Parallel Computing 17 (1991) 763–778
North-Holland

Practical aspects and experiences

A class of Lanczos-like algorithms implemented on parallel computers

S.K. Kim and A.T. Chronopoulos

Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455, USA

Received February 1990
Revised November 1990

Abstract

Kim, S.K. and A.T. Chronopoulos, A class of Lanczos-like algorithms implemented on parallel Computers, Parallel Computing 17 (1991) 763–778.

The Lanczos algorithm is most commonly used in approximating a small number of extreme eigenvalues and eigenvectors for symmetric large sparse matrices. Main memory accesses for shared memory systems or global communications (synchronizations) in message passing systems decrease the computation speed. In this paper, the standard Lanczos algorithm is restructured so that only one synchronization point is required; that is, one global communication in a message passing distributed-memory machine or one global memory sweep in a shared-memory machine per each iteration is required.

We also introduce the *s*-step Lanczos method for finding a few eigenvalues of symmetric large sparse matrices in a similar way to the *s*-step Conjugate Gradient method [2], and we prove that the *s*-step method generates reduction matrices which are similar to reduction matrices generated by the standard method. One iteration of the *s*-step Lanczos algorithm corresponds to *s* iterations of the standard Lanczos algorithm. The *s*-step method has improved data locality, minimized global communication and has superior parallel properties to the standard method. These algorithms are implemented on a 64-node NCUBE/seven hypercube and a CRAY-2, and performance results are presented.

Keywords. Lanczos algorithm; eigenvalue problem; symmetric sparse matrices; multiprocessor systems; CRAY-2; NCUBE.

1. Introduction

It is now feasible to achieve high performance in numerical computations with parallel processing technology. In order to use parallel computers in a specific application, algorithms need to be developed and mapped onto a parallel computer architecture [1,2,3]. Parallel computers consist of many processors communicating through an interconnection network. Depending on the structure of the memory system, two extremes of parallel computers are shared-memory multiprocessors, in which all memory modules are equally accessible to all

The research was partially supported by University of Minnesota Graduate School grant 0350-210407, and NSF grants CER DCR-8420935 and CCR-8722260. The Minnesota Super Computing-Institute provided time on Cray-2.

0167-8191/91/$03.50 © 1991 – Elsevier Science Publishers B.V. All rights reserved

Table 1
Serial and parallel complexity of Lanczos part

Operation	Sequential	Parallel
Vector update	$O(N)$	$O(1)$
Inner products	$O(N)$	$O(\log_2 N)$
Matrix–vector products	$O(n_d N)$	$O(\log_2 n_d)$

processors, and distributed-memory parallel processors, in which each memory module is physically associated with each processor. The significant difference between the two types is that a distributed-memory parallel computer has no shared memory. Consequently, to use data in a remote memory it is necessary to explicitly get the data from that remote memory. This and all other interprocessor communication is done by passing messages among the processors.

Memory contention on shared memory machines constitutes a severe bottleneck for achieving their maximum performance. The same is true for communication costs on a message passing system. It would be desirable to have methods for specific problems which have low communication costs compared to the computation costs. This is interpreted as a small number of global memory accesses for the shared memory systems and a small number of global communications for the message passing systems. Also, we consider the design issues involved in partitioning and mapping data parallel algorithms [12]. Data parallel algorithms are suitable for problems with a large volume of data. Parallelism is achieved by partitioning the data set rather than partitioning the control of the program. The algorithm must be designed so that both computation and data can be distributed to the processors in such a way that computations can be run in parallel, balancing the loads of the processors.

Many important scientific and engineering problems require the computation of a small number of eigenvalues of symmetric large sparse matrices. The most commonly used algorithm for solving such an eigenvalue problem is the Lanczos algorithm. The Lanczos algorithm has three basic types of operations: matrix–vector products, inner products and the vector updates. Let A be a sparse matrix of dimension N. The sequential and parallel complexity [3] of these computations is shown in *Table 1* (n_d = the number of nonzero diagonals for a sparse banded matrix A). In *Table 1* the parallel system is assumed to have $O(N)$ processors. From *Table 1* we can draw the conclusion that for massively parallel systems the inner products may be the slowest of the three operation types in the Lanczos method. The same might be true for hypercube systems if the communication delay is much longer than the floating point operation execution time. Thus, grouping together for execution the inner products of each iteration in the Lanczos method may lead to a speed up on this type of computer.

On shared memory systems with a memory hierarchy such as the CRAY-2 the data locality of the computations is very important in achieving high execution speed. A good measure of the data locality of a comparison is the size of the

$$Ratio = (Memory\ References)/(Floating\ Point\ Operation).$$

The data locality of a computation is good if this ratio is as low as possible. This ratio being low implies that data can be kept for 'a long time' in fast registers or local memories and many operations can be performed on them. Thus restructuring the three types of operations in the Lanczos method may lead to a speed up on shared memory systems with a memory hierarchy.

In this paper we also introduce the s-step Lanczos method. In the s-step method s consecutive steps of the standard method are performed simultaneously. This means, for example, that the inner products (needed for s steps of the standard method) can be performed simultaneously and the vector updates are replaced by linear combinations. In the s-step Lanczos method, the computational work and storage are increased slightly compared to the

standard one, however the parallel properties and data locality are improved and the s-step method is expected to have only one global communication for one s-step iteration. (i.e. the $2s$ inner products are executed simultaneously.)

In Section 2 we discuss the standard Lanczos algorithm. In Sections 3 and 4 we describe the NCUBE and the CRAY-2 in detail and discuss how to implement efficiently the Lanczos algorithm. No reorthogonalization is used in this paper. In Section 5 we restructure the Lanczos algorithm to gain better performance. In Section 6 we develop the s-step method which is the new version of the Lanczos method. In Section 7 we give numerical examples and present experimental results on the NCUBE/7 hypercube and Cray-2.

2. The Lanczos method

The Lanczos algorithm for computing extreme eigenvalues of symmetric matrices is based on the Lanczos recursion for tridiagonalization of a real symmetric matrix [4,7,14]. The basic Lanczos procedure can be viewed as the Gram–Schmidt orthogonalization of the Krylov subspace basis $\{q_1, Aq_1, \ldots, A^{j-1}q_1\}$. Furthermore, for each j,

$$T_j = Q_j^{\mathrm{T}} A Q_j \tag{2.1}$$

is the orthogonal projection of A onto the subspace spanned by the Lanczos vectors $Q_j = \{q_1, \ldots, q_j\}$ such that $Q_j^{\mathrm{T}} Q_j = I_j$ where I_j is the identity matrix of order j. The eigenvalues of the Lanczos matrices T_j are called Ritz values of A in Q_j. For many matrices and for relatively small j several of the extreme eigenvalues of A, that is several of the algebraically-largest or algebraically-smallest of the eigenvalues of A, are well approximated by eigenvalues of the corresponding Lanczos matrices. The Ritz vector $Q_j y (= z)$ obtained from an eigenvector y of a given T_j is an approximation to a corresponding eigenvector of A. The standard Lanczos algorithms is as follows:

Algorithm 2.1. The Lanczos Algorithm.
Choose q_1 with $\| q_1 \| = 1$, $q_0 = 0$.
For $j = 1$ **until Convergence Do**
1. Compute and store Aq_j.
2. Compute (Aq_j, q_j).
3. $\alpha_j = (Aq_j, q_j)$.
4. $r_j = Aq_j - \beta_{j-1} q_{j-1} - \alpha_j q_j$.
5. Compute (r_j, r_j).
6. $\beta_j = \sqrt{(r_j, r_j)}$.
7. $q_{j+1} = r_j / \beta_j$.

EndFor

Let T_j be the tridiagonal matrix at step j

$$T_j = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \bullet & \bullet & \bullet & \\ & & \bullet & \bullet & \bullet \\ & & & \beta_{j-1} & \alpha_j \end{bmatrix}.$$

These matrices explicitly satisfy the equation

$$AQ_j = Q_j T_j + r_j e_j^{\mathrm{T}}. \tag{2.2}$$

where $e_j^T = (0, 0, \ldots, 0, 1)$ is a j-dimensional vector. The simplest a posteriori bound on the accuracy of a Ritz value λ is obtained from the residual norm of the associated Ritz vector z, $\|Az - z\lambda\|$. It is possible to estimate the residual norm of the Ritz vector without computing the Ritz vector [14]. In fact,

$$\|Az_i - z_i\lambda_i\| = \beta_j |s_{ji}| \quad \text{for} \quad i = 1, \ldots, j, \tag{2.3}$$

where s_{ji} is the last element of the ith eigenvector of T_j. This is used as a stopping criterion. The following step is added after 6 in Algorithm 2.1.

6′. compute β_j, s_{ji}. If $\beta_j |s_{ji}| < \epsilon$ then stop.

The quantity β_j is calculated by the algorithm, and since j is usually very small compared to n the calculation of s_{ji} is relatively inexpensive. Also the calculation of s_{ji} can be performed in parallel with β_j. Thus 6′ does not add an extra synchronization point.

The Lanczos vectors $q_1, q_2, q_3, \ldots$ lose their mutual orthogonality as the number of steps increases. To obtain good numerical accuracy, the algorithm has to be augmented with an expensive reorthogonalization step [4,14]. For the explicit orthogonalization of q_{j+1} against all previous Q's, the following step is added after 4 in Algorithm 2.1.

4′. $r_j = r_j - q_i(q_i^T r_j)$, $i = j, j-1, \ldots, 2, 1$.

We have to keep all previous Lanczos vectors in accessible computer storage through the entire Lanczos computation with reorthogonalization because we use them in a direct reorthogonalization or use them to generate particular Ritz vectors which are used in the reorthogonalization. Paige showed that it is possible to obtain accurate eigenvalues by using the method in an iterative manner using no reorthogonalization [13]. A Lanczos procedure is said to be iterative if within the Lanczos procedure a sequence of Lanczos matrices is generated, each of which corresponds to a different starting vector. The Lanczos procedure with no reorthogonalization has minimal storage requirements and can therefore be used on very large matrices. The memory requirement is two vectors of length N, the order of the given matrix A, if only approximations to eigenvalues are required. The storage requirement increases if some eigenvectors are also needed, but sometimes none and often only one or two are required.

3. Parallelism on the NCUBE hypercube system

3.1. A hypercube computer

A hypercube model is a particular example of a distributed-memory message passing parallel computer. In a hypercube of dimension d there are 2^d processors. Assume that these are labeled $0, 1, \ldots, 2^d - 1$. Two processors i and j are directly connected iff the binary representation of i and j differ in exactly one bit. Each edge of *Fig. 1* represent a direct connection of a

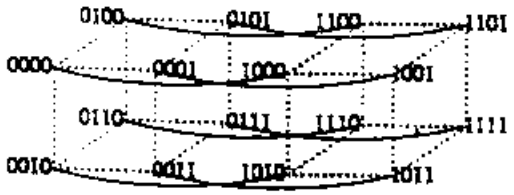


Fig. 1. Dimension 4 hypercube.

Table 2
Computation and communication time on NCUBE/7 hypercube

Operation	time (μs)	Comm./Comp.
8 byte transfer	470	
8 byte real addition	11.2	42 times
8 byte real multiplication	14.7	32 times

dimension 4 hypercube (lines and dotted lines are communication links). Thus, in a hypercube of dimension d each processor is connected to d others and 2^d processors may be interconnected such that the maximum distance between any two is d.

Table 2 shows a summary of an experimental study on interprocessor communication time and time to perform arithmetic operations on the NCUBE [15]. We see that an 8 byte message transfer between two directly connected processors takes 42 times the time for an 8 byte real addition and 32 times that of an 8 byte real multiplication. Furthermore, longer messages are transferred at a higher rate (i.e. bytes per second) than shorter ones going the same distance. In a linear time model of nearest-neighbor communication a message of length M bytes requires approximately $446.7 + 2.4M$ μs [15], where the constant term is a startup time which consists of the software overhead at each end and the time to set up the circuit, and the second term is the actual transmission time. The startup time is the dominating factor in the communication cost for short messages on the NCUBE.

In a hypercube with a high communication latency, the algorithm designer must structure the algorithm so that large amounts of computation are performed between communication steps: an algorithm requiring frequent and random exchange of messages will not perform well.

The two main issues in programming this machine are load balancing and reduction of communication overhead. A program is load balanced if all the processors are busy all the time. If one processor has most of the work then the others will end up being idle most of the time and the program will run inefficiently. To arrive at an efficient algorithm for a hypercube one needs to consider that both computations and data should be distributed to the processors in such a way that computational tasks can be run in parallel, balancing the computational loads of the processors as much as possible.

3.2. Mapping the Lanczos algorithm on the NCUBE

The Lanczos algorithm has three basic types of operations: matrix–vector products, inner products and the vector updates. The matrix vector multiplication can be performed concurrently by distributing the rows of A and the corresponding elements of vectors among processors of the NCUBE (*Fig. 2*). We divide the vector length N by the number of processors P. Each processor gets a subvector. When P does not divide n exactly, one processor gets a

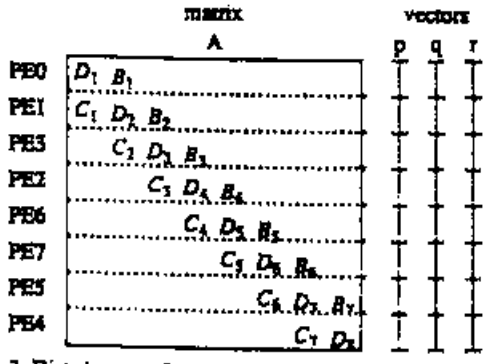


Fig. 2. Distribution of matrix and vectors to each processor.

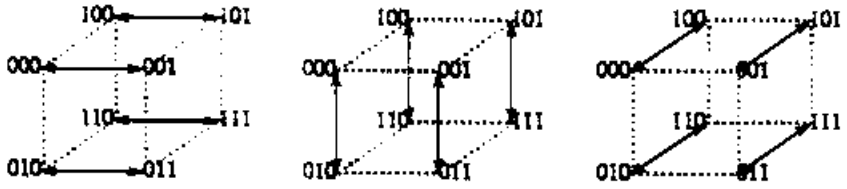


Fig. 3. Inner product process on hypercube.

shorter vector. The simplest representations of an $N \times N$ matrix are row or column oriented [11]. In the contiguous-row representation, each row is stored entirely in one processor. During a distributed matrix–vector product computation, processors need the most recently updated values of the vector elements which are mapped to neighbor processors, if the coefficient matrix A is sparse and banded. In *Fig. 2* D_i is a tridiagonal matrix and B_i, C_i are diagonal matrices. Here, only nearest-neighbor communication (i.e. local interprocessor communication) is required. The distributed vector updates can be performed concurrently without interprocessor communication only after each processor has computed the two required global scalar values in the Lanczos method.

For uniprocessors or multiprocessors with a small number of processors (e.g. four or eight processors) the matrix–vector products dominate the computation, whereas on parallel computer systems having the hypercube interconnection network the inner products dominate because they require global communication (synchronization of all processors) of the system. An inner product is computed by assigning an equal part of the vector (if possible) to each node. This allows each processor to work on local segments independent of other processors for most operations. Each node computes in parallel the sum of squares of its part of the vector. Then we use the Exchange–Add algorithm [1,9]. Processors $P_{(z_{d-1},\ldots,z_{i+1},0,z_{i-1},\ldots,z_0)}$, $(i = 0,\ldots,d-1)$ concurrently exchange their most recent partial sum with their neighbor $P_{(z_{d-1},\ldots,z_{i+1},1,z_{i-1},\ldots,z_0)}$ and then concurrently from their new partial sum. At the end of $2d$ concurrent nearest-neighbor communication steps, each processor has its own copy of the inner product. The Exchange–Add algorithm is illustrated in *Fig. 3* ($d = 3$).

4. Parallelism on the CRAY-2

The CRAY-2 is an example of a shared-memory four processor computer with memory hierarchy. All processors have equal access to a very large central memory of 256 Megawords and each processor has its own local memory. Each CRAY-2 processor has 8 vector registers (each 64 words long) and has data access through a single path between its vector registers and main memory. Each processor has 16 Kwords of local memory with no direct path to central memory but with a separate data path between local memory and its vector registers and the six parallel vector pipelines: the common memory to vector register, the vector register to local memory, the floating additive/subtractive, the floating multiplicative/divisible, the integer additive/subtractive and the logical pipeline. It is possible to design assembly language kernels which exhibit a performance commensurate with the 4.2 ns cycle time of the CRAY-2 if the computations allow it. This means that a rate of 459 Megaflops is possible on one processor if all arithmetic pipes can be kept busy [5]. The combination of fast cycle time, a local memory and a large central memory makes the CRAY-2 an exciting choice of computer for use in large-scale scientific computation.

The maximum performance of the CRAY-2 for specific applications comes from data movement minimization, good vectorization and division into multiprocessing tasks. Because of

the single paths between vector register and central or local memory on the CRAY-2 system, memory transfers constitute a severe bottleneck in achieving maximum performance. Therefore, minimization of data movement results in faster execution times. Algorithms must provide good data locality.

Macrotasking (often called multitasking on a data parallel algorithm is most often applied to parallel work found in the independent iterations of DO loops. If the loop has N iterations we map the N iterations into P processors or tasks so that each task has the same amount of work to do. To achieve load balancing, we must consider static and dynamic partitioning. We use static partitioning when the times for each of the loop iterations are approximately equal.

Example of contiguous static partitioning with $P = 4$:

processor	assigned iterations
P0	$I = 1, N/4$
P1	$I = (N/4) + 1, 2N/4$
P2	$I = 2(N/4) + 1, 3N/4$
P3	$I = 3(N/4) + 1, N$

The distribution of matrix and vectors is similar as in *Fig. 2* with 4 processors. The extremely large memory of the CRAY-2 means that many jobs can usually be resident in the main storage at the same time, leading to very efficient multiprogramming. We must minimize calls to the multitasking library because multitasking introduces an overhead that increases CPU time.

5. The modified Lanczos method

In the standard Lanczos algorithm iteration (algorithm 2.1), the inner products cannot be performed in parallel. Algorithms based on restructuring the standard Lanczos algorithm to decrease the global communication cost and to get better performance in distributed-memory message passing systems are introduced here.

For shared-memory systems with few processors processor synchronization is fast but accessing data from the main memory may be slow. Thus the data localities of the three basic operation parts of the Lanczos algorithm determine the actual time complexity of the algorithm. The data locality of the modified Lanczos algorithm is better than that of the standard algorithms. The conjugate gradient algorithm for solving symmetric and positive definite linear systems has the same shortcomings for parallel processing as the Lanczos algorithm. Examples of parallel conjugate gradient algorithms are discussed in [1,2].

In Algorithm 2.1, step 3 and step 6 are two synchronization points because α_j (or β_j) must be computed before the vector r_j (or $q_j + 1$) computation. This forces double access of vectors q, r and Aq from the main memory at each standard lanczos iteration.

Algorithm 5.1. The modified Lanczos algorithm.
Choose r_0 with $r_0 \neq 0$, $q_0 = 0$.
For $j = 0$ **until** Convergence **Do**
1. Compute and store Ar_j.
2. Compute (Ar_j, r_j), (r_j, r_j).
3. $\beta_j = \sqrt{(r_j, r_j)}$.
4. $\alpha_{j+1} = (Ar_j, r_j)/(r_j, r_j)$.
5. $q_{j+1} = r_j/\beta_j$.
6. $r_{j+1} = Ar_j/\beta_j - \beta_j q_j - \alpha_{j+1} q_{j+1}$.

EndFor

Algorithm 5.1 is a variant of Algorithm 2.1 and the orthonormal vectors q_j are generated in the same way as the standard Lanczos method. Paige considered the numerical properties of a few variants of the Lanczos algorithm [13]. Algorithm 2.1 is stable numerically because β_j is calculated by computing the norm of residual vector r_j not by computing $q_j^T A q_{j+1}$ [13]. Computationally the difference between Algorithms 2.1 and 5.1 is the computation of α_j and r_j. The computation of β_j is the same in Algorithms 2.1 and 5.1. Some loss of orthogonality from a very small residual vector is unavoidable in any of the algorithms [13]. We need one more vector operation to compute r_j in Algorithm 5.1. However, Algorithm 5.1 seems more promising for parallel processing because the two inner products required to advance each iteration can be executed simultaneously. Also, one memory sweep through the data is required to complete each iteration allowing better management of slower memories in a memory hierarchy computer.

If reorthogonalization is used in Algorithms 2.1 and 5.1 an additional synchronization point is required in each algorithm. But this does not affect the fact that the two inner products to compute α_j and β_j can be executed simultaneously in Algorithm 5.1, and the reorthogonalization step is added after step 6 of Algorithm 5.1 in a similar way to Algorithm 2.1. The stopping criterion requires the computation of β_j and eigenvectors of the reduced matrix T_j. Each processor of a parallel computer has T_j and computes the stopping criterion after computation of β_j in Algorithm 2.1 or computation of β_j and α_{j+1} in Algorithm 5.1. Therefore an additional synchronization point is not required in Algorithms 2.1 and 5.1.

In the next Section we propose an s-step Lanczos algorithm, which executes simultaneously in a certain sense s consecutive steps of Algorithm 2.1.

6. s-step Lanczos algorithm

One way to obtain an s-step Lanczos algorithm is to use the s linearly independent vectors $(v_k^1, Av_k^1, \ldots, A^{s-1}v_k^1)$ in building the Lanczos vector sequence. Let us denote by $\overline{V}_k$ the vector set $(v_k^1, v_k^2, \ldots, v_k^s)$. The subspace $\overline{V}_k$ is spanned by $(v_k^1, Av_k^1, \ldots, A^{s-1}v_k^1)$, so that $\overline{V}_k$ is made orthogonal to all preceding subspaces $\overline{V}_{k-1}, \overline{V}_{k-2}, \ldots, \overline{V}_1$.

Each subspace $\overline{V}_k$ can be decomposed into $\overline{Q}_k * \overline{R}_k$, where $\overline{Q}_k$ is an orthonormal basis of $\overline{V}_k$ and $\overline{R}_k$ is an $s \times s$ upper triangular matrix.

Remark 6.1. Let $\overline{V}_{i_1}$ be orthogonal to $\overline{V}_{i_2}$ for $i_1 \neq i_2$. Then $V_k = (\overline{V}_1, \overline{V}_2, \ldots, \overline{V}_k)$ can be decomposed into $\overline{Q}_k * R_k$ where $\overline{Q}_k = (\overline{Q}_1, \overline{Q}_2, \ldots, \overline{Q}_k)$ and $R_k = \mathrm{diag}(\overline{R}_1, \overline{R}_2, \ldots, R_k)$.

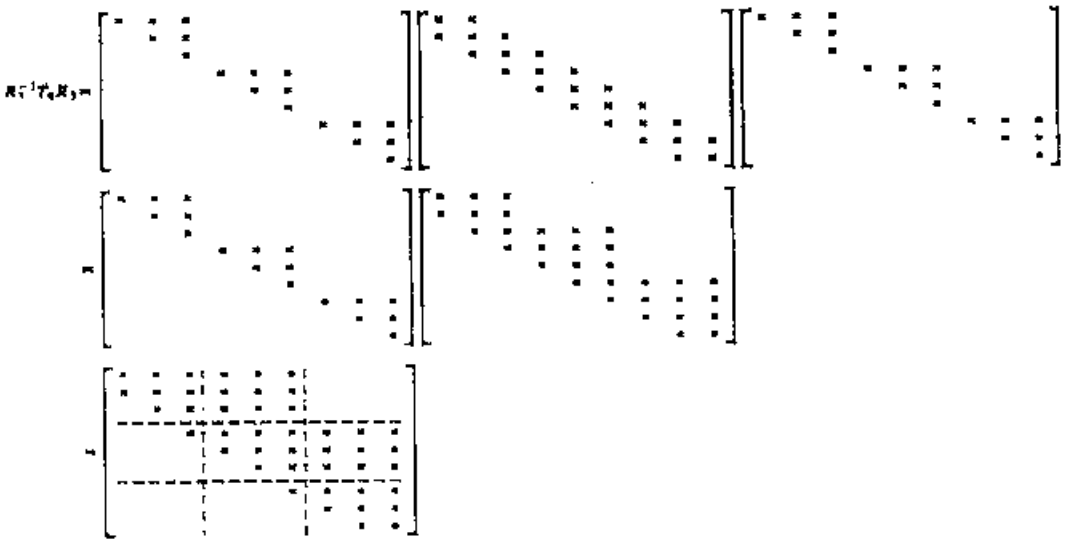
Proposition 6.1. *Let $\overline{T}_j$ be a symmetric tridiagonal matrix and $\overline{T}_k = R_k^{-1} \overline{T}_j R_k$ for $j = sk$. Then $\overline{T}_k$ is similar to the matrix $\overline{T}_j$ and $\overline{T}_k$ is a block tridiagonal matrix of the form:*

$$\overline{T}_k = \begin{bmatrix} G_1 & E_1 & & & \\ F_1 & G_2 & E_2 & & \\ & \bullet & \bullet & \bullet & \\ & & \bullet & \bullet & \bullet \\ & & & F_{k-1} & G_k \end{bmatrix},$$

where G_i and E_i are $s \times s$ matrices. The matrix F_i is an $s \times s$ matrix whose only nonzero element is at location $(1, s)$.

Proof. $\overline{R}_i$ is nonsingular if and only if $v_i^1, \ldots, v_i^s$ are linearly independent. By Remark 6.1, R_k is nonsingular if all $\overline{R}_i$ for $i = 1, \ldots, k$ are nonsingular. So $\overline{T}_k$ is similar to the matrix $\overline{T}_j$. Since

$R_k = \mathrm{diag}(\bar{R}_1, \bar{R}_2, \ldots, \bar{R}_k)$ and $\bar{R}_i$ for $i = 1, \ldots, k$ is an $s \times s$ upper triangular matrix. R_k^{-1} has the same structure as R_k. Thus the product $R_k^{-1}\bar{T}_j R_k$ for $j = sk$ is a block tridiagonal matrix with lower diagonal blocks in a special form. We will demonstrate this for the special case $s = 3$, $k = 3$. The general case is shown similarly but the description is more complicated.



□

We will use the following s-dimensional column notations for the matrices G_i and E_i.

$$G_i = \left[\alpha_i^j\right], \quad E_i = \left[\gamma_i^j\right], \quad \text{for } j = 1, \ldots, s,$$

where $\alpha_i^j = [\alpha_i^{1,j}, \ldots, \alpha_i^{i,j}]^T$ and $\gamma_i^j = [\gamma_i^{1,j}, \ldots, \gamma_i^{i,j}]^T$.

Theorem 6.1. *Let A be a $N \times N$ symmetric matrix and $V_m = (\bar{V}_1, \bar{V}_2, \ldots, \bar{V}_m)$ for $N = sm$. Let*

$$V_m^{-1} A V_m = \bar{T}_m, \tag{6.1}$$

then $\bar{T}_m = R_m^{-1} T_N R_m$ where T_N is the Lanczos tridiagonal matrix.

Proof. By multiplying both sides of the Eq. (6.1) by V_m we obtain:

$$A V_m = V_m \bar{T}_m. \tag{6.2}$$

From Remark 6.1 we have:

$$A(\ddot{Q}_m R_m) = (\ddot{Q}_m R_m)\bar{T}_m.$$

By multiplying both sides of the equation by R_m^{-1} we obtain:

$$A\ddot{Q}_m = \ddot{Q}_m R_m \bar{T}_m R_m^{-1}.$$

By multiplying both sides of the equation by $\ddot{Q}_m^T$ we obtain:

$$\ddot{Q}_m^T A \ddot{Q}_m = R_m \bar{T}_m R_m^{-1}.$$

From Remark 6.1 $\ddot{Q}_m$ is an orthogonal matrix and from Proposition 6.1 $R_m \bar{T}_m R_m^{-1}$ is a symmetric tridiagonal matrix. Therefore, by the implicit Q theorem [7]. $R_m \bar{T}_m R_m^{-1}$ is the same as the Lanczos tridiagonal matrix T_N. Also $\ddot{Q}_m$ is the same sequence of vectors as the standard Lanczos vectors Q_N if the initial vector v_1 is the same. □

Corollary 6.1. *The block tridiagonal matrix $\overline{T}_k$ for $k = 1, \ldots, m-1$ has the same eigenvalues as the Lanczos reduction matrix T_j for $j = sk$.*

Proof. By Proposition 6.1, the matrices $\overline{T}_k$ and T_j for $j = sk$ are similar. □

If $k < m$ then by equating column blocks in Eq. (6.2) we obtained the following equation:

$$A\overline{V}_k = \overline{V}_k\overline{T}_k + u_k e_{sk}^T \quad \text{for} \quad k = 1, \ldots, m-1, \tag{6.3}$$

where u_k is the residual vector. From Eq. (6.3) we derive the following block equations:

$$A\overline{V}_k = \overline{V}_{k-1}E_{k-1} + \overline{V}_k G_k + u_k e_{sk}^T, \tag{6.4}$$

where $\overline{V}_0 = 0$ and $E_0 = 0$.

Equations (6.2), (6.3) and (6.4) motivate the derivation of an s-step Lanczos algorithm. Initially we can form $\overline{V}_1 = [v_1^1, Av_1^1, \ldots, A^{s-1}v_1^1]$, then we select the new Krylov vector from equation (6.4)

$$u_1 = Av_1^s - \overline{V}_0\gamma_0^s - \overline{V}_1\alpha_1^s.$$

If we choose $v_2^1 = u_1$ (i.e. normalization is not applied) then the nonzero entry of F_1 is 1. We from the vector $(v_2^2, \ldots, v_2^s)$ in $\overline{V}_2$ from the vectors $(Av_2^1, \ldots, A^{s-1}v_2^1)$ by orthgonalizing them against $\overline{V}_1$. Thus v_2^j, for $2 \leqslant j \leqslant s$, are determined by the linear combinations:

$$v_2^j = A^{j-1}v_2^1 - \sum_{i=1}^{s} v_1^i t_2^{i,j} \quad \text{for} \quad j = 2, \ldots, s,$$

where $(t_2^{i,j})$, for $1 \leqslant i \leqslant s$ and $2 \leqslant j \leqslant s$, are parameters to be determined. Let $t_k^j = [t_k^{1,j}, \ldots, t_k^{s,j}]^T$ denote the parameters defining v_k^j. We now give the defining equations of the s-step Lanczos in the form of an algorithm.

Algorithm 6.1. s-step Lanczos Algorithm.
$\overline{V}_0 = 0$, $[\gamma_0^i] = 0$, $1 \leqslant i \leqslant s$.
$\overline{V}_1 = [v_1^1, Av_1^1, \ldots, A^{s-1}v_1^1]$.
For $k = 1$ **until Convergence Do**
Select $[\alpha_k^i]$, $[\gamma_{k-1}^i]$, $1 \leqslant i \leqslant s$, to orthogonalize $\overline{V}_{k-1}$ against $\overline{V}_k$ in Eq. (6.4). This gives also:

$$v_{k-1}^1 = Av_k^s - \overline{V}_{k-1}\gamma_{k-1}^i - \overline{V}_k\alpha_k^i. \tag{6.5}$$

Select $[t_k^j]$, $2 \leqslant j \leqslant s$, to orthogonalize $(Av_2^1, \ldots, A^{j-1}v_2^1)$ against $\overline{V}_k$ which gives

$$v_{k+1}^j = A^{j-1}v_{k+1}^1 - \overline{V}_k t_k^j \quad \text{for} \quad j = 2, \ldots, s. \tag{6.6}$$

EndFor

Next, we demonstrate how to determine the parameters α_k^i, γ_{k-1}^i, t_k^j in Algorithm 6.1. Equation (6.4) multiplied by $\overline{V}_k^T$ from the left yields

$$\overline{V}_k^T A\overline{V}_k = \overline{V}_k^T\overline{V}_k G_k. \tag{6.7}$$

Also, Eq. (6.4) multiplied by $\overline{V}_{k-1}^T$ from the left yields

$$\overline{V}_{k-1}^T A\overline{V}_k = \overline{V}_{k-1}^T\overline{V}_{k-1}E_{k-1}. \tag{6.8}$$

Equation (6.6) multiplied by $\overline{V}_k^T$ from the left yields

$$0 = \overline{V}_k^T A^{j-1}v_{k+1}^1 - \overline{V}_k^T\overline{V}_k t_k^j \quad \text{for} \quad j = 2, \ldots, s. \tag{6.9}$$

Equations (6.7), (6.8) and (6.9) determine $[\alpha_k^i]$, $[\gamma_{k-1}^i]$, $1 \leqslant i \leqslant s$, and $[t_k^j]$, $2 \leqslant j \leqslant s$, as solutions of $3s-1$ linear systems of size s. We will introduce some notation in order to describe these linear systems.

Remark 6.2. Let $W_k = \overline{V}_k^{\mathrm{T}}\overline{V}_k = ((v_k^i, v_k^j))$, $1 \leqslant i, j \leqslant s$, then W_k is symmetric and it is nonsingular if and only if $v_k^1, \ldots, v_k^s$ are linearly independent.

Remark 6.3. From Eqs. (6.7), (6.8) and (6.9) and Remark 6.2 follows that the following linear systems must be solved to determine $[\alpha_k^i]$, $[\gamma_{k-1}^i]$, $1 \leqslant i \leqslant s$, and $[t_k^j]$, $2 \leqslant j \leqslant s$:

$$W_{k-1}\gamma_{k-1}^i = c_{k-1}^i, \quad \text{where} \quad c_{k-1}^i = \left[\left(v_{k-1}^1, Av_k^i\right), \ldots, \left(v_{k-1}^s, Av_k^i\right)\right]^{\mathrm{T}}, \tag{6.10}$$

$$W_k\alpha_k^i = d_k^i, \quad \text{where} \quad d_k^i = \left[\left(v_k^1, Av_k^i\right), \ldots, \left(v_k^s, Av_k^i\right)\right]^{\mathrm{T}}, \tag{6.11}$$

$$W_k t_k^j = b_k^j, \quad \text{where} \quad b_k^j = \left[\left(v_k^1, A^{j-1}v_{k+1}^1\right), \ldots, \left(v_k^s, A^{j-1}v_{k+1}^1\right)\right]^{\mathrm{T}}. \tag{6.12}$$

The following corollary simplified the computation of the vectors $b_k^{j_1}$ and $c_{k-1}^{j_1}$.

Corollary 6.2. *The right-hand side vectors* $c_{k-1}^1, \ldots, c_{k-1}^s$ for the linear systems (6.10) and $b_k^2, \ldots, b_k^s$ for the linear systems (6.12) become:

$$c_{k-1}^i = \left[0, \ldots, 0, \left(v_{k-1}^i, Av_k^i\right)\right]^{\mathrm{T}} = \left[0, \ldots, 0, \left(v_k^1, A^{i-1}v_k^1\right)\right]^{\mathrm{T}}, \quad 1 \leqslant i \leqslant s,$$

$$b_k^j = \left[0, \ldots, 0, \left(v_k^{s-j+2}, A^{s-1}v_{k+1}^1\right), \ldots, \left(v_k^s, A^{s-1}v_{k+1}^1\right)\right]^{\mathrm{T}}, \quad 2 \leqslant j \leqslant s.$$

Proof. We use the symmetry of A, Eqs. (6.5), (6.6) and the fact that the subspace $\overline{V}_k$, $(\overline{V}_{k+1})$ is orthogonal to $\overline{V}_{k-1}$, $(\overline{V}_k)$. □

Using this result and the fact that A is symmetric, the following Corollary reduces the matrix W_k and the vectors b_k^j to the previously computed scalars and the $2s$ inner products

$$\left(v_k^1, v_k^1\right), \left(Av_k^1, v_k^1\right), \ldots, \left(A^{2s-1}v_k^1, v_k^1\right). \tag{6.13}$$

These inner products are the first $2s$ moments of the vector v_k^1 with respect to the matrix A.

Corollary 6.3. *The computation of matrix* $W_k = (v_k^i, v_k^j)$, $1 \leqslant i, j \leqslant s$, *and the vectors* $b_k^2, \ldots, b_k^s$ *can be reduced to the first* $2s$ *moments of* v_k^1 *and previously computed scalars.*

Proof. We use the symmetry of A, the block orthogonality of $\{\overline{V}_k\}$ and Eqs. (6.5) and (6.6). The matrix of inner products W_k can be formed from the first $2s$ moments of v_k^1 and the s-dimensional vectors b_{k-1}^j as follows:

$$\left(v_k^i, v_k^j\right) = \left(v_k^i, A^{j-1}v_k^1\right)$$

$$= \left(v_k^1, A^{(i+j)-2}v_k^1\right) - \sum_{l=r}^{s} t_{k-1}^{l,j-1}\left(v_{k-1}^i, A^{l-1}v_k^1\right).$$

Similarly the inner products in the vectors b_k^j are computed as follows:

$$\left(v_k^i, A^{j}v_{k+1}^1\right) = \left(v_k^1, A^{i+j-1}v_{k+1}^1\right)$$

$$= \left(v_{k+1}^1, A^{(i+j)-(s+1)}v_{k+1}^1\right) + \sum_{l=r}^{s} \alpha_k^{ls}\left(v_k^l, A^{(i+j)-(s+1)}v_{k+1}^1\right),$$

where r is $2(s+1)-(p+q)$. □

Also, the vectors $d_k^1, \ldots, d_k^s$ can be reduced to computing the first $2s$ moments of v_k^1 and previous scalar work using a proof similar to that of Corollary 6.3. Thus there are only $2s$ inner products (on vectors of size N) required in forming $\overline{T}_k$ and $\overline{V}_{k+1}$ in Algorithm 6.1.

We now reformulate the s-step Lanczos algorithm taking into account the theory developed above.

Algorithm 6.2. s-step Lanczos algorithm.
Select v_1^1.
Compute $\overline{V}_1 = [v_1^1, Av_1^1, \ldots, A^{s-1}v_1^1]$.
Compute $2s$ inner products.
For $k = 1$ **until** Convergence **Do**
1. Call Scalar 1.
2. Compute $v_{k+1}^1 = Av_k^s - \overline{V}_{k-1}\gamma_{k-1}^s - \overline{V}_k\alpha_k^s$.
3. Compute $Av_{k+1}^1, A^2v_{k+1}^1, \ldots, A^sv_{k+1}^1$.
4. Compute the $2s$ inner products in (6.13).
5. Call Scalar2.
6. Compute $v_{k+1}^j = A^{j-1}v_{k+1}^1 - \overline{V}_k[t_k^j]$ for $j = 2, \ldots, s$.

EndFor

Scalar 1: Decompose W_k and solve $W_{k-1}\gamma_{k-1}^i = c_{k-1}^i$, $W_k\alpha_k^i = d_k^i$, for $i = 1, \ldots, s$.
Scalar 2: Solve $W_k t_k^j = b_k^j$ for $j = 1, \ldots, s$.

From Eq. (6.3) follows that in the s-step Lanczos method the Ritz values of A in V_k are the eigenvalues λ_k and $\overline{T}_k$ and the Ritz vectors are vectors $\overline{V}_k x_k (= z_k)$, where the eigenvectors x_k of $\overline{T}_k$ are associated with the λ_k. The residual norms of the Ritz value λ and Ritz vector z can be computed by using the formula $\|(A - \lambda_i I)z_i\| = \|u_k\| \, |\bar{s}_{ki}|$, for $i = 1, \ldots, sk$, where $\bar{s}_{ki}$ is the last element of the ith eigenvector of $\overline{T}_k$. This can be used as a stopping criterion.

In *Table 3* we compare the computational work of the s-step Lanczos method to the standard Lanczos method. We only present the vector operations on vectors of dimension N and neglect the operations on vectors of dimension s.

7. Numerical experiments

Large, sparse problems arise frequently in the numerical integration of partial differential equations (PDEs). Thus we borrow our model problem from this area. The test problem was derived from the five point discretization of the following partial differential equation. Problem:

$$-(bu_x)_x - (cu_y)_y + (du)_x + (eu)_y + fu = g$$

on the unit square, where

$$b(x, y) = e^{-xy}, \quad c(x, y) = e^{xy}, \quad d(x, y) = \beta(x + y),$$
$$e(x, y) = \gamma(x + y) \quad \text{and} \quad f(x, y) = 1/(1 + x + y)$$

Table 3
Vector Ops for s iterations of the standard Lanczos and one iteration of s-step Lanczos method

operation	standard Lanczos algorithm	s-step Lanczos algorithm
Inner products	$2s$	$2s$
Vector updates	$5s$	$2s(s+1)$
Matrix * Vector	s	$s+1$

Table 4
Largest eigenvalues using the Lanczos methods on the CRAY-2

T_j	*Code**	standard	modified	2-step
10×10	0.11086467E+02	0.10704428E+02	0.10704428E+02	0.10704428E+02
20×20	0.11086467E+02	0.11083956E+02	0.11083956E+02	0.11083956E+02
30×30	0.11086467E+02	0.11086467E+02	0.11086467E+02	0.11086467E+02
40×40	0.11086467E+02	0.11086467E+02	0.11086467E+02	0.11086467E+02
T_j	3-step	4-step	5-step	6-step
10×10	–	–	0.10704427E+02	–
20×20	–	0.11083956E+02	0.11083955E+02	–
30×30	0.11086467E+02	–	0.11086460E+02	0.11086387E+02
40×40	–	0.11086467E+02	0.11086460E+02	–

subject to the Dirichlet boundary conditions $u(x, y)$ equals the solution on the boundary. The right-hand side g was chosen so that the solution was known to be $e^{xy} \sin(\pi x) \sin(\pi y)$. The parameters β and γ are useful for changing the degree of symmetry of the resulting coefficient matrix of the linear systems. Note that the matrix A resulting from the discretization remains positive real, independent of these parameters. We denote by n the number of interior nodes on each side of the square and by $h = 1/(n+1)$ the mesh size. In this paper we set $\gamma = 0$ and $\beta = 0$ for a symmetric matrix A of dimension $N = n^2$.

The experiments were conducted on the NCUBE/7 with 64 processors and on the CRAY-2 multiprocessor system at the University of Minnesota. In the accuracy test, *Table 4* shows that matrices generated by the standard, modified and s-step Lanczos method have the same largest eigenvalues, but for $s > 5$ in the s-step method loss of accuracy for eigenvalues can be observed. We tested the methods on the problem of size $N = 4096$. We also compared the largest eigenvalues computed by the standard, the modified and the s-step method with those by Saad's program (*code**) [16]. Saad's code uses reorthogonalization and deflated iteration to compute eigenvalues. The stopping criterion is $\epsilon = 10^{-6}$ in Saad's program. In the standard, modified and s-step Lanczos method, we find the largest eigenvalues after a reduced matrix of a certain size is generated, so these methods require minimal storage and time. Reorthogonalization or deflated iteration are required to get a good approximation for the extreme eigenvalues. However we have not yet incorporated these techniques in the implementation of our Lanczos algorithms.

We reduced the different size matrices A of the model problem to 20×20 tridiagonal or block tridiagonal matrices using the standard, modified and 5-step Lanczos algorithms. *Figure 4* shows the time these methods took to make 20×20 reduced matrices for different size test problems on the CRAY-2 with 4 processors. *Figure 5* shows the speedup (P1/P4) when 1 processor and 4 processors are used in the CRAY-2 for the standard and 5-step method. In the modified and s-step method memory reference time and calls to the macrotasking library are decreased by the possibility of making the grain size large and by decreasing the number of synchronization points. This accounts for the superior performance of these methods over the standard Lanczos method on the CRAY-2.

Figure 6 shows the time for the inner products part of the Lanczos algorithm which require global communications on the NCUBE/7 (problem size $N = 4096$). The speedup factor (standard/modified) for the inner product part is 1.5 in *Fig. 6*. The speedup of the modified Lanczos method over the standard Lanczos method results only from the decreased time for the inner products part on the NCUBE. So the modified method is expected to give better performance over the standard one as more processors are used. The total time to make 20×20 tridiagonal matrices for the standard and modified Lanczos method on different numbers of

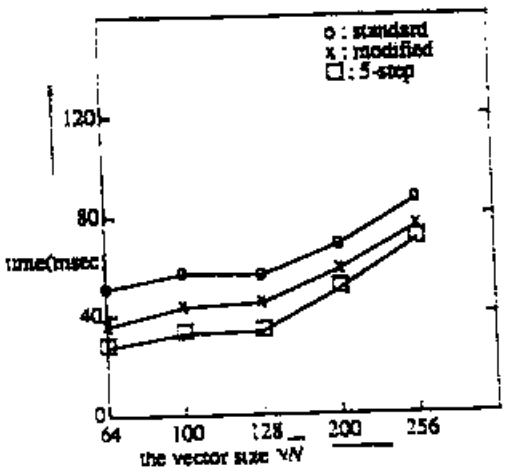


Fig. 4. CRAY-2 Performance (msec) using 4 processors for the Lanczos methods.

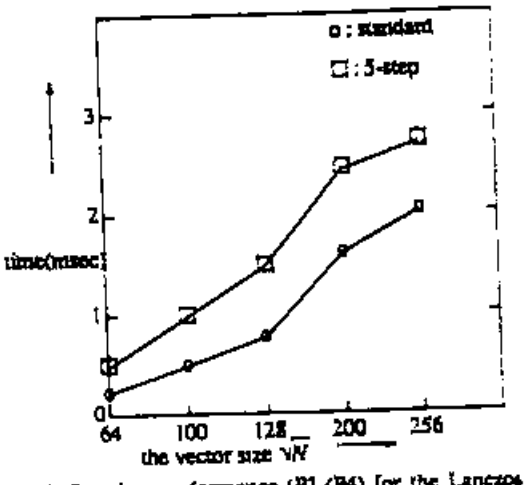


Fig. 5. Speedup performance (P1/P4) for the Lanczos methods on the CRAY-2.

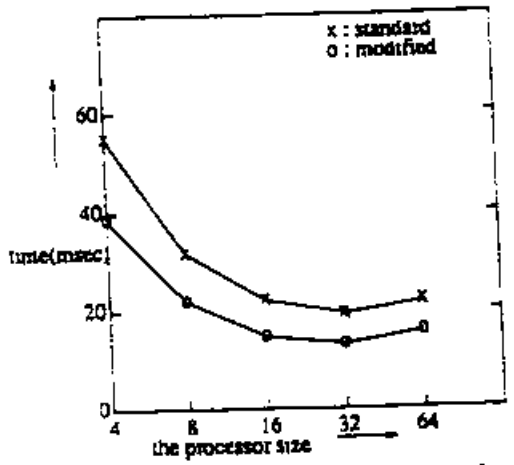


Fig. 6. NCUBE/7 performance for two inner products per one iteration in the standard and modified Lanczos method.

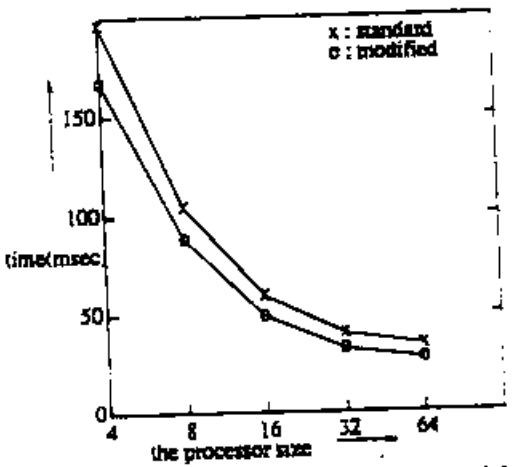


Fig. 7. NCUBE/7 performance for one iteration of the standard and modified Lanczos method.

processors of the NCUBE/7 with $N = 4096$ are given in *Fig. 7*. *Table 5* shows the total time of the standard and modified and 1-step method for different sizes of test problems with 64 nodes on the NCUBE/7. On the NCUBE/7, the 1-step Lanczos method gives the fastest perfor-

Table 5
NCUBE/7 performance (sec) using 64 nodes for the Lanczos methods

$\sqrt{N}$	standard	modified	1-step
64	0.645	0.536	0.471
128	1.457	1.208	1.114
192	2.792	2.307	2.171
256	4.691	3.858	3.642

mance because it requires one vector updates less than the standard Lanczos method and only one global communication per one iteration. The s-step Lanczos method with $s > 1$ has vector operations overhead (over the standard one) both in the linear combinations and matrix–vector products. As a result the gain due to reduced global communications cannot offset the time required for the overhead, and the s-step for $s > 1$ Lanczos method is slower than the standard Lanczos method on the NCUBE/7.

8. Conclusion

The Lanczos algorithm was restructured in this paper. The modified algorithm decreases the global communication bottleneck of the basic Lanczos algorithm by restructuring computations in such a way as to increase the number of inner products that are accumulated during one iteration. The modified algorithm has better data locality and decreases the memory contention bottleneck.

We have also introduced the s-step Lanczos method and proved that it generates reduction matrices which are similar to those generated by the standard Lanczos method. The resulting algorithm has better data locality and parallel properties than the standard one. In the s-step method the inner products needed for s steps of the standard method can be performed simultaneously and the vector updates are replaced by linear combinations. A disadvantage of the s-step Lanczos method is that additional operations (compared to that of the standard Lanczos method) are required. The design of a stable s-step Lanczos method with no additional vector operations compared to the standard method remains an open question. For large $s > 5$ loss of accuracy for eigenvalues has been observed.

Acknowledgement

We thank the anonymous referees and editor whose comments helped enhance significantly the quality of presentation of this article.

References

[1] Cevdet Aykanat, Fusun Ozguner, Fikret Ercal and Ponnuswamy Sadayappan, Iterative Algorithms for Solution of Large Sparse Systems of Linear Equations on Hypercubes, *IEEE Trans. Comput.* 37, (12) (December, 1988) 1554–1568.

[2] A.T. Chronopoulos and C.W. Gear, s-step iterative methods for symmetric linear systems, *J. Comput. Appl. Math.* 25 (1989) 153–168.

[3] A.T. Chronopoulos and C.W. Gear, On the efficient implementation of preconditioned s-step conjugate gradient methods on multiprocessors with memory hierarchy, *Parallel Comput.* 11 (1989) 37–52.

[4] Jane K. Cullum and Raph A. Willoughby, *Lanczos Algorithms for Large Symmetric Eigenvalues Computation* (Birkhauser, Boston, 1985).

[5] Ameet K. Dave and Iain, S. Duff, Sparse matrix calculations on the CRAY-2, *Parallel Comput.* 5 (1987) 55–64.

[6] J.J. Dongarra and D.C. Sorenson, in: M. Feilmeier et al., eds, Linear Algebra on High-Performance Computer, Parallel Comput. 85 (Elsevier Sci. Publishers, Amsterdam, 1986).

[7] G.H. Golub, C.F. Van Loan, *MATRIX Computations* (Johns Hopkins Univ. Press, Baltimore, MD, 1989) 219–225.

[8] John L. Gustafson, Gary R. Montry and Robert E. Benner, Development of Parallel Methods for a 1024-Processor Hypercube, *SIAM J. Sci. Stat. Comput.* 9 (4) (July, 1988).

[9] Robert F. Lucas, Tom Blank and Jerome J. Tiemann, A Parallel Solution Method for Large Sparse Systems of Equations, *IEEE Trans. Comput.-Aided Des. Integrated Circuits Syst.*, CAD-6 (6) (November, 1987).

[10] Gerard Meurant, Multitasking the conjugate gradient method on the CRAY X-MP/48, *Parallel Comput.* 5 (1987) 267–280.
[11] Oliver A. McBryan and Eric F. van der Velde, Matrix and vector operations on hypercube parallel processors, *Parallel Comput.* 5 (1987) 117–125.
[12] Lionel M. Ni, Chung-Ta King and Phillip Prinz, Parallel Algorithm Design Considerations for Hypercube Multiprocessors, *Proc. of the 1987 Internat. Conf. on Parallel Processing*, Aug. 1987.
[13] C.C. Paige, Computational Variants of the Lanczos Method for the Eigenproblem, Inst. Math. Appl. 10 (1972) 373–381.
[14] B.N. Parlett, *The Symmetric Eigenvalues Problem* (Prentice-Hall, Englewood Cliffs, NJ, 1980).
[15] Sanjay Ranka, Younggu Won and Sartaj Sahni, Programming the NCUBE Hypercube, Tech. Rep. CSci No. 88-13, Univ. of Minnesota, Mpls. MN, 1988.
[16] Youcef Saad, Partial eigensolutions of large nonsymmetric matrices, Research Report YALUE/DCS/RR-397, June 1985.
[17] Paul E. Saylor, Leapfrog variants of iterative methods for linear algebraic equations, *J. Comput. Appl. Math.* 24 (1988) 169–193.
[18] J.H. Wilkinson, *The Algebraic Eigenvalue Problem*, (Oxford Univ. Press, Oxford, 1965).