Fully Vectorized Solver for Linear Recurrence Systems with Constant Coefficients

Przemysław Stpiczyński[1] and Marcin Paprzycki[2]

[1] Department of Computer Science, Marie Curie-Sklodowska University, Plac Marii Curie-Sklodowskiej 1, 20–031 Lublin, Poland, phone: +4881 5376102, fax: +4881 5333669, e-mail: przem@golem.umcs.lublin.pl

[2] Scientific Computing Program, University of Southern Mississippi, Hattiesburg, MS 39406-5106, USA, phone: 601-266-6639, fax: 601-266-6452, e-mail: marcin@orca.st.usm.edu

Abstract. We describe the use of BLAS kernels as a key to efficient vectorization of m-th order linear recurrence systems with constant coefficients. Applying the Hockney-Jesshope model of vector computation, we present the performance analysis of the algorithm which considers also the influence of memory bank conflicts. The theoretical analysis is supported by experimental data collected on two Cray vector computers.
Keywords. m-th order linear recurrence systems, BLAS, LAPACK, vectorization, memory bank conflicts, speedup.
Conference topics. Numerical methods, Parallel and distributed algorithms.

1 Introduction

The critical part of several numerical algorithms reduces to the solution of a linear recurrence system of order m for n equations with constant coefficients [13, 16]:

$$x_k = \begin{cases} 0 & \text{for } k \leq 0 \\ f_k + \sum_{j=1}^{m} a_j x_{k-j} & \text{for } 1 \leq k \leq n. \end{cases} \tag{1}$$

The efficient solution to this problem is of particular interest in case of vector computers as optimizing compilers are not able to generate machine code that would fully utilize the underlying hardware. As our experiments show, even Cray's Fortran compiler, usually recognized as the best vectorizing compiler on the market, is in this category (see Section 5). In addition, numerical libraries (like LAPACK [1], implemented in the Cray's **scilib** library) instead of problem (1) provide a solution to a more general problem:

$$x_k = \begin{cases} 0 & \text{for } k \leq 0 \\ f_k + \sum_{j=k-m}^{k-1} a_{kj} x_j & \text{for } 1 \leq k \leq n. \end{cases} \tag{2}$$

Solution to this problem requires more memory and, in the case of LAPACK routines, the computational efficiency is obtained primarily by solving it for

multiple right hand sides. In case when the original problem (1) is solved, a simple application of a LAPACK routine does not result in achieving maximum performance (see Section 5). The aim of our work is thus to find the performance-optimal solver for the original problem (1). Based on our earlier work [9, 10, 11, 14] we have decided to approach the problem by augmenting the *divide-and-conquer* approach proposed there by application of BLAS kernels. We then proceeded to establish the optimal parameters to obtain maximum efficiency and to eliminate memory bank conflicts.

We proceed as follows. In the next section we introduce the algorithmic framework used in our work. We follow with the description of implementation details of the proposed algorithm. We then sketch the theoretical analysis of computational complexity. We complete our report by describing and analyzing results of our experiments performed on Crays C-90 and SV-1.

2 Algorithm description

In our considerations we will assume that $n \gg m$, i.e. the order of a recurrence system is rather small. The idea of the algorithm is to rewrite (1) as the following block system of linear equations

$$\begin{pmatrix} L & & & \\ U & L & & \\ & \ddots & \ddots & \\ & & U & L \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_p \end{pmatrix} = \begin{pmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \vdots \\ \mathbf{f}_p \end{pmatrix}, \tag{3}$$

where for $q = n/p > m$ we have

$$L = \begin{pmatrix} 1 & & & & & \\ -a_1 & \ddots & & & & \\ \vdots & \ddots & \ddots & & & \\ -a_m & & \ddots & \ddots & & \\ & \ddots & & \ddots & \ddots & \\ & & -a_m \cdots -a_1 & & 1 \end{pmatrix}, \quad U = \begin{pmatrix} -a_m & \cdots & -a_1 \\ & \ddots & \vdots \\ & & -a_m \\ 0 & & \end{pmatrix} \in \mathbb{R}^{q \times q}. \tag{4}$$

Note that L is a Toeplitz matrix, what means that entries are constant along each diagonal. The system (3) corresponds to the following recurrence system

$$\begin{cases} \mathbf{x}_1 = L^{-1}\mathbf{f}_1 \\ \mathbf{x}_j = L^{-1}\mathbf{f}_j - L^{-1}U\mathbf{x}_{j-1} \text{ for } j = 2, \ldots, p. \end{cases} \tag{5}$$

To solve this system let us consider the structure of the matrix

$$U = -\sum_{k=1}^{m}\sum_{l=k}^{m} a_{m+k-l}\mathbf{e}_k\mathbf{e}_{q-m+l}^T, \tag{6}$$

where $\mathbf{e}_k$ denotes k-th unit vector of $\mathbb{R}^q$. Obviously, equation (5) reduces to the form

$$\begin{cases} \mathbf{x}_1 = L^{-1}\mathbf{f}_1 \\ \mathbf{x}_j = L^{-1}\mathbf{f}_j + \sum_{k=1}^{m} \alpha_j^k \mathbf{y}_k \text{ for } j = 2, \ldots, p \end{cases} \tag{7}$$

where $L\mathbf{y}_k = \mathbf{e}_k$ and $\alpha_j^k = \sum_{l=k}^{m} a_{m+k-l} x_{(j-1)q-m+l}$. Note that to compute vectors $\mathbf{y}_k$ we need to find only the solution of the system $L\mathbf{y}_1 = \mathbf{e}_1$, namely $\mathbf{y}_1 = (1, y_2, \ldots, y_q)^T$. We can now form vectors $\mathbf{y}_k$ as follows

$$\mathbf{y}_k = (\underbrace{0, \ldots, 0}_{k-1}, 1, y_2, \ldots, y_{q-k+1})^T. \tag{8}$$

This yields that the number of subsystems we must solve does not depend on the order of the system. To find vectors $\mathbf{z}_j$ and $\mathbf{y}_1$ we must solve $p+1$ recurrence systems of order m for q equations.

3 Implementation details

Now let us consider the possible implementations of the proposed algorithm. We can omit the assumption that $n = pq$ because after we choose integers p and q we can apply (7) to find $x_1, \ldots, x_{pq}$ and (1) to find $x_{pq+1}, \ldots, x_n$. First we have to find vectors $\mathbf{z}_j$ and $\mathbf{y}_1$. We can do it efficiently by using a sequence of `_AXPY` operations $\mathbf{y} \leftarrow \mathbf{y} + \alpha\mathbf{x}$. Note that `_AXPY` consists of $2N$ floating point operations and it can be computed in a simple loop of length N. So let us define matrices

$$Z = (\mathbf{z}_1, \ldots, \mathbf{z}_p, \mathbf{y}_1), \; F = (\mathbf{f}_1, \ldots, \mathbf{f}_p, \mathbf{e}_1) \in \mathbb{R}^{q \times (p+1)}$$

and denote $Z_{k,*}$ as a k-th row of Z. Now we can find the solution of the system $LZ = F$ using the formula

$$Z_{k,*} = \begin{cases} \mathbf{0} & \text{for } k \le 0 \\ F_{k,*} + \sum_{j=1}^{m} a_j Z_{k-j,*} & \text{for } 1 \le k \le q. \end{cases} \tag{9}$$

Initially columns of the matrix F can be stored in a one-dimensional array $\mathbf{x}$, so Z can be computed using the following code

```
do k=1,q
  do j=1,min(m,k-1)
    call saxpy(p+1,a(j),x(k-j),q,x(k),q)
  end do
end do
```

It can be easily calculated that the number of `_AXPY` operations is equal to $m(q - \frac{m+1}{2})$ and thus the total number of operations needed to find vectors $\mathbf{z}_j$ and $\mathbf{y}$ can be expressed as

$$C_1 = 2(p+1)m(q - \frac{m+1}{2}). \tag{10}$$

As soon as the matrix Z is calculated its last column ought to be copied to a new array y such that y(-m:0)=0.0.

```
call scopy(q,x(p*q+1),1,y(1),1)
do j=-m,0
  y(j)=0.0
end do
```

Now vectors $\mathbf{x}_j$, $j = 2, \ldots, p$, can be computed. For each vector we should compute coefficients α_j^k using the following code

```
do k=1,m
  call saxpy(m+1-k,a(m+1-k),x(q*(j-1)-m+k),1,alpha,1)
end do
```

and then find $\mathbf{x}_j$ using a sequence of _AXPY calls

```
do k=1,m
  call saxpy(q,alpha(k),y(2-k),1,x(q*(j-1)+1),1)
end do
```

The total number of floating-point operations in this part of the algorithm is

$$C_2 = 2(p-1)\left(\sum_{k=1}^{m}(m+1-k)+mq\right). \tag{11}$$

Now let us consider possible modifications of the proposed algorithm. First, observe that the last step of the algorithm can be implemented in terms of level 2 BLAS using one call of _GEMV. More precisely, when we form

$$W = (\mathbf{y}_1, \ldots, \mathbf{y}_m) \in \mathbb{R}^{q\times m} \tag{12}$$

then instead of the last loop above, we can use

```
call sgemv('N',q,m,1,w,ldw,alpha,1,1,x(q*(j-1)+1),1)
```

Note that the use of _GEMV requires additional space for qm entries of W.

Let us now observe that for finding Z we can consider the use of the routine _TBTRS from the LAPACK library [1] which solves a system $AX = B$ where A is a triangular banded matrix. Thus instead of the sequence of _AXPY calls based on (9) we would have the following LAPACK call

```
call stbtrs('L','N','U',q,m,p+1,ab,ldab,x,q,info)
```

We have to recall that this routine does not take into account the Toeplitz structure of the matrix L and requires additional space for $m+1$ diagonals of L, i.e. for $(m+1)q$ additional values.

In the table below we summarize algorithms that can be used to solve the original problem (1):

Algorithm	Description
Scalar	Scalar code based on a direct implementation of (1)
Algorithm 1A	The main algorithm based on calls to the _AXPY routine
Algorithm 1B	As Algorithm 1A but the last step is calculated by one call of the level 2 BLAS routine _GEMV
Algorithm 2	The system $LZ = F$ solved by a call to the LAPACK routine _TBTRS and the last step calculated by the call to _GEMV
Algorithm 3	LAPACK _TBTRS routine called for one RHS

4 Performance analysis

To study the performance of the algorithm let us consider the theoretical model of vector computations introduced by Hockney and Jesshope [6, 2].

The performance r_N of a loop of length N can be expressed in terms of two parameters r_∞ and $n_{1/2}$ which are specific for a kind of loop and vector computer. The first parameter represents the performance in Mflops for a very long loop, while the second the loop length for which a performance of about $r_\infty/2$ is achieved. Then

$$r_N = \frac{r_\infty}{n_{1/2}/N + 1} \text{ Mflops.} \tag{13}$$

This yields that the execution time of `_AXPY` is

$$T_{AXPY}(N) = \frac{2N}{10^6 r_N} = \frac{2 \cdot 10^{-6}}{r_\infty}(n_{1/2} + N) \text{ seconds.} \tag{14}$$

From (10), (11) and (14) we get that the total execution time of our algorithm can be estimated as follows

$$T(p,q) = \frac{2 \cdot 10^{-6}}{r_\infty} m \left(2pq + 2n_{1/2}p + n_{1/2}q - 2.5n_{1/2} - 0.5n_{1/2}m - m - 1\right),$$

where $n = pq$. It can be easily verified that $T(p,q)$ reaches its minimum at the point

$$(p,q) = (\sqrt{n/2}, \sqrt{2n}). \tag{15}$$

Thus the optimal choice of p and q depends only on the problem size n and because these numbers should be integers we choose $q = \lfloor\sqrt{2n}\rfloor$ and $p = \lfloor n/q \rfloor$. Here, the last $n - pq$ elements of the solution $\mathbf{x}$ can be computed by a scalar algorithm based on (1).

Sometimes these chosen parameters have to be adjusted to avoid memory bank conflicts. Vector computers usually store data so that contiguous words (e.g. elements of arrays) are in separate memory banks. Usually the number of banks in the memory system is a power of two. Memory bank conflicts may occur when an array's stride (the difference in the index between two successive iterations) is a multiple of a power of two. Then the memory cannot be efficiently

used because CPU must wait until a former memory request to the same bank is completed. Thus to avoid memory bank conflicts the parameter q should be chosen as follows

$$q = \begin{cases} \lfloor\sqrt{2n}\rfloor - 1 & \text{if } \lfloor\sqrt{2n}\rfloor \text{ even,} \\ \lfloor\sqrt{2n}\rfloor & \text{otherwise.} \end{cases} \tag{16}$$

Finally let us calculate the number of floating point operations performed by the algorithm. Adding C_1 and C_2 defined by (10) and (11), and the number of flops required for finding the last $n - pq$ entries of the solution we get

$$C_{n,m}(p,q) = C_1 + C_2 + m(n - pq - \frac{m+1}{2}) = 3mpq - \frac{5}{2}m(m+1) + mn. \tag{17}$$

5 Results of experiments

The method has been implemented in FORTRAN and tested on a single processor of the Cray C-90 and SV-1 vector computers. We have used the optimized versions of BLAS and LAPACK available in the **scilib** library. Each algorithm was tested varying the problem sizes n and m and values of parameter q. CPU time was measured using the *second* function and the presented results represent the best values from multiple runs.

Figures 1 and 2 illustrate the dependency between the performance of Algorithm 1A and the value of parameter q for $m = 1$ and $n = 64000$ and $n = 1024000$ respectively. Results for both Cray's are reported in $Mflops$.

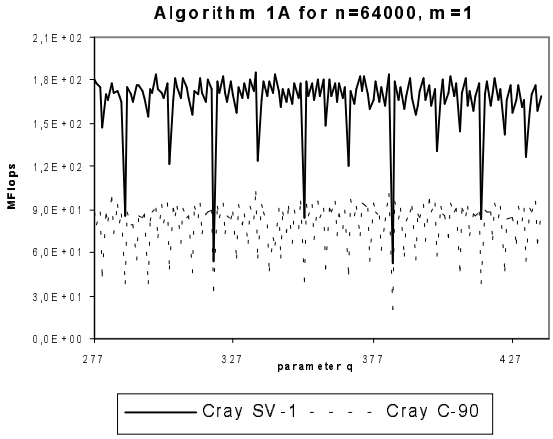

Fig. 1. Performance of Algorithm 1A for various values of q

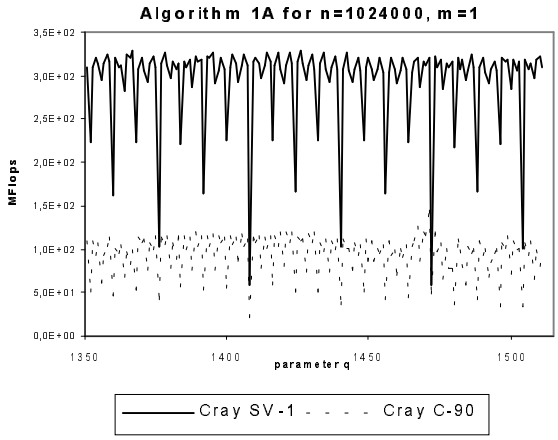

Fig. 2. Performance of Algorithm 1A for various values of q

It was shown above (see Section 4) that the optimal value of the parameter q depends only on the size of the problem n. Our experiments support this claim and show that this result holds for both machines (the optimal value is the same on both Crays) even though they have different characteristic parameters r_∞ and $n_{1/2}$. The experimental optimal value of q has been found to be in close proximity of the theoretically predicted one (excluding values which are powers of 2 for which the memory bank conflicts affect performance). Thus, in computational practice, the theoretically predicted optimal value of q can be used to implement the code.

Figures 3 and 4 depict the relationship between the performance (in *Mflops*) and the size of the problem n and the order of the recurrence m (for these experiments the theoretically predicted optimal value of q was used). In Figure 3 we report the results for $n = 64000$ and $m = 1, 2, \ldots, 6$ for both Crays and all five algorithms. In Figure 4 we present similar results for $n = 1024000$.

First, let us observe that the qualitative behavior of the five algorithms is the same for both machines and is independent of the problem size n.

For $m = 1$ the Algorithm 1A is the most efficient. For Algorithms 2 and 3 a performance dip manifests itself for $m = 2$. Starting from $m = 2$ further increase in m results in the performance increase. Interestingly, for all values of m, Algorithms 2 and 3 which utilize LAPACK library routine `_TBTRS` are substantially less efficient than Algorithms 1A and 1B and only barely more efficient than the Scalar code.

As m increases, Algorithm 1B outperforms Algorithm 1A. This can be explained as an effect of the application level 2 BLAS matrix-vector multiplication _GEMV.

Finally, note that the performance of the two Crays depends on the problem

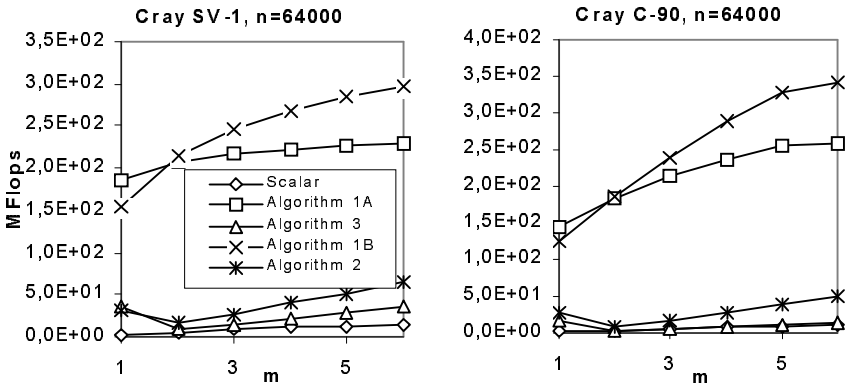

Fig. 3. Performance of the algorithms for various m

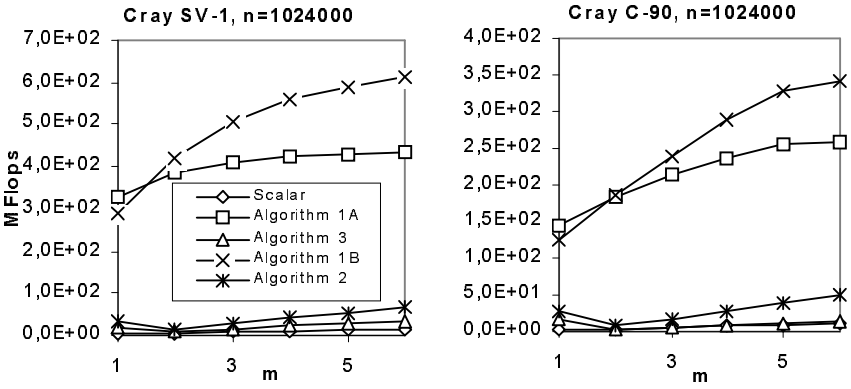

Fig. 4. Performance of the algorithms for various m

size (n). For small n Cray C-90 matches the performance of the newer SV-1 (for $m = 6$ it even outperforms it slightly). The situation changes radically for $n = 1024K$. Here, the Cray SV-1 is almost twice as fast as the Cray C-90.

We believe that from the point of view of the user one of the more interesting parameters is the speedup of the "fancy" algorithms over the basic Scalar approach. We illustrate this aspect of the problem in Figures 5 and 6. Here we report the speedup as the function of the problem size n for both machines for $m = 1$ and $m = 4$ respectively. As previously, the optimal theoretical value q was used for algorithms 1A, 1B and 2.

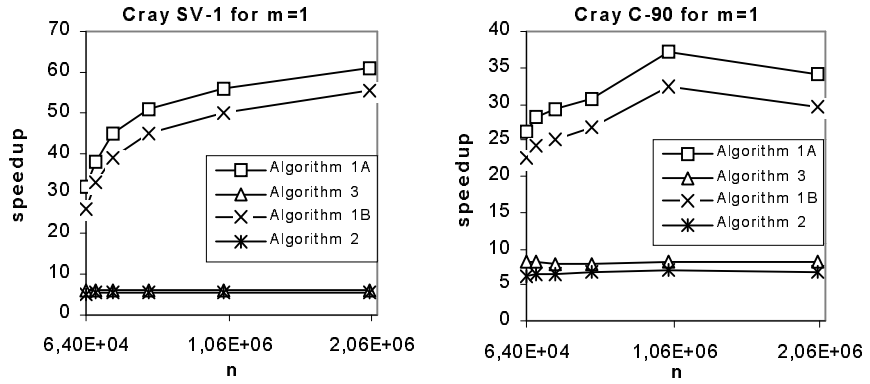

Fig. 5. Speedup of the algorithms for various n

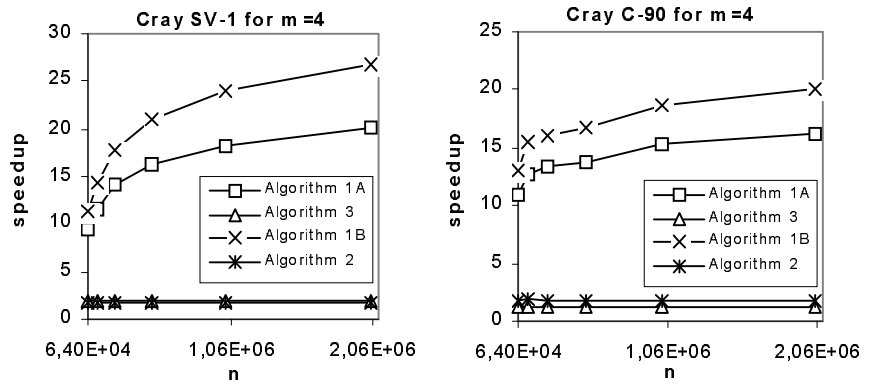

Fig. 6. Speedup of the algorithms for various n

As previously, the results are qualitatively similar for both machines. In all cases (independently of n) Algorithms 2 and 3 do not result in a significant speedup over the Scalar approach. Interestingly, while as n increases (for a fixed m, speedup of Algorithms 1A and 1B over Scalar increases, as m increases (for a given m) the speedup decreases. This indicates that the code generated by the compiler from the Scalar algorithm for increasing m results in improved efficiency.

Finally let us summarize the results of experiments

– Algorithms 1A and 1B achieve the best performance for values of the param-

eter q close to the theoretical optimal value. The optimal choice of q depends only on the problem size (and memory bank conflicts).
- The use of Algorithm 1B instead of 1A is profitable when $m > 2$. This is caused by the use of the level 2 BLAS routine `_GEMV`. However, use of `_GEMV` requires additional space for qm entries of W.
- The speedup of Algorithms 1A and 1B over the Scalar code increases when the problem size n increases and decreases when the order of the system m increases.
- The MFlop performance increases when the problem size n increases as well as when the order of the system m increases.
- When $q = a2^k$ (for integer a, k), performance rapidly decreases. Increase in the value of k results in further substantial performance degradation. This is the effect of memory bank conflicts.
- The performance of Algorithm 2 and 3 is rather poor and the algorithms require additional space. This is a result of the fact that the _TBTRS routine from LAPACK solves more general problem (2) and does not utilize the special Toeplitz structure of the matrix L.
- For first order linear recurrences ($m = 1$) Algorithm 1A is approximately six times faster then the Algorithms 2 and 3 which use _TBTRS routine from LAPACK and for large n achieves speedup up to 60 against the Scalar algorithm based on (1).

6 Acknowledgments

Computer time grants from MCSR in Oxford, Mississippi and NPACI in Austin, Texas are kindly acknowledged. We would also like to express our gratitude to the anonymous referee who helped us improve the paper.

References

1. E. Anderson, Z. Bai, C. H. Bischof, J. W. Demmel, J. J. Dongarra, J. J. Du Croz, A. Greenbaum, S. J. Hammarling, A. McKenney, S. Ostruchov and D. C. Sorensen, *LAPACK User's Guide* (SIAM, Pfiladelphia, 1992).
2. J. Dongarra, I. Duff, D. Sorensen and H. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers* (SIAM, Pfiladelphia, 1991).
3. J. Dongarra and L. Johnsson, Solving Banded Systems on Parallel Processor. *Parallel Comput.* 5(1987) 219–246.
4. D. Heller, A survey of parallel algorithms in numerical linear algebra. *SIAM Review* 20(1978) 740–777.
5. A.C. Greenberg, R.E.Lander, M.S. Paterson and Z. Galil, Efficient parallel algorithms for linear recurrence computation, *Inf. Proc. Letters* 15(1982) 31–35.
6. R. Hockney and C. Jesshope, *Parallel Computers: Architecture, Programming and Algorithms* (Adam Hilger Ltd., Bristol, 1981).
7. D.J. Kuck, *Structure of Computers and Computations* (Wiley, New York, 1978).

8. J. Modi, *Parallel Algorithms and Matrix Computations* (Oxford University Press, Oxford, 1988).
9. M. Paprzycki and P. Stpiczyński, Solving linear recurrence systems on a Cray Y-MP, in: J.Dongarra, J. Waśniewski eds., Lecture Notes in Computer Science 879, (Springer–Verlag, Berlin, 1994) 416–424.
10. M. Paprzycki and P. Stpiczyński, Solving linear recurrence systems on parallel computers, *Proceedings of the Mardi Gras '94 Conference*, Baton Rouge, Feb. 10–12, 1994 (Nova Science Publishers, New York, 1995) 379–384.
11. M. Paprzycki and P. Stpiczyński, Parallel solution of linear recurrence systems. *Z. Angew. Math. Mech.* 76(1996) Suppl. 2, 5-8.
12. A.H. Sameh and R.P. Brent, Solving triangular systems on a parallel computer. *SIAM J. Numer. Anal.* 14(1977) 1101–1113.
13. J. Stoer and R. Bulirsh, *Introduction to Numerical Analysis* (Springer, New York, 2nd ed., 1993).
14. P. Stpiczyński, Parallel algorithms for solving linear recurrence systems, in: L. Bougé et al. eds., Lecture Notes in Computer Science 634, (Springer–Verlag, Berlin, 1992) 343–348.
15. P. Stpiczyński, Error analysis of two parallel algorithms for solving linear recurrence systems, *Parallel Comput.* 19(1993) 917–923.
16. P. Stpiczyński and M. Paprzycki, Parallel algorithms for finding trigonometric sums, in: D.H. Bailey et al. eds., *Parallel Processing for Scientific Computing*, (SIAM, Philadelphia, 1995) 291–292.
17. H. A. Van Der Vorst and K. Dekker, Vectorization of linear recurrence relations, *SIAM J. Sci. Stat. Comput.*, 16(1989) 27-35.

This article was processed using the LaTeX macro package with LLNCS style