IMPLEMENTATION OF A DIVIDE AND CONQUER CYCLIC REDUCTION ALGORITHM ON THE FPS T-20 HYPERCUBE

Christopher L. Cox, Dept. of Mathematical Sciences, Clemson University, Clemson, SC 29634-1907

Abstract

A simple variant of the odd-even cyclic reduction algorithm for solving tridiagonal linear systems is presented. The target architecture for this scheme is a parallel computer with nodes which are vector processors, such as the Floating Point Systems T-Series hypercube. Of particular interest is the case where the number of equations is much larger than the number of processors. The matrix system is partitioned into local subsystems, with the partitioning governed by a parameter which determines the amount of redundancy in computations. The algorithm proceeds after the distribution of local systems with independent computations, all-to-all broadcast of a small number of equations from each processor, solution of this subsystem, more independent computations, and output of the solution. Some redundancy in calculations between neighboring processors results in minimized communication costs. One feature of this approach is that computations are well balanced, as each processor executes an identical algebraic routine.

A brief description of the standard cyclic reduction algorithm is given. Then the divide and conquer strategy is presented along with some estimates of speedup and efficiency. Finally, an occam program for this algorithm which runs on the FPS T-20 computer is discussed along with experimental results.

1. Introduction

In this paper we present an extension of the cyclic reduction algorithm to a method for use on a distributed-memory parallel processor with nodes which are able to perform vector arithmetic. The emphasis is on exploitation of vector capabilities while keeping communication costs to a minimum. This section contains a description of the standard cyclic reduction algorithm and an outline of the rest of this paper.

We are concerned with the solution of a system of linear equations, Ax=f, for which the coefficient matrix A is symmetric, tridiagonal and positive definite. This matrix system has the form

$$\begin{bmatrix} b_1 & a_2 & & & \\ a_2 & b_2 & a_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & a_n \\ & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \cdot \\ \cdot \\ \cdot \\ f_n \end{bmatrix} \qquad (1.1)$$

Letting $x_0 = x_{n+1} = a_1 = a_{n+1} = 0$, (1.1) is equivalent to the system of equations

$$a_i x_{i-1} + b_i x_i + a_{i+1} x_{i+1} = f_i, \; i = 1,n \qquad (1.2)$$

Suppose for convenience that $n=2^k-1$. Cyclic reduction consists of two phases, the reduction phase and the backsubstitution phase. First, the system (1.2) is reduced to a system of (n-1)/2 by (n-1)/2 equations in the unknowns x_2, $x_4, \ldots, x_{n-1}$ by dropping the odd equations after using them to substitute for the odd numbered unknowns in the even equations. That is, we have

$$a_i^{(2)} x_{i-1}^{(2)} + b_i^{(2)} x_i^{(2)} + a_{i+1}^{(2)} x_{i+1}^{(2)} = f_i^{(2)}, \; i = 1, \frac{n-1}{2} \qquad (1.3)$$

where $x_0^{(2)} = x_{\frac{n+1}{2}}^{(2)} = 0$. This procedure is repeated k-2 times until one equation is left,

$$b_1^{(k)} x_1^{(k)} = f_1^{(k)} \qquad (1.4)$$

The complete algorithm for the reduction phase follows.

For j = 1, k-1

  For i = 1, $\frac{n+1}{2^j}$

$$a_i^{(j+1)} = \frac{-a_{2i}^{(j)} a_{2i-1}^{(j)}}{b_{2i-1}^{(j)}} \quad (\text{if } i \neq 1)$$

$$b_i^{(j+1)} = b_{2i}^{(j)} - \frac{(a_{2i}^{(j)})^2}{b_{2i-1}^{(j)}} - \frac{(a_{2i+1}^{(j)})^2}{b_{2i+1}^{(j)}} \qquad (1.5)$$

$$f_i^{(j+1)} = f_{2i}^{(j)} - \frac{a_{2i}^{(j)} f_{2i-1}^{(j)}}{b_{2i-1}^{(j)}} - \frac{a_{2i+1}^{(j)} f_{2i+1}^{(j)}}{b_{2i+1}^{(j)}}$$

  Next i

Next j

After solving for $x_1^{(k)}$ in (1.4), the backsubstitution phase proceeds with the algorithm

For j = k-2, 1

$$x_{2i}^{(j)} = x_i^{(j+1)},\ i=1,\frac{n+1}{2^j}-1 \qquad (1.6)$$

$$x_{2i-1}^{(j)} = \frac{1}{b_{2i-1}^{(j)}}(f_{2i-1}^{(j)} - a_{2i-1}^{(j)}x_{2i-2}^{(j)} - a_{2i}^{(j)}x_{2i}^{(j)}),\ i = 1,\ \frac{n+1}{2^j}$$

Next j

where $x_0^{(j)} = x_{\frac{n+1}{2^{j-1}}}^{(j)} = 0,\quad 1 \le j \le k-2$

The algorithm can be easily generalized for arbitrary values of the dimension n. We have presented the algorithm in terms of retaining the even numbered equations at each step, and factoring out the odd equations. Clearly, one could instead retain the odd numbered equations, or alternate even and odd between cycles to carry certain unknowns through the entire reduction step. The method generalizes in an obvious way to systems with a block tridiagonal coefficient matrix. Cyclic reduction can be used for nonsymmetric systems. Because we are interested primarily in applications which produce symmetric systems, we limit our discussion to that case.

An important point to make is that the procedures for calculation of the coefficients in (1.5) and unknowns in (1.6) are inherently parallel with respect to i. The exploitation of this fine-grain parallelism has been analyzed for implementation on vector processors and on various parallel architectures. Hockney and Jesshope present a version for solving an n-dimensional system on n processors keeping the level of parallelism constant at n, [1]. Kershaw considered using cyclic reduction as a preconditioner for conjugate gradients to solve a block tridiagonal system on a Cray 1, [2]. Johnsson has formulated cyclic reduction algorithms for various configurations, including binary trees and n-cubes, [3]. Lambiotte and Voigt used cyclic reduction on a CDC STAR-100 computer, [4]. They point out that the method can be described as an LTU decomposition of a permutation on the original linear system. This rearrangement of the equations and unknowns yields an algorithm consisting of operations on vectors whose elements are accessed contiguously, eliminating strides called for in (1.5) and (1.6). Additional references on cyclic reduction can be found in Ortega and Voigt, [5].

In this work we present a simple adaptation of cyclic reduction to a parallel computer whose nodes are vector processors. Vector arithmetic will be used in connection with the fine-grain, or statement-level parallelism. A "divide and conquer" approach will provide coarse-grain, or subroutine-level parallelism which will correllate in terms of hardware to parallelism over multiprocessors. The divide and conquer algorithm is presented in detail in Section 2. We assume that the number of processors is much smaller than the number of equations, so that many equations will be passed to each node. We will limit this discussion to the case where the coefficients and unknowns in (1.1) are scalars, though the block case is of significant interest. In Section 3 the algorithm is discussed again in the context of LTU decomposition similar to that in [4]. An example of a parallel processor in which each processor performs vector arithmetic is the FPS T-Series hypercube. The implementation of the divide and conquer algorithm on the 16 processor FPS T-20 hypercube and results are presented in Section 4. The program is an outgrowth of an occam program written by Vandiver, [6]. Her implementation of a simple version of the algorithm presented here, using integer arithmetic, ran on a parallel processor simulator on a VAX 8600.

2. Divide and Conquer Algorithm

Our main interest is a variant on the standard cyclic reduction algorithm for a parallel processor with vector nodes, based on a 'divide and conquer' strategy. If there are p processors, the system of n equations is partitioned into p local systems so that the reduction and backsolving phases can be performed on each local system independently of the others. This necessitates some redundancy in calculations for the sake of lessened communications. Each local system is reduced to q equations at which point there is no overlap of equations. From a global perspective, the system of n equations has been reduced to pq equations in pq unknowns. An all-to-all broadcast of these pq equations takes place, and each processor solves the pqxpq system, then proceeds with backsubstitution to determine the unknowns associated with its local system. The distribution of coefficients and the reduction process is illustrated in Figure 2.1 for the case with n=19, p=2, and q=2. In this example there will be two reduction steps. The i^{th} equation after j-1 steps is denoted $E_i^{(j)}$.

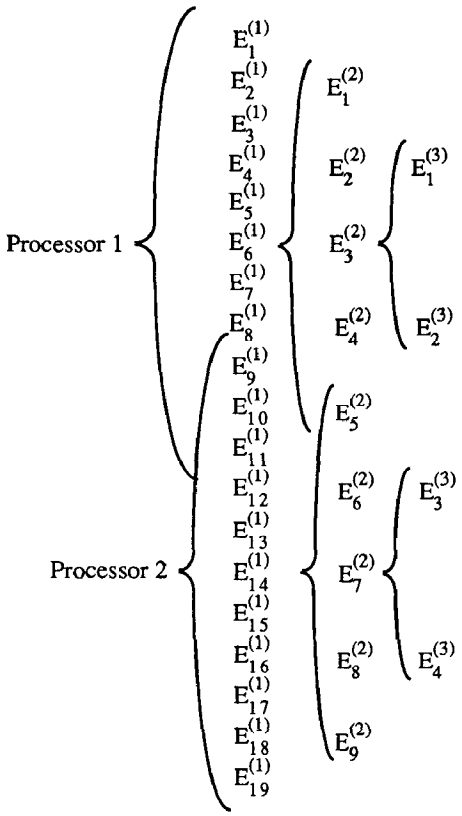

Figure 2.1 Example with n=19, q=2, p=2

The entries of the right hand side are distributed in a similar way.

Letting n_p be the local system size, we can determine the relationship between n, p, q and n_p. For convenience we assume $n_p=(q+1)2^{k-1}-1$ for some integer k so that the q unknowns in each local system are carried through a reduction procedure where the odd unknowns are always factored out. A simple proof by induction verifies the following proposition.

Proposition 2.1: Suppose the tridiagonal system of n equations is to be divided into p overlapping partitions each with

$n_p = (q+1)2^{k-1}-1$ equations, q and k integers. If

$$n = pn_p - (p-1)\left(\frac{n_p-q}{q+1}\right) \qquad (2.1)$$

then k-1 reduction steps (always factoring out the odd unknowns) will reduce each local system to q equations in q+2 unknowns where each reduced local system shares one unknown with its neighbor on either side.

In practice, n and p, or an upper limit for p, are given, and one must choose q and n_p. One way to determine these parameters would be to find n_p from (2.1) using various values of q, choosing the q which yields the value for n_p closest to but not greater than $(q+1)2^{k-1}-1$ for some integer k. The global matrix system could then be extended with 1's on the main diagonal and 0's elsewhere to length corresponding to q and $n_p=(q+1)2^{k-1}-1$. A short algorithm could be written to implement this procedure, choosing q from a specified range.

As one would expect, the efficiency of the divide and conquer method is directly related to q, which governs the amount of redundancy in computation. This will be restated more precisely in another proposition, but this is evident when one realizes that the number of equations which a processor originally shares with the processors on either side is the ratio which is multiplied by p-1 in (2.1).

Before reduction begins, each local system will have the form

$$a_i x_{i-1} + b_i x_i + a_{i+1} x_{i+1} = f_i, \; i = 1, n_p \qquad (2.2)$$

In consideration of the global system, on the first processor we can let $x_0=0$ and on the last processor set $x_{n_p+1}=0$. Otherwise these terms are nontrivial because of the overlap. Thus each local system is a system of n_p equations in n_p+2 unknowns.

On each processor, the reduction algorithm (1.5) is performed k-1 times with $n_1=n_p$ so that each local system is reduced to one of the form

$$a_i^{(k)} x_{i-1}^{(k)} + b_i^{(k)} x_i^{(k)} + a_{i+1}^{(k)} x_{i+1}^{(k)} = f_i^{(k)}, \quad i = 1, q \qquad (2.3)$$

A communication step takes place now. We have to solve a pqxpq tridiagonal system, and each processor holds q equations of the system. There are a variety of ways to communicate the coefficients and solve for the pq unknowns. We propose a simple 'all-to-all broadcast', i.e. for j=1,p, processor j sends the coefficients in its version of (2.3) to every other processor. Each processor will solve the same pqxpq system, then use the q+2 unknown values associated with that processor's local system. Then the backsubstitution algorithm (1.6) is carried out on each processor, beginning with q+2 known values rather than 1.

Now the solution is complete, and the values can be gathered according to the following rule. On processor 1, with $1 \le i \le p$, variables locally numbered 1 through n_p have global number $\frac{i-1}{q+1}[q(n_p+1)]+1$ to $\frac{i-1}{q+1}[q(n_p+1)]+n_p$.

Now we consider efficiency. We assume q is small enough that the time in communicating and solving the pqxpq system is negligible with regard to the time spent in reduction and backsubstitution. Timing formulas for vectorized cyclic reduction applied to an nxn system on a single processor have the form

$$\text{Total Time} = C_1 + C_2 \log_2 n + C_3 n \qquad (2.4)$$

(For example, see [4]). Thus, for suitably large values of n, time is essentially linear with respect to n. If this is the case, then we have the following.

Proposition 2.2: If the time-intensive stages of the divide and conquer algorithm are dominated by operations which are $O(n_p)$, then the speedup S_p, and efficiency, E_p, satisfy the inequalities

$$S_p \ge \frac{pq}{q+1} \qquad E_p \ge \frac{q}{q+1} \qquad (2.5)$$

Proof: From (2.1) we have

$$n_p = \frac{(q+1)n + (1-p)q}{pq+1} \qquad (2.6)$$

By definition,

$$S_p = \frac{\text{time to solve on one processor}}{\text{time to solve on p processors}} \quad \text{and} \quad E_p = \frac{S_p}{p}$$

Thus,

$$S_p = \frac{n}{n_p} = \frac{n(pq+1)}{(q+1)n+(1-p)q} \ge \frac{pq+1}{q+1} \ge \frac{pq}{q+1}$$

and the result for E_p follows immediately.

Remark: An obvious motivation for using multiple processors is the case in which the problem is too large to fit on one processor. In this case, one may wish to consider the speedup and efficiency with respect to a base timing on r processors, r greater than 1. With respect to r, for p greater than r, speedup and efficiency for p processors may be defined as

$$S_{\frac{p}{r}} = \frac{\text{time to solve on r processors}}{\text{time to solve on p processors}}, \quad E_{\frac{p}{r}} = \frac{S_{\frac{p}{r}}}{\frac{p}{r}} \qquad (2.7)$$

It must be pointed out that the efficiency relative to r processors is exactly that - it should not be assumed that the efficiency relative to one processor will be the same. Under the same hypothesis as in Proposition 2.2, the conclusion in (2.5) holds for the relative efficiency. Using (2.6) and (2.7), we have

$$S_{\frac{p}{r}} = \frac{n_r}{n_p} = \left(\frac{(q+1)n + (1-r)q}{(q+1)n + (1-p)q}\right)\left(\frac{qp+1}{qr+1}\right) \ge \frac{pq+1}{rq+1} \ge \frac{pq}{r(q+1)}$$

so that

$$E_{\frac{p}{r}} \ge \frac{q}{q+1}$$

For a complete analysis of efficiency, other factors must be considered. Proposition 2.2 implies that efficiency of the reduction and backsubstitution phases improves as q gets large. As q increases, the amount of time in broadcasting and solving the pqxpq system is no longer insignificant. An appropriate range for q depends on n, p, and parameters governing the particular computer being used such as scalar arithmetic speed, vector arithmetic start-up times and communication times.

Stopping the reduction procedure before there is just one equation left, while using a single-processor vector computer, is mentioned with additional references in Ortega and Voigt, [5]. This strategy makes sense because once vector lengths decrease to a certain size, use of the vector arithmetic unit is no longer worthwhile. A scalar method is used to solve for the unknowns at that level and backsubstitution proceeds from there.

Johnsson discusses a hybrid method for a parallel

processor which uses Gaussian elimination at the local level and cyclic reduction at the global level, (the opposite of our method), which he calls GECR, [3]. This may be appropriate for the case where there is no vector arithmetic capability.

3. Rearranging for Contiguous Access

The Divide and Conquer algorithm as described in Section 2 works efficiently under the assumption that the parallel processor being used is capable of vector arithmetic with strides, i.e. there is a reasonably fast gather-scatter capability. This is not always the case. For example, an FPS T-Series hypercube works most efficiently when vector operations involve elements which are stored contiguously. Toward this end, a reduction step of the cyclic reduction algorithm can be described in terms of a block LTU (for the symmetric case, LTL^T) decomposition of a permuted matrix [4]. The rows and columns of the matrix are rearranged so that each step in the cyclic reduction algorithm can be expressed with vector operations which require no gather-scatter operations. We describe one reduction step using this approach, applied to the local system Ax=f, which has the form

$$\begin{bmatrix} a_1 & b_1 & a_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n_p} & b_{n_p} & a_{n_p+1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \cdot \\ \cdot \\ x_{n_p} \\ x_{n_p+1} \end{bmatrix} = \begin{bmatrix} f_1 \\ \cdot \\ \cdot \\ f_{n_p} \end{bmatrix} \quad (3.1)$$

At the beginning of the reduction phase, each local system will consist of n_p equations in n_p+2 unknowns. Assume that n is odd and that the odd numbered unknowns are to be factored out, (i.e., $n_p=(q+1)2^{k-1}-1$). The (local) rearranged system will have the form

$$\hat{A}\hat{x} = \begin{bmatrix} B_1 & A_1 \\ A_2 & \bar{B}_2 \end{bmatrix} \begin{bmatrix} X_1 \\ \bar{X}_2 \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \end{bmatrix} = \hat{f} \quad (3.2)$$

where

$$A_1 = \begin{bmatrix} a_1 & A_2^T & a_{n_p+1} \end{bmatrix}, \quad \bar{B}_2 = \begin{bmatrix} 0 & & 0 \\ 0 & B_2 & 0 \\ 0 & & 0 \end{bmatrix}, \quad \bar{X}_2 = \begin{bmatrix} x_0 \\ X_2 \\ x_{n_p+1} \end{bmatrix}$$

$$B_1=\text{diag}(b_1, b_3, \ldots, b_{n_p}),\ B_2=\text{diag}(b_2, b_4, \ldots, b_{n_p-1})$$

$$X_1=\text{col}(x_1, x_3, \ldots, x_{n_p}),\ X_2=\text{col}(x_2, x_4, \ldots, x_{n_p-1})$$

$$F_1=\text{col}(f_1, f_3, \ldots, f_{n_p}),\ F_2=\text{col}(f_2, f_4, \ldots, f_{n_p-1})$$

$$A_2 = \begin{bmatrix} a_2 & a_3 & & & \\ & a_4 & a_5 & & \\ & & \ddots & \ddots & \\ & & & \ddots & \ddots \\ & & & & a_{n_p-1} \; a_{n_p} \end{bmatrix}$$

Note that X_1 contains the unknowns which are factored out. Setting $j=(n_p+1)/2$ and $m=j-1$, B_1 and B_2 have dimension jxj and mxm. A_1 and $\bar{B}_2$ have dimension jx(j+1) and mx(j+1). Because $\hat{A}$ is not symmetric, an LTU decomposition is performed, with L and U given by

$$L = \begin{bmatrix} I_{jxj} & 0 \\ A_2B_1^{-1} & I_{mxm} \end{bmatrix} \quad U = \begin{bmatrix} I_{jxj} & B_1^{-1}A_1 \\ 0 & I_{(m+2)x(m+2)} \end{bmatrix} \quad (3.3)$$

The lower right block T_{22} of the matrix T is the new coefficient matrix resulting from one reduction step. T_{22} has dimension mx(j+1) and is tridiagonal, having form

$$T_{22} = \begin{bmatrix} \nu_1 & \mu_1 & \nu_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \ddots \\ & & & \nu_m & \mu_m \; \nu_{m+1} \end{bmatrix} = \bar{B}_2 - A_2B_1^{-1}A_1 \quad (3.4)$$

To express the computation of T_{22} in vector arithmetic, define vectors β, γ, α, and δ as

$$\beta_i = \frac{1}{b_{2i-1}},\ i=1, m \qquad \gamma_i = b_{2i},\ i=1, m$$
$$\alpha_i = a_{2i},\ i=1, j \qquad \delta_i = a_{2i-1},\ i=1, j \quad (3.5)$$

Then we have

$$\mu_i = \gamma_i - (\alpha_i^2\beta_i + \delta_{i+1}^2\beta_{i+1}),\ \ i=1,m \quad (3.6)$$

and

$$\nu_i = -\alpha_i\delta_i\beta_i,\quad i=1, j \quad (3.7)$$

The right hand side associated with T_{22} is computed using the equation

$$(\tilde{F}_2)_i := (F_2)_i - [\alpha_i\beta_i (F_1)_i + \delta_{i+1}\beta_{i+1} (F_1)_{i+1}],\ i=1,m \quad (3.8)$$

After k-1 reduction steps each processor holds a system of q equations in q+2 unknowns. These coefficients are shared in an all-to-all broadcast. Each processor solves the pqxpq system and then extracts the q+2 values associated with its local system.

The backsubstitution step, performed k-1 times, to determine X_1 can be deduced from (3.2),

$$X_1 = B_1^{-1} (F_1 - A_1 \bar{X}_2)$$

and formulated in vector arithmetic as

$$(X_1)_i=\beta_i\{(F_1)_i - [\alpha_i(\bar{X}_2)_i + \delta_i(\bar{X}_2)_{i-1}]\},\ i=1, j \quad (3.9)$$

The vector product terms βF_1, $\beta\alpha$, and $\beta\delta$ can be computed in the reduction phase and stored for use in backsubstitution.

4. Implementation and Results

An occam program for the divide and conquer algorithm was written for the Floating Point Systems T-20 Hypercube at Clemson University. This is a 16 node MIMD computer with nodes which perform vector arithmetic.

Each T-Series node contains 1 Mbyte of random-access memory linked by vector registers to a 64-bit vector-arithmetic unit capable of 10 MFLOPS (peak). The memory is divided into four subbanks each containing 256 1024-byte slices, [7]. These subbanks form two banks, bank B having three subbanks, and bank A having one. After allowing for storage of the operating system and user code, the user can allocate sections of memory somewhat freely. Optimum data transfers

between memory and the vector unit require that input operands are in different banks and that the operands are aligned, which means that each vector begins at a slice boundary. Vector arithmetic and communication between processors can be performed concurrently.

The algorithm consists of six stages, the first being initialization of the local matrix systems. This can be done in one of two ways, either by dividing up the global system on node 0 and broadcasting the local systems, or simply constructing each local system on its corresponding processor. The former strategy would be useful in the case where the system is assembled on a serial computer and sent to the hypercube to be solved. Our primary interest is the solution of systems generated by numerical methods for partial differential equations. It has been recognized that there is parallelism as well as opportunity for vectorization in the assembling of these systems, [8]. Thus we have each processor construct its local system.

The second stage consists of the reduction phase performed on each local system. The moves and arithmetic operations in one loop of the reduction procedure corresponding to equations (3.5)-(3.8) can be arranged into 8 steps (each corresponding to a move operation), as shown in figure 4.1. Each step consists of a move (gather/scatter or shift for alignment) and one or two vector computations. These steps are ordered so that moves and arithmetic operations could be executed concurrently, if the computer has that capability. The product terms $\alpha\beta$, $\delta\beta$ and βF_1 are saved for use in the backsolving phase. The gather move in step VIII is also executed once before the first loop begins, and is not executed in the final reduction loop.

Step	Move	Compute
I.	α (gather)	β
II.	δ (gather)	$\alpha\beta$, $\alpha\alpha\beta$
III.	F_1 (gather)	$\delta\beta$, $\alpha\beta\delta$
IV.	γ (gather)	$\delta\delta\beta$, $-\alpha\beta\delta$
V.	$\delta\delta\beta$ (shift)	$\delta\beta F_1$, βF_1
VI.	$\delta\beta F_1$ (shift)	$\alpha\alpha\beta+\delta\delta\beta$, $\gamma-(\alpha\alpha\beta+\delta\delta\beta)$
VII.	F_1 (gather)	$\alpha\beta F_1$, $\alpha\beta F_1+\delta\beta F_1$
VIII.	b (gather)	$F_2-(\alpha\beta F_1+\delta\beta F_1)$

Figure 4.1 Steps in one reduction loop

Next it is necessary for each processor to share its local system (2.3) with the others. We use a routine written by Dan Warner which implements a variant of the Johnsson and Ho all-to-all broadcast based on rotated binomial trees, [9], [10]. The T-20 is self-synchronizing so the user only has to make sure that each send has a corresponding receive.

The fourth stage of the algorithm is the solution on each processor of the same pq by pq system. Obviously this is redundant, but it is not wasteful with respect to time and it saves a communication step that would be needed after one processor solved the system while the others sat idle. Originally we used (software) double precision floating point arithmetic to solve but discovered that the vectorized cyclic reduction routine was faster even for small dimensions.

Following the solution of the reduced system is the backsolving stage, (3.9), one loop of which is shown in figure 4.2. Note again that in the first two steps, data moves are coupled with calls to the vector arithmetic unit. The moves into shift registers for the computation steps must be given priority over the shifting and scatter moves. The occam PRI PAR statement is useful for this purpose.

In the sixth and final stage one can gather the global solution through an all-to-one broadcast of the local solutions. This will permit the user to send the global solution through node 0 to the host computer. The communication step is unnecessary if one merely desires a printout of the solution, as each processor can print its values to the screen (or to a log file). For test purposes we solve a system for which the exact solution is known, so we have each processor compute and print the mean square error of its associated unknowns.

Step	Move	Compute
I.	X_2 (shift)	$\delta\beta X_2$, $\beta F_1-\delta\beta X_2$
II.	X_2 (scatter)	$\alpha\beta X_2$, $(\beta F_1-\delta\beta X_2)-\alpha\beta X_2$
III.	X_1 (scatter)	

Figure 4.2 Steps in one backsolve loop

Numerical results are presented in two parts. To study aspects of vectorization, a series of experiments were conducted on a single node. Then a number of multi-node tests were conducted to examine the properties (especially speedup and efficiency) of the Divide and Conquer algorithm.

Table 4.1 is a listing of timings for portions of the cyclic reduction algorithm on one processor only. Here n is the dimension of the system and times are in microseconds (as are all subsequent timings). In all cases, q=1, so $n=2^k-1$ and thus the reduction and backsolve steps each were carried out in k-1 loops. Each time given is a sum of timings of k-1 vector operations on vectors of varying size. The first three times show how much of the total computing time was taken by each type of vector arithmetic operation. For example, for n=31, the vector reciprocals in step II of the reduction phase took a total of 1704 microseconds. The vector negative in step VIII took 415 microseconds. Each term-by-term multiply, addition, or difference of vectors took, on the average, 528 microseconds. (Times actually ranged from 521 to 539). Also listed are the total times spent in vector arithmetic and times spent in vector moves (either gather/scatter or shifts). Unrolling gather/scatter move loops by a factor of 4 resulted in a 20% time savings for the larger dimensions. All arithmetic operations are coupled with moves, as shown in figures 4.1 and 4.2. If there was true concurrency between arithmetic and moves, then the total time for reduction and backsolving (items 7 and 8 in Table 4.1) would be the same as the move time in 5, since this would overshadow the time for computing.

Two additional remarks about vectorization are in order. With the addition of fast scalar arithmetic (coming on the next T-Series generation), it will be worthwhile to stop the reduction early, say when the reduced system is m by m rather than 1 by 1. The unknowns associated with that level can be solved for by the tridiagonal algorithm, then the backsolving phase is begun with m known values. Also, we discovered that vector operations ran faster when operands filled slices fully, i.e. vectors had lengths which were multiples of 128. This may be related to the fact that we used the FPS higher level generic and single node subroutines, [11].

Table 4.1 provides some insight into what may be expected in the multi-processor results. Our primary goal is to compare results with the inequalities (2.5) in Proposition 2.2. Rows 7 and 8 reveal how large n_p must be for the hypothesis of Proposition 2.2 to be met. Certainly one would not expect to see optimal results for efficiency and speedup if n_p were less than 1000. A set of timings for various hypercube dimensions is given in Table 4.2. Here p is the number of processors, the local system size is $n_p=(q+1)2^{k-1}-1$, and the global size n is given by (2.1). In each case q=8 except for p=1 for which q=1. Though the values for n differ, we compare results as though they were all equal to 8191. (This is giving an advantage to the p=1 results.) The sum of the times spent in the reduction and backsubstitution phases is given first. Next

are the times for communication and solution of the pq by pq system. Here and in all subsequent results, speedup and efficiency are based on the total of these three timings.

Proposition 2.2 predicts an efficiency of 87.5%. Discrepencies in the numerical results are caused by two factors. First, the solver used for the reduced global system is not as efficient as we would like it to be. The capability for fast scalar arithmetic will improve this considerably. Second, as mentioned above, the local size n_p=575 is below the range of values for which the conditions of Proposition 2.2 are satisfied. From the perspective that n is approximately equal to p times n_p, it is not worthwhile to use 16 processors to solve systems which have less than 16,000 unknowns.

Memory limitations made it inconvenient to let n_p be larger than 8191. In order to keep the n_p for p=16 in the range of interest (larger than 1000), we use the generalized definitions of speedup and efficiency given in (2.7). Efficiency and speedup in subsequent tables are calculated according to (2.7), with r equal to the least number of processors used in each set of results. Three sets of results, with q held constant in each set, are displayed in Table 4.3. In these cases n varies somewhat yet we compare results as if it were constant. The predicted efficiencies for these three sets are 75%, 86% and 94%, respectively. A more efficient algorithm for the intermediate solve step would bring the weaker results closer to the predicted values. Finally, in Table 4.4, we no longer hold q constant but instead compare cases for which n is exactly the same. Because q is greater than or equal to 3, one would expect efficiencies no worse than 75%. The results vary within about 10% of this prediction.

5. Summary

In this paper we have described a divide and conquer cyclic reduction algorithm for a parallel processor with vector processing nodes. A rough estimate for efficiency was derived, based on a parameter which controls redundancy in calculations. Numerical results generated on an FPS T-20 hypercube computer lent credence to this estimate and highlighted some shortcomings as well. This estimate was based on the assumption that the time to communicate and solve the system at the most reduced level is insignificant in comparison to the times taken by the reduction and backsolving phases. Further analysis will yield more precise performance estimates.

Obviously, these results are machine-dependent. Improvements in hardware such as fast scalar arithmetic, which is promised for the next generation of T-Series computers, will influence the way in which this algorithm is implemented. Scalar arithmetic will replace vector arithmetic earlier in the reduction process. Faster gather-scatter operations may affect the balance in timings between moves and computation.

The next stage in this effort is to generalize this algorithm to the block case. Also, an abbreviated version of this method may be useful as a preconditioner for a conjugate gradient solver.

References

[1] R. Hockney and C. Jesshope, (1981), *Parallel Computers: Architecture, Programming and Algorithms,* Adam Hilger, Bristol.

[2] D. Kershaw, (1982), *Solution of single tridiagonal linear systems and vectorization of the* ICCG *algorithm on the* CRAY-1, in *Parallel Computations*, (1982), G. Rodrigue, ed., Academic Press, New York.

[3] L. Johnsson, (1984), *Odd-even cyclic reduction on ensemble architectures and the solution of tridiagonal systems of equations*, Dept. Computer Science Report YALEU/CSD/RR-339, Yale Univ., New Haven, CT.

[4] J. Lambiotte and R. Voigt, (1975), *The solution of tridiagonal linear systems on the* CDC STAR-100 *computer,* ACM Trans., Math Software, 1, pp. 308-329.

[5] J. Ortega and R. Voigt, (1985), *Solution of Partial Differential Equations on Vector and Parallel Computers,* SIAM Rev. 27, pp.149-240.

[6] K. Vandiver, (1987), *An occam implementation of a divide and conquer cyclic reduction algorithm,* Master's project report, Department of Mathematical Sciences, Clemson University.

[7] FPS T Series Users Guide, Floating Point Systems, Inc., Sept. 1987.

[8] O. Edwards, (1986), *Finite element stiffness calculation on supercomputers,* Master's Thesis, Department of Mathematics, Carnegie-Mellon University.

[9] D. Warner, *Hypercube Communications Algorithms,* presented to the Workshop on Parallel Computing, Clemson University, Clemson, SC, Nov. 18-19, 1987.

[10] L. Johnsson and C.-T. Ho. *Spanning Graphs for Optimum Broadcasting and Personalized Communication in Hypercubes,* Dept. Comp. Sci. Tech. Rep. 500, Yale Univ., Nov., 1986.

[11] FPS T Series Math Library Manual (Release B), Floating Point Systems, Inc., February, 1987.

n:	31	63	127	255	511	1023	2047	4095	8191
times (μsecs):									
1. vector recip.:	1704	2126	2547	2984	3775	5329	8378	14419	26455
2. vector neg.:	415	519	622	729	865	1061	1386	1956	3023
3. vector *,+,-:	528	659	792	923	1092	1332	1716	2387	3637
4. total vector arithmetic op. time:	10565	13194	15839	18479	22106	27698	37221	54574	87673
5. moves (unrolled):	3968	6464	11456	20608	39168	75648	148928	293888	584512
6. moves (without unrolling):	4416	7104	13248	25344	48704	95552	188800	375808	747904
7. reduction:	11520	15552	21184	29952	45760	75648	133696	248000	475008
8. backsolve:	3456	4736	6720	9856	15680	26752	48448	91136	175936

Table 4.1 Timings for single processor cyclic reduction

p	k	n_p	n	red&back	comm	solve	total	speedup	efficiency
1	11	8191	8191	653760	--	--	653760	--	--
2	10	4607	8703	374080	768	17216	392064	1.67	83%
4	9	2303	8447	197120	2048	24384	223552	2.92	73%
16	7	575	8255	61056	16000	54976	132032	4.95	31%

Table 4.2 Results comparing efficiency and speedup for q=8

					q=3			(relative)	
p	k	n_p	n	red&back	comm	solve	total	speedup	efficiency
4	12	8191	26623	647296	1728	11456	660480	--	--
8	11	4095	25599	335360	5568	16256	357184	1.85	92%
16	10	2047	25087	178176	15808	22592	216576	3.05	76%
					q=6			(relative)	
p	k	n_p	n	red&back	comm	solve	total	speedup	efficiency
4	11	7167	25599	567744	1920	16256	585920	--	--
8	10	3583	25087	294400	5696	22656	322752	1.82	91%
16	9	1791	24831	156672	15936	32320	204928	2.86	71%
					q=15			(relative)	
p	k	n_p	n	red&back	comm	solve	total	speedup	efficiency
2	10	8191	15871	642240	1024	15808	659072	--	--
4	9	4095	15615	329472	2496	21568	353536	1.86	93%
8	8	2047	15487	171968	6528	29952	208448	3.16	79%
16	7	1023	15423	92032	16768	43520	152320	4.33	54%

Table 4.3 Results comparing efficiency and speedup for various values of q and n

						n=6655			
p	q	k	n_p	red&back	comm	solve	total	speedup	efficiency
1	12	10	6655	528832	--	11520	540352	--	--
2	6	10	3583	295808	640	11520	307968	1.75	88%
4	3	10	2047	179072	1728	11520	192320	2.8	70%
						n=12799		(relative)	
p	q	k	n_p	red&back	comm	solve	total	speedup	efficiency
2	12	10	6655	528000	896	16448	545344	--	--
4	6	10	3583	295552	1920	16448	313920	1.74	87%
8	3	10	2047	179136	5568	16448	201152	2.71	68%
						n=24831		(relative)	
p	q	k	n_p	red&back	comm	solve	total	speedup	efficiency
4	24	9	6399	504192	3072	32512	539776	--	--
8	12	9	3327	272768	6208	32512	311488	1.73	87%
16	6	9	1791	157056	15936	32512	205504	2.63	66%
						n=50175		(relative)	
p	q	k	n_p	red&back	comm	solve	total	speedup	efficiency
8	6	11	7167	569728	5632	22784	598144	--	--
16	3	11	4095	336704	15680	22784	375168	1.6	80%

Table 4.4 Results comparing efficiency and speedup for identical n values.