Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J.J.F.M. Schlichting, H.A. van der Vorst

Solving bidiagonal systems of linear equations
on the CDC CYBER 205

Department of Numerical Mathematics Report NM-R8725 November

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Solving Bidiagonal Systems of Linear Equations on the CDC CYBER 205

J.J.F.M. Schlichting
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
Control Data BV, P.O. Box 111, 2285 VL Rijswijk, The Netherlands

H.A. van der Vorst
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
Technical University Delft, Faculty of Mathematics and Informatics,
Julianalaan 132, 2628 BL Delft, The Netherlands

This paper examines the efficiency of some different techniques for the solution of bidiagonal systems of linear equations on a CDC Cyber 205. Special attention is paid to exploiting the capabilities of the scalar processor and the estimation of execution times. Three categories of algorithms for the solution of bidiagonal systems of linear equations are described, The first category consists of straightforward scalar algorithms written in standard FORTRAN, optimized by means of commonly known techniques like loop unrolling. The second category consists of vector algorithms on recursive doubling, cyclic reduction and a partitioning technique. The third category consists of scalar codes written in assenbly code designed to fully exploit the parallelism in the scalar processor; a method is developed to predict the execution time of optimal code for recursive problems. The predicted and measured performances of the routines described are compared and analysed.

1980 Mathematics Subject Classification (1985 Revision): 65F05, 65V05, 69C12.
1982 CR Categories: 5.14, 4.6.
Key Words & Phrases: bidiagonal linear system, Cyber 205, scalar optimization, cyclic reduction, recursive doubling, vectorization.
Note: The first author designed, implemented and tested the subroutines described in this article. The second author provided the basic algorithms and the literature references.

I. Introduction

In this report we study the efficient implementation on CDC Cyber 205 computers of the solution of a normalized lower bidiagonal system of linear equations. Such a system can be represented as

$$Ax = \begin{bmatrix} 1 & & & \\ a_1 & 1 & & O \\ & a_2 & 1 & \\ & & \ddots & \ddots \\ O & & & a_{n-1} \quad 1 \end{bmatrix} \begin{bmatrix} x_0 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ b_{n-1} \end{bmatrix} = b \tag{1}$$

or $Ax = b$, for short.

The solution of (1) can be expressed by the recurrence relations

$$x_0 = b_0 \quad x_i = b_i - a_i x_{i-1}, \quad i = 1,2, \cdots, n-1. \tag{2}$$

It is well known, that recurrence relations normally prevent vectorization on vector computers and therefore many algorithms and strategies have been proposed to replace (2) by better vectorizable alternatives. Specially algorithms based on cyclic reduction [5] and the partitioning algorithm [6, 9]

offer good alternatives. For a discussion on the partitioning algorithm for bidiagonal systems on vector computers see [10].

In the sequel we will discuss some of those algorithms together with some new ideas, in an attempt to determine the fastest algorithm for large systems on a Control Data Cyber 205 computer with one, two or four vector pipes.

In Section II we discuss a number of implementations of a straightforward algorithm in standard FORTRAN for which we compare hand optimization with compiler optimization.

In Section III we briefly present some relevant features of the vector processor of the Cyber 205 and we describe a few vector algorithms based on recursive doubling [4] and cyclic reduction [5].

In Section IV we briefly discuss the scalar processor of the Cyber 205, and analyse the execution time of sequences of scalar instructions. A number of scalar algorithms for solving bidiagonal systems is discussed.

In Section V we present estimated and actual performances of the algorithms described in Sections II, III and IV. We discuss the effects of memory bank conflicts.

In Section VI we develop a simple algorithm for the *optimal ratio of reduction* in scalar code. We derive estimates for the performance of some algorithms on half precision data.

In Section VII a summary and some conclusions are given.

Performance is expressed throughout in Cyber 205 machine cycles of 20 nanoseconds per unknown solved.

II. FORTRAN SUBROUTINES

A number of scalar routines in FORTRAN have been tested, using the CDC FORTRAN compiler FTN200.

FTN200 offers several optimization options, which may be freely combined. For the scalar routines in this section the following options are relevant:

S Instruction Scheduling
D DO-loop optimization and
R Remove redundant code

whenever S or D are used, the specification of R gives better results. We have tested the following combinations:

$$OPT = 0, \quad OPT = RS \quad \text{and} \quad OPT = DRS.$$

The FORTRAN compiler used is FTN200 cycle 678A.

1. Straightforward Fortran code:

```
C     ALGORITHM  A1
      SUBROUTINE  BIDIAG (X,A,B,N)
      DIMENSION  X(0:N-1), A(1:N-1), B(0:N-1)
      X(0) = B(0)
      DO   10  J=1,N-1
 10   X(J) = B(J)-A(J)*X(J-1)
      RETURN
      END
```

Execution time for N=25600:

51 cycles per equation when OPT=0
35 cycles per equation when OPT=DRS

Unrolling the do-loop once, leads to the code:

```
C     ALGORITHM  A2
      SUBROUTINE  BIDIAG (X,A,B,N)
      DIMENSION   X(0:N-1), A(1:N-1), B(0:N-1)
      X(0) = B(0)
      DO  10  J=1,N-2,2
      X(J) = B(J)-A(J)*X(J-1)
 10   X(J+1) = B(J+1)-A(J+1)*X(J)
      IF (J.LT.N-1) X(N-1) = B(N-1)-A(N-1)*X(N-2)
      RETURN
      END
```

Execution time for N=25600:

48.5 cycles per equation when OPT=0
34.5 cycles per equation when OPT=DRS

2. Separate Store and Load Ahead

Each element of the result vectors is stored and then loaded again, thus requiring 2 memory accesses rather than one. Also the elements of the input vectors must be loaded before being processed. Since in the Cyber 205 load instructions, are slow compared with arithmetic instructions load ahead will improve performance. These considerations lead to the code:

```
C     ALGORITHM  B1
      SUBROUTINE  BIDIAG (X,A,B,N)
      DIMENSION   X(0:N-1), A(1:N-1), B(0:N-1)
      XC = B(0)
      X(0) = XC
      IF (N.EQ.0) RETURN
      AC = A(1)
      BC = B(1)
      DO  10  J=1,N-2
      XC = BC-AC * XC
      AC = A(J+1)
      BC = B(J+1)
 10   X(J) = XC
      X(N-1) = BC-AC * XC
      RETURN
      END
```

Execution time for N=25600:

24 cycles per equation when OPT=0

18 cycles per equation when OPT=DRS

We unroll the DO-loop once, using different local variables for the odd and even elements of X,A and B.

```
C     ALGORITHM  B2
      SUBROUTINE BIDIAG(X,A,B,N)
      DIMENSION  X(0:N-1), A(1:N-1), B(0:N-1)
      X2 = B(0)
      X(0) = X2
      IF (N.EQ.0) RETURN
      B1 = B(1)
      A1 = A(1)
      IF (N.EQ.1) GOTO 11
      B2 = B(2)
      A2 = A(2)
      DO  10  J=1,N-4,2
      X1 = B1-A1*X2
      X2 = B2-A2*X1
      A1 = A(J+2)
      B1 = B(J+2)
      A2 = A(J+3)
      B2 = B(J+3)
      X(J) = X1
      X(J+1) = X2
10    CONTINUE
      X1 = B1-A1*X2
      X2 = B2-A2*X1
      X(J)   = X1
      X(J+1) = X2
      IF (J.EQ.N-2) RETURN
11    X(N-1) = B(N-1)*A(N-1)*X2
      RETURN
      END
```

Execution times for N = 25600

17 cycles per equation when OPT=0
13.5 cycles per equation when OPT=RS or OPT=DRS

We unroll further to solve 4 equations in the body of the DO-loop. Different forms are possible, but the best results were obtained with a DO-loop of the form:

```
      DO  10  J=1,N-8,4
      XD = B1-A1*XC
      A1 = A(J+4)
      B1 = B(J+4)
      XE = B2-A2*XD
      A2 = A(J+5)
```

```
      B2 = B(J+5)
      XF = B3-A3*XE
      A3 = A(J+6)
      B3 = B(J+6)
      XC = B4-A4*XF
      A4 = A(J+7)
      B4 = B(J+7)
      X(J) = XD
      X(J+1) = XE
      X(J+2) =XF
10    X(J+3) = XC
```

Execution time FOR N=25600:

14.25	cycles per equation when	OPT=0
13.02	cycles per equation when	OPT=RS or OPT=DRS

With other forms we obtained the same performance for OPT=RS, and a lower performance in all cases for OPT=0 and in most cases for OPT=DRS.

Unrolling further to solve 8 equations in each iteration increases the performance slightly to:

13.5	cycles per equation when	OPT=0
12.28	cycles per equation when	OPT=RS
12.66	cycles per equation when	OPT=DRS

3. Deferred Store and Load Ahead

The placement of the store of the results in the result vector plays an important role; storing the result of the preceding rather than the current iteration yields the code:

```
C     ALGORITHM  C1
      SUBROUTINE  BIDIAG (X,A,B,N)
      DIMENSION X(O:N-1), A(1:N-1), B(O:N-1)
      XC = B(0)
      X(0) = XC
      IF (N.EQ.1) RETURN
      AC = A(1)
      BC = B(1)
      IF (N.EQ.1) GOTO 11
      DO  10  J=1,N-2
      X(J-1) = XC
      XC = BC-AC*XC
      AC = A(J+1)
10    BC = B(J+1)
      X(J-1) = XC
11    X(N-1) = BC-AC*XC
      RETURN
      END
```

Execution times for N=25600:

17	cycles per equation when	OPT=0
18	cycles per equation when	OPT=RS

We unroll once, placing the store into the result vector after the use of the result for the computation of the next result:

```
C     ALGORITHM  C2
      SUBROUTINE  BIDIAG (X,A,B,N)
      DIMENSION  X(0:N-1), A(1:N-1), B(0;N-1)
      XC = B(0)
      X(0) = XC
      IF (N.EQ.0) RETURN
      AD = A(1)
      BD = B(1)
      J = -1
      IF (N.LE.2) GOTO 11
      AC = A(2)
      BC = B(2)
      XD = BD-AD*XC
      AD = A(3)
      BD = B(3)
      DO  10  J=1,N-5,2
      X(J-1) = XC
      XC = BC-AC*CD
      AC = A(J+3)
      BC = B(J+3)
      X(J) = XD
      XD = BD-AD*XC
      AD = A(J+4)
10    BD =B(J+4)
      X(J-1) = XC
      XC = BC-AC*XD
      X(J) = XD
11    XD = BD-AD*XC
      X(J+1) = XC
      X(J+2) = XD
      IF(J.NE.N-3)  X(N-1) = B(N-1)-A(N-1)*XD
      RETURN
      END
```

The execution time now decreases to:

13.02	cycles per equation when	OPT=0
11.54	cycles per equation when	OPT=PRS

We unrolled the loop once more to produce 4 results in each pass through the loop. The best results were obtained when we increased the load ahead factor to 3, but even then the optimized code executed slower than the preceding algorithm.

We also did an experiment with a DO-loop in which 8 results are computed in each iteration. Except for a load ahead factor of 4 this leads to the code of algorithm C2 unrolled 3 times.

The execution times are

11.54	cycles per equation when	OPT=0
12.03	cycles per equation when	OPT=RS
12.28	cycles per equation when	OPT=DRS

In an attempt to further improve the code we broke the computation of the last but one result, computed in the loop, into two statements in order to define more precisely the desired sequence of execution for the individual instructions. This modification is expected to improve the performance in non-optimized mode. For 4 results per iteration this led to a routine of which we show the do-loop only:

```
C     ALGORITHM  C4
      DO  10  J=1, N-8,4
      X1 = B1-A1*X4
      X(J-1) = X3
      A4 = A(J+4)
      B4 = B(J+4)
      X2 = B2-A2*X1
      X(J) = X4
      A1 = A(J+5)
      B1 = B(J+5)
      AX3 = A3*X2
      A(J+1) = X1
      A2 = A(J+6)
      B2 = B(J+6)
      X3 = B3-AX3
      X(J+2) = X2
      A3 = A(J+7)
      B3 = B(J+7)
      X4 = B4-A4*X3
10    CONTINUE
```

Execution time for N=25600:

10.78	cycles per equation when	OPT=0
12.52	cycles per equation when	OPT=RS
12.02	cycles per equation when	OPT=DRS

If we apply the same modification to the code solving 8 equations in the body of the do-loop, we obtain the following results:

10.53	cycles per equation when	OPT=0
12.77	cycles per equation when	OPT=RS
19.91	cycles per equation when	OPT=DRS

4. SEPARATE STORE WITH UNROLLING

We have seen so far that with load ahead and heavy unrolling the benefits of optimization are quite disappointing. Now we investigate unrolling without load ahead in optimized mode. We use separate stores as in algorithm B and we show only a version in which 8 results are computed in each pass through the loop.

The DO-loop is shown below:

```
      DO   10  J=1, N-8,8
      X1 = D(J)-A(J)*X8
      X(J) = X1
      X2 = B(J+1)-A(J+1)*X1
      X(J+1) = X2
       :
       :
      X8 = B(J+7)-A(J+7)*X7
10    X(J+7) = X8
```

Execution times for N=25600:

42.16	cycles per equation with	OPT=0
15.02	cycles per equation with	OPT=RS
12.52	cycles per equation with	OPT=DRS

5. DEFERRED STORE WITH UNROLLING

Finally we extend the previous experiment with deferred stores, still computing 8 results in the body of the DO-loop which has the form:

```
      DO   10  J=1, N-8,8
      X1 = D(J)-A(J)*X8
      X(J-1) = X8
      X2 = B(J+1)-A(J+1)*X1
      X(J) = X1
       :
       :
      X7 = B(J+6)-A(J+6)*X6
      X(J+5) = X6
10    X8 = B(J+7)-A(J+7)*X7
```

Execution times for N=25600:

35.15	cycles per equation when	OPT=0
11.89	cycles per equation when	OPT=DRS

Below we give a summary of the results.

In table I *U* indicates the number of equations solved in one pass through the DO-loop, OPT=0 indicates no optimization and OPT stands for OPT=RS or OPT=DRS, whichever produced the best results.

ALGORITHM	$U=1$ OPT=0	$U=1$ OPT	$U=2$ OPT=0	$U=2$ OPT	$U=4$ OPT=0	$U=4$ OPT	$U=8$ OPT=0	$U=8$ OPT
A	51.07	35.01	48.65	34.53	-	-	-	-
B	24.01	18.02	17.02	13.52	14.28	13.02	16.16	12.28
C	17.02	18.03	13.02	11.54	10.78	12.03	10.53	12.03
D	-	-	-	-	-	-	42.16	12.52
E	-	-	-	-	-	-	35.15	12.03

TABLE I

We conclude that Algorithm C gives the best results. When we unroll to 4 or more results per iteration, *hand-optimization* as in C4, may further improve the performance: this has been done for $U=4$, and for $U=8$.

If we want to use only unrolling and standard compiler optimization, it pays to apply deferred store (Algorithm E with $U=8$)

III VECTORIZED ALGORITHMS

We use both subarray references and vector syntax as they are defined in [2].

A subarray reference of the form: $A(I:J:K)$ denotes the vector consisting of the elements $A(I)$, $A(I+K), \cdots, A(I+NK)$, where N is defined by:

$$I+NK \leqslant J < I+(N+1)K.$$

With vector syntax $A(I;K)$, denotes the vector containing the K elements $A(I), A(I+1), \cdots, A(I+K-1)$.

III.1 Cyber 205 Vector Processor

The vectors processed by the Cyber 205 Vector Processor reside in memory, occupying a block of consecutive locations. Three types of vectors are distinguished:

- Data-vectors, also called vectors, of which the elements are full precision (64 bit) or half-precision (32 bit) floating point numbers.
- Bit vectors of which the elements are one bit quantities.
- Index vectors of which the elements occupy 64 bits and contain integer values, to be used as indices.

In most vector instructions a data-vector operand may be replaced by a broadcast value (i.e. a single value).

A Cyber 205 processor may be equipped with one, two our four vector pipes. Each pipe is capable of producing one full precision result or two half precision results per cycle of 20 nanoseconds.

We denote the number of vector pipes by p.

The execution time of a vector instruction may be written as $S+EN$, where

S is the vector start-up time
E is the time per element processed and
N represents the number of elements processed.

For most instructions $S=51$ and $E=1/p$.

Normal vector instructions operate on one or two data vectors and produce a data vector as result; all vector pipes are used simultaneously giving a peak rate of

50p megaflops for full precision operands or

100p megaflops for half precision operands.

Control store capability allows the specification of a bit vector operand; only those elements for which the corresponding bit-vector element has a specified value (1 or 0) are stored in the result vector.

A special *link instruction* allows two normal vector instructions of which the second uses the result of the first as an operand, to be executed simultaneously, provided the two instructions have at most two non-broadcast operand vectors. As an example the operation

$$C(1{:}n) = A(1{:}n) + S^{*}B(1{:}n) \quad \text{where } S \text{ is a scalar value,}$$

may be linked in a so-called linked triad with a peak rate of 2p results per cycle (or 100 p megaflop for full precision operands).

The COMPRESS instruction forms an output vector C from those elements of an input vector A, for which the corresponding elements of a bit vector Z have a specified value (0 or 1). We will often use a bit vector with alternating values zero and one, called the alternating bit vector. The compression of C into A under control of the alternating bit vector can be written in FORTRAN as

$$C(1{:}N/2) = A(1{:}N{:}2)$$

where N is the length of vector A.

The COMPRESS instruction requires $1/p$ cycles per element of the input vector A.

The MERGE instruction forms an output vector C from the input vectors A and B under control of bit vector Z; each element of C is replaced by the next element of A or B depending on the value of the corresponding element of bit vector Z.

The merging of vectors A and B into vector C under control of the alternating bitvector may be written in Fortran by:

$$C(1{:}N{:}2) = A(1{:}N/2)$$

$$C(2{:}N{:}2) = B(1{:}N/2).$$

The MERGE instruction requires $S + N/p$ cycles, where N is the length of vectors C and Z.

The MASK instruction forms an output vector C from the input vectors A and B under control of a bit vector Z; each element of C is replaced by the corresponding element of A or B depending on the value of the corresponding element of Z. This operation with the alternating bit vector is described in FORTRAN by

$$C(1{:}N{:}2) = A(1{:}N{:}2)$$

$$C(2{:}N{:}2) = B(2{:}N{:}2).$$

The MASK instruction requires $S + N/p$ cycles where N is the length of vectors C and Z.

Sparse vector instructions operate on so-called sparse vectors, that are represented by a data vector and a bit vector. The represented virtual vector consists of the elements of the data vector in the positions where the bit vector has a one-valued element and a default value (0.0 or 1.0) in the positions where the bit vector has a zero; a data vector can thus be viewed as the result of compressing the virtual vector under control of the bit vector. A sparse vector instruction specifies a floating point operation, a logical operation, two sparse vector operands (data vectors A and B with associated bit vectors) and a sparse vector result (data vector C with associated bit vector Z).

The operation of a sparse vector instruction may be described as consisting of the following steps.

1. Set the broadcast value BC to the unit value for the floating point operation specified, i.e. 0.0 for an additive and 1.0 for a multiplicative operation.
2. Form the virtual operand vectors AX and BY by merging data vector A, respectively B with broad cast value BC under control of bit vector X resp. Y.
3. Apply the specified logical operation (AND, OR, Exclusive OR or Implication) to bit vectors X

and Y yielding bit vector Z.

4. Apply the specified floating point operation to virtual vectors AX and BY yielding virtual result vector CZ.
5. Compress virtual vector CZ under control of bit vector Z to produce result vector C.

The execution time of a sparse vector instruction is the time for step 4: $S+EN$, where $S=51$, $E=1/p$ and N is the length of bit vector X or Y, whichever is larger.

We will use sparse vector instructions only with logical operation AND; bit vector X will contain only one- valued elements and bit vector Y will be the alternating bit vector. The sparse vector operation for multiplication may then be described in FORTRAN by

$$C(1{:}N/2) = A(1{:}N{:}2)*B(1{:}N/2).$$

The GATHER instruction forms an output vector C from the elements of an input vector A indexed by the elements of index vector B: each element of C is replaced by the element of A indexed by the corresponding element of B.

In FORTRAN:

```
      DO  10  I=1,N
10    C(I) = A(B(I))
```

Instead of the index vector B one may alternatively specify the distance D between consecutively referenced elements of A.

In FORTRAN:

```
      DO  10  I=1,N
10    C(I) = A(1+(I-1)*D)
```

Similarly the SCATTER instruction stores each element of A into the element of C, indexed by the corresponding element of B.

In FORTRAN:

```
      DO  10  I=1,N
10    C(B(I)) = A(I)
```

Again, a fixed distance D may be specified in lieu of index vector B, giving in FORTRAN

```
      DO  10  I=1,N
10    C(1+(I-1)*D) = A(I)
```

The time required for GATHER and SCATTER instructions is $S+GN$, where G usually varies between 1.25 and 1.75, depending on B or D. If D is a multiple of 64, G will be as large as 4.0.

2. Recursive Doubling

Algorithm F.

Recursive doubling transforms the bidiagonal system of equations (2) into a new system of equations of the form

$$\begin{aligned} x_i &= b_i' \quad (i=0,1) \\ a_i' x_{i-2} + x_i &= b_i' \quad (i=2,3,\cdots,N-1) \end{aligned} \tag{4}$$

By substitution of the i-th equation from (2) in the $(i-1)$-th equation we obtain:

$$a_i a_{i-1} x_{i-2} + x_i = b_i - a_i b_{i-1}$$

and hence

$$\begin{aligned} b_0' &= b_0 \\ b_i' &= b_i - a_i b_{i-1} \quad (i=1,2,\cdots,N-1) \\ a_i' &= a_i a_{i-1} \quad (i=2,3,\cdots,N-1) \end{aligned} \tag{5}$$

In matrix-vector notation the new system has the form:

$$A'x = b' \quad \text{where}$$

$$A' = \bar{A}A \quad \text{and} \quad b' = \bar{A}b, \quad \text{with} \quad \bar{A} = 2E - A.$$

$$A' = \bar{A}A = \begin{pmatrix} 1 & & & & \\ -a_1 & 1 & & & \\ & -a_2 & 1 & & \\ & & \ddots & \ddots & \\ & & & -a_{n-1} & 1 \end{pmatrix} \begin{pmatrix} 1 & & & & \\ a_1 & 1 & & & \\ & a_2 & 1 & & \\ & & \ddots & \ddots & \\ & & & a_{n-1} & 1 \end{pmatrix} = \begin{pmatrix} 1 & & & & \\ 0 & 1 & & & \\ a_1 a_2 & 0 & 1 & & \\ & \ddots & & \ddots & \ddots \\ & & a_{n-2}a_{n-1} & 0 & 1 \end{pmatrix}$$

and thus

$$\begin{pmatrix} 1 & & & & & \\ 0 & 1 & & & & \\ a_2' & 0 & 1 & & & \\ & a_3' & 0 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & a'_{n-2} & 0 & 1 & \\ & & & a'_{n-1} & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} b_0' \\ b_1' \\ b_2' \\ \cdot \\ \cdot \\ \cdot \\ b'_{n-1} \end{pmatrix}.$$

The new system (4) now consists of two independent systems for the odd and even numbered elements of vector x, respectively.

The number of floating point operations required is 3N giving an execution time of

$$153/N + 3/p \quad \text{cycles per equation.}$$

3.3. Recursive doubling recursively applied

Application of recursive doubling to the transformed system (4) yields a new system of the form:

$$\begin{aligned} x_i &= b_i'' \quad (i=0,1,2,3) \\ a_i'' x_{i-4} + x_i &= b_i'' \quad (i=4,5,\cdots,n-1) \end{aligned} \tag{4}$$

where analogously to (5)

$$
\begin{aligned}
b_i'' &= b_i' && (i=0,1,2,3)\\
b_i'' &= b_i' - a_i' b'_{i-2} && (i=2,3,\cdots,n-1)\\
a_i'' &= -a_i' a'_{i-2} && (i=4,5,\cdots,n-1)\,.
\end{aligned}
\tag{7}
$$

The system (6) now consists of 4 independent systems for x_{4i}, x_{4i+1}, x_{4i+2} and x_{4i+3} $(i=0,1,2,\cdots,(n-1)/4)$, respectively.

The k^{th} step of recursive doubling yields a system of the form

$$
\begin{aligned}
x_i &= b_i^0 && (i=0,1,\cdots,2^k-1)\\
a_i^0 x_{1-2^k} + x_i &= b_i^0 && (i=2^k,2^{k+1},\cdots,n-1).
\end{aligned}
\tag{8}
$$

This system consists of 2^k independent systems with the solution

$$
\begin{aligned}
x_i &= b_i^0 && (i=0,1,\cdots,2^{k-1})\\
x_i &= -a_i^0 x_{1-2^k} + b_i^0 && (i=2^k,2^{k+1},\cdots,n-1)
\end{aligned}
$$

and may thus be computed in groups of 2^k unknowns using vector instructions on vectors of length 2^k.

```
C     ALGORITHM  G
      SUBROUTINE  BIDIAG (X,A,B,N)
      DIMENSION  A(1:N), B(0:N), X(0:N)
      PARAMETER  (K=k)
      X(0) = B(0)
      X(1:N) = B(1:N)-A(1:N)-B(0:N-1)
      L1 = 1
      DO  100  J=1,K
      L2 = L1*2
      A(L2:N) = -A(L2:N) * A(L1:N-L1)
      X(L2:N) = X(L2:N)-A(L2:N)*X(0:N-L2)
100   L2 = L1
      A(L2:N) = -A(L2:N) * A(L1:N-L1)
C     Solve remaining system directly using vectors of length L2
      DO  200  J=L2,N,L2
      L = MIN0 (N,J+L2-1)
200   X(J:L) = X(J:L)-A(J:L)*X(J-L2:L-L2)
      RETURN
```

The time required for the j^{th} step of recursive doubling is

$$T_j = 3(N+1-2^{**}j)/p + 3S \quad \text{cycles.}$$

The time required to solve the system after k steps of recursive doubling is

$$T_s = 2^*2^{k*}((N+1-2^k)/(p^*2^k)+S) \quad \text{cycles.}$$

The total execution time in number of cycles per equation, for large N and small k, may thus be approximated by

$$T_{ek} = (3K+2)/p + S/2^{k-1},$$

where, as before, p denotes the number of vector pipes and S the start-up time of 51 cycles. The number of cycles per equation required for 1 to 8 steps of recursive doubling for a Cyber 205 with 1,2 or 4 vector pipes is tabulated below.

steps of recursive doubling	number of vector pipes		
	1	2	4
1	56	53.5	52.25
2	33.5	29.5	27.5
3	23.75	18.25	15.5
4	20.38	13.38	9.88
5	* **20.19**	12.44	7.44
6	21.60	* **11.60**	6.60
7	23.80	12.30	* **6.55**
8	26.40	13.40	6.80

where for each number of vector pipes the lowest number of cycles is preceded by an asterisk.

3.4. Cyclic reduction

Cyclic reduction is similar to recursive doubling, but the substitution of the equation for x_{i-1} in the one for x_i is performed for even values of i only. Thus where recursive doubling transforms the system into a new system of the same size that consists of two independent systems, cyclic reduction generates only one of those independent systems; after the solution of the reduced system the other equations are solved by substitution of the solution of the reduced system in the original system.

In matrix-vector notation we premultiply both sides of the original system $Ax = b$ by a matrix $\bar{A}$ of the form (n assumed to be odd):

$$\begin{bmatrix} 1 & & & & & & \\ -a_1 & 1 & & & & & \\ 0 & 1 & & & & & \\ & -a_3 & 1 & & & & \\ & & \ddots & & 1 & & \\ & & -a_{n-1} & 0 & 1 & & \end{bmatrix}$$

yielding

$$\begin{bmatrix} 1 & & & & & & \\ a_1 & 1 & & & & & \\ a_2' & 0 & 1 & & & & \\ & a_3 & 1 & & & & \\ & a_4' & 1 & & & & \\ & & \ddots & \ddots & & & \\ & & a_n' & 1 & & & \\ & & & 0 & 1 & & \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ \cdot \\ \cdot \\ \cdot \\ b'_{n-2} \\ b'_{n-1} \end{bmatrix}$$

We then disregard the even numbered rows and columns yielding

$$\begin{pmatrix} 1 & & & & \\ a_2' & 1 & & & \\ & a_4' & 1 & & \\ & & \ddots & \ddots & \\ & & & a'_{n-1} & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_2 \\ x_4 \\ \\ x'_{n-1} \end{pmatrix} = \begin{pmatrix} b_0' \\ b_2' \\ . \\ . \\ b'_{n-1} \end{pmatrix}$$

for the coefficient and right-hand side elements of the new equation we have, analogously to (5),

$$\begin{aligned} b'_0 &= b_0 \\ b'_{2i} &= b_{2i} - a_{2i} b_{2i-1} \qquad (i = 1,2, \cdots n-1) \\ a'_{2i} &= a_{2i} * a_{2i-1} \qquad (i = 2,3, \cdots n-1). \end{aligned} \tag{9}$$

The solution is given by

$$\begin{aligned} x_0 &= b_0 \\ x_{2i} &= b'_{2i} - a'_{2i} * x_{2i-2} \qquad (1 = 1,2, \cdots n-1) \end{aligned} \tag{10}$$

and back substitution gives

$$x_{2i+1} = b_{2i+1} - a_{2i+1} x_{2i} \qquad (1 = 0,1, \cdots n-1). \tag{11}$$

The number of floating point operations required is then

for cyclic reduction (10) 1.5 N
for back substitution (11) N.

We may thus solve the bidiagonal system $Ax = b$ by the following FORTRAN-like code, in which we omitted declarations and allocation of temporary vectors for reasons of brevity:

```
C     ALGORITHM  H1
      N1 = N-1
      AB(0) = B(0)
C     cyclic reduction : 3 vector instructions
      AB (1:N1:2) = -A(2:N1:2) * A(1:N1:2)
      AB (2:N1:2) = B(2:N1:2)-A(2:N1:2) * B(1:N1:2)

C     solve the reduced system by some (scalar) code equivalent to:
C     X(0) = AB(0)
C     DO  20  I=2,N1,2
C2    X(I) = AB(I)-AB(I-1) * X(I-2)
C     Back substitution  : 2 vector instructions
      X(1:N1:2) = B(1:N1:2)-A(1:N1:2) * X(0:N1-1:2)
```

We use *control store* capability with all 5 vector instructions specifying a bit vector consisting of alternating ones and zeroes. The execution time in cycles, excluding the solution of the reduced system, may then be approximated by: 5N/p.

In this code we used a single array AB to store the coefficients and right-hand side elements of the reduced system, since this allows faster scalar code for the solution of the reduced system, as we will see in the next section (algorithm R). However, in most cases we require the coefficients and right-hand side of the reduced system in contiguous vectors. We compress the odd and even numbered elements of coefficients and right-hand side (arrays A and B) into four separate arrays by means of 4

compress instruction requiring each $S+N/p$ cycles. The cyclic reduction may then be performed by three vector instructions operating on contiguous vectors of length $N/2$ requiring $3S+1.5N/p$ cycles. Back substitution is performed on similarly contiguous vectors of length $N/2$ and thus requires $2S+N/p$ cycles. Since we now have 2 contiguous vectors, one holding the odd and the other the even numbered elements of the result, we require a merge instruction of length N to produce a single result vector: the execution time for this merge is $S+N/p$ cycles. Summing up we obtain an execution time for cyclic reduction and back substitution of approximately $7.5N/p$ cycles.

Sparse vector instructions make it possible to improve this algorithm as follows:

C ALGORITHM H2

1. Compress the even elements of the coefficient vector requiring $S+N/p$ cycles.
2. Perform the cyclic reduction proper in three sparse vector instructions, each requiring $S+N/p$ cycles.
3. After solving the reduced system we expand the result vector of length $N/2$ into the even elements of a vector of length N, using a merge instruction requiring $S+N/p$ cycles.
4. Perform the back substitution by means of two vector instructions with control store capability using the alternate bit vector. Since the length of the vectors used is N this step requires $2S+2N/p$ cycles.

Summing up, approximately $7N/p$ cycles are required.

```
C    ALGORITHM  H2
     N1 = N-1
     N2 = N1/2
C    Compress                                          N/p cycles
     AA(1:N2) = A(2:N:2)
C    3 sparse vector instructions                      3 N/p cycles
     BB(1:N2) = -AA(1:N2) * B(1:N1:2)
     AA(1:N2) = -AA(1:N2) * A(1:N1:2)
     BB(1:N2) =  BB(1:N2) + B(1:N1:2)

C    Solve reduced system  (XX,AA,BB,N2)
C    Expand solution into even elements of X           N/p cycles
     X (2:N1:2) = XX(1:N1)
C    Back substitution with control store capability   N/p cycles
     X(1:N1:2) = -A(1:N1:2) * X(0:N1:2)
     X(1:N1:2) =  B(1:N1:2) + A(  ) * X(1:N1:2)
C    Total                                             N/p cycles
```

In the preceding algorithm we used a COMPRESS instruction to extract the even or odd elements of an array: the MASK instruction provides a faster method, be it at the cost of the ordering of the extracted elements.

We split the coefficient and right-hand side vectors into two halves and use a MASK instruction to combine the odd or even elements of both halves into one vector.

This method may be illustrated by the following piece of FORTRAN code (declarations omitted) in which we assume that N is a multiple of 4.

```
C    ALGORITHM  H3
     N1 = N-1
```

```
      N2 = N1/2
C     4 mask instructions of length N/2
      AA(1:N2:2) = A(1:N2:2)
      AA(2:N2:2) = A(N2+1:N1:2)

      AA(N2+1:N1:2) = A(2:N2:2)
      AA(N2+2:N1:2) = A(N2+2:N1:2)

      BB(1:N2:2) = B(1:N2:2)
      BB(2:N2:2) = B(N2+1:N1:2)

      BB(N2+1:N1:2) = B(2:N2:2)
      BB(N2+2:N1:2) = B(N2+2:N1:2)
C     3 vector instructions length N/2

      BB(N2+1:N1) =  BB(N2+1:N2)-AA(N2+1:N2) * BB(1:N2)
      AA(N2+1:N1) = -AA(N2+1:N2) * A(1:N2)

C     Solve reduced system to XX(N2+1:N1)
      N4 = N2/2
      XP = B(0)
      DO  100  J=2,3
      DO  100  I=1,N4
      XP = BB(J*N4+I)+AA(J*NU+I)*XP
 100  XX(J*N4+I) = XP
      XX(N2) = XX(N1-2)
      XX(N2-1) = B(0)
C     2 Vector instruction length N/2
      XX(1:N2) = BB(1:N2)-A(1:N2) * XX(N2-1:N1-2)
C     2 mask instruction of length N/2
      X(1:N2:2) = XX(1:N2:2)
      X(2:N2:2) = XX(N2+1:N1:2)
      X(N2+1:N1:2) = XX(2:N2:2)
      X(N2+2:N1:2) = XX(N2+2:N1:2)
```

The execution time of cyclic reduction and back substitution is now estimated as follows:

Reorder A into AA	N/p
Reorder B into BB	N/p
Cyclic reduction proper	1.5 N/p
Back substitution proper	N/p
Reorder XX into X	N/p

	5.5 N/p

5. Repeated Cyclic Reduction

Of course we may apply cyclic reduction to the reduced system too. Algorithm H1 generates the coefficients and right-hand side elements into the even elements of vectors of length N, (the size of the original system). Therefore repeated application of algorithm H1 requires 5N/p cycles of execution times for the first as well as for the second application. However, algorithms H2 and H3 use contiguous arrays of length N/2 for the reduced system and thus each application of H2 or H3 requires half the time of the preceding application.

A routine applying algorithm H2 repeatedly is shown below. For brevity we use a pseudo Fortran, featuring dynamic allocation and recursive subroutines, using self explanatory syntax extensions:

```
C     ALGORITHM  J2
      SUBROUTINE BIDIAG (N,A,B,X)
      Parameter  (LIM=50)
      Dimension  A(1:N), B(0:N), X(0:N)
      Dimension  AA(*), BB(*)
      N1 = N-1
      N2 = N/2
      ALLOCATE  (A,1,N2)
      ALLOCATE  (BB,0,N2)
      BB(0) = B(0)
C     Compress the even elements of A in          N/p cycles

      AA(1:N2) = A(2:N:2)
C     multiply, using sparse vector instructions in  N/p cycles
      BB(1:N2) = -AA(1:N2) * B(1:N1:2)
C     multiply, using sparse vector instructions in  N/p cycles
      AA(1:N2) = -AA(1:N2) * A(1:N1:2)
C     Add, using sparse vector instructions in    N/p cycles
      BB(1:N2) =  BB(1:N2) + B(1:N1:2)

      IF (N2 .GT. LIM)  THEN
      CALL BIDIAG (N2,AA,BB,BB)
      ELSE

      DO  20  I = 1,N2
 20   BB(I) = BB(I)-AA(I) * BB(I-1)
      END IF
C     expand, using merge instruction             N/p cycles
C
      X(2:N1:2) = BB(1:N2)
C     use control store capability                N/p cycles
      X(1:N1:2) = -A(1:N1:2) * X(0:N1:2)
      X(1:N1:2) = X(1:N1:2) + B(1:N1:2)
      END
```

The execution time for routine J2 may, for large N, be approximated by

$$7N/p + 7N/2p + \cdots = 14N/p.$$

Repeated application of algorithm H3 leads to a faster but more complex algorithm.

In order to simplify the code we placed the data movements involving mask instructions in a

separate subroutine FOLD, and we also use a recursive subroutine BIDIAGR.

```
      SUBROUTINE  FOLD (S,D,NB,LB)
      DIMENSION   S(LB,2,NB,2), D(LB,2,NB,2)
      DO   10  L=1,2
      DO   10  K=1,NB
      DO   10  J=1,2
      DO   10  I=1,LB
10    D(I,J,K,L) = S(I,L,K,J)
      RETURN
      END
```

In consecutive recursions LB has the values 1,2,4,... etcetera. The three innermost loops are implemented by a single MASK instruction with a bit vector consisting of blocks of length LB: odd numbered blocks consist of ones, even numbered blocks of zeroes.

```
      SUBROUTINE BIDIAG (X,A,B,N)
      DIMENSION  X(0:N-1), A(N-1), B(0:N-1)
      DIMENSION  XX(65535), AA(65535), BB(65535)
      PARAMETER  (LIM=100)
      N4 = 0
      N2 = N-1
      IF(N2 .LT. LIM) GOTO 15
      K = 0
10    CONTINUE
      K = K+1
      N2 = N2/2
      IF(N2 .GE. LIM) GOTO 10
      N4 = (N2/2**K)*4**K
      X(0) = B(0)
      CALL BIDIAGR (X(1),A,B(1),N-1,XX,AA,BB,B(0),1,LIM)
15    CONTINUE
      DO   20  I = N4+1,N-1
20    X(I) = B(I)-A(I) * X(I-1)
      RETURN
      END


      SUBROUTINE BIDIAGR (X,A,B,N,XX,AA,BB,XZ,J,LIM)
      DIMENSION  X(N),A(N),B(N),XX(N),AA(N),BB(N)
      DIMENSION
      LB = 2**(J-1)
      N2 = N/2
      NB = N2/(2*LB)
      CALL  FOLD (A,AA,NB,LB)
      CALL  FOLD (B,BB,NB,LB)
      NH = N2+1
C
      BB(NH;N2) = BB(NH;N2)-AA(NH;N2) * BB(1;N2)
      AA(NH;N2) = -AA(NH;N2) * A(1;N2)
```

```
      IF (N2 .GE. LIM) THEN
C     RECURSIVE CALL
      CALL BIDIAGR (XX(NH),AA(NH),BB(NH),N2,X(1),X(NH),AA(NH),
      $XZ, J+1)
      ELSE
C     DIRECT
      DO  101  L=1,J
 101  IX(I) =1
      DO  102  L=1,J
      DO  102  I=2**(L-1),2**J,2**(L-1)
 102  IX(I) = IX(I)+2**(J-L)
      XP = XZ
      DO  103  L=1,2**J
      DO  103  I=IX(L),N2,2**J
      XP = BB(N2+I)-A(N2+I)*XP
 103  XX(N2+I) = XP
      ENDIF

      LG = 2**J
      DO  110  I=0,J-1
      L = 2**(I-1)
 110  XX(N2-LG+L;L) = XX(N-2-L;L)
      XX(N2-LG) = XZ

      AA(1;N2) = -XX(N2-LG;N2)*AA(NH;2)
      XX(1;N2) = BB(1;N2)+AA(1;N2)

      CALL FOLD (XX,X,NB,LB)
      END
```

The execution time is now estimated by

$$5.5N/p+5.5N/2p+5.5N/4p+\cdots = 11N/p \quad \text{cycles}$$

3.6. Cascaded cyclic reduction

This is basically the partitioning algorithm, see [6, 9]. We subdivide the coefficient matrix in K submatrices of M rows each, assuming $N-1=k*m$ and apply cyclic reduction successively to all the second, the third etc. rows of all submatrices simultaneously. We then have

$$\left.\begin{aligned} a'_{jm+1} &= a_{jm+1} \\ a'_{jm+i} &= -a'_{jm+i}*a'_{jm+i-1} \quad (i=2,\cdots) \end{aligned}\right\} j=1,2,\cdots,k$$

and

$$\left.\begin{aligned} b'_{jm+1} &= b_{jm+1} \\ b'_{jm+i} &= b'_{jm+i}-a'_{jm+i}*b'_{jm+i-1} \end{aligned}\right\} \begin{aligned} &j=1,\cdots,k \\ &(i=1,\cdots m-1,\ j=1,2,\cdots,k). \end{aligned}$$

These definitions are recursive in i, but not in j; the solution is given by

$$\left.\begin{aligned} x_i &= b'_i \quad (i=0,1,\cdots,m-1) \\ x_{jm+i} &= b'_{jm+i}-a'_{jm+i}*x_{jm-m} \end{aligned}\right\} (i=0,1,\cdots,m)\ \ (j=1,2,\cdots,k-1).$$

The reduced system has the form $A'x = b'$

$$\begin{bmatrix} 1 & & & & \\ a'_2 & 1 & & & \\ \vdots & & & & \\ a'_k & & 1 & & \\ & a'_{k+1} & & 1 & \\ & \vdots & & & \\ & a'_{2k} & & & \\ & & a'_{(m-1)k+1} & & \\ & & \vdots & & 1 \\ & & a'_{mk} & & \end{bmatrix} \begin{bmatrix} x_2 \\ \\ x_k \\ x_{k+1} \\ \\ x_{2k} \\ \\ \\ x_{mk} \end{bmatrix} \begin{bmatrix} b'_2 \\ \vdots \\ b'_k \\ b'_{k+1} \\ \vdots \\ b'_{2k} \\ \\ \vdots \\ b'_{mk} \end{bmatrix}$$

The algorithm can be described by the code:

```
C       ALGORITHM  L
        SUBROUTINE BIDIAG
        DIMENSION  A(M,K),B(M,K),X(M,K)

C       CYCLIC REDUCTION PHASE
        X(1:1;K) = B(1;K,1)
        DO  200  I=2,M
        X(I,1:K) = B(I,1;K)-A(I,1;K)*X(I-1,1;K)
 200    A(I,1;K) = -A(I,1;K)*A(I,1,;K)

C       SUBSTITUTION PHASE
        DO  300  J=2,K
 300    X(1,J;M) = X(1,J;M)-A(1,J;M)*X(M,J-1)
        RETURN
        END
```

This algorithm requires $5N$ floating operations and is completely vectorized.

However, the 200-loop has stride M: on the Cyber 205 in each step of the loop 2 gather and 2 scatter operations must be used. The 300-loop consists of $k-1$ linked triads, that must be executed in the order indicated, because, for $j=3,k$, the constant used in the j^{th} linked triad is the last element of the vector computed in the $j-1^{th}$ linked triad.

The execution time is estimated as follows.

200-loop	scatter/gather	4×1.25 N	=	5	N cycles + 4MS
	floating operations	$3\times$ N/p	=	3	N/p cycles + 3MS
300-loop	linked triad		.		N/p cycles + KS

This amounts to a total of $(4/p+5)N+(7M+K)$ cycles where S is the vector start up time (51 cycles). We may split the cyclic reduction phase in three subphases

1. Reorder the arrays A and B by means of 2M gather instructions
2. Perform the reduction proper, computing A and B

3. Reorder the newly computed A and B arrays by means of 2M scatter instructions.

For the Cyber 205 the routine may then be speeded up by carrying out the substitution phase immediately after the reduction proper phase and reordering the resulting X array, rather than both the arrays A and B. The disadvantages are that we must compute $X(M, 1{:}K)$ in another way and that the substitution is no longer a linked triad.

This leads to the following routine, in which we have split some loops in order to facilitate the coming discussions.

```
C       ALGORITHM  L
        SUBROUTINE BIDIAG (X,A,B,XT,AT,M,KR,N)

        DIMENSION  A(M,K),B(M,K),X(M,K)
        DIMENSION  AT(K,M),XT(K,M)
        DO  300  I=1, M-1
 300    AT(2:K,I) = A(2:k,I) * XT(1:k-1,M)
        DO  310  I=1,M-1
 310    XT(2:K,I) = X(2:K,I)-AT(2:K,I)
C       REORDER PHASE
        DO  400  I=1,M
 400    X(I,1:K) = XT(1:K,I)
C       ORDER PHASE
        DO  100  I=1,M
 100    AT(1:K,I) = A(I,1:K)

C       REDUCTION PHASE

        DO  200  I=2,M
 200    XT(1:K,I) = XT(1:K,I)-AT(1:K,I)*XT(1:K,I-1)

        DO  210  I=2,M
 210    AT(1:K,I) = -AT(1:K,I)*AT(1:K,I-1)
C       SOLVE THE REDUCED SYSTEM

        DO  250  J=2,K
 250    XT(J,M) = XT(J,M)-AT(J,M)*XT(J-1,M)

C       SUBSTITUTION PHASE
```

Using the estimate $G = 1.40$, we obtain

$$T = (4.2+5/p)N+90\sqrt{N} \quad (\text{or} \quad 4.2+5/p+90/\sqrt{N} \quad \text{cycles per equation.})$$

For $N = 25600$, have actually observed 9.92 cycles per equation.

In many cases an iterative process requires the solution of a bidiagonal system in each iteration with minor modifications to coefficient matrix and/or right hand side vector.

In such cases a faster algorithm is possible, in which the user reorders the coefficient matrix and right-hand side, before calling the bidiagonal system solver. In each iteration the result vector is used to modify the coefficient and/or right hand side in their reordered form.

The following subroutine solves a bidiagonal system with reordered coefficient and right hand side vectors, delivering the result vector also in reordered form.

An additional entry point is provided that may be used when the coefficient matrix is not changed since the last call of the routine. Also the requirement that N is a multiple of K has been dropped.

```
C       ALGORITHM L
        SUBROUTINE BIDIAGT (A,B,X,AA,N,M,K)
        DIMENSION  A(K+1,M),B(K+1,M),X(K+1,M),AA(K,2:M)

        AA(2:K,2) = -A(2:K,1)*A(2:K,1)
        DO  200  I=3,M
200     AA(2:K,I) = -A(2:K,I)*AA(2:K,I-1)

        ENTRY BIDIAGR

        X(1:K,2) = B(1:K,2)-A(1:K,2)*B(1:K,1)
        DO  210  I=3,M
210     X(1:K,I) = B(1:K,I)-A(1:K,I)*X(1:K,I-1)

        DO  250  J=2,K
250     X(J,M) = X(J,M)-AA(J,M)*X(J-1,M)

        X(1,1) = B(1,1)
        X(2:K,1) = X(2:K,1)-A(2:K,1)*X(1:K-1,M)
        DO  300  I=2,M-1
300     X(2:K,I) = X(2:K,I)-AA(2:K,I)*X(1:K-1,M)
        KM = K*M
        L=N-KM
        XL = X(K,M)
        DO  320  J=1,L
        XL = B(K+1,J)-A(K+1,J)*XL
320     X(K+1,J) = XL
        RETURN
        END
```

The 250-loop may be replaced by a call to some fast routine for the solution of a bidiagonal system, e.g., one of the routines discussed in this paper. Denoting the execution time per equation in cycles for such a routine by x, we estimate the execution time of algorithm L (in cycles) as follows:

Reduction of coefficients	(200-loop)	$(M-1)K/p+(M-1)S$
Reduction of right hand side	(210-loop)	$2(M-1)K/p+2(M-1)S$
Solve reduced system	(250-loop)	xK
Substitution phase	(300-loop)	$2(M-1)K/p+(M-1)S$
Total:		$5(M-1)K/p+5(M-1)S+xK=$ $5N/p-5K/p+5MS-5S+xK,$

where N is the number of equation
K is the reordering parameter
p is the number of vector pipes

s is the start-up time and
x is the execution time in cycles per equation for the routine replacing the 250-loop.

We replace the 250-loop by Algorithm R, as described in section IV.5, and estimate the value of x by 9. Since $M*K=N$, we find an optimum, if

$$\frac{dT}{dM} = 4S-(x-5/p)*N/M^2=0.$$

For a one pipe Cyber this gives $M\approx\sqrt{N/50}$ and $T\approx 5N+64\sqrt{N}$. For $N=25600$ we thus expect an execution time per equation of $5+64/160=5.4$ cycles. We have measured 5.48 cycles for $M=\sqrt{N/50}$. In case the coefficient matrix is not changed between calls we use the entry point BIDIAGR; as we do not carry out the reduction of the coefficients (200-loop), the execution time is then estimated by

$$4N/p-4k/p+4Ms-5s+xK.$$

Using the same solution process for the remaining loop of length M/K, we estimate the execution time per equation as 4.37 cycles we have and measured 4.45 cycles for a system of 25600 equations.

IV Cyber 205 scalar optimization

IV.1. Cyber 205 Scalar Processor

The Cyber 205 scalar processor contains a number of independent segmented functional units, capable of accepting one scalar instruction every cycle and delivering the result of the instruction in a fixed number of cycles, dependent on the instruction executed. The issue unit is capable of accepting every minor cycle one instruction to the functional unit for the instruction issued; the one exception relevant to our discussion is the store instruction that requires two cycles in the issue unit. The times spent in the functional unit for relevant types of instructions are shown below.

Floating point add/subtract	5 cycles
Floating point multiply	5 cycles
Integer arithmetic	1 cycles
Load	15 cycles
Branch (within instruction buffers)	9 cycles
Branch (out of instruction buffers)	24 cycles

The execution time, for a series of instructions is at least equal to the maximum of the minimal issue time (i.e. the number of cycles required to issue all instructions, provided no delays occur) and minimal functional unit time (i.e. the sum of the time required in the functional unit for any subseries of instructions, of which each requires the result of a preceding instruction of the subseries as an operand).

In the further discussion we will assume (and prove) that by means of the usual optimization techniques such as loop unrolling and load ahead, the actual execution time can be kept as low as the maximum of minimal issue time and minimal functional unit time.

The scalar processor has 256 arithmetic registers of 64 bits each. The arithmetic registers are organized in the so-called register file, a memory at its own right, capable of performing two reads and one write every cycle. Each instruction result generated by a functional unit is written into the register file and may be read from the register file 3 cycles later. However, the result is available as an operand to another instruction in the cycle in which the instruction is completed; this is achieved by a bypass around the register file called shortstop. The functional unit times given above are shortstop times, e.g. the result of a floating point addition may be used as an operand to some

functional unit 5 cycles times after the issue of the floating point addition through the short stop bypass or at least 8 cycles after issue of the floating addition from the register file. Since the register file is capable of performing only one write per cycle, two instructions with different execution times, delivering their results in the same cycle, cause a so-called register file write conflict and the issue of the faster instruction is delayed by the issue unit.
The swap instruction is executed mainly in the vector processor, but is discussed here because we use it only explicitly in scalar code.

The swap instruction allows to load a number of consecutive registers from a vector in memory and simultaneously store a number of consecutive registers into a vector in memory. The execution time is independent of the number of vector pipes and amounts to $56+N/2$ cycles, where N is of the numbers of registers loaded or stored, which ever is greater. Conventionally this instruction is used at entry and exit of a subroutine to exchange the local variables of caller and callee.

The scalar processor has 8 buffers, each capable of holding 16 half-words of instructions. It attempts to maintain a load ahead of 2 buffers. Reading a whole buffer requires 4 minor cycles of memory bank busy time only.

The Branch instruction requires 9 cycles if the target instruction is already in an instruction buffer and 24 cycles otherwise. Hence, loops that fit in 6 instruction buffers may be executed without an instruction buffer load.

On the other hand a sequence of instructions that does not fit in 6 instruction buffers and contains load/store instructions may be delayed because of memory bank conflicts between the load/store instruction and the loading of instruction buffers.

IV.2. Straight forward scalar code
The formula (2) is represented by the code

```
C     ALGORITHM  M
      X(0) = B(0)
      DO  10  I=1,N-1
10    X(I) = B(I)-A(I)*X(I-1)
```

The body of the loop contains the following instructions:

2	load instructions for $A(I)$ and $B(I)$
2	floating point instructions
1	store instruction for $X(I)$
1	increment and branch instruction

The minimal issue time for these instructions is 7 minor cycles. Since the increment and branch instruction requires 9 cycles functional unit time, we unroll this loop many times and we use load ahead to avoid delays from the 15 cycle load functional unit time.

The minimal functional unit time for one equation is then the sum of the functional unit times of the two floating point instructions i.e. 10 minor cycles. When unrolling 256 times, we actually observed an execution time of 10.11 cycles per equation.

IV.3. Cyclic reduction on the fly

In the straight forward Algorithm M discussed in the preceeding Section IV.2 the execution time was determined by the minimal functional unit time. The application of cyclic reduction increases the number of instructions and hence the minimal issue time, but reduces the recursiveness and hence the minimal functional unit time.

Fortran code:

```
C     ALGORITHM  N1
      X(0) = B(0)
      DO  10  I=1,N-2,2
      X(I) = B(I)-A(I)*X(I-1)
      BB = B(I+1)-A(I+1)*B(I)
      AA = A(I+1)*A(I)
 10   X(I+1) = BB-AA*X(I-1)
```

The body of the loop handles 2 equations and contains

4	load instructions
7	floating point instructions
2	store instructions
1	index increment instruction

The minimal issue time is 16 cycles and the minimal functional unit time is 10 cycles per 2 equations.

We unroll this loop twice, thus handling 6 equations in the body of the DO-loop. We apply load ahead and deferred store when necessary to avoid conflicts (using 3 instances of the same variable). Below we show the DO-loop of the resulting routine:

```
C     ALGORITHM  N1.1
 10   CONTINUE
      A1 = A(I+6)          A3 = A(I+6)          A5 = A(I+6)
      A2 = A(I+7)          A4 = A(I+7)          A6 = A(I+7)
      I = I+2              I = I+2              I = I+2
      BB = B4-AB           BB = B6-AB           BB = B2-AB
      AAX = AA*X2          AAX = AA*X2          AAX = AA*X2
      AX = A3*X2           AX = A5*X2           AX = A1*X2
      X(I-1) = X2          X(I-1) =X2           X(I-1) = X2
      B1 = B(I+4)          B3 = B(I+4)          B5 = B(I+4)
      B2 = B(I+5)          B4 = B(I+5)          B6 = B(I+5)
      AA = A5*A6           AA = A1*A2           AA = A3*A4
      AB = B5*A6           AB = B1*A2           AB = B3*A4
      X2 = BB+AAX          X2 = BB+AAX          X2 = BB+AAX
      X(I-2) = X1          X(I-2) = X1          X(I-2) = X1
      X1 = B3-AX           X1 = B5-AX           X1 = B1-AX
                                                IF (I.LT.NX) GOTO  10
```

For each of the 43 lines of the code, exactly one machine instruction is generated. The minimal issue

time is 57 cycles, computed as follows

12	loads	12
6	stores	12
21	floating point	21
3	index increment	3
1	test and branch	9
		---
		57 cycles

The minimal functional unit time is determined by the 3 lines computing AAX, the three instructions computing X2 and the test and branch instruction. Since we do not use the short stop feature in this routine, we need $6\times 8+9=57$ cycles.

We thus expect an execution time of close to $57/6=9.5$ cycles per equation for large systems; we actually measured 9.53 cycles per equation for a bidiagonal system of 25600 equations.

We may improve the algorithm by unrolling once: Algorithm N1.2 handles 12 equations in each pass through the loop and requires $2\times 48+9=105$ cycles for 12 unknowns. We expect close to $105/12=8.75$ cycles per equation and measured 8.78 cycles for a system of 25600 equations.

The number of lines of FORTRAN code, as well as the number of instructions generated for the body of the loop, is now increased to $6\times 14+1=85$: Those 85 instructions still fit in 6 instruction buffers of 16 instructions each. Further unrolling will cause the body of the loop to exceed the capacity of the instruction buffers. The test and branch instruction will then require 24, rather than 9, cycles and the instruction buffers will be reloaded cyclically during execution of the loop.

Algorithm N1.3 is constructed by unrolling the loop 31 times; 384 equations are solved for each trip through the loop; we thus expect an execution time of

$$16/2+24/384 = 8.0625 \quad \text{cycle per equation.}$$

However for a system of 25600 equations we observed 8.81 cycles per equation. The difference of 0.75 cycles per equation is caused by the occurrence of memory bank conflicts between the load and store instructions in the code and the loading of instruction buffers. We will discuss this phenomenon in Section V.3.

Algorithm N2

Because in the preceeding algorithm the minimal issue time is larger than the minimal functional unit time we modify the code in order to reduce the number of instructions and hence the minimal issue time, be it at the expense of an increase in minimal functional unit time. We do this by computing $XAA = A(i)*A(i+1)*X(i-1)$ in a different way. This leads to the following code for the loop:

```
C     ALGORITHM  N2.1
      X(0) = B(0)
      DO  10  I=1,N-2,2
      XA = A(I)*X(I-1)
      BB = B(I+1)-A(I+1)*X(I-1)
      XAA = A(I+1)*XA
 10   X(I+1) = BB-XAA
```

The instructions in this code are listed with their minimal issue time

4	load instructions	4

6	floating point instructions	6	
2	store instructions	4	
1	index increment instruction	1	
		---	
	total minimal issue time	15	cycles .

The three floating point instructions computing XA,XAA and $X(I+1)$ in the code above constitute the critical path, hence the minimal functional unit time is $3\times5=15$ cycles. We coded this algorithm in FORTRAN.

As before we rewrite the code given above, so that each line of code produces a single machine instruction giving for the body of the DO-loop:

sequence	fortran code	minimal issue time	minimal functional unit time
1	$A1=A(I)$	1	
2	$A2=A(I+1)$	1	
3	$B1=B(I)$	1	
4	$B2=B(I+1)$	1	
5	$XA=A1*X2$	1	5
6	$AB=A2*B1$	1	
7	$BB=B2-AB$	1	
8	$X1=B1-XA$	1	
9	$XAB=A2*XA$	1	5
10	$X(I)=X1$	2	
11	$X2=BB-XA$A	1	5
12	$X(I+1)=X2$	2	
13	$I=I+2$	-----	-----
		15	15

Since this code is designed to be unrolled many times we ignore the time for the test and branch instruction.

We reorder the instructions to obtain the minimal issue time of 15 cycles per 2 equations. First we select the cycles in which the three floating point instructions on the critical path will be issued. We select cycles 4,9 and 14. Then we determine the issue time for the other three floating point instructions, so as to avoid delays due to operands not being available. Where necessary we compute ahead: and we do the same for the remaining instructions. The highest load/compute ahead factor required is 3, thus for some variables we need three instances and the code must be unrolled to a multiple of three copies that differ only in the cyclic interchange of the local variables. We show the final code for 2 equations below:

seq. no.	old seq. no	statement	cycle of issue	result available at cycle	result required	load/compute ahead
1	1	$A1=A(I2+6)$	0	15	4+2*15	3
2	2	$A2=A(I2+7)$	1	16	13+15	3
3	12	$X(I-1)=X6$	2	-		-1
4	5	$XA=A3*X2$	4	12(9)	9	1
5	13	$I4=I4+6$	5	9	0+15	1

6	7	$BB=B4-AB4$	6	14	14	1
7	10	$X(I2)=X1$	7	-	-	0
8	9	$XAA=XA*A4$	9	17(14)	14	1
9	3	$B1=B(I2+6)$	10	25	12+2*15	3
10	4	$B2=B(I2+7)$	11	26	6+2*15	3
11	8	$X3=B3-AX$	12	20	7+15	1
12	6	$AB6=A6*B5$	13	21	6+15	2
13	11	$X4=BB-XAA$	14	22(19)	4+15	1

The column "result available" lists the cycle at which the result of the instruction is available in the register file. For those instructions of which the result is used at short stop time, the cycle of short stop availability is indicated in parentheses. The column "result required" lists the cycle at which the first instruction requiring the result is issued: each entry is written as the sum of the relative cycle in the basic block of 15 cycles and the relative block number of usage multiplied by 15. The column "load/compute ahead" lists the related block number of first usage of the result. As we see the entries in the result "available column" are all different modulo 15. This is necessary because the register file is capable of one write per cycle only. The use of load/compute ahead necessitates the use of more than one instance of some variables. Those instances are cyclically interchanged between the three columns of instructions. The cyclically interchanged instances of variables are listed below, where each triple of instances of the same *variable* is enclosed in parentheses.

$$(A1,A3,A5),(A2,A4,A6),(B1,B3,B5),(B2,B4,B6)$$

$$(X1,X3,X5),(X6,X2,X4),(AB4,AB6,AB2),(I2,I4,I6)$$

The number of registers required for this code is quite large and includes at least

24 registers for cyclically interchanged variables,
6 registers for the addresses A(6), A(7), B(6), B(7), X(0), X(1),
4 registers for XA, XAA, BB, N,
4 registers for the parameters X, A, B, N,

and a number of registers for the standarized interface between the system, caller and callee. The development of this type of code is therefore possible only on machines with a large number of registers like the Cyber 205.

The execution time expected for this algorithm, that handles 6 equations each iteration, is

$$(3\times15+9)/6 = 9.0 \quad \text{cycles per equation.}$$

We actually measured 9.03 cycles per equation for a system of 25600 equations.

As before we may unroll the DO-loop once to solve 12 equations in the body of the DO-loop yielding algorithm N2.2, for which we expect an execution time close to.

$$(6\times15+9)/12 = 8.25 \quad \text{cycles per equation}$$

for a large system. We measured 8.28 cycles per equation for a system of 25600 equations.

As before further unrolling will cause the capacity of the instruction stack to be exceeded and thus the test and branch instruction will require 24 rather than 9 cycles. Unrolling the loop 31 times to handle 384 equations on each pass through the loop, we hoped to achieve an execution time close to

$$15/2+24/384 = 7.57 \quad \text{cycles,}$$

but we measured 8.69 cycles per equation for a system of 25600 equations. The difference is caused by memory bank conflicts and will be discussed in more detail, in Section V.3.

ALGORITHM N3

Algorithm $N1$ needed per 2 equations a minimal issue time of 16 cycles and a minimal functional unit time of 10 cycles. We did not use the short stop feature and the functional unit time was thus increased to $2 \times 8 = 16$ cycles. In algorithm $N2$ both minimal issue time and functional unit time were 15 cycles per 2 equations. With algorithm $N3$ we reduce the minimal issue time to 15 cycles while retaining the minimal functional unit time of 10 cycles. We do this by removal of the index increment instructions used in both $N1$ and $N2$ and we use one index register with preset values for each pair of equations solved in the body of the loop modifying 2 base addresses for each of the three arrays A,B and X used. Algorithm $N3$ cannot be coded in FORTRAN; we use the assembler instead.

The body of the loop contains for each pair of equations the following instructions, listed with minimal issue time

		issue	
4	loads	4	
2	stores	4	
7	floating point	7	
	total	15	cycles

and at the bottom of the loop we have the following instructions:

		issue	
6	base register increment	6	
1	test and branch	9/24	
	total	15/30	cycles for a loop fitting/not fitting in the instruction buffers

The minimal functional unit time is determined as before by the two floating point instructions on the critical path and is thus 10 cycles. Since the minimal issue time exceeds 13 cycles we use short stop only once and thus increase the functional unit time to 13 cycles.

Algorithm N3.1 handles 6 equations at each pass through the loop; the execution time is estimated by

$$(3 \times 15 + 15) / 6 = 10 \text{ cycles}$$

we measured 10.3 cycles for a system of 25600 equations. Similarly algorithm N3.2, is derived by unrolling N3.1 once, solves 12 equations per pass through the loop and has an estimated execution time of

$$(6 \times 15 + 15) / 12 = 8.75 \text{ cycles.}$$

We measured 8.78 cycles for a system of 25600 equations.

Algorithm N3.4 solves 408 equations per iteration but the body of the loop exceeds the capacity of the instruction buffers. The estimated execution time is

$$15 / 2 + 30 / 408 = 7.57 \text{ cycles.}$$

We measured 8.08 cycles on a system of 25600 equations. Algorithm N3.3 solves 384 (a multiple of 64) equations per iteration; we measured 7.93 cycles per equation for a system of 25600 equation. Algorithms N3.3 and N3.4 suffer from delays caused by memory bank conflicts, just as N1.3 and N2.3, but to a lesser extent We will discuss this phenomenon in Section V.3.

IV.4 Swap and Cyclic Reduction on the Fly

In all algorithms discussed so far we required two loads and one store instruction for each equation. However, as described in Section IV.1, a faster transfer of data between memory and the register file may be achieved by means of the so-called swap instruction. In algorithm *P* we use the swap instruction to load elements of the right hand side vector *B* and simultaneously store elements of the result vector *X*. We will describe the algorithm in pseudo-fortran, in which the section of the register file into which, and from which, we swap data is represented by a so-called register file array. Syntactically such arrays will be distinguished from normal arrays by the use of square brackets, rather than parentheses, in declarations as well as in references. Semantically the use of register file arrays is restricted to subarray references in swap statements and references to elements with constant indices outside swap-statements. Loops in which register file arrays are indexed by the loop variable must thus be fully unrolled. The code shown below contains a parameter statement defining LRF used as length of the subarray swapped. LRF is determined by the number of registers available: we assume that $N-1$ is a multiple of LRF. In the code shown below, the elements of the coefficient vector *A* are loaded ahead into a cyclic buffer in the register file; we use indices modulo 4 with array A. Since the loops in which those loads occur are fully unrolled the actual indices are constants, the modulo function is computed at assembly time.

```
C     ALGORITHM  P
      SUBROUTINE BIDIAG (X,A,B,N)
      DIMENSION X(0:N-1),A(1:N-1),B(0:N-1)
      PARAMETER (LRF=216)
C     REGISTER FILE ARRAYS
      DIMENSION  RF[0:LRF+2],AR[0:7],XA[0:1],AA[0:1]
C     LOAD AHEAD FROM A INTO AR
C     THIS LOOP IS UNROLLED COMPLETELY
      DO 5  I=1,6
 5    AR [I-1] = A(I)

      RF[0] = B(0)
C     COMPUTE AHEAD
      AB = B(1)*AR[1]
      AA[1] = AR[0]*AR[1]
      BB = B(2)-AB
C     SWAP IN FIRST BLOCK OF RIGHT-HAND SIDE
      RF[1:LRF+2] = B(1:LRF+2)
      J = 1
      GOTO 16
 15   RF[0] = RF[LRF]
C     SWAP BLOCKS OF RIGHT-HAND SIDE AND RESULT
      X(J-LRF;LRF) = RF(1;LRF)
      RF(1;LRF+2) = B(J;LRF+2)
C     THIS LOOP IS UNROLLED COMPLETELY
      DO  10  I=1, LRF,2
      AB = RF[I+2]*AR[MOD (I+3,8)]
      XAA = RF[I+1]*AA[MOD(I/2,2)]
      AR[MOD(I+5,8)] = A(J+I+5)
      AA[MOD(I/2+1,2)] = AR[MOD(I+2,8)]*AR[MOD(I+3,8)]
      XA[MOD(I/2,2)] = RF[I-1]*AR[MOD(I-1,8)]
      AR[MOD(I+6,8)] = A(J+I+6)
```

```
      RF[I+1] = BB * XAA
      IF(I.NE.1) RF[I-2] = RF[I-2]-XA[MOD(I/2+1,2)]
      BB = RF[I+3]-AB
C     INDEX INCREMENT
      I = I+2
 10   CONTINUE
      RF[LRF-2] = RF[LRF-2]-XA[1]
      J = J+LRF
      IF(J+LRF.LT.N) GOTO 15
C     SWAP LAST BLOCK OF RESULT VECTOR
      X(J-LRF;LRF) = RF[1;LRF]
      RETURN
      END
```

Each statement in the DO-loop with end-label 10 corresponds to a single instruction. The body of the loop handles 2 equations and contains

2 load instructions
7 floating point instructions and
1 index increment instruction

The minimal issue time is therefore 10 cycles, the minimal functional unit time is determined by the two floating point instructions computing XAA and RF[I+1] and is therefore $2\times5=10$ cycles for 2 equations.

Each iteration over the loop starting label 15 handles 216 equations. Its execution time is computed as follows:

SWAP instructions $S+N/2=56+216/2$	=	164	cycles
DO-loop end label 10 $216/2\times10$	=	1080	cycles
Test and branch, etc.		40	
		1284	cycles

or $1284/216=5.95$ cycles per equation.

We measured 6.28 cycles per equation on a system of 25600 equations. Estimating the overhead at 0.03 cycles, we note a discrepancy of $6.28-0.03-5.95=0.30$ cycles per equation. Again this difference must be attributed to memory bank conflicts between the load instructions and the loading of instruction buffers; we discuss this phenomenon in Section V.3.

IV.5. Vectorized Cyclic Reduction, Swap and Cyclic Reduction on the Fly.

In Algorithm R we avoid all load and store instructions in the scalar code.

We first apply cyclic reduction in vector mode using algorithm H1: we compute the right-hand side and coefficient elements of the reduced system in alternating positions of a single array AB. We can thus swap in the right-hand side and coefficient vector elements in a single swap instruction. In the scalar code we replace the right-hand side elements by the corresponding elements of the result vector. We swap out an array in which the odd numbered elements contain the result elements of the reduced system, i.e. the even numbered result elements of the original system of equations. Finally we perform back substitution in vector mode using *control store* capability. The vectorized cyclic reduction is rewritten to provide a copy of the odd elements of the right-hand side vector; this is necessary if

the right-hand side and result reside in the same vector, because the swap-out of the even result elements destroys the odd elements in the result vector. The scalar code will now contain for every 2 elements only the 7 floating point instructions, since the load and store instructions are replaced by the swap instructions and the index increment instruction is not needed. We thus obtain a minimal issue time of 7 cycles, retaining the 10 cycles functional unit time. Since in optimal code the minimal issue time and minimal functional unit time should be equal, we may issue more instructions if that leads to a lower functional unit time. Therefore in the scalar code we reduce three out of every five equations rather than one out of every two. As shown in the annotated code, given below, for every 5 equations in the reduced system we now obtain a minimal issue time of 19 cycles and a minimal functional unit time of 20 cycles and thus an execution time of 4 cycles per equation rather than 5 cycles as for algorithm P. For reasons of readability we have not rewritten the code to generate one instruction per statement. Also we use a DYNAMIC statement to allocate dynamically temporary arrays.

```
C       ALGORITHM  R
        SUBROUTINE BIDIAG (X,A,B,N)
        DIMENSION X(0:N-1),A(1:N-1),B(0:N-1)
        PARAMETER (LRF=220)
C       DIMENSION RF [-1:N)
        ARRAY RF IN THE REGISTERS
        DYNAMIC D(1:N-1), AB(N)
C       VECTORIZED CYCLIC REDUCTION
C       MASK USING ONE VECTOR INSTRUCTION
        D(1:N-1:2) = A(2:N-1:2)
        D(2:N-1:2) = B(1:N-1:2)
C       D HOLDS COPY OF ODD ELEMENTS OF B
C       TO BE USED IN BACK SUBSTITUTION
        AB(2:N) = -A(1:N-1)*D(1:N-1)
        AB(1:N-1,2) = B(2:N:2)-D(2:N:2)
        RF [-1] = B(0)
        X(0) = RF[-1]
        RF[1;LRF+10] = AB(1;LRF+10)
        J = 1
        GOTO 110
 100    CONTINUE
C       SWAP
        RF[-1] = RF[LRF-1]
        X(J-LRF;LRF) = RF[1;LRF]
        RF[1;LRF+10] = AB(J;LRF+10)
C 10-LOOP UNROLLED                                  ISSUE    FUNCT. UNIT
C                                                   TIME     TIME
C 110   DO  10  I=1,LRF,10
C       REDUCE EQUATIONS I+2,I+6 and I+8
        AA2 = RF[I+3] * RF[I+1]                      1
        BB2 = RF[I+2] * RF[I+3] * RF[I]              2
        AA6 = RF[I+7] * RF[I+5]                      1
        BB6 = RF[I+6] * RF[I+7] * RF[I+4]            2
        AA8 = RF[I+9] * AA6                          1
        BB8 = RF[I+8]-RF[I+9] * BB6                  2
C       SOLVE 5 EQUATIONS
        RF[I] = RF[I]-RF[I+1] * RF[I-2]              2
```

```
      RF[I+2] = BB2-AA2 * RF[I-2]                    2          10
      RF[I+4] = RF[I+4]-RF[I+5] * RF[I+2]            2
      RF[I+6] = BB6-AA6 * RF[I+2]                    2
      RF[I+8] = BB8-AA8 * RF[I+2]                    2          10
10    CONTINUE
C     TOTAL ISSUE AND FUNCT. UNIT TIMES             19          20
      J = J+LRF
      IF(J+LRF.LE.N) GOTO 110
C     SWAP OUT LAST BLOCK OR RESULT
      X(J-LRF;LRF) = RF(1;LRF)

c     BACK SUBSTITUTION IN VECTOR MODE
      X(1:N-1:2) = D(2:N:2)-A(1:N-1:2) * X(0:N-1:2)
      RETURN
      END
```

The execution time in cycles per equation is estimated as follows

Vectorized Cyclic Reduction		3/p	
Swap (56+220/2)/220	0.76		
Scalar code 20/10	2.0		
Loop overhead 33/220	0.15		
Vectorized back substitution		2/p	
	2.91 +	5/p	cycles

We measured 7.95 cycles on a one pipe Cyber for a system of 25600 equations.

IV.6. Vector/Scalar Overlap

Since the Cyber 205 Vector and Scalar processor can operate simultaneously and independently as long as the scalar processor does not access memory, we attempted to modify algorithm *R*, described in the previous section, by overlapping the execution of the scalar code in the DO-loop reduction (with end label 10) by the vector instructions in the vectorized cycle reduction and back substitution phases.

To this end we split the vector instructions so that each vector instruction processes five times the number of elements processed in the scalar loop and at the beginning of every execution of the scalar loop we start one of the vector instructions.

Since we now need some registers to keep track of the partially executed cyclic reduction and back substitution we reduce the number of registers swapped (PARAMETER LRF) from 220 to 210.

Because of the added complexity we increase the estimate for the overhead of the innermost loop from 40 to 100 cycles. The estimated execution time is

Swap 210 registers 56+210/2 =	161	cycles
Non shareable vector start-up	17	cycles
	178	cycles

Further we add the overlap time, which is the maximum of the scalar processing time and the shareable vector processing time. The scalar time is

loop body 210/10×20 =	420	cycles
loop overhead	100	cycles
total	520	cycles

The shareable vector time is 34+1050p cycles.

Hence the total time is computed as

	$p=1$	$p=2$	$p=4$
non-shared	178	178	178
shared = max (34+1050/p, 520) =	1084	559	520
total for 210 equations	1262	737	698
total per equation	6.0	3.5	3.4

However, on a one pipe Cyber we measured 8.55 cycles per equation of a system of 25600 equations.

The reason for the discrepancy turned out to be, that the execution of a vector instruction cannot be overlapped with the reading of instruction buffers.

We still mention this algorithm, because we expect a better result for it on the ETA[10] (the successor of the Cyber 205).

V. Obtained results

V.1. Predicted and Actual execution times

In table V.1 below we list the predicted and actual execution times of the algorithms discussed in Chapters II, III and IV.

The column headed ALG. refers to the mnemonic of the algorithm, the column headed UNR/VEC contains the letter *V* for vectorized algorithms and /or the number of equations treated in one pass through the innermost scalar DO-loop. The right-most 3 columns list the asymptotical predicted execution times, in cycles per equation, for large systems for one, two and four vector pipes. The column headed "ACTUAL 1 pipe" lists the best execution times obtained on a one pipe Cyber for a system of 25600 equations.

ALG.	UNR/ VEC	Technique	Actual 1 pipe	PREDICTED no of pipes 1	2	4
		II. FORTRAN SUBROUTINES				
A	2	Straightforward	34.5			
B	8	Load A head and Separate Store	12.28			
C	4	Load A head and Deferred Store	10.78			
C	8	Load A head and Deferred Store	12.03			
D	8	Separate Store Unrolled	12.52			
E	8	Deferred Store Unrolled	11.89			
		III. VECTOR ALGORITHMS				
F	V	Recursive Doubling (one iteration only)	-	3.0	1.5	0.75
G	V	Repeated Recursive Doubling	-	20.19	11.60	6.85
N1	V	Cyclic Reduction (one iteration only)	-	5.0	2.5	1.25
H2	V	Cyclic Reduction Compressed	-	7.0	3.5	1.75
H3	V	Cyclic Reduction Folded	-	5.5	2.75	1.375
J2	V	Repeated Cyclic Reduction Compressed	-	14.0	7.00	3.50
J3	V	Repeated Cyclic Reduction Folded		11.0	5.50	2.75
K	V	Cascaded Cyclic Reduction	9.92	9.90	7.40	6.15
L1	V	Cascaded Cyclic Reduction (Reordered)	5.48	5.45	2.95	1.70
L2	V	Cascaded Cyclic Red. (Reordered, same coeff.)	4.45	4.40	2.40	1.40
		IV SCALAR ALGORITHMS				
M	256	Straightforward optimization	10.11	10.0	10.00	10.00
N1.2	12	Cyclic Reduction on the Fly	9.53	9.50	9.50	9.50
N1.3	384	Cyclic Reduction on the Fly	8.61	8.06	8.06	8.06
N2.2	12	Cyclic Reduction on the Fly	9.03	9.00	9.00	9.00
N2.3	384	Cyclic Reduction on the Fly	8.69	7.57	7.57	7.57
N3.3	384	Cyclic Reduction on the Fly	8.08	7.59	7.59	7.59
N3.4	408	Cyclic Reduction on the Fly	7.93	7.59	7.59	7.59
P	216	Swap and Cyclic Reduction on the Fly	6.28	5.95	5.95	5.95
R	V220	Vect. Cycl. Red, SWAP, Cycl. Red. on the Fly	7.95	7.90	5.40	4.15
S	V210	V. Cycl. R, SWAP, Cycl. Red. Fly Overlap	8.55	6.15	3.55	3.40

TABLE V.1. Predicted and Actual execution times

V.2. Actual Execution times for different System Sizes.

In table V.2 we list the execution times in cycles per equation observed for with some of the algorithms described in Chapters II, III and IV for systems of different sizes.

The top line of the table lists the mnemonic of the algorithm, where *Q* designates the library routine *Q*8SM011. The second line lists the unroll count, i.e. the number of equations solved in the innermost loop of the algorithm and the third line shows a letter *V* to for a vectorized algorithm.

The third line lists the routines used to solve the reduced system, where applicable. The left most column lists the size of the tested system. In the column the entry "25600*" marks the row with execution times for systems of 25600 equations, where the operand vectors have been aligned in memory to minimize memory bank conflict overhead. The entry "APT" signals the "Asymptotic Predicted Times" for the algorithm, while the rows marked N(1.1) resp. N(1.5) display the system sizes for which execution times of 1.1 resp. 1.5 times the execution time for N = 25600 are attained. The entry VEC lists the time in cycles per equations spent in the vector pipe on a one pipe Cyber: this figure allows to estimate the execution time on a 2 or 4-pipe Cyber.

Algorithm	Q	B	C	C	K	L_1	L_2	M	N_2	N_3	P	R
UNROLL		8	4	8				256	12	384	216	220
VECTOR					V	V	V				V	V
SIZE/ROUT						Q	R					
1	392	788	640	1096	-	-	-	146	556	118	167	121
50	29.52	25.29	23.20	26.29	-	-	-	13.52	21.04	20.33	18.54	23.70
100	22.76	18.60	17.77	16.05	-	-	-	11.79	15.52	14.64	12.42	15.73
200	18.38	15.46	14.26	14.05	-	-	-	10.98	11.41	11.07	9.54	12.14
400	16.12	13.95	12.50	12.28	19.39	12.90	11.03	10.52	10.07	9.61	8.06	10.19
800	15.29	13.10	11.63	11.39	14.61	10.07	7.61	10.33	9.05	8.80	7.12	9.03
1600	14.87	12.68	11.19	10.95	12.98	8.31	6.19	10.22	8.70	8.47	6.71	8.48
3200	14.66	12.46	10.97	10.73	12.46	7.17	5.44	10.16	8.45	8.25	6.49	8.20
6400	14.55	12.30	10.86	10.62	11.38	6.46	4.97	10.14	8.37	8.16	6.37	8.06
12800	14.50	12.28	10.81	10.56	10.46	6.00	4.64	10.12	8.31	8.11	6.31	7.99
25600	14.48		10.78	10.53	9.92	5.67	4.45	10.11	8.28	8.09	6.29	7.96
25600*									-		6.27	-
APT		12.25	10.75	10.50	9.90	5.60	4.40	10.00	8.25	7.57	5.95	7.99
VEC	0	0	0	0	9.2	5.0	4.0	0	0	0	0	5.0
N(1.1)	550	600	750	900	10000	10000	6400	200	800	700	1200	1000
N(1.5)	90	100	140	180	800	2000	1200	30	175	185	200	190

TABLE V.2. Execution times for different system sizes

The fastest algorithm for a large system is thus:
on a 1 pipe Cyber 205: algorithm P (scalar)
on a 2 pipe Cyber 205: algorithm R (mixed)
on a 4 pipe Cyber 205: algorithm $J3$ (vector)
Note that algorithms L1 and L2 assume reordered input vectors and deliver reordered result vectors.

V.3. THE EFFECT OF MEMORY BANK CONFLICTS ON EXECUTION TIME

In several of the described routines the inner most DO-loop contains memory references and exceeds the capacity of the instruction buffers in the Cyber 205 scalar processor. Such routines are delayed to some extent by memory bank conflicts between the load and/or store instructions on the one hand and the loading of instruction buffers on the other hand.
As may be seen in the row marked "25600" in table V.2 the delay is only slightly dependent on relative allocation of the parameter arrays. Because in these routines the loads and/or stores access the elements of parameter arrays sequentially the loads and/or stores access all memory banks in sequence at a fixed rate. The loading of instruction buffers of 16 instructions or halfwords also accesses all memory banks sequentially, but at a faster rate than the loads and/or stores in the code. We can therefore predict how often memory bank conflicts between those streams of accesses will occur. The code in the DO-loops is periodic by nature. In Table V.3 we list some relevant properties of the five routines concerned.

		ALGORITHM				
	Characteristics	M	N1	N2	N3	P
N	Number of equations	1	2	2	2	2
F	Floating point per period	3	8	7	7	8
L	Loads per period	2	4	4	4	2
S	Stores per period	1	2	2	2	0
I	Instructions per period	6	14	15	13	10
IS	Issue time in cycles	7	15	15	15	10
FU	Functional unit time in cycles	10	16	15	15	10
E	Execution time in cycles (no delay)	10	16	15	15	10
W	Register file writes	4	12	11	11	10
SS	Short stopped functional unit times	10	0	15	5	10
RC	Conflict predicted every RC cycles	160	112	125	125	160
RD	Ratio delay/execution time	0	0.07	0.16	0.05	0.07
DC	Delay in cycles per predicted conflict	0	8	10	3	10

TABLE V.3. Properties of Routines affected by bank conflicts

Whenever a bank conflict between a load/store instruction and the loading of an instruction buffer occurs, either may be delayed, although load and store instructions have the higher priority. Delaying the instruction buffer load, normally will not delay the execution of instructions. The delay of a store instruction does not directly delay the execution of the routines concerned, unless it causes a delay in subsequent load instructions. When a load instruction suffers from a bank conflict the execution of the load rather than its issue is delayed and the writing of the data loaded from memory into the register file is postponed by at least 4 cycles. The respective routines feature load ahead to such an extent that the delay in the availability of the data itself will not cause a delay in the execution. However, the register file write of the data loaded from memory now takes place at a non-predictable time and may cause another instruction to be delayed because the register file is only capable to handle one write operation per cycle. The probability of such a register file write conflict obviously depends on the number of cycles per period in which data is written in the register file (see row *W* in Table V.3).

Also when the execution of a floating point instruction is delayed because of a register file write conflict, the instruction may be buffered in front of the floating point unit and the next instruction may be issued, unless it is a floating instruction too, in which case the issue of the next instruction is delayed. Thus the denser the floating point instructions in the code, the greater the probability of secondary delays (see row *F*).

Also when the execution of a floating point instruction is delayed, the register file write of its result may conflict with the register file write resulting from an earlier non-delayed load instruction. When one floating point instruction uses the result of an earlier floating point instruction at the short stop time, the delay of the issue of any instruction between those two, will cause the former to move the short stop time causing a delay of at least 3 cycles. We may conclude that delays due to memory bank conflicts are positively correlated with the number of floating point instructions and the use of

the short stop feature.

VI. Generalization and notes

VI.1. The Ratio of the Reduced Equations

In some of the algorithms described in Chapters II, III and IV we used cyclic reduction on the fly. In algorithm *R* we reduced 3 out of every 5 equations rather than 1 out of every 2, thereby achieving a performance improvement from 10 cycles per 2 equations to 20 cycles per 5 equations. We will discuss the optimal ratio of equations to be reduced in a more general fashion below.

We define:

M the number of equations in a group
T the number of equations to be reduce per group of *N* equations

and

I the minimal issue time for one equation,
F the minimal function unit time for one equation,
R the minimal issue time for the reduction of one equation.

For a bidiagonal system of equations we have F=10 and R=3. The minimal issue time I depends on the residence of coefficients, right-hand side and result, i.e. on the number of load and/or store instructions required. In any case we need two floating point instructions. The minimal issue times I for the scalar loop in some of the algorithms described above are:

Algorithm M,N:	2 loads, 1 store, 2 floating point	I=6
Algorithm P:	1 load, 2 floating point	I=3
Algorithm R:	2 floating point	I=2

For algorithms, that contain load and/or store instructions, we need index-increment instructions in most cases at least one per group. As an exception in algorithm N3 we perform index incrementation for a number of groups at a time.

We define

X the number of index increment instructions per group of N equations.

We further define

J minimal issue time for a group of *N* equations,
G minimal functional unit time for a group of *N* equations,
E execution time per equation.

By definition we have

$$J = M \times I + T \times R + X$$

$$G = F(M - T)$$

and for the minimal execution time per equation *E*, we have

$$E = 1/M \times \max(J,G)$$

Since for any given N,J increases and G decreases with increasing T we find a minimum when J equals G, i.e. the minimal issue time equals the minimal functional unit time.
We thus have the fastest algorithm when

$$J = M \times I + R \times T + X = F(M - T) = G$$

or

$$M(F-I)-X = T(F+R). \tag{1}$$

For $X=0$, i.e. no index increment instructions required, we find the solutions

$$M = k(F+R), \quad T=k(F-I) \tag{2}$$

where k is rational, such that N and T take integer values greater than zero.

Selecting $k=1$, we have

$$G = F(M-T)=F(I+R)$$

and hence

$$E = G/M=F(I+R)/(F+R). \tag{3}$$

When $X>0$, we may set $X=(F-I,F+R)$ we find a family of solutions for (1) of the form

$$M = M_0+k(F+R) \quad \text{and} \quad T=T_0+k(F-I), \tag{4}$$

M_0 and T_0 are defined by

$$M_0(F-I) = X \quad \text{MOD}(F+R), \tag{5}$$

$$T_0 = (M_0(F-I)-X)/(F+R).$$

For the scalar loop solving the reduced system in algorithm R with $I=2,F=10,R=3$ and $X=0$ we find by application of (2) with $k=1$, $N=13$ that $T=8$ and hence (3) gives

$$E = 10(13-8)/(10+3)=50/13=3.92.$$

In algorithm R, described in Chapter V.5, the values $N=5$, $T=3$ are used, and hence

$$E = \max(J,G)/M = \max(5\times2+3\times3,10(5-3)/5=\max(19,20)/5=4.$$

We may thus reduce the execution time per cycle from 7.96 to 7.92, which is a marginal improvement.

Algorithm improved may be too:

We have $I=3$, $F=10$, $R=3$ and $X\neq0$. Setting $X=G.C.D(R-I,\ F+R)=GGD(7.13)=1$ we find using (5):

$$M_0*(F-I)+1=0\ MOD\ (F+R) \qquad \text{or}$$

$$M_0*7 = MOD\ 13 \quad \text{and thus} \quad M_0=2, \quad T_0=1$$

Applying (4) we obtain:

$$M=13k+2, \quad T=7k+1.$$

In the implementation of algorithm P, as described in Section IV.4, we used $k=0$, giving $N=2$ and $T=1$, hence

$$J = G=F(M-T)=10 \quad \text{and} \quad e=G/M=5.$$

For $k=1$, we get $M=15$ and $T=8$; hence

$$J = 6=F(M-T)=70 \quad \text{and} \quad E=G/M=70/15/4.66.$$

Hence, the code for one roup of M (15) equations contains

M times 1 load + 2 floating point
T times 3 floating point
X index increment instructions

which amounts to 15 loads, 54 floating point and 1 index increment instructions.

We can only achieve the execution time of 4.66 cycles for the scalar loop, if we succeed in sequencing the above 70 instructions so that all conflicts are avoided. The problem here is to avoid register file write conflicts, in spite of the different functional unit times of loads and floating point instructions. Note that in algorithm R discussed above we used only floating point instructions that write their result to the register file 8 cycles after issue, so that register file write conflicts will not occur.

As an example we investigate the minimal execution time of the solution of a bidiagonal system in which all elements of the subdiagonal are equal to -1. The equations are then reduced to

$$x_0 = b_0$$
$$x_i - x_{i-1} = b_i \quad (1=1,2,\cdots N-1)$$

with the solution

$$x_0 = b_0$$
$$x_i = b_i + x_{i-1} \tag{6}$$

Analogously to algorithm P we subdivide the X and B arrays in blocks of length LRF, where LRF is the number of available registers, say 220. For each 220 equations we then perform the following steps:

1. swap the B-array -segment into the register file
2. compute the 220 elements of the x-array
3. swap the computed values into the x-array segment.

As before steps 1 and 3 may be combined in a single swap instruction.

We apply our analysis to step 2. The code required for one equation consists of a single floating point instruction . To reduce one equation consider

$$x_{i+1} = b_{i+1} + x_1 = (b_{i+1} + b_i) + x_{i-1}.$$

To reduce the equation we need a single floating point addition.

We then find for step 2:

$$I = 1,\ F=5,\ R=1 \quad \text{and} \quad X=0.$$

From (3) and (4) we obtain

$$E = F(I+R)/(F+R) = 5(1+1)/(5+1) = 1.66$$
$$M = 6k,\ T=2k$$

for $k=1/2$ we find $M=3,\ T=1$.

The optimal algorithm solves groups of 3 equations as follows:

	issue	functional unit
$B2=B(I)\times B(I+1)$	1	
$B34=B2\times B(I+2)$	1	
$X(I)=B(I)+X(I-1)$	1	
$X(I+1)=B2+X(I-1)$	1	
$X(I+2)=B3+X(I+2)$	1	5
	5	5

such a group can be executed in 5 cycles. The total execution per equation for the solution of the

system with large N is then approximated as follows

SWAP (220/2 + 56)/220 =		0.76	cycles
Scalar loop		1.66	cycles
overhead	50/220 =	0.23	cycles

		2.65	cycles per equation

We measured 2.66 cycles per equation for $N=25600$.

Another possibility is an algorithm analogous to algorithm N. We may use load and store instructions rather than a swap instruction. We then need for each equation:

1 load
1 floating point instruction
1 store instruction,

and thus we have $I=4$, $F=5$, $R=1$, $X\neq 0$
As before we set $X=G.C.D(F-I,F+R)=GCD(1,6)=1$ and use (4) and (5) to solve

$$M_0(F-I)+X=0 \ MOD \ (F+R),$$

giving $M_0=1$, $T_0=0$ and hence

$$M = M_0+k(F+R)=1+6k$$

$$T = T_0+k(F-I)=k$$

The minimal execution time E for different values of k is then:

$k=0$ $G=5.5=5$ $E=5/M=5$
$k=1$ $G=5.6=30$ $E=30/M=30/7=4.29$
$k=2$ $G=5.11=55$ $E=55/M=55/13=4.23$

which is slower than the previous method.

Finally we may use this analysis to investigate the trade off between vector and scalar code.

As an example we reconsider algorithm R, that consists of 3 steps:

1 Vectorized cyclic reduction,
2 Scalar solution of reduced system,
3 Back substitution in the reduced equations,

and investigate, whether it would be advantageous to include step 3 in the scalar code of step 2. The back substitution generates the odd elements of the results by

$$x_{2i+1} = b_{2i+1}-a_{2i+1}\times x_{2i}$$

In vector mode 2 vector instructions of length N (control store capability) at a cost of $2N/p$ cycles are issued.

In order to incoporate this in step 2, we must add in step 2, for each equation of the reduced system, 2 loads and 2 floating point instructions with an issue time of 4 cycles.

Setting $E=F(I+R)/(F+R)=FR/(F+R)+IF/(F+R)$ we see that for each cycle added to I,E increases by $F/(F+R)=10/13$ cycles. Since the reduced system has $N/2$ equations the extra time in the scalar code is 20/13 N cycles, in contrast to the $2N/p$ cycles in vector mode. Thus inclusion of step 3 in step 2 gives a better performance for $p=1$ (one-pipe Cyber) and a poorer performance for $p>1$ (two or four pipe Cyber).

VI.2. Half Precision

In this section we estimate the execution time, for the algorithms described in Chapters III and IV, for half precision floating point operands.

Vector instructions on half precision operands require $1/2p$ cycles per operation rather than the $1/p$ cycles for full precision operands.

The swap instructions on half precision data require 1/4 cycle per swapped register rather than the 1/2 cycle required for full precision registers.

The start-up time for vector and swap instruction, as well as the issue and functional unit times for scalar instructions, are identical for both full and half precision operands. Two loads or stores of serially accessed elements of half precision vectors may be replaced by a single full word load or store. Thus for several accesses, load and stores are effectively twice as fast for half precision operands.

The algorithms containing DO-loops with load and/or store instructions must be redesigned for half precision. For all other algorithms the execution times are easily estimated. In the previous chapter we introduced the quantities:

I minimal issue time for the solution of one equation,
R minimal issue time for the reduction of one equation and,
F the minimal functional unit time for the solution of one equation.

R and F relate to scalar floating point instructions only and are therefore identical for full and half precision. I is different for half precision for those algorithms that use load and/or store instructions in the innermost loop. The only algorithms affected are M, N and P; we confine ourselves to $N3$ and P.

Algorithm $N3$ requires in its inner loop 2 loads, 2 floating point instructions and 1 store per equation.

For half precision data we require for each 2 equations 2 loads, 4 floating point instructions and 1 store, for a minimal issue time of 8 cycles: for one equation we thus have

$$I=4,\ F=10,\ R=3,\ X\neq 0$$

we set $X=G.C.D(F-I,\ F+R)=G.C.D(G.13)=1$ and compute M,T and E as before:

$M_0(F-I)=1$ modulo $(F+R)$
$M_0=1$ modulo 13, giving

$$M_0=11 \text{ and } T_0=((M_0x(F-I)+X)/(F+R)=5$$

and $N=11+13k, T=5+6l$.

For $k=0$ we find

$$E=G/M=F(M-T)/M=60/11=5.5$$

cycles per equation.

For algorithm P we have in the inner loop 1 load and 2 floating point instructions. In half precision we have for 2 equations 1 load and 4 floating point instructions, giving a minimal issue time of 5 cycles for 2 equations.

We then have per 2 equations

$$I=5,\ F=20,\ T=6.$$

Set $X=G.C.D(F-C,F+R)=G.C.D.(15,26)=1$ and solve M_0 from

$M_0(F-I)=X$ modulo $F+R$
$15M_0=1$ modulo $F+R$

This yields $M_0=7$, $T_0=(M_0*(F-I)-X)/(F+R)=104/26=4$, and hence

$$M=7+26k,\ T=4+15k.$$

For $k=0$: $G=F(M-I)=60$, $E=60/7=8.6$,
$k=1$: $G=F(M-T)=280$, $E=280/33=8.5$.

We will use an estimate of $8.6/2=4.3$ cycles per equation.

In table VI.1 we list the estimated execution times in full and half precision of a number of algorithms, for a 1,2 and 4 pipe Cyber 205, obtained by similar analysis.

Algorithm	No. of vec. op	Swap + gather		Scalar loop		over head	Execution time in cycles					
							1-pipe		2-pipes		4-pipe	
		full	half	full	half	full	half	full	half	half	full	half
J2	14						14	7	7	3.5	3.5	1.75
J3	11						11	5.5	5.5	2.75	2.75	1.38
K	5	4.2	4.2	0.4	0.3	0.2	9.9	7.4	7.4	6.15	6.15	5.53
L1	5			0.4	0.3	0.1	5.5	2.9	3.0	1.65	1.75	1.03
L2	4			0.4	0.3	0.1	4.5	2.4	2.5	1.4	1.5	0.9
N3	-			7.5	5.5	0.58	8.08	6.08	8.08	6.08	8.08	6.08
P	-	0.75	0.5	5.0	4.3	0.59	6.29	5.14	6.29	5.14	6.29	5.14
R	5	0.75	0.5	2.0	2.0	0.29	7.99	5.24	5.49	3.99	4.24	3.42

TABLE VI.1.

The column "No of vec. op" lists the number of operations in vector mode per equation. To determine the contribution to the execution time, the entries in this column must be divided by p for full precision and by $2p$ for single precision. The entries in the column headed "overhead" give the delays caused by memory bank conflicts.

VII. SUMMARY AND CONCLUSIONS

A number of algorithms for the solution of a bidiagonal system of linear equations on a Cyber 205 is analyzed and tested and the estimated and actual execution times are found to be in good accordance.

Unrolling, statement sequencing and other well known optimization techniques are applied to the straightforward solution of the problem, coded in standard FORTRAN. The performance improvements obtainable by means of hand optimization and compiler optimization options are measured and analyzed.

The key element to further optimization is cyclic reduction. Cyclic reduction first allows the solution algorithm to be at least partly vectorized. On the Cyber 205. The performance is hindered by the so-called stride problem, since peak performance requires contiguous vectors on the Cyber 205; the problem may be miligated by means of the compress and sparse vector instructions or by the application of a folding technique allowing fast compactification of sparse data at the cost of some increased computational complexity.

In scalar algorithms for recursive problems cyclic reduction is helpful, since it allows the computational parallelism in the scalar processor to be exploited.h

The swap instruction exchanges data between register file and memory up to four times faster than

load and store instructions.

A simple algorithm has been developed, by which the ratio of equations, that must be reduced to obtain the fastest scalar code, can be computed.

On a Cyber 205, equiped with a single vector pipe, the fastest algorithm is fully scalar. For two pipes a partially vectorization gives the best result and on a four pipe Cyber 205 a fully vectorized algorithm is required to obtain optimal performance.

In half precision mode the highest performance is achieved by partial vectorization for a one pipe Cyber and by full vectorization on a two or four pipe Cyber 205.

REFERENCES.

[1] *CDC Cyber 205 Model 205 Computer System Hardware Reference Manual.* Pub. no. 60256020 Revision E (01-10-86)

[2] *FORTRAN 200 Version 1, for use with CDC Cyber 200 Virtual Storage Operating System,* Reference Manual Pub.nr. 60480200 Revision D (03-30-84)

[3] H.A. VAN DER VORST and J.M. VAN KATS. *The Performance of Some Linear Algebra Algorithms on CRAY-1 and Cyber 205 Supercomputers.* Technical Report TR-17 (18-04-1984) ACCU-Reeks nr 38.

[4] P. DUBOIS, and G. RODRIGUE, (1977). An analysis of the recursive doubling algorithm. In: *Kuck et al.* (eds.), *High Speed Computer and Algorithm Organization,* Academic Press, New York.

[5] D. HELLER, (1978). Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems, SIAM J. Numer. Anal., 13 (4), pp. 484-496.

[6] H.H. WANG, (1981). A parallel method for tridiagonal equations, ACM Trans. on Math. Softw. 7 (2), pp. 170-183.

[7] J.J. LAMBIOTTE, and R.G. VOIGT, (1974). The solution of tridiagonal linear systems on the CDC-STAR-100 computer, ICASE Report, NASA Langley Research Center, Hampton, VA.

[8] H.S. STONE, (1973). An efficient parallel algorithm for the solution of a tridiagonal linear system of equations, JACM, Vol. 20 (1), pp. 27-38.

[9] S.C. CHEN, D.J. KUCK, and A.H. SAMEH, (1978). *Practical Parallel Band Triangular System Sovers,* ACM Trans. on Math. Softw. 4 (3), pp. 270-277.

[10] H.A. VAN DER VORST, (Januari 1988). *Parallel Solution of Bidiagonal Systems Coming from Discretised PDE's,* to appear in: IEEE Trans. on Magnetics, Januari 1988.

TABLE OF CONTENTS