A PARALLEL ALGORITHM FOR COMPUTING THE EXTREMAL EIGENVALUES OF VERY LARGE SPARSE MATRICES

FREDRIK MANNE [*]

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 computer show that the algorithm converges fast ant that it also scales well as more processors are applied. Comparisons show that the convergence of the presented algorithm is much faster on the given test problems than using ARPACK [10].

Key words. sparse matrix algorithms, eigenvalue computation, parallel computing, graph coloring, Cray Origin 2000

AMS subject classifications. 05C50, 05C85, 15A18, 65F15, 65F50, 65Y05, 65Y20

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 [2] for a survey. More recent methods include among others the implicitly re-started Arnoldi iteration [10] 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 [3, 16] 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 to a few vectors and, as noted by Shavitt et. al [17], it is also possible to neglect coordinates if their contribution to the overall solution is insignificant.

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 [6].

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 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 of the vertices of this graph it is possible to divide the updating of the

[*]Department of Informatics, University of Bergen, N-5020 Bergen, Norway, Fredrik.Manne@ii.uib.no . An extended abstract of this paper was published in [11]

eigenvector into parallel tasks. Since the sparse graph changes from each iteration of the algorithm the graph coloring has to be performed repeatedly.

The present algorithm has been implemented and tested on a Cray Origin 2000 computer. In our tests we use diagonal dominant matrices from molecular quantum mechanics. The results show the scalability of the algorithm. We also report on using ARPACK [10] to solve the same problems.

2. The Coordinate Relaxation Method. In this section we review the coordinate relaxation (CR) method and its convergence properties. For simplicity we will 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 (or largest) eigenvalue λ and its corresponding eigenvector x. We denote the i'th column of A by A_i and the number of nonzeros in A_i by $nonz(A_i)$.

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

$$R(x') = \frac{x'^T Ax'}{x'^T x'} = \frac{x^T Ax + 2\alpha y^T Ax + \alpha^2 y^T Ay}{x^T x + 2\alpha x^T y + \alpha^2 y^T y}$$

is minimized.

Differentiating $R(x')$ with respect to α one obtains the equation

$$\frac{dR(x')}{d\alpha} = \frac{a\alpha^2 + b\alpha + c}{x^T x + 2\alpha x^T y + \alpha^2 y^T y}$$

where

$$a = y^T Ax * y^T y - y^T Ay * x^T y \tag{2.1}$$
$$b = x^T Ax * y^T y - y^T Ay * x^T x \tag{2.2}$$
$$c = x^T Ax * x^T y - y^T Ax * x^T x \tag{2.3}$$

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

$$F = Ax \tag{2.4}$$
$$p = x^T Ax \tag{2.5}$$
$$q = x^T x \tag{2.6}$$

we get

$$yAx = f_i \tag{2.7}$$
$$y^T Ay = a_{ii} \tag{2.8}$$
$$y^T y = 1 \tag{2.9}$$
$$y^T x = x_i \tag{2.10}$$

This simplifies the quadratic equation for determining α to

$$\alpha^2(f_i - a_{ii}x_i) + \alpha(p - a_{ii}q) + px_i - f_i q = 0. \tag{2.11}$$

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

$$p' = p + 2\alpha f_i + \alpha^2 a_{ii} \tag{2.12}$$

$$q' = q + 2\alpha x_i + \alpha^2 \tag{2.13}$$

$$x' = x + \alpha e_i \tag{2.14}$$

$$F' = Ax' = Ax + \alpha A e_i = F + \alpha A_i \tag{2.15}$$

$$\lambda' = \frac{p'}{q'} \tag{2.16}$$

Updating p, q, x, and λ involves only a few scalar operations. The most time consuming work 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. However, as noted in [17] it is possible to ignore coordinates if their contribution to λ is insignificant. A threshold that is successively lowered is used to determine if a coordinate should be used or not. This way the most significant updates are performed first. The complete algorithm is as follows:

```
Coordinate Relaxation
Determine initial values for $\lambda$ and $x$
Repeat
  Do $i = 1, n$
      Calculate $\alpha$ using (2.11)
      Calculate the improvement on $\lambda$ by using $\alpha$
      If (improvement $\geq$ threshold)
         Update $p, q, x, F$, and $\lambda$ according to eq. (2.12)- (2.16)
  End Do
  Lower threshold
Until Convergence
```

The CR method can also be used to solve the general eigenvalue problem $(A - \lambda B)x = 0$. If we update x by $x' = \omega x + \alpha y$ where ω is a relaxation factor we obtain the *coordinate overrelaxation method.* If ω is chosen correctly this can speed up the convergence of the method. See [15, 16] for the details. These modifications do not change the algorithm significantly and for clarity we ignore them. Once the smallest eigenvalue λ_1 has been determined it is possible to shift the system to $A - \lambda_1 I$ and use the CR method to determine the second smallest eigenvalue. But as noted by Davidson [2] this does not appear to be efficient. Thus the CR method should mainly be used for determining the extremal eigenvalues.

If one assumes that every coordinate of x is used in one iteration then the computation of F can be viewed as computing the sparse matrix-vector product $F = A\bar{\alpha}$ where the values of $\bar{\alpha}$ must be computed sequentially and α_j depends on α_i, $i > j$, if $a_{ji} \neq 0$. In this way the CR method is very similar to the method of Gauss-Seidel for solving linear systems [6]. This is also reflected in the convergence analysis in the next section.

2.1. Convergence Properties. Consider the general eigenvalue problem $(A - \lambda B)x = 0$. Let x^0 be the the initial eigenvector and x^k the eigenvector after the kth iteration of the algorithm. In [16] it is shown that x converges to the eigenvector corresponding to the smallest eigenvalue λ_1 of A if the following criteria are met:

1. λ_1 is simple.
2. $R(x^0) < \lambda_2$, where x^0 is the starting vector.
3. $R(x^0) < \min_j \frac{a_{jj}}{b_{jj}}$.

Let $C = A - \lambda_1 B = E + D + E^T$ where E is a strict lower matrix and D a diagonal matrix. Then

$$x^{k+1} \approx (\omega E + D)^{-1}[(1-\omega)D - \omega E^T]x^k \equiv M(\omega, \lambda_1)x^k$$

If the relaxation factor $\omega = 1$ and $B = I$ this simplifies to $x^{k+1} \approx (E+D)^{-1}(-E^T)x^k$.

Thus x^{k+1} is asymptotically obtained by multiplying x^k by the iteration matrix $M(\omega, \lambda_1)x^k$. The convergence of the coordinate relaxation method is governed by the theory of the successive over-relaxation method for symmetric and semi-definite linear systems. If the above criteria are met, λ_1 has a good separation, and A is strictly diagonal dominant the convergence of the method is usually fast [6]. For further properties of the CR method and its convergence properties see [16].

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 we discuss how the actual computations are performed. Our computational model is a parallel computer with distributed memory. Communication is done by message passing.

In order to develop an efficient parallel algorithm we must locate independent tasks. This should be done such that the communication/computation ratio is low. Moreover, communication tasks should be grouped in order to reduce the number of times communication is initiated. In this way the accumulated latency of the communication can be kept low.

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 (2.15). We note, however, that once a particular value of α has been determined, F can be updated in parallel. Using coordinate i the total amount of work that can be performed in parallel is then $nonz(A_i)$. For a sparse matrix this is most likely not sufficient for obtaining any significant speedup and the algorithm will be dominated by the time required to distribute the values of α. Thus we seek to accumulate more operations in each parallel step and in this way reduce the time spent on communication.

3.1. Finding Candidates. In order to accumulate more work in each parallel step we first consider how multiple coordinates can be chosen for updating the solution before the actual updates on F are performed.

In the sequential algorithm a coordinate is chosen if its contribution to the current value of λ is larger than the threshold. We suggest using 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. In this way we postpone the updating of the solution until after we have determined which coordinates to use.

This scheme might cause some coordinates that would have been used in the sequential algorithm not to be chosen in the current pass. Similarly, some coordinates that would not have been used in the sequential algorithm might be used. We note that using extra coordinates will not decrease the quality of the solution.

After testing each coordinate the chosen ones are used to update the solution. 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. The only load imbalance that might occur is if some processor finds more candidates than others and have to spend more time in storing its results.

3.2. Updating the Solution. Assuming that we have collected a set K of coordinates as described in Section 3.1 we now consider how it is possible to accumulate more work in each parallel step.

The computation of α, p, q, and λ as specified by eq. (2.11)-(2.13) and (2.16) is inherently sequential. However, these computations involve only a few scalar operations. Updating x according to eq. (2.14) involves only one multiplication and one addition and can be done in parallel at the end of each iteration. We now show how it is possible to postpone and thus accumulate the updating of F.

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 $a_{ij} \neq 0$. But since the coordinates in K were chosen based on their contribution to λ at the *start* of the iteration, it follows that the order in which the coordinates of K are applied is not crucial for their contribution to λ. Thus it is possible to reorder the elements of K.

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: It is possible to update only those values of F whose coordinates are in C_2:

$$f_i = f_i + \alpha_j a_{ij}, \quad i \in C_2, j \in C_1, a_{ij} \neq 0 \tag{3.1}$$

It is also possible to update all the values corresponding to coordinates in $K - C_1$ before proceeding with C_2. A third option is to immediately perform the complete update of F:

$$F = F + \sum_{i \in C_1} (\alpha_i * A_i) \tag{3.2}$$

If we only perform selective updates on F we must perform one step at the end of the iteration where the rest of F is updated:

$$f_i = f_i + \alpha_j a_{ij}, \quad i \notin K, j \in K, a_{ij} \neq 0 \tag{3.3}$$

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. Depending on when we chose to update F we can either try to save as much work as possible towards the end of each iteration or we can perform the work as it is generated. The selective updating schemes require more attention to the structure of A than if one performs the complete update of F for each C_j. We therefore chose to perform a complete update of F for each C_j.

We now consider how the updates on F can be performed in parallel. There are two obvious ways in which this can be done, depending on whether we associate the work related to one column or to one row of A with one processor. In the column scheme one processor calculates the values from $\alpha_i A_i$ that are needed for updating F. The results are then merged using a binary tree into the complete F. This requires at most $\log |C_j|$ communication steps where the partial results are sent. In the row scheme one processor is responsible for updating f_i for each row assigned to it. This requires that each processor has access to the necessary values of α. In both schemes each processor must have access to the corresponding columns (or rows) of A.

We choose the row scheme since the only communication required is the distribution of the α's. 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, $i \in C_j$ before the sequential computation of the α's. We do this on processor 0 which then computes the values of α.

We now address how one can partition K into $\{C_1, C_2, \ldots, C_r\}$ such that $a_{ij} = 0$ for all $i, j \in C_k$, $1 \leq k \leq r$. Construct the adjacency graph $G(K) = (V, E)$ with $|V| = |K|$ and an edge (i, j) if and only if $i, j \in K$ and $a_{ij} \neq 0$. Then C_i must consist of coordinates whose vertices are non-adjacent in $G(K)$. In other words they must form an *independent set.* Partitioning the vertices of $G(K)$ into independent subsets reduces to the well known graph-coloring problem:

> Given a graph G, color the nodes in G using as few colors as possible under the restriction that no adjacent vertices are colored using the same color.

If we order the number of colors used from 1 to r, C_i consists of the coordinates whose corresponding vertices are colored with color i.

Computing the solution using the fewest colors is known to be NP-hard [4], it is also difficult to find a solution that approximates the optimal solution well [7]. However, there exists a number of algorithms that run in linear time and produce relatively good colorings [8].

The complete parallel algorithm is as follows:

```
**Parallel Coordinate Relaxation**
Calculate initial value of $\lambda$ 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 $\alpha$
```

```
            Broadcast the values of $\alpha$
            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. Note again that the calculation of p, q, and α must be carried out in a sequential manner whereas finding candidates and updating F can be carried out in parallel.

4. Numerical Experiments. In this section we present results on applying the parallel CR algorithm developed in Section 3 to problems from quantum mechanical calculations. We first give a description of the test problems.

4.1. Test matrices. The eigenvalue problems that have been used to test the algorithm are due to Røeggen [14]. These problems arise in quantum mechanical calculations of electron correlation energies in molecules using extended geminal models. Røeggen proposed using the sequential CR method for solving these and showed the feasibility of this approach [13].

The test matrices we use are strictly diagonal dominant symmetric matrices where a_{11} gives a good approximation of the smallest eigenvalue λ_1. The size of a matrix A is specified through a parameter m. Each row and column of A is labeled by four indices (i_1, i_2, j_1, j_2) where $1 \le i_2 < i_1 \le m$ and $1 \le j_2 < j_1 \le m$. Thus each element of A is specified by two quadruples: $(i_1, i_2, j_1, j_2), (i'_1, i'_2, j'_1, j'_2)$ giving the row and the column indices of the element. An element of A is non-zero if and only if

$$|\{i_1, i_2\} \cap \{i'_1, i'_2\}| + |\{j_1, j_2\} \cap \{j'_1, j'_2\}| \ge 2. \tag{4.1}$$

Let $S = \frac{m(m-1)}{2}$. Then the dimension of A is S^2 and the number of non-zero elements is given by the polynomial $f(m) = 1.25m^6 - 6.75m^5 + 13.5m^4 - 11.75m^3 + 3.75m^2$. The number of non-zeros in each column of A is $m^2 - 17m + 5$.

The matrix consists of $S \times S$ blocks each of size $S \times S$. Each element of the same block has the same initial coordinates $(i_1, i_2), (i'_1, i'_2)$. The diagonal blocks are full. If $|\{i_1, i_2\} \cap \{i'_1, i'_2\}| = 1$ the block has the same element structure as the block structure of A. If $|\{i_1, i_2\} \cap \{i'_1, i'_2\}| = 0$ the block only contains diagonal elements.

For $m = 8$ this gives a matrix as shown if Figure 4.1. Here the block size S is 28, the dimension of the matrix S^2 is 784, and the number of non-zeros is 156016.

The values of the non-zero elements of A are constructed from two tables h and t giving respectively the one and two electron integrals. Let $T = \frac{1}{2}m(m+1)$. Then the table t is of size T^2 and h of size T. Between 2 and 6 elements are taken from h and t and summed to construct each a_{ij}. It should be noted that although there is some structure in how the elements of t and h are accessed this is not sufficient to achieve good memory locality when generating the columns of A.

4.2. Experimental results. All experiments have been performed on a Cray Origin 2000 supercomputer with 128 MIPS R10000 processors. Each processor has 4 Mbytes of primary cache and 192 Mbytes of memory. Physically this is a distributed memory computer but the operating system also supports shared memory execution. We present results for two matrices, one with $m = 38$ and one with $m = 68$.

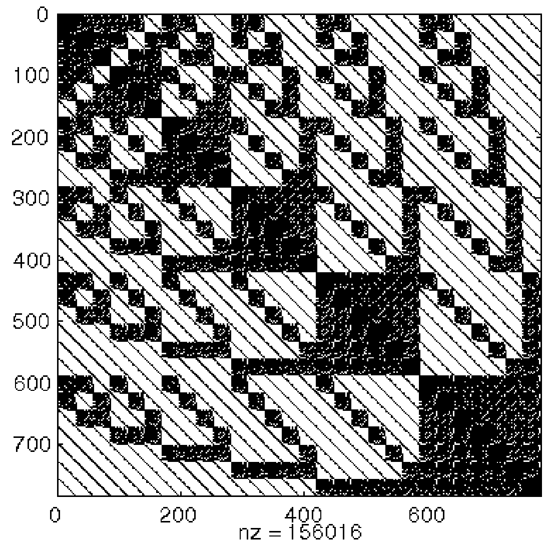

FIG. 4.1. *Structure of A with $m = 8$*

For $m = 38$ and $m = 68$ the order of A is respectively 494209 and 5189284 and contains on the order of 3×10^9 and 1.1×10^{11} non-zero elements. Storing A in a compressed scheme requires 8 bytes for each floating point value and 4 bytes for its row index. In addition one needs S^2 integers for the column pointers. Thus one needs approximately 1.3 Terra bytes to store A in the case when $m = 68$. On todays computers this is not feasible and A is therefore generated as needed.

When the algorithm starts the tables t and h are copied to each processor. The dimension of A is split into groups of S consecutive elements and assigned to the processors in a round robin fashion. Each processor is then responsible for finding candidates among its assigned columns of A and for updating the corresponding elements of F and x. In this way each processor gets blocks of consecutive elements from F and some reuse of index calculations and values from t and h are possible.

When a set of candidates has been selected the structure of its adjacency graph is given implicitly by their indices. Thus there is no need to construct and store the entire graph prior to performing the graph coloring. For each vertex we mark the colors of its adjacent vertices as used and search to see if there is any color available or if a new color is needed. The way in which this search is performed can influences issues such as load balancing and the number of colors used. We choose to perform a linear search. Other schemes such as starting the search from a random point were tried but this did not have any significant impact on the running time.

Two iterations of the algorithm are made before lowering the threshold. The threshold starts at 10^{-5} and is lowered by a factor of 10 down to 10^{-15}.

Tables 4.1 and 4.2 gives the execution time in seconds and the speedup of the parallel CR algorithm for the two matrices. The first column "Proc" denotes the number of processors used, "Total" is the total time of the algorithm, while "Candidates", "Compute", and "Color" gives the time needed for finding candidates, for updating the solution, and for performing the graph coloring.

In Table 4.3 we display more detailed information about the case when $m = 68$ run on 10 processors. Each row displays information about one iteration of the algorithm.

Proc.	Total	Speedup	Candidates	Compute	Color
1	162.4	1.0	1.4	157.7	0.6
5	40.7	4.0	0.9	39.1	0.7
10	19.8	8.2	0.4	18.6	0.7
20	12.3	13.2	0.2	11.3	0.8
30	9.2	17.7	0.1	8.2	0.9
40	8.9	18.2	0.1	7.9	0.9

TABLE 4.1
Execution times and speedup for $m = 38$.

Proc.	Total	Speedup	Candidates	Compute	Color
10	1027	1.0	3.9	1011	11.8
16	662	1.6	2.5	646	13.5
26	407	2.5	1.5	392	13.5
31	336	3.1	1.2	322	12.8
41	260	4.0	0.9	246	13.1
51	209	4.9	0.8	196	12.2
75	156	6.6	0.5	142	13.6
100	128	8.0	0.4	113	14.8

TABLE 4.2
Execution times and speedup for $m = 68$.

The column labeled "$|K|$" gives the number of candidates in each iteration, "nonz" gives the number of edges in the adjacency graph $G(K)$, and "Max deg" the maximum degree of any vertex in $G(K)$. "Col Time" and "Compute" gives the time spent on coloring and updating F in each iteration, "Colors" gives the number of colors used, and λ shows the development of the desired eigenvalue.

For $m = 68$ the number of candidates in each iteration increases from 200 up to 210000. The total number of candidates found over all iterations is just over 700000. These numbers vary slightly with the number of processors used. Note that this is approximately 14 % of the dimension of A. The parallel and sequential algorithms both produce the same answer. For $m = 68$ the difference between the sequential solution and the parallel one was never more than 2×10^{-10} and the number of candidates found differ by less than 2 %.

As can be seen from Tables 4.1 and 4.2 almost all of the time is spent updating F. The time to calculate the values of α and spreading them around is insignificant compared to the time needed to do the updates. The algorithm scales fairly well in the case when $m = 68$. For $m = 38$ the speedup is hampered by the lack of work as the number of processors increases.

For each color the time spent updating F is dominated by the processor with the most work. In the case of $m = 68$ on 10 processors a perfect load distribution would have resulted in an additional speedup of approximately 18 %.

The main obstacle to performing the update of F more efficiently is the unstructured memory references in t when generating A_i. Note that this is not an effect of the CR method but fundamental to the specification of A.

To compare the CR algorithm with other methods we have used ARPACK [10] to solve the eigenvalue problem. ARPACK is an implementation of the implicitly

$\|K\|$	nonz	Max deg	Col Time	Colors	Compute	λ
232	7626	78	0.01	61	0.35	-7.6541875434878532
18	42	6	0.00	6	0.03	-7.6561355264945705
751	69325	750	0.02	190	1.09	-7.6818445199279886
42	262	20	0.01	19	0.08	-7.6823068415640785
1077	117908	555	0.02	228	1.53	-7.6864848434976496
41	262	24	0.01	15	0.07	-7.6865343557708634
1979	159847	1023	0.03	257	2.83	-7.6871480107658794
202	1412	48	0.01	17	0.31	-7.6871714818722117
15248	2339389	922	0.14	225	22.51	-7.6875980661644352
1224	18314	150	0.01	32	1.81	-7.6876133224398302
51164	14037455	1257	0.61	305	73.77	-7.6877872855141725
2768	66972	385	0.02	87	4.20	-7.6877910195333827
90980	31859150	1391	1.27	328	128.40	-7.6878243771514958
4193	112720	466	0.02	107	6.26	-7.6878249335847650
132099	54061398	1440	2.05	348	184.26	-7.6878298362686373
6616	203923	512	0.03	116	9.75	-7.6878299196455533
179214	87415369	1563	3.25	389	246.65	-7.6878306053042396
9346	339152	500	0.05	119	13.68	-7.6878306171439403
217718	119863154	1611	4.36	390	297.14	-7.6878307015535938
10821	414196	461	0.06	109	15.63	-7.6878307028821737

TABLE 4.3
Sizes and times for $m = 68$ run on 10 processors

re-started Arnoldi iteration available through Netlib. To use this one must supply a matrix-vector multiplication routine for computing $y = Ax$ where x is supplied by the package. For large A almost all the computation is spent on this matrix vector product. This computation also lends itself well to parallelization. Initial tests showed that the method did not converge. As suggested by Morgan and Scott [12] we instead applied the algorithm to the preconditioned system $L^{-1}(A - \lambda I)L^{-T}$ where λ is the current estimate of the eigenvalue and $M = LL^T$ is the diagonal of $A - \lambda I$. With this approach the algorithm converged but only after performing more than 50 matrix-vector multiplications. Even if the matrix-vector operations were performed in parallel this took significantly longer time than the sequential CR method.

We have also implemented the sequential CR method and parallelized the updating of F through the use of shared memory. This did not show any significant speed-up. We believe this is due to the fact that the elements of A_i are not generated in row order but in a more unstructured manner. Thus two processors might try to update elements of F that are close enough to be on the same cache-line in memory and thus cause communication.

5. Conclusion.

Through some restructuring of the CR algorithm and the use of a graph coloring algorithm we developed a parallel version of the CR algorithm. Experimental results on very large problems from quantum mechanics showed that the algorithm scaled well as more processors were applied.

The close similarity between the CR algorithm and the Gauss-Seidel algorithm is to a somewhat lesser extent also reflected in parallel implementations of the two algorithms. Koester et. al. [9] presented a parallel Gauss-Seidel algorithm for sparse matrices based (partly) on performing a graph coloring of the matrix. They showed that computations associated with one color can be performed in parallel. This is similar to the presented algorithm. However, the graph coloring is part of a larger reordering step that is only performed initially before the computation starts. Moreover this reordering step takes significant longer time than the numerical solution itself. Thus the time must be amortized over many computations with the same structure.

It is fairly expensive to generate the elements of A since the access of t and h is not done in a systematic enough way. This is typical for these types of problems [2]. Thus performing the updates of F takes sufficiently long time to justify performing the graph coloring in each iteration of the algorithm. If the matrix A had been directly available the graph coloring would most likely have taken a more significant amount of the total time. In this case it would have been possible to try and use a parallel graph coloring algorithm [1, 5]. This was not pursued as the relative time spent on the graph coloring compared to the updating of F is small.

If t and h are large making a separate copy for each processor might require to much memory. It is conceivable that one could design an algorithm where only one copy of t and h is used by all the processors. These would then be stored in a distributed fashion and shared either by making explicit request for elements or through the use of shared memory. However, this would most likely decrease the performance of the algorithm.

The success of the CR method depends on being able to ignore updates that contribute insignificantly to the solution. If this approach could be used in other methods then these might be able to compete with the CR method on the type of problems that were used in the present work.

6. Acknowledgment. The author thanks Inge Røeggen and Tor Sørevik for constructive comments and discussions.

REFERENCES

[1] J. Allwright, R. Bordawekar, P. D. Coddington, K. Dincer, and C. Martin, *A comparison of parallel graph coloring algorithms*, Tech. Report NPAC technical report SCCS-666, Northeast Parallel Architectures Center at Syracuse University, 1994.

[2] E. R. Davidson, *Super-matrix methods*, Computer Physics Communications, (1988), pp. 49–60.

[3] D. K. Faddeev and V. N. Faddeeva, *Computational Methods of Linear Algebra*, W. H. Freeman and Co., San Francisco, CA., 1963.

[4] M. R. Garey and D. S. Johnson, *Computers and Intractability*, Freeman, 1979.

[5] R. K. Gjertsen Jr., M. T. Jones, and P. Plassman, *Parallel heuristics for improved, balanced graph colorings*, J. Par. and Dist. Comput., 37 (1996), pp. 171–186.

[6] G. H. Golub and C. F. V. Loan, *Matrix Computations*, North Oxford Academic, 2 ed., 1989.

[7] M. Halldorsson, *A still better performance guarantee for approximate graph coloring*, Inf. Proc. Letters, (1993), pp. 19–23.

[8] M. T. Jones and P. Plassman, *A parallel graph coloring heuristic*, SIAM J. Sci. Comput., (1993), pp. 654–669.

[9] D. P. Koester, S. Ranka, and G. Fox, *A parallel gauss-seidel algorithm for sparse power system matrices*, in Proceedings of Supercomputing '94, 1994, pp. 184–193.

[10] 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.

[11] F. Manne, *A parallel algorithm for computing the extremal eigenvalues of very large sparse matrices (extended abstract)*, in proceedings of Para98, Workshop on Applied Parallel Computing in Large scale scientific and Industrial Problems, vol. 1541, Lecture Notes in Computer Science, Springer, 1998, pp. 332–336.

[12] R. B. Morgan and D. S. Scott, *Preconditioning the Lanczos algorithm for sparse symmetric eigenvalue problems*, SIAM J. Sci. Comput., 14 (1993), pp. 585–593.

[13] I. Røeggen. Private communications.

[14] I. Røeggen and P. A. Wind, *Electron correlation, extended geminal models, and intermolecular interactions: Theory*, J. Chem. Phys., (1996), pp. 2751–2761.

[15] A. Ruhe, *SOR-methods for the eigenvalue problem with large sparse matrices*, Math. Comp., 28 (1974), pp. 695–710.

[16] H. R. Schwarz, *The method of coordinate overrelaxation for* $(A - \lambda B)x = 0$, Numer. Math., (1974), pp. 135–151.

[17] 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.