Solving Large Sparse Eigenvalue Problems on Supercomputers

Bernard Philippe
Youcef Saad

December, 1988

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 88.38

NASA Cooperative Agreement Number NCC 2-387

RIACS

Research Institute for Advanced Computer Science

Solving Large Sparse Eigenvalue Problems on Supercomputers

*Bernard Philippe**
Youcef Saad

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 88.38
December, 1988

An important problem in scientific computing consists in finding a few eigenvalues and corresponding eigenvectors of a very large and sparse matrix. The most popular methods to solve these problems are based on projection techniques on appropriate subspaces. The main attraction of these methods is that they only require to use the matrix in the form of matrix by vector multiplications. We compare the implementations on supercomputers of two such methods for symmetric matrices, namely Lanczos' method and Davidson's method. Since one of the most important operations in these two methods is the multiplication of vectors by the sparse matrix, we first discuss how to perform this operation efficiently. We then compare the advantages and the disadvantages of each method and discuss implementations aspects. Numerical experiments on a one processor CRAY 2 and CRAY X-MP are reported. We also discuss possible parallel implementations.

This paper is to appear in the Proceedings of the International Workshop on Parallel Algorithms and Architectures, October, 1988, Bonas, France.

*Bernard Philippe is located at the IRISA/INRIA in Rennes, France.

Work reported herein was supported by the National Science Foundation under Grants No. US NSF-MIP-8410110 and US NSF DCR85-09970, the US Department of Energy under Grant No. DOE DE-FG02-85ER25001, the ARO contract DAAL03-88-K-0085, and by Cooperative Agreement NCC 2-387 between the National Aeronautics and Space Administration (NASA) and the Universities Space Research Association (USRA).

1 Introduction

The numerical solution of large sparse eigenvalue problems arises in numerous important scientific applications that can be termed supercomputing applications. The advances in supercomputing technology allow today to tackle very large eigenvalue problems that were not feasible a few years ago. Moreover, this trend is likely to continue as the scientific models will become more sophisticated and as the numerical methods will improve in efficiency and reliability. This paper considers the problems encountered when adapting two well-known numerical techniques for solving eigenvalue problems, to supercomputing environments. The two methods considered are the Lanczos algorithm and Davidson's method. The first is a well-known technique which has had impressive success in the last two decades, for solving standard symmetric eigenvalue problems. It main attraction is that it can build, at least in theory, an orthogonal basis of the so-called Krylov subspace $K_m = span\{v, Av, ..., A^{m-1}v\}$ with the help of a simple three-term recurrence. To extract the Ritz eigenvalues from this subspace, one only needs to compute eigenvalues of a tridiagonal matrix whose coefficients are obtained from the three-term recurrence. In Davidson's method one constructs explicitly an orthogonal basis of a certain subspace which is obtained by adding a vector of the form $(M - \theta I)^{-1}(A - \theta I)v$ where M is a preconditioner of A, θ is a shift close to the eigenvalue being computed and v is the previous basis vector. This can be viewed as a preconditioned version of the Lanczos algorithm. The advantage is faster convergence, i.e., the subspace for which convergence is achieved in Davidson's method is of much smaller dimension than with Lanczos. On the other hand each iteration now costs much more since the three term recurrence is lost and one must explicitly orthogonalize each new vector against all previous ones. However, for many problems arising in specific applications such as chemistry, Davidson's method is still superior.

Though very similar in theory, the two methods are different from the computational point of view and the problems encountered when implementing them on supercomputers are not the same. Their only common point is that they both use at each step a matrix by vector multiplication. There are issues, such as reorthogonalization for the Lanczos algorithm, that are proper to one of the methods only.

In the next section we describe the two methods and compare their advantages and disadvantages. Then we will discuss the problem of performing a matrix by vector product on supercomputers. In Section 4, we will describe implementation details and numerical experiments for the CRAY 2 and for the CRAY X-MP. Finally in Section 5, we will give some ideas on how to adapt the two methods on a parallel machine with a moderate number of processors.

2 The Lanczos and Davidson methods

2.1 The Lanczos Algorithm

The Lanczos algorithm in its basic form, is an algorithm for computing an orthogonal basis of a Krylov subspace, i.e., a subspace of the form $K_m = span\{v_1, Av_1, \ldots, A^{m-1}v_1\}$. The

algorithm is as follows.

Start: Choose an initial vector v_1.

Iterate: for $j = 1, 2, .., m, ..$ Do

- $y := Av_j$
- If $j > 1$ compute $y := y - \beta_j v_{j-1}$
- $\alpha_j := (y, v_j)$
- $y := y - \alpha_j v_j$
- $\beta_{j+1} := \|y\|_2$
- $v_{j+1} := y/\beta_{j+1}$

The main iteration in the above algorithm can be succinctly described by the following three term recurrence:

$$\beta_{j+1} v_{j+1} = Av_j - \alpha_j v_j - \beta_j v_{j-1} \tag{1}$$

where α_j and β_{j+1} are selected so that the vector v_{j+1} is orthogonal to both v_j and v_{j-1}. It can be shown that this ensures that the sequence of vectors v_i forms an orthonormal sequence.

If we denote by V_m the N x m matrix

$$V_m \equiv [v_1, v_2, \ldots, v_m]$$

and by T_m the tridiagonal matrix:

$$T_m = \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & . & . & . & \\ & & . & . & \beta_m \\ & & & \beta_m & \alpha_m \end{pmatrix} \tag{2}$$

then we have the relations,

$$AV_m = V_m T_m + \beta_{m+1} v_{m+1} e_m^T$$

and

$$V_m^T A V_m = T_m$$

Therefore, it is natural to approximate the eigenvalues of A by some of the eigenvalues of T_m. This will constitute a Rayleigh-Ritz [21] procedure on the Krylov subspace K_m. It is known [21] that the outmost eigenvalues of T_m, will rapidly become good approximations to those of A. In this paper we will assume that we are interested in the smallest eigenvalues so we number all eigenvalues, exact and approximate, increasingly and denote by $\lambda_i^{(m)}$ the $i-th$ eigenvalue of T_m, and $y_i^{(m)}$ the corresponding eigenvector. The corresponding approximate eigenvector of A is given by $u_i^{(m)} = V_m y_i^{(m)}$.

Although the above procedure seems very simple to implement there are several pitfalls. One of the first observations made on the Lanczos algorithm is the loss of orthogonality of the vectors v_i. We can only summarize the main findings concerning the analysis on loss of orthogonality but we refer the readers to [21] for details. Loss of orthogonality is triggered by convergence of some eigenvector and is avoidable only at the price of some form of reorthogonalization. The simplest reorthogonalization possible is full reorthogonalization: at every step one reorthogonalizes the current Lanczos vector against all previous ones. However, it was discovered that loss of orthogonality does not prevent convergence, it can at worst slow it down [19]. As a result several authors suggested using the algorithm without reorthogonalization. The implementation of the algorithm without reorthogonalization is complicated by several factors. First, the approximate eigenvalues may now appear several times. Moreover, one must be careful about monitoring convergence of eigenvalues: some eigenvalues will seem to converge and then disappear for a while only to reappear later. This does not happen with full reorthogonalization. The advantages of the algorithm without reorthogonalization are clear, the avoidance of storing all the Lanczos vectors being perhaps the most important one. ¿From the point of view of cost, orthogonalization by Gram Schmidt is rather expensive.

For additional details on the Lanczos method, see [4,13,23,25,26].

2.2 The Davidson Algorithm

Similarly to the Lanczos algorithm, the Davidson algorithm [6] is based on the projection of the matrix over a sequence of subspaces of increasing dimension. In some sense, the method can be considered as a preconditioned version of the Lanczos method [17] although the context is rather different from that of preconditioned techniques for solving linear systems. If the preconditioner is efficient, the convergence can be very fast. The major drawback of the method comes from the amount of work involved in one iteration, which increases with the dimension of the subspace : in contrast with the Lanczos algorithm, the restricted matrices are full and they must explicitly be computed; moreover, it is now necessary to store the orthonormal basis and its resulting transformation by the original matrix. Hence, the process must be restarted periodically with the current best estimate of the wanted eigenvector. The algorithm is built around two embedded loops. To compute the largest (resp. smallest) eigenvalue of A, the process can be described as follows.

Start: Choose an initial unit vector v_1.

Iterate: for $iter = 1, 2, \ldots$ Do

Iterate: for $j = 1, \ldots, k$ Do

- $W_j := AV_j$ (only the last column is computed)
- $H_j := V_j^T W_j$ (only the last column is computed)
- computation of the largest eigenpair of H_j : (λ, y)
- $x := V_j y$ (Ritz-vector)
- $r := W_j y - \lambda x$ (residual)
- test for convergence
- If $j < k$:

  $t := (M - \lambda I)^{-1} r$ (M is the preconditioner)

  $V_{j+1} := MGS([V_j, t])$ (Modified-Gram-Schmidt procedure)

$v_1 := x$

The simplest and most common preconditioner M is the main diagonal of A (Jacobi Preconditioner). It can only be used when the matrix A is nearly diagonal in the sense that its matrix of eigenvectors is close to the identity ; this is often the situation in Quantum Chemistry and this is the reason for the method is popular there.

It should be noticed that, if no preconditioner were to be used (ie. $M = Id$), then the sequence of subspaces would become identical with that of the Krylov subspaces and then both methods, Lanczos and Davidson, would theoretically be equivalent. However, since the considered orthonormal basis is not the same, the computation will remain heavier with the Davidson method.

When one seeks several eigenvalues or when one knows that the desired eigenvalue is very close from some others, a block version of the algorithm can be used : several eigenpairs of H_j are computed at the same time and then several vectors are added to the basis V_j.

2.3 Complexity of the Algorithms

We now consider the complexity of one outer iteration of the Davidson algorithm when only one vector is added at every inner step to the basis. The cost of each such outer loop is roughly,

$$\begin{aligned} C_{out} &\approx \sum_{j=1}^{k} (n_z + jn + 2j^3 + jn + (j+1)n + p_r + (2j+1)n \\ &\approx k(n_z + p_r) + 5/2k(k + 9/5)n + k^2(k+1)^2 \end{aligned}$$

where n, n_z and p_r stand respectively for the order of the matrix A, its number of non-zero entries and the complexity of the preconditioning step. When the Jacobi preconditioner is used $p_r = n$ and the average cost for the inner iteration is then

$$C_{inn} \approx n_z + p_z + 5/2(k + 9/5)n + k(k + 1)^2$$

This shows that the cost per step increases quadratically with the step number k, and a reasonable upper limit for k would be $k = \sqrt{n}$ in which case the average cost per inner-step comes to

$$C_{inn} \approx n_z + p_z + 7/2n^{3/2} + O(n).$$

This number has to be compared to $(n_z + 5n)$ which is the complexity for each Lanczos step, without reorthogonalization. It becomes clear that, in order to be competitive, the method needs a preconditioner which will strongly reduces the number of steps to achieve convergence.

For the block version of the Davidson algorithm, the complexity per outer step remains roughly unchanged, but of course the number of necessary steps for convergence increases with the number of eigenpairs sought.

3 The problem of matrix by vector multiplications

Since both methods use a matrix by vector multiplication at each step the procedure to perform this operation must be carefully implemented. The typical matrices dealt with are sparse and often unstructured and as a result very poor performance may prevail if not enough care is devoted to optimizing this operation. We review here some of the possible options we have to improve the speed of this basic kernel by exploiting vectorization and parallelization.

To parallelize the multiplication, it is easy to consider A as a sum of some elementary matrices $A = \sum_{t=1}^{nproc} A_t$, to compute in parallel $(y_t := A_t x)_{t=1,nproc}$ and then accumulate the partial results $y := \sum_{t=1}^{nproc} y_t$. Because the vector x must be read by all the processors, it should remain in the global memory or be duplicated in the local memories. The usual way to perform the partitioning is to define blocks of consecutive rows of A in order to bypass the accumulation step. A special case consists of considering one row only per block and performing all the scalar products in parallel. Workload would obviously be balanced when the number of non-zero entries per processor is roughly the same.

It may be pointed out that in the block version of the Davidson method parallelism can also be achieved by performing independently several multiplications. We will not discuss this obvious additional possibility which can also be exploited in the block form of the Lanczos algorithm.

We now would like to examine in detail the procedure on one processor which is assumed to be a vector processor. The first observation that has been made in this context is that this operation can be performed by diagonals when the matrix is regularly structured, i.e., when it consists of a few diagonals [12]. The matrix can be stored in a rectangular array $DIAG(1 : N, 1 : NDIAG)$ and the offsets of these diagonals from the main diagonal may

be stored in a small integer array $IOFF(1 : NDIAG)$. After initializing the vector y to zero, the main loop for computing $y = Ax$ is expressed in FORTRAN 8-X as follows.

```
      DO 10 J=1, NDIAG
         JOFF = IOFF(J)
         Y(1:N) = Y(1:N) + DIAG(1:N,J)*X(JOFF+1:JOFF+N)
10    CONTINUE
```

Excellent megaflops rates can be reached on vector machines when the matrix is large enough.

For general sparse matrices there has been several attempts to obtain similar performances by either generalizing the diagonal storage scheme [18,27] or by reordering the matrix so as to obtain a diagonal structure [1,22]. We will only discuss the first approach here. This approach is of interest only for matrices whose maximum number of nonzeros per row $jmax$ (which is called the degree of the row) is small. One then stores the entries of the matrix in a real array $COEFF(1 : n, 1 : jmax)$ together with an integer array $JCOEFF(1 : n, 1 : jmax)$ that stores the column numbers of each entry of $COEFF$. We refer to this as the ITPACK format. The above FORTRAN loop then becomes,

```
      DO 10 J=1, NDIAG
            Y(1:N) = Y(1:N) + COEFF(1:N,J)*X(JCOEFF(1:N,J))
10    CONTINUE
```

The main difference between this loop and the previous one is the presence of indirect addressing in the innermost computation. Note that if the degree of the rows varies substantially, then many zero elements must be stored unnecessarily, and this scheme may become inefficient.

The above storage schemes are somewhat specialized to certain types of matrices. These can be very useful in many instances but their lack of generality is a serious limitation. Unfortunately, as is often the case, there is a conflict between generality and efficiency.

On of the most general schemes for storing sparse matrices uses a real array $A(1 : NNZ)$ which contains the nonzero elements of the matrix, stored row-wise, an integer array $JA(1 : NNZ)$ which stores the column positions of the corresponding elements in the real array A, and finally a pointer integer array $IA(1 : N + 1)$ the i-th element of which points to the beginning in the arrays A and JA of the consecutive rows. This data structure is often referred to as the general sparse format, or the A, JA, IA format. With this storage scheme each component of the resulting vector y can be easily computed independently as the dot product of the i-th row of the matrix with the vector x. We can write this as

```
      DO 10 I=1, N
            K1 = IA(I)
            K2 = IA(I+1)-1
            Y(I) = DOTPRODUCT( A(K1:K2) , X(JA(K1:K2)) )
10    CONTINUE
```

The outer loop can be performed in parallel, as mentioned before. On a machine like the

	2-D	3-D	Markov
matrix order	2765	3096	14917
non-zero entries	16238	40606	128773
maximum degree	9	38	17
minimum degree	3	6	1
average degree	6	13	9

Table 1: Characteristics of three sparse matrices.

Alliant FX-8, the synchronization of this outer loop is inexpensive and the performance of the above program can be excellent.

The indirect addressing involved in the second vector in the dot product loop is handled by a special hardware instruction called a *Gather* operation. The vector $X(JA(k1 : k2))$ is first gathered from memory into a vector of contiguous elements. The dot product is then carried out as regular dot product operation. The first vector machines that appeared did not perform too well on sparse computations because they were not equipped with special instructions for Gather and Scatter. The beneficial impact of Hardware Scatter and Gather on vector machines has been discussed in [14].

For vector machines the previous two techniques are likely to perform very poorly because they involve vectors that are usually very short. For instance on CRAY-2, the sparse dot-product reaches half of the asymptotic speed with vectors of length 150 while the average degree per row of sparse matrices usually lies in the range 5-50. An alternative is to use one of the schemes based on diagonal and generalized banded format described above. However, the following scheme is more general.

We start from the A, JA, IA data structure and build a new one by constructing what we call jagged diagonals [24]. This scheme is related to the stripe structure [16] or to the generalized-column wise storage [8]. We store as a dense vector the leftmost element from each row, together with two integer vectors containing the row and column positions of each element. This is followed by the second jagged diagonal consisting of the elements in second position from the left. This storage may also be done starting from the main diagonal, especially when storing only the upper-triangular part of the symmetric matrix. As we build more and more jagged diagonals, their length decreases. The number of j-diagonals is equal to the largest degree of the rows. As an illustration, three sparse matrices have been considered for the efficiency of this storage : two matrices arising from the triangularization of 2-D and 3-D domains and a stochastic transition probability matrix of a Markov chain; their characteristics are described in Table 1 and the lengths of their jagged diagonals are pictured in Figure 1. These examples show that, with this storage, most of the non-zero entries belong to long vectors.

If IDIAG(j) is the pointer to the beginning of the j-th jagged diagonal and IROW(k) and JDIAG(k) are, respectively, the row position and the column position of the element stored in A(k), then the product $y = Ax$, can be computed as follows,

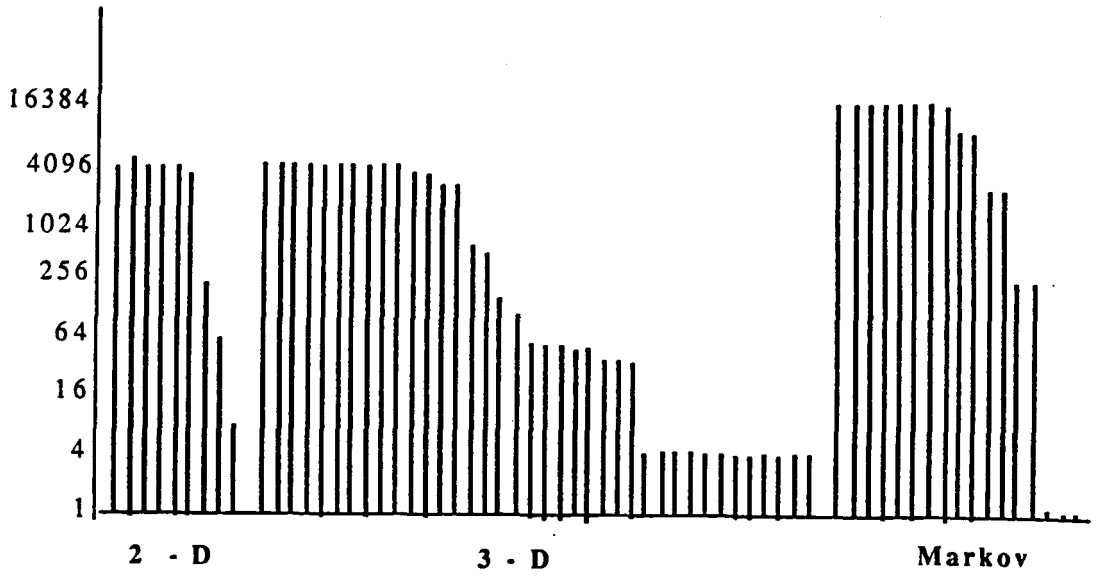

Figure 1: Lengths of jagged diagonals for three test matrices.

```
      DO 10 J = 1, NDIAG
           K1 = IDIAG(J)
           K2 = IDIAG(J+1)-1
           Y(IROW(K1:K2)) = Y(IROW(K1:K2)) + A(K1:K2)*X(JDIAG(K1:K2))
10    CONTINUE
```

The asymptotic speed which can be obtained on CRAY 2 is 29 MFLOPS. To increase speed, one may proceed in two ways. A first way consists of reordering the rows by decreasing degree in order to be able to pick the non-zero entries of each jagged diagonal of A from consecutive rows; this allows us to avoid the indirect load and store on Y. On CRAY 2, the asymptotic speed is then 39 MFLOPS. The second way to increase speed is to *unroll* the loop by blocking several j-diagonals together. Let us assume, that now the sparse matrix is stored in a two-dimensional array A such that A(*,k) represents all the entries of the k-th jagged diagonals of all the blocks ; NS is the number of diagonals in a block ; IROW(k) and JDIAG(k,j) are the row and column indices of the entry stored in A(k,j). This definition implies that all the diagonals of a block have the same structure ; it can be achieved by adding some virtual non-zero entries. Let NB be the number of blocks and IDIAG(j) the pointer to the j-th block. The program becomes

GRID SIZE	METHOD				
	1	2	3	4	5
20 x 20	6.56	0.19	8.04	9.43	7.11
20 x 20 x 10	7.06	0.21	7.013	10.13	4.05
30 x 30	6.7	0.19	8.98	10.83	8.19
30 x 30 x 10	5.42	0.21	6.48	9.69	3.32

Table 2: Megaflop rates for five matrix by vector multiplication kernels on an Alliant FX-80 (double precision arithmetic).

```
      DO 10 J=1, NB
         K1 = IDIAG(J)
         K2 = IDIAG(J+1)-1
         Y(IROW(K1:K2)) = Y(IROW(K1:K2))
                        + A(K1:K2,1)*X(JDIAG(K1:K2,1))
     *                  + A(K1:K2,2)*X(JDIAG(K1:K2,2))
     *                  + ...
     *                  + A(K1:K2,NS)*X(JDIAG(K1:K2,NS))
10    CONTINUE
```

The parameter NS is called *depth of unrolling.* The improvement achieved here is due to the fact that there is only one indirect load and store of the vector Y for NS indirect loads of the vector X. On CRAY 2, the asymptotic speed for $NS = 4$ is about 55 MFLOPS.

It is obviously possible to combine both improvements (reordering rows and blocking diagonals) to reach higher rates of execution.

As an illustration we show in the next table the performance, on an Alliant FX-80, of the following five different ways of multiplying a matrix by a vector:

1. Row-wise storage , (sparse dot product form);
2. Column wise storage, (sparse saxpy form);
3. Itpack format
4. Diagonal storage, (triad form);
5. Jagged diagonal format(with reordered rows);

The technique of unrolling has not yet been tested on Alliant. We took 5-point and 7-point matrices for 2-D and 3-D rectangular grids. Results are displayed in Table 2. Notice the wide differences in performance obtained between the various ways of performing the same operation. On the Alliant FX-80, method 2, using the column-wise storage is the worst performer. Also of interest, and to some extent disturbing, is the variation in performance obtained for different matrices *with the same kernel.* These discrepancies are especially noticeable in Kernel 5, using the Jagged diagonal format.

One problem that seems to have not yet been studied in the literature is that of performing simple operations with general sparse matrices on SIMD machines of the type of the MPP or the Connection machine. On such machines much of what has been accomplished is to test the usual symmetric conjugate gradient method for easy model problems [3,2]. The difficulty with the more realistic general sparse problems is the apparent necessity of resorting to indirect addressing, a difficult operation on these architectures. Hammond and Law [9] propose a hardware solution based on systolic arrays. This challenging problem is to be solved to satisfaction before SIMD machines are to be considered real contenders to MIMD ones, in the race for usable supercomputers.

4 Implementations and results on vector processors

4.1 Implementation of the Lanczos method

The questions that must be addressed when implementing a Lanczos type algorithm are numerous:

- Should reorthogonalization be used and which form of reorthogonalization?
- How often should one compute the approximate eigenvalues and how does one monitor convergence.
- What is the best way of computing or updating the eigenvalues of T_m?

We will now answer some of these questions. The trade-offs between reorthogonalizing and not reorthogonalizing in a Lanczos code are similar to those on a scalar machine, namely cost versus ease of implementations and simplicity. There are circumstances where reorthogonalization of some sort is essential. For example if the Lanczos algorithm is used in conjunction with shift-and-invert then the price of reorthogonalization is worth paying. The reason is that the major cost in this case is the factorization and there will be more factorizations if Lanczos is slower to converge as is the case when reorthogonalization is skipped. Relatively speaking, factorizations of unstructured sparse matrices are even more expensive on vector machines than on scalar machines so the case for even a full reorthogonalization in shift and invert is stronger. If a large number of eigenvalues must be computed, then reorthogonalization is an option that might become too expensive.

We are interested in the problem of computing a few of the largest or smallest eigenvalues of A. We chose to implement an algorithm without reorthogonalization only because of the interesting challenges that the implementation of this technique brings to vector and parallel processing. When full reorthogonalization is used one has only to solve the problems of implementing a good Gram-Schmidt algorithm and a good matrix by vector multiplication routine. The question of tracking convergence and analyze T do not matter as much. The other forms of reorthogonalization, such as Selective Orthogonalization or Partial Reorthogonalization [21] are in fact similar in nature to those of Lanczos without reorthogonalization but more complex to implement.

Here are a few points of detail on SLAN a simple Lanczos code without reorthogonalization, which we implemented and tested.

- No reorthogonalization of any form is used.
- The algorithm computes only the k leftmost or k rightmost eigenvalues of A.
- Eigenvectors associated with these eigenvalues may be computed if desired. If so, these are accumulated from recomputed Lanczos vectors at the end of the process.
- The code eliminates any new eigenvalue that is very close to an already computed eigenvalue.
- Convergence is tracked from left to right for the smallest eigenvalues and from right to left for the largest ones. As a result, we only extract the eigenvalues of the tridiagonal matrix that are likely to converge next. The number of eigenvalues to extract is estimated from the number of desired eigenvalues and the rates of convergence.

4.2 Analyzing T on vector and parallel processors

Although computing the eigenvalues of a tridiagonal matrix in the Lanczos algorithm may seem negligible on a scalar machine, it is no longer the case in a supercomputing environment. The standard algorithm used for computing eigenvalues of tridiagonal matrices is the QL algorithm. In the context of the Lanczos algorithm it is more natural to use bisection type algorithm for several reasons [20]. In order to exploit vectorization we will use a form of multisectionning similar to that proposed by Lo and Philippe [15]. As in [15], we start by a first phase of isolation, where we seek an interval for each eigenvalue that does not contain any other eigenvalue. This process is sequential and consists of sweeping the data points from left to right and to determine which of the points is the closest to a given eigenvalue from the left and then from the right. Once a separate interval has been found for each eigenvalue, we can proceed to the next phase which consists of a refinement phase. Let us assume that we have a set of intervals $[a_j, b_j]$ each of which contains exactly one eigenvalue and let $x_j = (a_j + b_j)/2$ be their middles. To be able to refine a given interval, we may use the usual Sturm sequence property and compute the (divided) Sturm sequences,

$$\sigma_i(x_j) = \alpha_i - x_j - \frac{\beta_i^2}{\sigma_{i-1}(x_j)} \tag{3}$$

starting from $\sigma_0(x_j) = 1$. As is well-known, the number $\nu(x_j)$ of negative signs in the sequence $\sigma_i(x_j), i = 1, 2, .., m$ counts the number of eigenvalues of T_m that are located at the left of x_j. Therefore, the left interval bound a_j will be moved to the middle x_j, whenever $\nu(x_j)$ is less than j otherwise b_j is moved to x_j. Note that the operations to compute the sequence $\sigma_i(x_j)$ is sequential with respect to the subscript i, but it vectorizes across the points x_j.

After a certain number of additional bisections steps we come close enough to the solution that we can use a modified Newton iteration. We call this the third phase of the algorithm.

The Secant method is used instead of the traditional Newton. The iteration is applied to the rational function $\sigma_m(t)$ defined above which, as is well-known, is nothing but the ratio of the characteristic polynomial of T_m over that of T_{m-1}. For any value t that is not an eigenvalue of the intermediate matrices $T_i, i = 1, 2, ..., m-1$, the function $\sigma_m(t)$ can be evaluated with the help of the recursion (3). If we call x_j the current approximation of the eigenvalue in the interval $[a_j, b_j]$, then we start by computing every $\sigma_m(x_j)$ from the above recurrence, and then we define the new iterate by a secant step, namely,

$$x_j := x_j - \sigma_m(x_j)\frac{b_j - a_j}{\sigma_m(b_j) - \sigma_m(a_j)} \tag{4}$$

The algorithm for computing the eigenvalues $\lambda_{m_1}, \ldots, \lambda_{m_2}$, can be sketched as follows.

Vectorized Multisectioning Algorithm (VMSEC)

A. Start:

- For $j = m_1, \ldots, m_2$, define $a_j = a, b_j = b$, with a, b = Gershgorin lower and upper bounds.
- Define $x_j = a + j(b-a)/(m_2 - m_1 + 1), j = m_1, \ldots, m_2$
- Define Phase = 1.

B. Sturm: For $k = 1, ..., m$ Do
Compute (Vector mode) $\sigma(x_j)$ and $\nu(x_j)$ for $j = m_1, \ldots, m_2$.

C. Phase 1: If Phase=1 then,

1. Start $x_* = x_{m_1}$
2. For $i = m_1 \ldots m_2$ Do:
   - Starting from previous x_* determine the smallest x_j that is larger than λ_i. Call this value x_* and denote the old x_* by x_{*-1}.
   - If $b_i > x_*$ then $b_i := x_*$,
   - If $a_i < x_{*-1}$ then $a_i = x_{*-1}$.
3. Test for Phase 2: If $\nu(b_i) = i$ and $\nu(a_i) = i-1$ for all $i = m_1, ..., m_2$ then Phase=2.
4. If Phase=1 then define new set of $x_j's$ by putting in each interval $[a_j, b_j]$ a number of $\nu(b_j) - \nu(a_j)$ equally distributed points.

D.Phase 2: If (Phase = 2) then

1. For $j = m_1, m_2$ Do
   If $\nu(x_j) \geq j$ then $b_j = x_j$ else $a_j = x_j$
2. For $j = m_1, \ldots, m_2$ Do $x_j = (a_j + b_j)/2$.
3. Test for Phase 3: if $b(j) - a(j) < tol * [a_{j+1} - b_{j-1}], j = m_1, ..., m_2$ Phase = 3.

E. Phase 3: If (Phase = 3) Compute new sequence $x_j, j = m_1, \ldots, m_2$ according to (4).

F. Convergence test: If all eigenvalues have converged then exit else goto B.

m_1, m_2	VMSEC					TQL1	TRIDIB
	PHASE 1	PHASE 2	PHASE 3	TOTAL	MFLOPS		
1 , 10	7.1	17	12	26.8	12	-	213
1 , 50	12.3	44.5	2.7	59.5	26	-	1,100
1 , 100	24.2	87.1	5.4	117	27	-	2,200
1 , 500	93.8	440	27.4	550	29	600	10,700

Table 3: Times for the different phases in Algorithm VMSEC on a Cray 2 and for the corresponding EISPACK routines (All times in milliseconds).

Note that the only part that does not vectorize in the above algorithm is Phase 1. One might ask what percentage of the total run time this phase will usually take. In Table 3 we show a break-down of the total time with respect to the three phases. The matrix is the classical $Tridiag[-1, 2, -1]$ of size $n = 500$. We consider four different cases for the values of m_1 and m_2.

As can be seen from Table 3, Phase 1 is by no means dominating even if one computes a large number of eigenvalues. In fact the number of Phase 1 iterations required is usually less than 3 or 4. Then the algorithm proceeds to the most time consuming part which is Phase 2. Phase 3 also takes a small number of steps, usually less than 5.

An interesting comparison to make is with standard routines from EISPACK. We compared VMSEC with both Tridib and TQL1 which are the two competitive routines from the scientific library. Table 3 gives also the times for the three methods for the same matrix and the same machine. We asked Tridib and VMSEC to compute exactly the same eigenvalues and gave the same initial intervals to both routines, namely the interval $[0, 4]$ given by Gershgorin's theorem for the test matrix considered. Note that even for computing all the eigenvalues of a large matrix VMSEC is now competitive with TQL1, the method of choice in this case. We should point out that if we do not need to compute the eigenvectors, as is the case in 'Analyze T', then Cuppen's divide and conquer algorithm [5,7] is not competitive. A remarkable observation is that when computing all the eigenvalues the two algorithms TQL1 and VMSEC are very close even for larger matrices. For computing a smaller number of eigenvalues VMSEC is invariably superior.

Also of importance in the Lanczos algorithm is the computation of residual norms $(A - \lambda_i^{(m)} I)u_i^{(m)}$, which as is well-known [21] can be readily obtained from the last component of the eigenvector $y_i^{(m)}$,

$$\|(A - \lambda_i^{(m)} I)u_i^{(m)}\|_2 = \beta_{m+1}|e_m^T y_i^{(m)}| \tag{5}$$

To compute inexpensively the last component of $y_i^{(m)}$, we exploit the observation made in [20] that this component is equal to the last term of the sequence $\sigma_k(\lambda_i^{(m)})$, which is available for free from the Sturm sequence computation.

4.3 Implementation of Davidson's method

The routine developed computes the lowest eigenvalues of a symmetric sparse matrix A and their associated vectors. The algorithm is expressed in a block version using the Jacobi preconditionner ; it is organized as follows

Initializations : initial guess for the first block ;

Outer loop :

Inner loop :

- CALL MATMULT (multiplication of the last block by A) ;
- CALL MVTW (update of the interaction matrix H) ;
- Computation of the NB lowest eigenvalues and of their associated vectors by EISPACK (TRED1 - TRIDIB - TINVIT - TRBAK1) ;
- CALL RITZ (computation of the Ritz vectors, which are candidate for a new block, and of their residuals) ;
- Test for convergence ;
- Test for ending the inner loop (too many vectors in the basis) ;
- CALL CORRECT (preconditioning every vector of the new block)
- CALL COMPL (projection of the new block onto the orthogonal complement of the subspace already spanned since the beginning of the outer iteration) ;
- CALL GRAMS (Modified Gram-Schmidt orthogonalization on the new block; only are kept the vectors having a significant contribution) ;

CALL GRAMS (On the last Ritz vectors which will be used to start a new outer iteration) ;

The following example illusrates a typical run. The test matrix has been randomly built by defining its sparsity (≈ 0.01), and by choosing the diagonal entries in the range $[-\alpha, 0]$ and the non-zero off-diagonal entries in the range $[-\beta, \beta]$, where $\alpha = 10$ and $\beta = 1$. The sparsity pattern is unstructured. In Table 4, the relative run times of the different parts of the computation are reported with corresponding MFLOPS rates. Convergence was obtained during the 4-th outer iteration. The computation involved 316 sparse matrix $\times$ vector multiplications. The subroutine which performs these multiplications (MATMULT) has to be provided by the user. Here only the upper-part of the matrix has been stored row-wise and the multiplication is performed using the SPAXPY and SPDOT routines. By considering full storage, reordering the rows in decreasing degree and by storing blocks of two jagged diagonals, the sparse multiplication reaches about 27 MFLOPS; in this situation, the corresponding part of the computation drops to 6 % of the overall process.

When calling the routine DAVID, the user defines the block size and the maximum size for the basis. To minimize the number of iterations, he must choose the smallest block size, i.e. the number of sought eigenpairs. However, if he knows that there are some close

computation	% of running time	MFLOPS
EISPACK	40 %	—
MATMULT	36 %	3
COMPL	14 %	37
RITZ	4 %	130
MVTW	4 %	130
Remaining	2 %	—

Table 4: Percentages of times for the different phases in the Davidson Algorithm on a CRAY 2 (1 processor)

eigenvalues, he may define the block size so that the Ritz vectors corresponding to these eigenvalues are computed. For the maximal size of the basis, the user is first limited by the memory capacity. He has also to maintain the orthogonality of the basis ; for that purpose, there is a control on the new vectors which are incorporated to the basis : those which are, before projection onto the complement of the basis, almost spanned by them, are lost. When there is some loss of orthogonality of the basis, the residuals stop to decrease and the Ritz vectors are a worse initial guess for the next outer iteration than they were some inner iterations before.

To conclude, we may assert that the algorithm is well suited for vector processors since the main part of the computation is expressed by vector operations or even by matrix operations. We have mainly tackled the important problem of sparse matrix by vector multiplication. It remains to optimize the solution of the eigenvalue problem that is solved at every step. In the current implementation, the routines of tridiagonalization and back transformation (TRED1 and TRBAK1) are well vectorized. In contrast TRIDIB and TINVIT are sequential ; the first one may be replaced by the algorithm VMSEC but a vector version of TINVIT has yet to be designed. Further improvements might then be achieved by using at one step the previous estimates of the eigenvalues and by computing the eigenvectors by inverse iterations from the untransformed interaction matrix.

4.4 Comparing the two methods

In this section, we discuss the domain of applicability of Davidson's method vs. the Lanczos algorithm. Since we restrict our study to the Jacobi preconditioner, we are interested at the effect of diagonal dominance of the matrix on execution times. Here, the term diagonal dominant matrix is loosely used to refer to a matrix whose diagonal entries vary substantially relatively to the other elements of the matrix. Table 5 reports run times, on a CRAY X-MP, for both methods when looking for the lowest five eigenvalues and corresponding eigenvectors of a randomly generated symmetric sparse matrix. Tests with our codes have been carried out by P. Harten [10]. The matrix is of order 1000 and has a full diagonal whose elements are first taken as random numbers between 0 and 1, and then multiplied by a diagonal scaling factor;

diagonal factor	Davidson	Lanczos
10	6.757	1.455
20	3.969	1.547
30	3.411	1.841
40	2.699	1.950
50	2.380	2.692
60	2.322	2.140
70	2.287	2.396
80	2.168	2.751
90	2.196	7.804
100	1.996	6.705
110	1.993	7.099

Table 5: Davidson - Lanczos run time comparisons (seconds) on a CRAY X-MP (1 processor).

the off-diagonal elements are random numbers between -1 and +1; the positions of these non-zero off-diagonal elements are randomly selected as to give a specified density, namely 0.01. The results clearly show that the methods have an opposite behavior with respect to diagonal dominance. However, it may be asserted that the Lanczos algorithm has a broader spectrum of applicability than Davidson's method since in case of no diagonal dominance, the second method may fail. Davidson's method is efficient in specific applications such as in Quantum Chemistry.

5 Towards Parallel Implementations

Parallelism can easily be exploited in almost all the steps of both methods: matrix or vector operations are well adapted since the length of the involved vectors is large and multiplying by the sparse matrix can be efficiently performed as already seen. Only two bottlenecks occur, namely

1. the eigenvalue problem for the reduced matrix (interaction matrix),
2. the orthogonalization process.

The eigenvalue problem in (1) is different for the two algorithms. In the Lanczos process, the matrix under consideration can be very large (especially if only eigenvalues are required) but it is tridiagonal. In contrast, for Davidson's method the interaction matrix is full and usually of smaller order; as is usually done, it must first be put into tridiagonal form. To parallelize the calculation of the eigenvalues, it is possible to split their range into several pieces which are distributed among all the processors and then apply algorithm VMSEC independently on them. However, if the number of sought eigenvalues is small the algorithm in [15] which relies only on parallelism and not of vectorization, can be more effective.

Parallelizing the computation of the eigenvectors is theoretically much easier since the inverse iteration can be done in parallel for every wanted vector. However, if eigenvalues are clustered then some attention must be paid to maintaining orthogonality. This was addressed in [15].

In these algorithms orthogonalization is performed by the Modified Gram-Schmidt process (MGS):

$$\begin{aligned}&\text{do k = 1, m}\\&\qquad\text{normalize } v_k\\&\qquad\text{do j = k+1, m}\\&\qquad\qquad v_j := v_j - v_k(v_j^t v_k)\end{aligned}$$

where the vectors to be orthogonalized are $v_k, k = 1, ..., m$. Obviously, the inner loop can be run in parallel. To increase the efficiency, it is possible to add synchronizing barriers in order to insure that as soon one vector has received all the corrections from the previous vectors, it is normalized and is then ready to correct the next vectors. On a CRAY X-MP/48 (4 processors) this modification improves the speed-up of the process from 2.2 to 3.2 [15]. Another algorithm consists of replacing in (MGS) the vectors by blocks of vectors; then the normalization of one vector is replaced by applying (MGS) to the block. This algorithm does not have equivalent stability properties but carefully used it can be very efficient, especially to exploit data locality in hierarchical memory systems [11].

The parallelization of the methods does not seem to be too difficult to carry out at least on a moderately large number of processors. Our first tests in this direction indicate that by using CRAY microtasking techniques fairly good speed-ups can be expected.

References

[1] L.M. Adams. *Iterative algorithms for large sparse linear systems on parallel computers.* PhD thesis, University of Virginia, Applied Mathematics, Charlottsville, VA 22904, 1982. Also available as NASA Contractor Report 166027.

[2] O. A. Mc Bryan. *The Connection Machine: PDE solution on 65,536 processors.* Technical Report LA-UR-86-4219, Los Alamos National Lab, Los Alamos, New Mexico, 1986.

[3] T. F. Chan, C. C. Kuo, and C. Tong. *Parallel Elliptic Preconditioners: Fourier Analysis and Performance Evaluation.* Technical Report 88-22, Computational and Applied Mathematics, UCLA, 1988.

[4] J. Cullum and R.A. Willoughby. *Lanczos and the Computation in Specified Intervals of the Spectrum of Large, Sparse Real Symmetric Matrices,* in Sparse Matrix Proc., ed. I.S. Duff and G.W. Stewart, SIAM Publications, Philadelphia, 1979.

[5] J.J.M. Cuppen. *A divide and conquer method for the symmetric tridiagonal eigenproblem.* Num. Math., 36:318–40, 1981.

[6] E. R. Davidson. *The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real symmetric matrices.* J. Comput. Phys., 17:87–94, 1975.

[7] J. Dongarra, DC. Sorensen. *A Fully Parallel Algorithm for the Symmetric Eigenvalue Problem.* SIAM J. Stat. Scient. Comput., 8:2, 1987.

[8] J. Erhel and B. Philippe. *Multiplication of a vector by a sparse matrix on supercomputers.* In M. Cosnard, editor, Proceedings of IFIP working conference on parallel processing, Pisa Italy 1988, North Holland, 1988.

[9] S. W. Hammond and K. H. Law. *Architecture and operation of a Systolic Engine for Finite Element Computations.* Technical Report 88-12, Rensselaer Polytechnic Institute, Computer Science Dept., Troy, NY, 1988.

[10] P. Harten, *CRAY Numerical Software Project Report.* Internal report - Kuck and Associates, Inc.,1988.

[11] W. Jalby, U. Meier. *Optimizing Matrix Operations on a Parallel Multiprocessor with a memory Hierarchy.* Proc. of the 1986 Int'l Conf. on Parallel Processing, St Charles, IL, 1986.

[12] T. I. Karush, N. K. Madsen, and G.H. Rodrigue. *Matrix Multiplication by Diagonals on Vector/Parallel Processors.* Technical Report UCUD, Lawrence Livermore National Lab., Livermore, CA, 1975.

[13] J.G. Lewis. *Algorithms for Sparse Matrix Eigenvalue Problems.* Technical Report STAN-CS-77-595, Department of Computer Science, Stanford University, Stanford, Calif., 1977.

[14] J.G. Lewis and H.D. Simon. *The impact of hardware scatter-gather on sparse gaussian elimination.* SIAM J. Stat. Scient. Comp., 9:304–311, 1988.

[15] S. S. Lo, B. Philippe, and A. Sameh. *A multiprocessor algorithm for symmetric tridiagonal eigenvalue problem.* SIAM J. Stat. Scient. Comput., 8:2, 1987.

[16] R. Melhem. *Solution of Linear Systems with Striped Sparse Matrices.* Technical Report ICMA-86-91, Univ. of Pittsburgh, 1986.

[17] R. B. Morgan and D. S. Scott. *Generalizations of Davidson's method for computing eigenvalues of sparse symmetric matrices.* SIAM J. Sci. Stat. Comput., 7:817–825, 1986.

[18] T. C. Oppe and D. R. Kinkaid. *The performance of ITPACK on vector computers for solving large sparse linear systems arising in sample oil reservoir simulation problems.* Communications in applied numerical methods, 2:1–7, 1986.

[19] C.C. Paige. *The computation of eigenvalues and eigenvectors of very large sparse matrices.* PhD thesis, London University, Institute of Computer Science, London, England, 1971.

[20] B. N. Parlett and B. Nour-Omid. *The use of refined error bounds when updating eigenvalues of tridiagonals.* Lin. Alg. Appl., 68:179–219, 1985.

[21] B.N. Parlett. *The Symmetric Eigenvalue Problem.* Prentice Hall, Englewood Cliffs, 1980.

[22] E.L. Poole and J.M. Ortega. *Multicolor ICCG methods for vector computers.* SIAM J. Numer. Anal., 24:1394–1418, 1987.

[23] A. Ruhe. *Implementation Aspects of Band Lanczos Algorithms for Computation of Eigenvalues of Large Sparse Symmetric Matrices.* Math. Comp. 33, pp. 680-87, 1979.

[24] Y. Saad and H. Wijshoff. *A benchmark package for sparse matrix computations.* In J. Lenfant and D. De groot, editors, Proceedings of ICS conference 1988, St Malo, France, pages 500–509, ACM, 1988.

[25] D.S. Scott. *Analysis of the Lanczos Process.* UCB-ERL Technical Report M78/40, University of California, Berkeley, 1978, PhD Thesis.

[26] R. Underwood. *An Iterative Block Lanczos Method for the Solution of Large Sparse Symmetric Eigenproblems.* Report STAN-CS-75-496, Department of Computer Science, Stanford University, Stanford, Calif., 1975.

[27] D.M. Young, T.C. Oppe, D. R. Kincaid, and L. J. Hayes. *On the use of vector computers for solving large sparse linear systems.* Technical Report CNA-199, Center for Numerical Analysis, Univ. of Texas at Austin, Austin, Texas, 1985.