Fast Banded Linear Equation Solvers

Russell K. Standish
ANU Supercomputing Facility
The Australian National University
GPO Box 4, Canberra, 2601, Australia

December 8, 1995

Abstract

A new algorithm for solving triangular linear equations is reported that is significantly faster than conventional methods on vector and parallel architectures for small bandwidths. In essence, it is a block version of the cyclic reduction algorithm used for tridiagonal systems of equations, and is applicable to general banded systems as well.

1 Introduction

A number of linear algebra packages, such as LAPACK[1] rely on the solution of a triangular system. Indeed, one of the 2nd level BLAS[2] routines (xTRBSV) provides this.

The usual way of implementing the solution is a form of Gaussian elimination, and can be expressed in the following way (for a lower unit triangular system Lx=r).

Algorithm 1
loop over i
    $x_i \leftarrow r_i$
    loop over $j < i$
        $x_i \leftarrow x_i - L_{ij} x_j$

This algorithm is efficient on vector processors only when the loop over j is large or alternatively, when the matrix has a large bandwidth. In the case of small to medium bandwidths, vector lengths are small, leading to inefficient vector execution. The outer loop cannot be vectorised because of the recursive data dependence on x. Algorithm 1 executes most efficiently as a scalar program, i.e. is of order nb for small bandwidth matrices.

In the limiting case of the bandwidth equal to 2, an alternative algorithm exists that executes in logarithmic time on parallel systems. It is the cyclic reduction method for tridiagonal systems [3], with one of the diagonals set to zero. Clearly, there is a need for algorithms that bridge the efficiency gap between tridiagonal cyclic reduction, and algorithm 1 for dense matrices.

2 Block Cyclic Reduction

A lower triangular system of equations with small bandwidth b may be considered as a bidiagonal system with $b \times b$ matrices for components. Without loss of generality, we will assume that the system is scaled such that all diagonal elements are unity.

$$\begin{pmatrix} T_1 & \cdot & \cdot & \cdots & \cdot & \cdot \\ A_2 & T_2 & \cdot & \cdots & \cdot & \cdot \\ \cdot & A_3 & T_3 & \cdots & \cdot & \cdot \\ \cdot & \cdot & \ddots & \ddots & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdots & A_n & T_n \end{pmatrix} \mathbf{X} = \mathbf{R} \tag{1}$$

where T_i are unit lower triangular matrices of order b and the A_i are diagonal free upper triangular matrices of the same order.

The basic block cyclic reduction algorithm involves taking pairs of equations, and eliminating the common variable. One is then left with a lower triangular system with half the number of equations. By repeating this process, one finally ends up with a single equation, or a decoupled pair of equations, which may then be solved. All other variables may then be obtained by backward substitution.

Consider a pair of $b \times b$ matrix equations:

$$A_{i-1}X_{i-2} + T_{i-1}X_{i-1} = R_{i-1} \tag{2}$$

$$A_iX_{i-1} + T_iX_i = R_i. \tag{3}$$

By multiplying (2) by $A_iT_{i-1}^{-1}$, and subtract it from (3), the resulting equation is:

$$-A_iT_{i-1}^{-1}A_{i-1}X_{i-2} + T_iX_i = R_i - A_iT_{i-1}^{-1}R_{i-1}. \tag{4}$$

The coefficients of the tridiagonal system at level l of the cyclic reduction may be found recursively from:

$$A_i^{(l)} = -A_i^{(l-1)}T_{i-1}^{-1}A_{i-1}^{(l-1)} \tag{5}$$

and

$$R_i^{(l)} = R_i^{(l-1)} - A_i^{(l-1)}T_{i-1}^{-1}R_{i-1}^{(l-1)}, \tag{6}$$

where $l = 1, \ldots \leq \log_2 n$ and $i = 2^l, 2\cdot 2^l, 3\cdot 2^l, \ldots \leq n$. Note that the coefficients $A_i^{(l)}$ and T_i are $b \times b$ matrices and $R_i^{(l)}$ is a b vector, where b is the bandwidth of

the original system. The coefficients T_i are independent of the level of reduction, a distinction that is not true in the case of tridiagonal systems. The coefficients $A_i^{(l)}$ start as upper triangular matrices with zero diagonal. For $l \geq 1$, these are "filled in", so that only the first column and last row are zero.

The solution vector X_i is computed by backward substitution once the equations have decoupled:

$$X_i = T_i^{-1}(R_i^{(l)} - A_i^{(l)} X_{i-2^l}), \tag{7}$$

with l and i spanning the previously given range, and $X_0 = 0$ by convention.

The algorithm as described is the serial cyclic reduction method, which is most appropriate for a vector or moderately parallel system (no. of processors $\lesssim$100). A variant of this algorithm called parallel cyclic redution [3] may well be more appropriate for massively parallel systems.

3 Implementation and Results

The initial implementation of this algorithm was for arbitrary bandwidth b. Since the hardware was a Fujitsu VP100, with an autovectorising compiler, the inner loop had always to be over the index i, with as many as three loops encasing the inner loop to implement the matrix multiplication, and algorithm 1 to multiply by T_i^{-1}. The resultant code when fully vectorised always ran slower than algorithm 1 due to the loop overhead. The algorithm was recast into a macro language [4] that allowed the inner loops to be expanded inline. This produced separate Fortran source codes for each of the cases $b = 2, 3, \ldots 10$.

A further optimisation was performed by employing a cutoff condition. For well conditioned matrices, $A_i^{(l)}$ diminishes exponentially with l [3], and so at some point equation (7) becomes trivial. This is monitored dynamically, and when the term $\left\| A_i^{(l-1)} T_{i-1}^{-1} A_{i-1}^{(l)} \right\|$ was less than machine accuracy for every i, the loop over l was exited, and backward substitution began. This leads to significant improvements in execution speed for $n \gtrsim h1000$. Since the reduction operation of taking the and of all elements in a vector is not available as a machine instruction, the conventional way of implementing the test is quite inefficient, i.e.

```
jump ⟵ true
loop over i
    jump ← jump and condition(i)
```

does not vectorise, and tended to dominate the code execution. However, it was found that a logical mask operation would vectorise, and so this method was chosen:

```
jump ← true
```

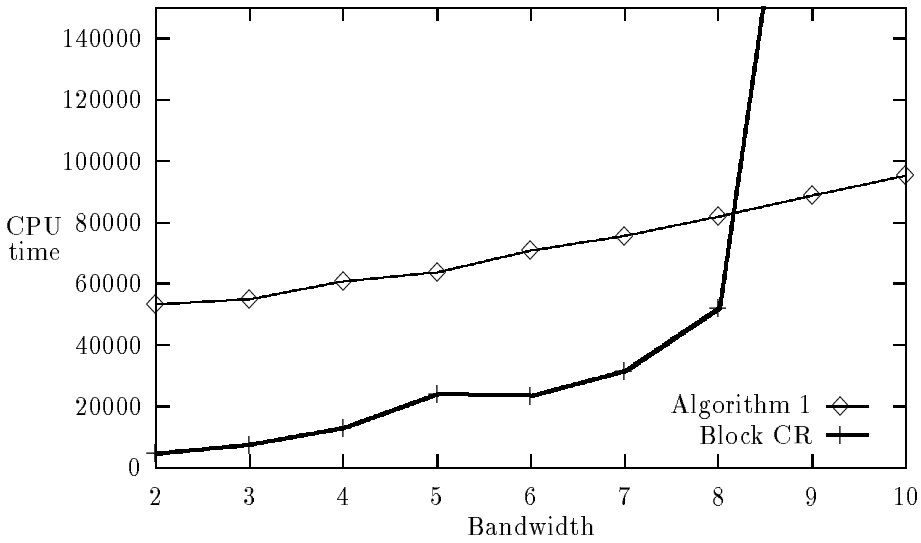

Figure 1: Execution time on VP100

Bandwidth	2	3	4	5	6	7	8	9	10
Speedup	11.8	7.2	4.7	2.7	3.0	2.4	1.6	0.34	0.15

Table 1: Speedup of Block Cyclic Reduction over algorithm 1

loop over i
    if not condition(i) then jump $\longleftarrow$ *false.*

Figure 1 shows the execution times for algorithm 1 and block cyclic reduction as a function of bandwidth b, for a system of 25 200 equations, where all off diagonal are set to 0.5. The speedup seen on the VP100 is reported in table 1. As can be seen, block cyclic reduction gives a significant performance improvement over the traditional algorithm 1 when the bandwidth is modest. The exact value of b at which algorithm 1 performs better will depend on the computer architecture, but is probably not too different to the results given here in general, as the complexity of the block cyclic reduction algorithm rises steeply with bandwidth. A "phase diagram" obtained empirically on the VP100 is shown in Figure 2, the marked areas corresponding to the regions in parameter space where that method performs best.

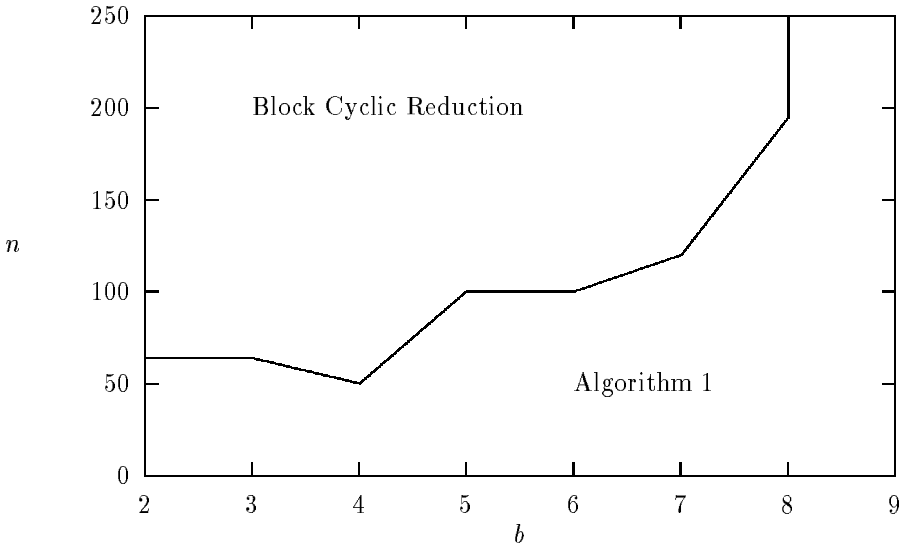

Figure 2: Phase diagram

4 Conclusion

In this paper a new algorithm for solving banded triangular systems is presented along with results obtained from a Fujitsu VP100 supercomputer. The algorithm shows significant performance gains on vector (and parallel) computers for modest bandwidths. One could easily extend the algorithm to general banded matrices, and it would be interesting to know if this gives better results than LAPACK coupled with the block CR algorithm given in this paper.

This algorithm isn't easily generalized to the case of sparse matrices because of the way that the offdiagonal blocks (A_i) "fill in". Even if memory was not a limitation, the overhead of manipulating dense block matrices dominate the total execution time.

References

[1] C. H. Bischoff and J. J. Dongarra. A linear algebra library for high performance computers. In G. F. Carey, editor, *Parallel Supercomputing: Methods, Algorithms and Applications*, pages 45–56. Wiley, New York, 1989.

[2] J. J. Dongarra, J. du Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Software*, 14(1):1–17, 1988.

[3] R. W. Hockney and C. R. Jesshope. *Parallel Computers*. Adam Hilger, Bristol, 1983.

[4] C. J. Ribbens, J. R. Rice, and W. A. Ward. A simple macro processor. *ACM Trans. Math. Software*, 10(4):410, December 1984.