A Parallel Algorithm for Computing the Extremal Eigenvalues of Very Large Sparse Matrices*

Fredrik Manne

Department of Informatics, University of Bergen, N-5020 Bergen, Norway
Fredrik.Manne@ii.uib.no

Abstract. Quantum mechanics often give rise to problems where one needs to find a few eigenvalues of very large sparse matrices. The size of the matrices is such that it is not possible to store them in main memory but instead they must be generated on the fly.
In this paper the method of coordinate relaxation is applied to one class of such problems. A parallel algorithm based on graph coloring is proposed. Experimental results on a Cray Origin 2000 super computer show that the algorithm converges fast and that it also scales well as more processors are applied.

1 Introduction

Frequently problems in quantum mechanics lead to the computation of a small number of extremal eigenvalues and associated eigenvectors of

$$Ax = \lambda Bx$$

where A and B are real symmetric and sparse matrices of very high order. The matrices are often of such a magnitude that it is neither practical nor feasible to store them in memory. Instead the elements of the matrices are generated as needed, commonly by combining elements from smaller tables.

A number of methods have been proposed for solving such systems. See Davidson [1] for a survey. More recent methods include among others the implicitly re-started Arnoldi iteration [4] by Lehoucq et. al. Common to most of these methods is that they only require the storage of a few dense vectors. In this paper we consider one of the early methods, the method of coordinate relaxation [2, 7] used for computing the smallest eigenvalue of a large sparse symmetric system. The coordinate relaxation method selects each coordinate of the approximate eigenvector and alters it so that the Rayleigh quotient is minimized. This is repeated until a converged solution is obtained. The advantage of the method is that the amount of memory is restricted and, as noted by Shavitt et. al [8], it is also possible to neglect coordinates if their contribution to the overall solution is insignificant.

* For full paper, see http://www.ii.uib.no/~fredrikm

However, the convergence of the method is only guaranteed if the smallest eigenvalue is simple and has a good separation. Moreover, a good approximation of the eigenvalue must exist. If these conditions are met and the matrix is strictly diagonal dominant, the convergence is usually fast [3].

We present an efficient parallel version of this method applicable for sparse matrices. Designing such an algorithm is a non-trivial task since the computation of the contribution from each coordinate in the algorithm depends directly on the previous computations.

We first show that it is possible to perform the selection of which coordinates to use in parallel. The selected coordinates define a sparse graph. By performing a graph coloring on the vertices of this graph it is possible to divide the updating of the necessary variables into parallel tasks. Since the sparse graph changes from each iteration of the algorithm the graph coloring has to be performed repeatedly.

The presented algorithm has been implemented and tested on a 128 processor Cray Origin 2000 computer using diagonal dominant matrices from molecular quantum mechanics. The results show the scalability of the algorithm.

2 The Coordinate Relaxation Method

Here we briefly review the coordinate relaxation (CR) method. For simplicity we assume that $B = I$ such that we are considering the simplified problem $Ax = \lambda x$.

Given a symmetric matrix $A \in \Re^{n \times n}$. We wish to compute the smallest eigenvalue λ and its corresponding eigenvector x.

Given an initial approximation x of the eigenvector and a search direction y we find a new eigenvector $x' = x + \alpha y$ where the scalar α is determined so that the Rayleigh quotient

$$R(x') = \frac{x'^T A x'}{x'^T x'}$$

is minimized.

In the CR method we take y equal to a unit vectors e_i. Combined with the notation

$$F = Ax \tag{1}$$

$$p = x^T A x \tag{2}$$

$$q = x^T x \tag{3}$$

we get the following quadratic equation for determining α

$$\alpha^2(f_i - a_{ii}x_i) + \alpha(p - a_{ii}q) + px_i - f_i q = 0. \tag{4}$$

When α has been determined one must update the values of p, q, x, F, and λ accordingly:

$$p' = p + 2\alpha f_i + \alpha^2 a_{ii} \quad (5)$$

$$q' = q + 2\alpha x_i + \alpha^2 \quad (6)$$

$$x' = x + \alpha e_i \quad (7)$$

$$F' = Ax' = Ax + \alpha A e_i = F + \alpha A_i \quad (8)$$

$$\lambda' = \frac{p'}{q'} \quad (9)$$

Updating p, q, x, and λ involves only a few scalar operations. The time consuming part of the algorithm involves computing F'. This involves not only $nonz(A_i)$ scalar operations but each element of A_i must also be generated. If A is to large to store in memory this must be done on the fly.

In one complete iteration of the CR method one cycles through every coordinate of x. A threshold that is successively lowered is used to determine if a coordinate should be used or not.

3 A Parallel Algorithm

We now present a parallel version of the CR method for sparse matrices. The algorithm operates in three stages. First we consider which coordinates should be used to update the eigenvector. Then we consider how the calculations can be ordered to allow for parallel execution, and finally how the actual computations are performed. Our computational model is a parallel computer with distributed memory. Communication is done by message passing.

The computation of the different values of α is inherently sequential with each step of the algorithm depending on the previous ones. As described in Section 2 the main work of the algorithm is in updating F according to (8).

Finding Candidates We use the initial values of λ, p, q, and F at the start of the iteration when testing each coordinate to see if it contributes enough to the solution. If so, the coordinate is added to the set of candidates that will be used to update the solution. Thus we postpone the updating of the solution until after we have determined which coordinates to use. To ensure that all significant contributions to the solution are acquired we make repeated passes over the matrix before lowering the threshold value.

By dividing the coordinates evenly among the processors we can now determine the candidates in parallel without the need of communication except for distributing the initial values.

Updating the Solution We show how it is possible to postpone and thus accumulate the updating of F. Let K the set of chosen coordinates. To be able to compute α_j the element f_j must be updated by each α_i where $i < j, i \in K$, and $a_{ij} \neq 0$. Let $K = \{C_1, C_2, ..., C_r\}$, $1 \leq r \leq |K|$ be a partitioning of K such that $a_{ij} = 0$ for $i, j \in C_k$ and $1 \leq k \leq r$. If the coordinates in C_1 are applied first, we can compute each α_i, $i \in C_1$, without performing any update on F. This follows from the fact that $a_{ij} = 0$ for $i, j \in C_1$. Thus the updating

of F can be postponed until each α_i, $i \in C_1$ has been computed. Note that the computation of the values of α is sequential. But since this only involves a few scalar operations for each α it can be performed relatively fast. Before we can compute the values of α corresponding to the coordinates in C_2 we must perform an update of F. This can be done in several ways. We choose to immediately perform the complete update of F:

$$F = F + \sum_{i \in C_1} (\alpha_i * A_i) \tag{10}$$

From a parallel point of view we have now restructured the algorithm to consist of fast sequential parts, each one followed by some communication and a potential larger parallel update of F.

To perform the updates on F in parallel we associate the work related to one row of A with one processor making it responsible for updating f_i for each row assigned to it. This requires that each processor has access to the necessary values of α and to the corresponding rows of A.

With this scheme the only communication required is the distribution of the αs. This can be done in one broadcast operation before the parallel update of F. The load balance now depends on how F is distributed and the structure of the rows of A corresponding to coordinates in each C_j. If we distribute x in the same way as F we must gather both x_i and f_i, $i \in C_j$ from each processor before the sequential computation of the α's. We do this on processor 0 which then computes the values of α.

In order to obtain the desired partitioning of K we perform a graph coloring on the adjacency graph $G(K)$. The set C_i now consists of the coordinates whose corresponding vertices are colored with color i. The complete parallel algorithm is as follows:

Parallel Coordinate Relaxation
Calculate initial value of λ and x
Repeat
  Do s times
    Find a set of candidates K
    Perform a graph coloring on $G(K)$
    For each color i:
      Gather f_j and x_j $j \in C_i$ on processor 0
      Processor 0: **For** each $e_j \in C_i$
        Calculate p, q, and α
      Broadcast the values of α
      Update F and x with the coordinates in C_i
  End do
  Lower threshold
  Processor 0: Distribute p and q
Until convergence

Here s is the number of passes we make over the matrix before lowering the threshold.

4 Results

We have performed experiments on a Cray Origin 2000 computer with 128 processors. Here we present results for a matrix from quantum mechanical calculation [5]. The matrix is of order 5189284, and contain on the order of 1.1×10^{11} non-zero elements. The elements are generated on the fly by combining elements from several smaller tables.

Table 1 displays the timings and speedups for one matrix when increasing the number of processors. All times are given in seconds. The quality of the solutions are as good as for the sequential algorithm, and the number of candidates found differ by less than 2%. Finding candidates takes 2.5 seconds on 16 processors

Proc	10	16	26	31	41	51	75	100
Time	1113	662	407	336	260	209	156	128
Speedup	1.0	1.7	2.7	3.3	4.3	5.3	7.1	8.7

Table 1. Execution times and speedup for $m = 38$.

and scales appropriately. The graph coloring takes between 12 and 14 seconds independent of the number of processors used.

To conclude we note that the presented algorithm gives a good speedup on realistic problems but also note that one should be careful to only use the CR method when the convergence criteria are met. Comparisons with the ARPACK parallel package for computing extremal eigenvalues [4] show that for our particular problems the coordinate relaxation method gives an order of magnitude faster convergence.

References

1. E. R. DAVIDSON, *Super-matrix methods*, Computer Physics Communications, (1988), pp. 49–60.
2. D. K. FADDEEV AND V. N. FADDEEVA, *Computational Methods of Linear Algebra*, W. H. Freeman and Co., San Francisco, CA., 1963.
3. G. H. GOLUB AND C. F. V. LOAN, *Matrix Computations*, North Oxford Academic, 2 ed., 1989.
4. R. B. LEHOUCQ, D. SORENSEN, AND P. VU, *ARPACK: An implementation of the implicitly re-started Arnoldi iteration that computes some of the eigenvalues and eigenvectors of a large sparse matrix.* Available from netlib@ornl.gov under the directory scalapack, 1996.
5. I. RØEGGEN. Private communications.
6. A. RUHE, *SOR-methods for the eigenvalue problem with large sparse matrices*, Math. Comp., 28 (1974), pp. 695–710.
7. H. R. SCHWARZ, *The method of coordinate overrelaxation for* $(A - \lambda B)x = 0$, Numer. Math., (1974), pp. 135–151.
8. I. SHAVITT, C. F. BENDER, A. PIPANO, AND R. P. HOSTENY, *The iterative calculation of several of the lowest or highest eigenvalues and corresponding eigenvectors of very large symmetric matrices*, Journal of Computational Physics, (1973), pp. 90–108.