notion in hypercube architectures is that of Gray-codes. A Gray code of order p is simply a sequence g_0, \dots, g_{2^p-1} , of all the p -bit binary numbers such that two consecutive elements of the sequence differ in exactly one bit, i.e., $H(g_i, g_{i+1}) = 1$, where $H(x, y)$ denotes the Hamming distance between x and y . In particular, the binary reflected Gray code

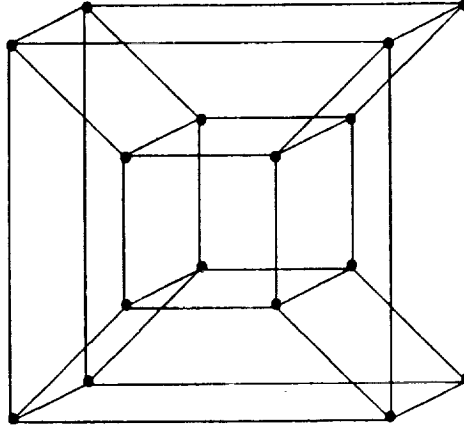


FIG. 3. 3-D view of the 4-cube.

sequence of order p , see [17] has the remarkable additional property that $H(g_i, g_{i+2^j}) = 2$, for $j \geq 1$. In the remainder of this paper we will use the term Gray code to refer to binary reflected Gray codes.

2.1.1. Interleaving right hand sides vertically across processors. We first describe a mapping of the data that leads to a simple and efficient algorithm for the case $m \gg P$. Considering each of the subvectors f_j of the right-hand-side f , we assign its l -th component $f_{j,l}$ to processor g_{l-1} for $l \leq P$ and more generally,

MAP1: Assign component $f_{j,l}$ to node $g_{\text{mod}(l-1, 2^p)}$.

Thus the processor labeled g_{l-1} will hold all the components $l + \nu P$, where $l + \nu P \leq m$, of each subvector f_j of f . A consequence of this mapping is that when solving the simultaneous tridiagonal systems with Gaussian elimination, the forward and backward sweeps will only require nearest neighbor communication.

In the very first step of block cyclic reduction, there are $2^{k-1} - 1$ tridiagonal linear systems to solve with the same matrix A , and different right hand sides, $f_2, f_4, \dots, f_{2(2^{k-1}-1)}$. As is illustrated in Figure 4 each of these independent tridiagonal systems is interleaved across the P processors. These linear systems can be solved by pipelined Gaussian elimination, which we now describe.

The first elimination step consists of communicating the first row and top element of the first right hand side down from processor g_0 to its neighbor g_1 which then performs the forward elimination step. In the second step processor g_1 sends the second row of the first right-hand-side to processor

P_{00}	x x	x	x	x
P_{01}	x x x	x	x	x
P_{11}	x x x	x	x	x
P_{10}	x x x	x	x	x
P_{00}	x x x	x	x	x
P_{01}	x x x	x	x	x
P_{11}	x x x	x	x	x
P_{10}	x x x	x	x	x
P_{00}	x x x	x	x	x
P_{01}	x x x	x	x	x
P_{11}	x x x	x	x	x
P_{10}	x x x	x	x	x
P_{00}	x x x	x	x	x
P_{01}	x x x	x	x	x
P_{11}	x x x	x	x	x
P_{10}	x x x	x	x	x
P_{00}	x x x	x	x	x
P_{01}	x x x	x	x	x
P_{11}	x x x	x	x	x
P_{10}	x x	x	x	x

FIG. 4. Interleaving assignment of three right-hand-sides across four processors.

g_2 while processor g_0 will send the first element of the second right hand side down to g_1 . Then g_1 performs the first elimination step for the second right hand side and g_2 the second step of elimination for the first right hand side. Note that here g_1 does less computation than g_2 . For later reference, we display in Figure 5 the main loop involved in the solution of a system with the tridiagonal matrix whose nonzero components in the i th row are $b(i), d(i), c(i)$.

The pivots z may be saved in place of $b(i)$ since they will be needed for the other right hand sides. The remainder of the process is similar. Initially most of the processors are idle but after the first P elimination steps, every processor becomes active and the granularity of the tasks increases. When the forward elimination is completed, the backward elimination is performed

```

c Forward solve:
do i=2, m
    z = b(i)/d(i-1)
    d(i) = d(i) - z*c(i-1)
    y(i) = y(i) - z*y(i-1)
enddo
c Backward solve
y(m) = y(m)/d(m)
do i = m-1, 1, -1
    y(i) = (y(i) - c(i)*y(i+1))/d(i)
enddo

```

FIG. 5. *Gaussian elimination for tridiagonal system.*

in the same way but backwards.

In subsequent steps of the parallel version of Buneman's algorithm described in Section 2, we have several tridiagonal systems to solve with the same right hand sides but different tridiagonal matrices. For simplicity we drop the superscript index (r) from the λ 's; it is understood however that the set of λ 's changes at each step of block cyclic reduction. Thus, in the second step of the forward reduction, we end up with systems of the form $(A - \lambda_0 I)(A - \lambda_1 I)X = Y$ where this time the number of right hand sides in Y is $2^{k-2} - 1$, or nearly half what it was in the first step. One possibility is to proceed sequentially with respect to the λ 's, i.e., we can use the pipelining procedure described above to solve $(A - \lambda_0 I)Z = Y$ and then $(A - \lambda_1 I)X = Z$. However, towards the end of the forward elimination, the number of right hand sides decreases and a better alternative is to use the parallel technique based on partial fractions, described in Section 2.1. This amounts to solving the independent tridiagonal systems $(A - \lambda_0 I)X_0 = Y$ and $(A - \lambda_1 I)X_1 = Y$ and then computing a linear combination of $X_i, i = 0, 1$. It is interesting to observe that from the point of view of implementation, one can consider that we are solving altogether twice as many tridiagonal systems independently. The right hand sides can therefore be duplicated as many times as there are λ 's and we are back to the situation of step one except that not all of the distinct right hand sides must be solved with the same tridiagonal matrix. For example, in step 2 we will have to solve $2[2^{k-2} - 1]$ independent tridiagonal systems, half of which involve the same matrix $(A - \lambda_0 I)$ and the other half the matrix $(A - \lambda_1 I)$. The pipelined procedure described for the first step can be used.

Let us now estimate the time that it takes to perform the r -th step of the forward elimination. For reasons that will be explained later we will assume that $m \geq n$. We will use the following standard and simple model.

To move a data set of j words from one processor to a neighbor takes a time of

$$(2.10) \quad \beta + j\tau$$

seconds, while performing j arithmetic operations in vector mode,

$$(2.11) \quad \gamma + j\omega$$

seconds. Note that this model includes the case where each processor is a scalar processor, by taking $\gamma = 0$. We will often refer to Buneman's algorithm described in Section 2.1. At the start of every step, the algorithm requires forming the matrix Y_r of right-hand-sides. Each column vector y_j of Y_r is a linear combination of the vectors q 's and p 's in the standard notation of Buneman's algorithm and they are obtained by the formula,

$$(2.12) \quad y_j = q_{2jh}^{(r)} + p_{(2j-1)h}^{(r)} + p_{(2j+1)h}^{(r)} \quad j = 1, 2, \dots,$$

Clearly, this requires no communication since a given processor contains the same components of each of the vectors to be combined. The time for this is approximately,

$$2\gamma + 2\lceil \frac{m}{P} \rceil \omega.$$

The next phase of the elimination step consists of solving 2^{r-1} tridiagonal systems each of dimension m and having $2^{k-r} - 1$ different right hand sides arranged in interleaved order. Following our previous discussion, we will assume that we must solve exactly $q = 2^{r-1}(2^{k-r} - 1)$ tridiagonal systems each with a different right hand side and a different diagonal. Clearly, this is not quite accurate since many of the matrices are identical as was seen above, but it gives an upper bound for the time estimates that is much simpler to derive.

In pipelined algorithms there is a pipe-fill time which corresponds to the first few steps before all processors reach their high regime of efficiency. The first $P - 1$ steps see processors g_0, \dots, g_{P-1} becoming gradually busy each doing one elimination step consisting of reading three floating point numbers from a previous processor and performing forward elimination at a total cost of

$$(P - 1)[\beta + 3\tau + 3\gamma + 5\omega]$$

Similarly, in the next P steps all processors will be busy but they will deal with two elements of the right hand side instead of just one. The new cost is

$$P[\beta + 6\tau + 6\gamma + 10\omega]$$

It is only after the first component of the last right hand side is processed, i.e., after step q , that the processors will start doing essentially the same work at every new step. If we let $s \equiv \lceil q/P \rceil$, then the total time that it takes before this is achieved is approximately,

$$(2.13) \quad \sum_{j=1}^s P[\beta + 3j\tau + 3\gamma + 5j\omega] \approx Ps[\beta + 3\gamma + 3\frac{s}{2}\tau + 5\frac{s}{2}\omega]$$

Note that a similar phenomenon takes place in reverse in the last q steps. Apart from these first and last q steps each of the remaining $m - 2q$ steps of the algorithm, takes the same amount of time which is approximately, $\beta + 3\gamma + 3s\tau + 5s\omega$ and the total becomes

$$(2.14) \quad \approx 2Ps[\beta + \gamma + 3\frac{s}{2}\tau + 5\frac{s}{2}\omega] + (m - 2q)[\beta + 3\gamma + 3s\tau + 5\omega]$$

Note here that we need to have $m - 2q > 0$ which, using the fact that q is a decreasing function of τ leads to $m > 2^k - 2 = n - 1$ or $m \geq n$, which justifies our earlier assumption.

The partial fraction expansion formula (2.7) requires that we now take a linear combination of 2^{r-1} matrices of $2^{k-r} - 1$ columns of length m each. These matrices are split equally among the processors and the time required for this linear combination is approximately,

$$2^{r-1}(2^{k-r} - 1)(\gamma + \lceil \frac{m}{P} \rceil \omega) = q(\gamma + \lceil \frac{m}{P} \rceil \omega)$$

assuming that the linear combinations are performed column-wise. Note that once more no communication is required.

Finally, we need to update the p and q vectors according to (2.4), (2.5). Again this requires no interprocessor communication and the arithmetic time is approximately, $\gamma + \lceil \frac{m}{P} \rceil \omega$ for (2.4) and $2\gamma + 2\lceil \frac{m}{P} \rceil \omega$ for (2.5).

Adding up all these times, and separating the communication from arithmetic complexities we obtain

$$(2.15) \quad T_{comm} \approx (2Ps + m - 5q)\beta + 3s(Ps + m - 2q)\tau$$

for communication and

$$(2.16) \quad T_{arith} \approx (2Ps + 3m - 5q)\gamma + (5Ps^2 + 5(m - 2q) + (q + 5)\lceil \frac{m}{P} \rceil)\omega$$

for arithmetic.

Note that since $q = 2^{k-1} - 2^{r-1}$, for q large relative to the number of processors, Ps will be of the order of q which implies that the number

of start-ups is of the order of $2q = 2^k - 2^r$ in both communication and arithmetic. Apart from the start-up coefficients, the algorithm is dominated by the $O(Ps^2)$ terms in both communication and arithmetic, which are of the same order as q^2/P in a typical situation.

When the number of right hand sides is small, blocking can be used to reduce the effect of start-up overhead. Going back to the original version of the algorithm, we notice that at every step only a few arithmetic computation and a few data transfers are performed at every step. If each processor has a vector processing capability then this may be inefficient. Similarly, many machines have high latency times in communication and it is always preferable when possible to transfer a sizable amount of data at once rather than just a few. The remedy to this is to use a simple and standard scheme in pipelining which consists of blocking the computations. Instead of treating only one right hand side at a time, we can deal with a group of ν right hand sides simultaneously at each time. Then each elementary step of Gaussian elimination can be performed as a vector operation across the ν right hand sides.

2.1.2. Horizontal distribution of the right hand sides. In this section we consider another implementation of BCR in which the right hand side vectors are not distributed vertically, but horizontally, i.e., a whole right hand side (vector) is now assigned to the same processor instead of being split and shared among several ones as in the previous subsection. Again we consider a hypercube system with $P = 2^p$ processors and assume that $P \ll n$, where n is of the form $n = 2^k - 1$.

For the purpose of illustration let us consider the simple case where $n = 15$ and the number of processors $P = 2^p$ is 8. We start by mapping two right hand sides per processor except for the last processor which will only hold one right hand side. We use the Gray code mapping of the right hands sides f_1, f_2, \dots, f_{15} , at the rate of two vectors per node, which consists of assigning f_1, f_2 to node 000, f_3, f_4 to node 001, ..., f_{2i+1}, f_{2i+2} to node g_i where $g_0, g_1, \dots, g_{2^p-1}$ is the standard binary reflected Gray code sequence. As is easily seen in this example, when combining three successive vectors $f_{2i-1}, f_{2i}, f_{2i+1}$ as in the first step of forward reduction, we only need nearest neighbor communication. After these linear combinations are completed, each processor solves a tridiagonal system involving A . In the later reduction steps, subvectors with subscripts differing by a power of two are combined. Because of a well-known property of the binary reflected Gray code, one observes that communication is kept at distance of exactly two. It is important to notice that after a certain number of steps, in our example after just the first step, many processors will have no right hand sides while others will have exactly one right hand side. In our example, in step 2, processors

g_2, g_4 and g_6 will have one right hand side to solve with a system which is a degree 2 polynomial in A . This raises the question of how to put the inactive processors to work, which will be addressed shortly.

Consider the situation in the general case. At each step of the reduction process there are $2^{k-r} - 1$ right hand sides involving a polynomial of degree 2^{r-1} for $r = 1, \dots, \log n - 1$. Once these systems are solved, the resulting vectors are combined in groups: the vectors corresponding to indices $i - 2^{r-1}, i, i + 2^{r-1}$ are added together. This gives a new set of independent systems with half the number of unknowns. With the exception that we are now dealing with vectors instead of scalars, the combination step is as in scalar cyclic reduction. A systematic way of mapping the right hand sides to the P processors is to start by partitioning the Gray code representation h_i of the index i of each right hand side (vector), for $1 \leq i+1 \leq n$, as

$$(2.17) \quad h_i = \overbrace{i_{k-1} \dots i_{k-p}}^{F_p(i)} i_{k-p-1} \dots i_0$$

As shown, for any integer i , $0 \leq i \leq 2^k - 1$, we denote by $F_p(i)$ the binary number formed by the p leading bits of h_i , the i -th element of the Gray code sequence of dimension k . Hence $F_p(i)$ can serve as an identifier tag for $P = 2^p$ processors. Then, the mapping rule used is as follows

MAP2: Assign component $f_{j,l}$, $l = 1, 2, \dots, m$ to node $F_p(j-1)$.

We then obtain the following theorem which can be used both in scalar and block cyclic reduction.

THEOREM 2.1. *If $n = 2^k - 1$ and rule MAP2 is in effect in the initial assignment of equations to processors for cyclic reduction, the elements to be combined at each step belong to nodes which are at a maximum distance of 2 apart on the hypercube graph.*

Proof. We discuss the case of BCR. The same arguments can be used to prove the result for scalar cyclic reduction. Consider block index $i+1$ with $i = \alpha 2^{k-p} + l$, where $0 \leq l < 2^{k-p}$ and $\alpha = 0, \dots, 2^p - 1$. At step r , each vector $i = 2^r \mu$ with $\mu = 1, \dots, 2^{k-r} - 1$ is combined with vectors $i - 2^{r-1}$ and $i + 2^{r-1}$. According to MAP2, the equation $i+1$ is in processor $F_p(i) = g_\alpha$. When $2^r \leq i + 2^{r-1} \leq 2^k - 1$, the equations $i + 2^{r-1}$ and $i - 2^{r-1}$ are in the same processor because the p leading bits of their indices are identical. It is only when $r > k - p$ that any of the vectors in g_α would be combined with elements in a processor other than $g_\alpha, g_{\alpha+1}$ or $g_{\alpha-1}$. From the Gray code assignment, processors g_α and $g_{\alpha \pm 1}$ are adjacent. When $r > k - p$, we write $r = k - p + t$ for $t \geq 1$ and combine i with $i \pm 2^{k-p+t}$, i.e.

$$\alpha 2^{k-p} + l \pm 2^{k-p+t} = 2^{k-p}(\alpha \pm 2^t) + l$$

By definition these lie in processors $g_{\alpha \pm 2^t}$. From a fundamental property

of binary reflected Gray codes these processors are at distance 2 away from processor g_α and the theorem is proved. ■

We distinguish two regimes in the algorithm. With $n = 2^k - 1$ and $P = 2^p$, the first regime is when the step number r does not exceed $k - p$, in which case each processor ends up with $2^{k-r}/2^p = 2^{k-r-p}$ right hand sides except for the last processor which will have one less right hand side. The degree of the polynomial is 2^{r-1} . Thus, the work is well balanced, at the exception of the slight difference with the last processor. More precisely, each processor will have a linear system of the form $p_{2^r}(A)X = Y$ to solve and there is no need to use partial fractions in order to load balance. In contrast, in the second regime, i.e., as soon as $r > k - p$, each processor of the form $J = F_p(j \times 2^r)$ ends up with exactly one right hand side while any other processor will not have accumulated any right hand side and would remain idle if no counter action is taken. However, the idea of partial fraction expansions described earlier can be employed to achieve load balancing when solving the polynomial systems $(A - \lambda_0 I)(A - \lambda_1 I) \dots (A - \lambda_{2^{r-1}-1} I)x = y$, of degree 2^{r-1} that are generated at the r^{th} reduction step in nodes labeled $F_p(j \times 2^r)$, $j = 1, \dots$. Again we start by illustrating the process with the particular case $n = 15$ and $P = 8$; i.e., $k = 4, p = 3$. After the first elimination process, all processors must solve a linear system of the form $(A - \lambda_0 I)x = y$ where y consists of one right hand side, and this constitutes the only step of the first regime where there is no need to load balance. In the second step, processors $F_3(2) = 001, F_3(4) = 010$ and $F_3(6) = 111$, will have to solve each a system of the form $(A - \lambda_0 I)(A - \lambda_1 I)x = y$, or equivalently, by the partial fraction decomposition, two independent linear systems $(A - \lambda_i I)x_i = y$, $i = 0, 1$. To make the other processors participate in solving these systems, we will have each of the three 'master nodes' 001, 010, and 111 distribute some of the linear systems $(A - \lambda_i I)x_i = y$ to slave processors. To do this in a systematic manner, each master processor will assign the system $(A - \lambda_i I)x_i = y$ to the (slave) processor whose label has the same leading 2 bits as those of the master node and the same trailing bit as that of the binary representation of i . Thus, node 001 sends the system associated with λ_0 to processor 000, and keeps the system with λ_1 . Similarly, node 010 sends the system associated with λ_1 to node 011 and node 111 sends the system associated with λ_0 to node 110. After these systems are solved they must be combined back in their master nodes.

More generally, at a given step of the second regime processors numbered $F_p(j \times 2^r)$, $j = 1, 2, \dots, 2^{k-r} - 1$ and only those processors, will have to solve systems of the form

$$(2.18) \quad (A - \lambda_0 I)(A - \lambda_1 I) \dots (A - \lambda_{2^{r-1}-1} I)x = y.$$

When $r = k - p + 1$, all the processors in the subcube of the nodes whose leading $p - 1$ bits are identical with those of the master node $F_p(j2^r)$, will have no system of the form (2.18) to solve. At any step $r = k - p + r_0$, where $r_0 = 1, 2, \dots$ serves as iteration counter for the steps of the second regime, each master node $F_p(j2^r)$ will have a system of the form (2.18) while all other nodes in the subcube of the nodes having the same $p - r_0$ leading bits, will have none. We can use the partial fraction expansion of Section 2 to decompose (2.18) into 2^{r-1} independent linear systems

$$(2.19) \quad (A - \lambda_i I)x_i = y, \quad i = 0, 1, \dots, 2^{r-1} - 1,$$

which should be followed by a linear combination of the x_i 's. Then, a simple strategy to distribute the linear systems (2.19) equally, is to have the master node broadcast the right hand side y to the nodes of its subcube consisting of all the nodes with the same leading $p - r_0$ bits, and have each of them solve the systems (2.19) for which the trailing r_0 bits of i match the trailing r_0 bits of the node's label.

In brief we have used the fact that the sets S_j consisting of the nodes ξ so that $F_p(\xi) = F_p(j2^r)$, form a partition of the p -cube into subcubes, and we have distributed the linear systems (2.19) equally in each subcube.

Therefore, if we denote by $L_{r_0}(i)$ the binary number consisting of the s trailing bits of the binary number i , we can summarize the rule by

LOAD BALANCING RULE: *At step $r = k - p + r_0$, each master node $F_p(j2^r)$ makes slave nodes $F_{p-r_0}(j2^r)L_{r_0}(i)$ solve systems (2.19).*

Note that there are $p - 1$ steps in the second regime. It is easy to see that each node will solve an equal number of linear systems which is constant and equal to $2^{r-1-r_0} = 2^{k-p-1}$, at the exception of a small number of the last nodes in the Gray code sequence which will have no system to solve. This process requires broadcasting of the right-hand-side in node J to its subcube and then gathering/summing of the data in the manner distributed inner products are usually computed in a hypercube. With the simplest broadcast algorithm, these operations cost $O(r_0 m)$ which means that the communication overhead in this second phase is higher than that of the first phase where the right hand sides y are formed. Hence depending on the relation between communication and arithmetic costs of the particular hypercube system it might be preferable not to distribute the work to all processors but only to those located in a smaller subcube.

A final operation required in the second regime is to accumulate the different solutions back to the master node. This is essentially a gather-combine operation and is done by exploiting the topology of the hypercube, in r_0 steps consisting of moving in a higher subcube closer to the master node and adding the intermediate results at each time.

To get an estimated time of the r -th step of this algorithm, we must distinguish between the two regimes. We first observe that the operation that consists of combining the vectors p 's and q 's to form the right hand sides is identical in both regimes except that we deal with more vectors in the first and that only the master nodes are actually active in the second. We can also view the right-hand sides in the first regime as forming a single long right vector of length $m' \equiv 2^{k-r-p} \times m$. Thus, the first operation in step r is to have each processor with label of the form $J = F_p[j \times 2^r]$ form a linear combination of q and p vectors of length m' to form their column of the right-side matrix Y . This costs $2\beta + 2m'\tau$ for communication (Bringing two vectors that are at most two hops away) and $2\gamma + 2m'\omega$ for their linear combination.

After the right-hand-sides are formed we need to solve the tridiagonal systems. This is processed differently in the two regimes. In the first regime, we need to solve in each processor independently, at most 2^r tridiagonal systems with 2^{k-r-p} right-hand-sides. Using standard Gaussian elimination this will consume a time of

$$(2.20) \quad 2^r \times m \times [3\gamma + 8 \times 2^{k-r-p}\omega]$$

assuming vectorization across the right-hand-sides.

On the other hand regime 2 requires first broadcasting the (single) right-hand-side to its subcube at the cost of

$$r_0 \times (\beta + m\tau),$$

and then having each of the slave processors solve $2^{r-1-r_0} = 2^{k-p-1}$ tridiagonal systems at the cost of

$$(2.21) \quad 2^{k-p-1} \times m \times [3\gamma + 8 \times \omega]$$

Moreover, the different solutions must be accumulated back to the master node at the cost of

$$r_0[\beta + m\tau + \gamma + m \times \omega].$$

Finally, we need to update the vectors p and q according to (2.4), (2.5). Note that the vectors involved in updating $p_{2jh}^{(r+1)}$ are the same processor, so there is no need for communication. The number of vectors that processor $g_{2j,2^r}$ combines is $2^{k-r-p-1}$ and the cost is $2^{k-r-p-1}(\gamma + m\omega)$ for (2.4). For (2.5), we need to bring the vectors $q_{(2j-1)h}^{(r)}$ at the cost of $2(\beta + m\tau)$, and then do the linear combination at the cost of $2\gamma + 2m\omega$. These computations are identical in both regimes. Note that many processors will be idle at the

end of the elimination phase and load balancing can also be performed, but we omit the details.

Summing up we find that each step in the first regime costs a total of

$$(2.22) \quad T_{comm} = 4\beta + 2^{k-r-p+1}m\tau$$

for communication and,

$$(2.23) \quad T_{arith} = (5 + 3 \times 2^{r-1}m)\gamma + (4 + 2^{k-r-p+1} + 2^{k-p+2})m\omega$$

for arithmetic. Similarly, each step in the second regime costs

$$(2.24) \quad T_{comm} = (r_0 + 4)(\beta + m\tau)$$

for communication and

$$(2.25) \quad T_{arith} = (5 + 3 \times 2^{k-p-1})\gamma + (r_0 + 6 + 2^{k-p+2})m\omega$$

and arithmetic. Note the high coefficient in front of the γ term in the second regime which simply indicates that there is no vectorization when solving the tridiagonal systems. It is rather difficult to compare the algorithm of Section 2.1.1 with the one described in this section based on these complexity results. Assuming that $m = n$, we only note that when the latter is dominated by the first regime, i.e., when n is very large compared with P then the second algorithm is likely to be slightly superior than the first. This can be seen by comparing all the coefficients of each of the constants $\beta, \tau, \gamma, \omega$ and making the simplification $P_s \approx 2^{k-1}$ which is valid for large n and small r , i.e., for the first reduction steps. If the algorithm is dominated by the second regime, i.e., when then n is of the same order as P and so there are only a very small number of steps in the first regime, then the arithmetic times of the two algorithms are comparable except that there are many more start-ups with the second. On the other hand, communication is less costly with the second algorithm.

2.2. A massively parallel block cyclic reduction algorithm. In this section we consider the extreme case where the number of processors is larger or equal to $N \equiv mn$ the number of grid points. In fact, consider the simple case where $m = 2^{p_1} - 1, n = 2^{p_2} - 1$ and $p = p_1 + p_2$, i.e., $P = m(n + 1) > N$. Each of the grid points is assigned to a different node and corresponds to a well-known mapping of a two-dimensional grid into the hypercube as illustrated in Figure 6. Thus, the physical grid is mapped into an array of processors imbedded in the hypercube. In terms of the right-hand-side vectors f_j the mapping rule is as follows:

MAP3: Assign component $f_{j,l}$ to node $g_{j-1}g_{l-1}$.

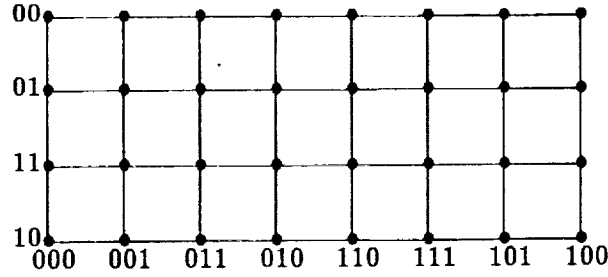


FIG. 6. Two dimensional Gray-code for an 8×4 grid.

Hence the subvectors $f_i, i = 1, 2, \dots, n$, are stored vertically, each in one vertical line of the array depicted in 6. Each horizontal line contains the same components of these subvectors. It is clear that if standard BCR were used, then the linear combination of the subvectors causes no difficulty and would require communication only between nodes that are on the same horizontal lines. Solving the tridiagonal systems requires interaction between grid points of the same subvectors, i.e., communication between nodes that are on the same vertical lines. If we use standard Gaussian elimination then only one of the nodes of each vertical line will be busy at any given step of the forward and backward sweeps. An obvious solution to this difficulty is to use scalar cyclic reduction. An alternative would be to use the parallel variant of cyclic reduction known as PARACR that avoids the back substitution phase ([12]). It was observed however that because of the additional communication involved in PARACR, cyclic reduction is more economical ([14]).

Without going into the complex details of the algorithms, we would like to estimate the order of the time required to execute it. We can adapt the algorithm of Section 2.1.2 to this case, except that we are now only considering the special situation where there is no first regime. There are a total of $k = p_2$ steps. Each step starts by assembling the right-hand-sides, one for each column of the grid corresponding to the nodes x_{j2r} . This requires communicating a few elements with processors that are at most 2 hops away in the x -direction, and is a constant with respect to r . The tridiagonal systems to be solved next will require to broadcast the right-hand-sides to the subcube associated with the master nodes, at a cost roughly proportional to the step number r . Assume that this time plus the time required to sum up the partial results later on, back to the master node is of the form $\alpha_r r$. Then the tridiagonal systems are solved independently at roughly a cost of the form $\alpha_t p_1$. The rest of the operations on p 's and q 's are again constant with respect to r . If we call c the total of all the times

that are constant with respect to r , each step will cost approximately

$$(2.26) \quad c + \alpha_b r + \alpha_t p_1$$

which yields a total over the $p_2 - 1$ steps of

$$(2.27) \quad T_{BCR} \approx p_2 \left[c + \alpha_R \frac{p_2 - 1}{2} + \alpha_t p_1 \right]$$

In other words the cost is of the order of $O[\log^2 n + \log m \log n]$. The term $\log^2 n$ comes entirely from the use of the partial fraction expansion and appears because of the need to load balance the computation.

It is interesting to observe the striking difference between the parallel complexity of BCR and FFT based algorithms. A simple application of the 2-D FFT algorithm (for Poisson's equation) or of the combined FFT and tridiagonal solve algorithm (for (2.1)) in the hypercube can yield a parallel arithmetic complexity of $O(\log n + \log m)$, i.e., the sum of the times for implementing FFT in the one direction and FFT or scalar cyclic reduction in the other. On the other hand the parallel version of BCR described here cannot achieve a time better than $O(\log n \log m)$. That this is an actual limitation of the algorithm rather than just a consequence of the implementation being used can be established by looking at the dependency graph of the parallel BCR.

However, as is shown in the next section, there exist alternative algorithms with a complexity of order $O(\log n + \log m)$, which is comparable to that of FFT based methods.

3. Explicit Methods. We now present an algorithm for the solution of (2.2) having parallel complexity $O(\log n + \log m)$ which is based on using the explicit inverse of \mathcal{A} .

Denote by $S_n(x)$ the shifted Chebyshev polynomial of degree n of the second kind defined for $0 \leq x \leq 2$ by

$$(3.1) \quad S_n(x) = \frac{\sin[(n+1)\theta]}{\sin \theta}, \quad \cos \theta = x/2.$$

The explicit inverse \mathcal{A}^{-1} can be written in block form, with block (i, j) given by, ([1])

$$(3.2) \quad \mathcal{A}_{ij}^{-1} = \begin{cases} S_n^{-1}(A) S_{i-1}(A) S_{n-j}(A), & j \geq i, \\ S_n^{-1}(A) S_{j-1}(A) S_{n-i}(A), & i \geq j \end{cases}$$

As a result we can write the solution u explicitly. The i^{th} block component is obtained by multiplying \mathcal{A}^{-1} by f blockwise, yielding:

$$u_i = \sum_{j=1}^n \mathcal{A}_{ij}^{-1} f_j$$

$$\begin{aligned}
&= \sum_{j=1}^{i-1} S_n^{-1}(A) S_{j-1}(A) S_{n-i}(A) f_j \\
&\quad + \sum_{j=i}^n S_n^{-1}(A) S_{i-1}(A) S_{n-j}(A) f_j
\end{aligned}$$

It is clear from (3.2) that each block \mathcal{A}_{ij}^{-1} can be expressed as a rational matrix function $q_{ij}(A)$ with denominator of degree n and numerator of degree less than n . As a result we can express each of these as the sum of n elementary fractions. Let the coefficients $\alpha_{ij}^{(k)}$, be the expansion coefficients of each of these rational functions with respect to the elementary fractions. In other words, let,

$$(3.3) \quad \sum_{k=1}^n \frac{\alpha_{ij}^{(k)}}{x - \lambda_k} = q_{ij}(x) = \begin{cases} S_n^{-1}(x) S_{i-1}(x) S_{n-j}(x) & \text{for } j \geq i, \\ S_n^{-1}(x) S_{j-1}(x) S_{n-i}(x) & \text{for } i \geq j \end{cases}$$

Then each of the components u_i is given by

$$\begin{aligned}
u_i &= \sum_{j=1}^n \mathcal{A}_{ij}^{-1} f_j \\
&= \sum_{j=1}^n q_{ij}(A) f_j \\
&= \sum_{j=1}^n \sum_{k=1}^n \alpha_{ij}^{(k)} (A - \lambda_k I)^{-1} f_j \\
&= \sum_{k=1}^n (A - \lambda_k I)^{-1} \sum_{j=1}^n \alpha_{ij}^{(k)} f_j
\end{aligned}$$

which results in the algorithm of Figure 7 for computing the solution.

From the above description the parallel computational complexity follows easily: Step (I) requires $\log n$ operations, if $n^3 m$ processors are available. The complexity for step (II) depends on the algorithm chosen to solve the systems. Since A is tridiagonal, scalar cyclic reduction can be used at a cost of $O(\log m)$ and $n^2 m$ processors. Step (III) can be completed in $\log n$ operations with $n^2 m$ processors. Summarizing, the algorithm has parallel computational complexity

$$T_{EES} = O(\log n + \log m).$$

To achieve this bound, $O(n^3 m)$ processors are necessary. The sequential complexity of the algorithm is $2n^3 m + O(mn^2)$ operations, which is much

Compute the n roots λ_k of $S_n(x)$ in (3.1).

Compute the coefficients $\alpha_{ij}^{(k)}$ in (3.3).

DOALL $i = 1, n$

DOALL $k = 1, n$

$$\text{compute } f_i^{(k)} = \sum_{j=1}^n \alpha_{ij}^{(k)} f_j \quad (\text{I})$$

$$\text{compute } \bar{u}_{ik} = (A - \lambda_k I)^{-1} f_i^{(k)} \quad (\text{II})$$

END DOALL

$$\text{compute } u_i = \sum_{k=1}^n \bar{u}_{ik} \quad (\text{III})$$

END DOALL

FIG. 7. Algorithm: Explicit Elliptic Solver

higher than the usual $O(mn \log n)$ of fast methods. As was just seen, the dominating cost is the computation of the intermediate vectors $f_i^{(k)}$ in the innermost loop. This complexity is comparable to that of a direct banded solver that does not exploit the special structure.

Note that the algorithm is as general as the BCR algorithm. It has the added benefit that it is very simple to program and that unlike BCR it does not require a special treatment when $n \neq 2^k - 1$.

We must point out however that the factor $O(\log n)$ lower complexity of the explicit algorithm compared with that of parallel BCR came at a very high price, namely a factor of $O(n^2)$ increase in the number of processors required. As we show next, by taking advantage of the structure of the problem, we can lower this processor requirement by a factor of $O(n)$.

We first observe that each of the coefficients $\alpha_{ij}^{(k)}$ is defined by

$$(3.4) \quad \alpha_{ij}^{(k)} = \frac{S_{j-1}(\lambda_k) S_{n-i}(\lambda_k)}{S'_n(\lambda_k)} \text{ for } i \geq j$$

with, the corresponding formulas for $i < j$ defined by interchanging i and j . Therefore, we can write

$$\begin{aligned} \alpha_{ij}^{(k)} &= \frac{\sin(j\theta_k) \sin((n+1-i)\theta_k)}{\sin^2 \theta_k S'_n(\lambda_k)} \\ &= \frac{\beta_{ij}^{(k)}}{\sin^2 \theta_k S'_n(\lambda_k)} \end{aligned}$$

where we have set

$$(3.5) \quad \beta_{ij}^{(k)} = \sin(j\theta_k) \sin((n+1-i)\theta_k)$$

with $\cos \theta_k = \frac{1}{2} \lambda_k$. We now use a well known trigonometric formula to obtain for $i \geq j$

$$(3.6) \quad \beta_{ij}^{(k)} = \frac{1}{2} [\cos((n+1-(i+j))\theta_k) - \cos((n+1-(i-j))\theta_k)]$$

To get the formula for the case $i < j$ we need only interchange the indices i and j , leading to the general expression

$$(3.7) \quad \beta_{ij}^{(k)} = \frac{1}{2} [\cos((n+1-(i+j))\theta_k) - \cos((n+1-|i-j|)\theta_k)]$$

One then notices that the matrix $B^{(k)} = (\beta_{ij}^{(k)})_{i,j=1,\dots,n}$ is the sum of the Hankel matrix

$$\left\{ \frac{1}{2} \cos(n+1-(i+j))\theta_k \right\}_{i,j=1,\dots,n}$$

and the Toeplitz matrix

$$\left\{ -\frac{1}{2} \cos(n+1-|i-j|)\theta_k \right\}_{i,j=1,\dots,n}.$$

It follows that the matrix $A^{(k)} = (\alpha_{ij}^{(k)})$ is also the sum of a Hankel and a Toeplitz matrix.

Observe that each of the m coordinates of the subvectors of $f_i^{(k)}$ in the previous algorithm, is the result of the product of the matrix $A^{(k)}$ by the vector obtained by extracting all the corresponding coordinates from the vectors f_j . In fact another way of expressing this is by writing that

$$(3.8) \quad (F^{(k)})^T = A^{(k)} F^T$$

where $F^{(k)}$ and F are $m \times n$ matrices whose column vectors are the $f_i^{(k)}$'s and f_i 's, respectively. Each of these products represents the product of a Toeplitz plus a Hankel matrix times a vector of size n . We write this as $(T_1 + H)x$ with T_1 Toeplitz and H Hankel. By definition, H can be rewritten as JT_2 , where J is the permutation matrix consisting of ones in the antidiagonal and 0 everywhere else. Hence the operation becomes $(T_1 + JT_2)x$. T_1x and T_2x are two Toeplitz matrix vector products computable by four FFT transforms of size $2n$ each, at the cost of $O(\log n)$ arithmetic operations and $2n$ processors per FFT. J is a permutation matrix and hence the only other computation required is the parallel addition of the two partial results requiring n processors.

This must be multiplied by the number of components in each subvector, and by n , the number of roots λ_k which leads to $2n^2m$ processors. The rest of the algorithm proceeds as before and requires $O(\log mn)$ operations with $O(mn^2)$ processors. We are thus led to the following theorem.

THEOREM 3.2. *Using $O(mn^2)$ processors, system (2.2) can be solved in $O(\log n + \log m)$ parallel steps.*

4. General Separable Equations. A second order finite difference discretization of equation (1.1) with Dirichlet boundary conditions leads to the block tridiagonal matrix \mathcal{A}

$$\begin{pmatrix} A + \alpha_1 I & \gamma_1 I & & & \\ \beta_2 I & A + \alpha_2 I & \gamma_2 I & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_k I & A + \alpha_k I & \gamma_k I \\ & & & \ddots & \ddots & \ddots \\ & & & \beta_{n-1} I & A + \alpha_{n-1} I & \gamma_{n-1} I \\ & & & & \beta_n I & A + \alpha_n I \end{pmatrix}$$

We summarize how the techniques we have described so far can deal with this case. As we shall see, this generalization comes at the cost of extra preprocessing overhead due to the need to compute polynomial roots or (equivalently) computing eigenvalues, which in practice may make these methods less attractive.

Block cyclic reduction has been extended by Swarztrauber ([22]) to handle (1.1). The difficulty here is that the roots of the polynomials generated in the course of the reduction are not analytically available as is the case for (2.1). As a result, in order to perform the partial fraction decomposition, they must be computed numerically. In [22, Table 2] there are timing results which show that this preprocessing overhead can be overwhelming.

Although (1.1) is the most general form we can have, it is not the most convenient to work with since it can lead to non-symmetric matrices. This is overcome by rewriting (1.1) (if the coefficients allow) or transforming it (multiplying by suitable integrating factors) to self-adjoint form. We thus assume next that such a transformation has been made and hence $\beta_i = \gamma_{i-1}$ for $i = 1, \dots, n$ and $A = A^T$.

The *matrix decomposition* algorithm described in [4] is based on the eigenvalue-eigenvector decomposition of the diagonal blocks of \mathcal{A} . When the equation is of Poisson type in one direction, a suitable ordering of the unknowns makes the orthogonal matrix of eigenvectors of A equal to a discrete Fourier transform operator thus allowing the use of FFT for the matrix-vector multiplications. For the above more general matrix \mathcal{A} however this is no longer true: The eigenvalues and eigenvectors of A must be computed numerically and matrix-vector multiplications must be performed explicitly. In the experiments described in [22], the generalized BCR seems to perform better than matrix decomposition.

The generalization of the explicit method of Section 3 requires the use

of a formula analogous to (3.2) for \mathcal{A}^{-1} . From [2, Theorem 2.2]

$$(4.1) \quad -\beta_n \mathcal{A}_{ij}^{-1} = \begin{cases} P_n^{-1}(A)P_{i-1}(A)R_{n-j}(A), & j \geq i, \\ P_n^{-1}(A)P_{j-1}(A)R_{n-i}(A), & i \geq j \end{cases}$$

where the n roots of P_n are the negatives of the eigenvalues of the symmetric tridiagonal matrix $\text{Tridiag}[\beta_{i-1}, \alpha_i, \beta_i]$. The polynomials P_i, R_j are computable by means of three term recurrence relations. Hence once the n eigenvalues of $\text{Tridiag}[\beta_{i-1}, \alpha_i, \beta_i]$ have been computed the algorithm can proceed as in Section 3.

5. ADI Algorithms. The Alternating Direction Implicit procedure of Peaceman and Rachford can be regarded as a fast algorithm for solving separable elliptic equations. Although this is an iterative method, if the equations resulting from the discretized partial differential equations are solved with an accuracy that is of the order of the discretization error, and if the optimal parameters are used, then the number of steps required for convergence is of the order of $O(\log^2 n)$, where n is assumed to be the larger of the two numbers of grid points in the x and y directions. This puts the total cost in the sequential case to $O(nm \log^2 n)$ [20]. Hence if one can achieve a parallelism of the order of nm then there is the possibility of reducing the cost of ADI to the same order as what we obtained with BCR in the best case. Several of the benefits of ADI have been mentioned in the literature for the general nonseparable case or the parabolic equation case ([10,14,15]). Here we would only like to mention some of the implementation aspects and discuss some advantages over BCR.

To describe the basic algorithm, consider the partial differential equation,

$$(5.1) \quad \frac{\partial}{\partial x} \left(a(x, y) \frac{\partial u(x, y)}{\partial x} \right) + \frac{\partial}{\partial y} \left(b(x, y) \frac{\partial u(x, y)}{\partial y} \right) = f(x, y)$$

on a rectangular domain with the Dirichlet boundary conditions.

If the equations are discretized using a mesh of $n + 1$ points in the x direction and $m + 1$ points in the y direction we get the system of equations:

$$(5.2) \quad A_x u + B_y u = f$$

in which the matrices A_x and B_y represent the 3-point central difference approximations to the operators $\frac{\partial}{\partial x}(a(x, y) \frac{\partial}{\partial x})$ and $\frac{\partial}{\partial y}(b(x, y) \frac{\partial}{\partial y})$ respectively.

The ADI algorithm consists of iterating by solving (5.2) alternatively in the x and y directions as follows:

$$(5.3) \quad (A_x + \rho_i I) u^{i+1/2} = (\rho_i I - B_y) u^i + f$$

1	2	3	4
2	3	4	1
3	4	1	2
4	1	2	3

FIG. 8. Domain decomposition and assignment of the square into a 4-processor ring.

$$(5.4) \quad (B_y + \rho_i A_x)u^{i+1} = (\rho_i I - A_x)u^{i+1/2} + f,$$

where $\rho_i, i = 1, 2, \dots$, is a sequence of acceleration parameters.

In the following we summarize some of the results described in [15] with emphasis on complexity. Observe that if the mesh points are ordered by lines in the x direction, then (5.3) constitutes a set of m independent tridiagonal systems of size n each. The system (5.4) can also be recast into a set of n independent tridiagonal systems of size m each, by reordering the grid points by lines, this time in the y direction. This requires essentially to transpose the matrix of the $n \times m$ grid points and constitutes the main difficulty in implementing ADI on parallel architectures. Another difficulty that has been traditionally associated with ADI is that classical algorithms for solving tridiagonal systems are sequential in nature.

Consider first the implementation of ADI on a simple ring of processors. To avoid transposing data in ADI as pointed out above, we consider the special assignment of the grid points into the ring of processors proposed in [15] and shown in Figure 8 for the case of a 4-processor ring. When iterating with ADI, the solutions of the systems (5.3) and (5.4) can be performed by a regular Gaussian elimination algorithm. Observe that all processors will be performing some work at any given stage of the iteration. Communication is facilitated by the fact that all neighboring subsquares of the domain are in neighboring processors and this is true in both the horizontal and vertical direction. The mapping can be succinctly described by

$$\text{MAP4: Assign component } f_{j,l} \text{ to node } \text{mod}[\lceil \frac{jP}{n} \rceil + \lceil \frac{lP}{m} \rceil - 2, P] + 1$$

Using the same model for estimating execution time as in Section 2.1, with $\gamma = 0$, a simple complexity analysis shows that the time for implementing such an algorithm on a ring of P processors is [15]

$$T(P) = 2(P-1)\beta + 2(m+n)\tau + \frac{8mn}{P}\omega.$$

If P is small compared with m and n , the above formula shows that the optimal speed-up of P is nearly reached provided the communication constants β, τ are not too big. However, as the number of processors increases the communication time may become too high. In fact for the case $m = n$, the minimal time that can be achieved on an arbitrarily large ring is $4(2\sqrt{\beta\omega} + 2\tau)m$, see [15] which is linear in m .

The next simple architecture to be considered is that of a two dimensional grid. In [15] it was shown for the case $m = n$, that mapping the n^2 grid points of the square homographically into a $\kappa \times \kappa$ grid of processors, and using a substructured Gaussian Elimination [10,19], the total time for one of the solution steps in ADI is of the form

$$T_G(P) \approx \alpha \frac{n^2}{P} + \delta \frac{n}{\sqrt{P}} + \eta \sqrt{P} + O(1),$$

where α, δ, η are constants independent of P . The minimum time for an arbitrarily large processor grid is of the form $O(n^{2/3})$. Multiplying this by the number of steps which is $O(\log^2 n)$ we arrive at an asymptotic complexity of

$$T_{opt,Grid} = O(n^{2/3} \log^2 n)$$

compared with $O(n^2 \log^2 n)$ in the sequential case. Note that the number of processors to achieve this optimal time is $O(n^{4/3})$.

We next consider the implementation of ADI on hypercubes. We simplify the exposition by assuming again that $m = n$. We use the same mapping as before by embedding the 2-D grid into the hypercube as was described in Section 2.2. Then, scalar cyclic reduction is employed to solve the successive tridiagonal systems in the algorithm. Assume that the 2-D mesh is first subdivided into small $(n/\kappa) \times (n/\kappa)$ squares and that the sub-square in position (i, j) is assigned to processor (i, j) of the grid. Then each of the solve phases in ADI amounts to solving in each row or column of the grid n/κ independent tridiagonal systems each of which is split into κ equal parts.

Consider the process on each of the n/κ tridiagonal systems separately. Each of the first $\log(n/\kappa)$ steps of cyclic reduction requires only communication between neighboring processors in which a fixed number of elements is transmitted to neighbors namely 4 elements from each direction. The total time for arithmetic operations of the forward and backward sweep is $O(n/\kappa)$ since it is similar to that of performing the cyclic reduction algorithm on a tridiagonal system of size n/κ on a single processor. After these $\log(n/\kappa)$ first steps are completed, each processor will end up with one equation of a $\kappa \times \kappa$ tridiagonal system. Cyclic reduction on such a system can be performed

in time $O(\log \kappa)$ thanks to the fact that the distance between equations i and $i + 2^j$ is constant due to the assignment using Gray codes [13].

The total time for all n/κ systems is of the form $O(\frac{n^2}{\kappa}) + O(\frac{n}{\kappa} \log P)$. Observe that for the maximum allowable value of P , $P = n^2$ we get a time of the form $O(\log P)$. Therefore, a logarithmic time in n is achievable for each step of ADI with the hypercube topology. Moreover, the total time over the $O(\log^2 n)$ steps required for convergence would become $T_{opt,hyp} = O(\log^3 n)$ which does not compare favorably with the $O(\log^2 n)$ of the hypercube BCR described earlier. On the other hand the implementation of ADI is far simpler than the parallel BCR. All that is required is to implement efficient multiple tridiagonal solvers in the x and y directions. Moreover, ADI is more general than block cyclic reduction, although the theory for nonseparable problems does not provide the optimal parameters and the number of steps may be much higher than what is obtained with separable problems.

6. Conclusions. We have proposed several parallel implementations of fast algorithms for solving elliptic equations. As was shown in [8] the block cyclic reduction algorithm using partial fraction expansions leads to a viable and efficient approach for computers with small numbers of processors. In this paper we have also considered the case where the number of processors is large compared with the problem size. One common feature of all the different variants is the complexity of their implementation. This problem becomes even more acute when the problem dimension is not carefully chosen (e.g. $n = 2^k - 1$ for BCR). It is not clear to us whether much simpler algorithms should not be preferred even at the expense of sacrificing some efficiency. The explicit methods described in Section 3 are simpler to implement but require far too many processors to be of practical use for problems of reasonable size. The alternative of the ADI techniques considered in Section 5 constitutes a good compromise between efficiency and ease of implementation.

Acknowledgement. The authors would like to thank George Cybenko for his helpful remarks.

The research of the first author was supported by the National Science Foundation under Grants No. US NSF-MIP-8410110, US NSF DCR85-09970, US NSF CCR-8717942 and by AT&T Grant AT&T AFFL67Sameh. The research of the second author was supported by NASA under USRA Grant No. NCC 2-387.

REFERENCES

- [1] R. E. BANK AND D. J. ROSE, *Marching algorithms for elliptic boundary value problems. I: the constant coefficient case*, SIAM J. Numer. Anal., 14 (October 1977),

pp. 792-829.

- [2] —, *Marching algorithms for elliptic boundary value problems. II: the variable coefficient case*, SIAM J. Numer. Anal., 14 (October 1977), pp. 950-969.
- [3] O. BUNEMAN, *A compact non-iterative Poisson solver*, Tech. Rep. 294, Stanford University Institute for Plasma Research, Stanford, Calif., 1969.
- [4] B. BUZBEE, G. GOLUB, AND C. NIELSON, *On direct methods for solving Poisson's equation*, SIAM J. Numer. Anal., 7 (December 1970), pp. 627-656.
- [5] D. FISCHER, G. GOLUB, O. HALD, C. LEIVA, AND O. WIDLUND, *On Fourier-Toeplitz methods for separable elliptic problems*, Math. Comp., 28 (April 1974), pp. 349-368.
- [6] E. GALLOPOULOS AND Y. SAAD, *Parallel elliptic solvers*, in Proc. Third SIAM Conference on Parallel Processing for Scientific Computing, G. Rodrigue, ed., SIAM, December 1987, pp. 51-55.
- [7] —, *Parallel block cyclic reduction algorithm for the fast solution of elliptic equations*, in Lecture Notes in Computer Science No. 297: Proc. 1987 First Int'l. Conf. on Supercomputing, T. S. Papatheodorou, E. N. Houstis, and C. D. Polychronopoulos, eds., Springer-Verlag, New York, Feb. 1988, pp. 563-575.
- [8] —, *Parallel block cyclic reduction algorithm for the fast solution of elliptic equations*, Parallel Comput., (to appear).
- [9] E. GALLOPOULOS AND A. H. SAMEH, *Solving elliptic equations on the Cedar multiprocessor*, in Aspects of Computation on Asynchronous Parallel Processors, M. H. Wright, ed., Elsevier Science Pub. B. V. (North-Holland), 1989, pp. 1-12.
- [10] D. GANNON AND J. V. ROSENDALE, *On the impact of communication complexity on the design of parallel numerical algorithms*, IEEE Trans. Comput., C-33 (December 1984), pp. 1180-1194.
- [11] R. HOCKNEY, *A fast direct solution of Poisson's equation using Fourier analysis*, J. Assoc. Comput. Mach., 12 (1965), pp. 95-113.
- [12] R. HOCKNEY AND C. JESSHOPE, *Parallel Computers*, Adam Hilger, 1983.
- [13] S. L. JOHNSON, *Odd-even cyclic reduction on ensemble architectures and the solution of tridiagonal systems of equations*, Tech. Rep. RR-339, Yale University, Department of Computer Science, October 1984.
- [14] S. L. JOHNSON AND C. T. HO, *Multiple tridiagonal systems, the alternating direction methods and boolean cube configured multiprocessors*, Tech. Rep. RR-532, Yale University, Department of Computer Science, 1987.
- [15] S. L. JOHNSON, Y. SAAD, AND M. H. SCHULTZ, *Alternating direction methods on multiprocessors*, SIAM J. Sci. Statist. Comput., 8 (September 1987), pp. 686-700.
- [16] H. T. KUNG, *New algorithms and lower bounds for the parallel evaluation of certain rational expressions and recurrences*, J. Assoc. Comput. Mach., 23 (April 1976), pp. 252-261.
- [17] E. M. REINGOLD, J. NIEVERGELT, AND N. DEO, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, 1977.
- [18] A. H. SAMEH, S. C. CHEN, AND D. J. KUCK, *Parallel Poisson and biharmonic solvers*, Computing, 17 (1976), pp. 219-230.
- [19] A. H. SAMEH AND D. J. KUCK, *On stable parallel linear system solvers*, J. Assoc. Comput. Mach., 25 (January 1978), pp. 81-91.
- [20] J. STOER AND R. BURLISCH, *Introduction to Numerical Analysis*, Springer-Verlag, New York, 1980.
- [21] P. N. SWARZTRAUBER, *Fast Poisson solvers*, in Studies in Numerical Analysis, G. H. Golub, ed., Mathematical Association of America, 1984, pp. 319-369.
- [22] —, *A direct method for the discrete solution of separable elliptic equations*, SIAM J. Numer. Anal., 11 (December 1974), pp. 1136-1150.

- [23] P. N. SWARZTRAUBER AND R. A. SWEET, *Vector and parallel methods for the direct solution of Poisson's equation*, J. Comput. Appl. Math., (To appear).
- [24] D. A. SWAYNE, *Matrix operations with rational functions*, in Proc. 7th Manitoba Conf. Numerical Mathematics, Utilitas Mathematica, Winnipeg, Manitoba, 1977, pp. 581-589.
- [25] R. A. SWEET, *A parallel and vector cyclic reduction algorithm*, SIAM J. Sci. Statist. Comput., 9 (July 1988), pp. 761-765.
- [26] ———, *A cyclic reduction algorithm for solving block tridiagonal systems of arbitrary dimension*, SIAM J. Numer. Anal., 14 (September 1977), pp. 707-720.
- [27] O. B. WIDLUND, *On the use of fast methods for separable finite difference equations for the solution of general elliptic problems*, in Sparse Matrices and their Applications, D. J. Rose and R. A. Willoughby, eds., Plenum Press, 1972, pp. 121-131.

~~SECRET~~ ~~UNCLASSIFIED~~ ~~CONFIDENTIAL~~