Journal of Computational Physics **180,** 256–269 (2002)
doi:10.1006/jcph.2002.7090

# Very Large Electronic Structure Calculations Using an Out-of-Core Filter-Diagonalization Method

Sivan Toledo* and Eran Rabani†

**School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel; and †School of Chemistry, Tel Aviv University, Tel Aviv 69978, Israel*
E-mail: stoledo@tau.ac.il and rabani@tau.ac.il

Received October 7, 2001; revised April 16, 2002

We present an out-of-core filter-diagonalization method which can be used to solve very large electronic structure problems within the framework of the one-electron pseudopotential-based methods. The approach is based on the following three steps. First, *nonorthogonal* states in a desired energy range are generated using the filter-diagonalization method. Next, these states are orthogonalized using the Householder $QR$ orthogonalization method. Finally, the Hamiltonian is diagonalized within the subspace spanned by the *orthogonal* states generated in the second step. The main limiting step in the calculation is the orthogonalization step, which requires a huge main memory for large systems. To overcome this limitation we have developed and implemented an out-of-core orthogonalization method which allows us to store the states on disks without significantly slowing the computation. We apply the out-of-core filter-diagonalization method to solve the electronic structure of a quantum dot within the framework of the semiempirical pseudopotential method and show that problems which require tens of gigabytes to represents the electronic states and electronic density can be solved on a personal computer. © 2002 Elsevier Science (USA)

*Key Words:* electronic structure; singular-value decomposition; $QR$ decomposition; out-of-core.

## 1. INTRODUCTION

One of the most challenging problems of computational physics is to extend the size of systems that can be studied computationally. For example, much attention has been given in recent years to the development of methods for large electronic structure calculations [1, 2].



0021-9991/02 $35.00
© 2002 Elsevier Science (USA)
All rights reserved.

Most electronic structure methods rely on solving an effective single-particle Schrödinger equation,

$$\mathcal{H}\psi_n(\mathbf{r}) = \left\{-\frac{1}{2}\nabla^2 + V(\mathbf{r})\right\} = \epsilon_n\psi_n(\mathbf{r}), \tag{1}$$

where $\psi_n(\mathbf{r})$ are the orthogonal single-particle electronic wavefunctions and $V(\mathbf{r})$ is the total potential of the system. The solution to the Schrödinger equation typically requires the calculation of *all* occupied eigenstates, since in many situations electronic densities and molecular geometries are needed. Furthermore, in many electronic structure theories, such as the Hartree–Fock approach [3] or the Kohn–Sham approximation to the density functional theory [4, 5], the solution to Eq. (1) needs to be done iteratively, since $V(\mathbf{r})$ depends functionally on all occupied solutions.

For small systems that contain a few tens of atoms, direct diagonalization of the Hamiltonian in Eq. (1) in a given basis is possible. The computational effort required by this approach scales as $O(N^3)$, where $N$ is the total number of basis functions. As a result, the application of this approach to large systems is mainly limited by the cubic scaling law. This problem has led to the development of direct minimization techniques such as the conjugate gradient method [1], and to linear-scaling techniques [2]. All these methods rely on the sparsity of the Hamiltonian matrix. The conjugate gradient method requires the storage of all occupied states, a task that becomes impractical for large systems, while the application of the linear-scaling techniques to realistic systems is still limited by the current available computer resources.

In this work we present a new approach, the out-of-core filter-diagonalization (OOC–FD) method, to solve large electronic structure problems. The method is simple and robust and can be implemented on a personal computer. It is based on the following three steps.

- Filtering step: States in a desired range of eigenvalues are generated and stored in files using the filter-diagonalization method [6–8]. These states are nonorthogonal and therefore are not eigenstates of the Hamiltonian; they may even be linearly dependent. This step does not require large memory and can be carried out in parallel.
- Orthogonalization step: The states generated in the first step are orthogonalized using the $QR$ and singular-value decomposition (SVD) methods. This step requires the storage of all states in memory, which for large systems becomes the bottleneck of the computation. To overcome this limitation we have developed and implemented an out-of-core orthogonalization method which allows us to store the states on disks without significantly slowing the computation.
- Diagonalization step: The Hamiltonian is diagonalized within the subspace spanned by the orthogonal states generated in the second step. We use an out-of-core matrix multiplication algorithm to generate the eigenstates of the Hamiltonian within the desired subspace.

While the first filtering step has been used for a wide variate of problems, including the study of electronic properties of large materials [9], the last two steps involve a novel out-of-core algorithm, which is applied to a realistic physical system for the first time. In the application reported below we have used the OOC–FD method to obtain the electronic states of a semiconductor nanocrystal that contains thousands of atoms and requires approximately 70 GB of disk storage to obtain the occupied states. To the best of our knowledge, this is significantly larger than a typical calculation based on a conventional application of electronic structure calculations within the same physical framework.

The OOC–FD method is mostly suited for "single-point" electronic structure calculations and for geometry optimizations. We do not claim that the method is computationally more efficient than other sparse-Hamiltonian methods; however, the structure of the algorithm enables an out-of-core implementation. This is the main advantage of the OOC–FD approach: the computational cost is comparable to other sparse-Hamiltonian methods; however it can be implemented on commodity hardware and applied to large system that require storage of tens of gigabytes. This is not possible using other sparse-Hamiltonian methods, such as the conjugate–gradient method, where an out-of-core implementations will significantly slow the computation, and therefore the application of other methods is mainly limited by the size of main memory.

The paper is structured as follows. In Section 2 we outline the OOC–FD method, and in Section 3 we describe in detail the most challenging phase in the OOC–FD method, namely the second stage that involve a new out-of-core $QR$ and SVD decomposition methods. The code's performance is summarized in Section 4, where we report the performance of the out-of-core stage for a model random matrix. The OOC–FD method is applied to study the electronic properties of a large semiconducting CdSe quantum dot in Section 5. Finally, in Section 6 we present our conclusions.

## 2. THE OUT-OF-CORE FILTER-DIAGONALIZATION METHOD

The overall objective of the out-of-core filter-diagonalization method is to compute the eigenvalues and eigenstates in a given range of eigenvalues of an equation of the general form

$$HC = CE, \tag{2}$$

where $H$ is a *sparse* Hermitian matrix, the columns of $C$ are the coefficients of the eigenstates, and $E$ is a diagonal matrix containing the eigenvalues. Since the matrix $H$ is too large to fit in the main memory and cannot be diagonalized directly, we must use an alternative approach to obtain the desired eigenvalues and eigenstates. The key point in the OOC–FD method outlined below is that the matrix $H$ is *sparse*[1], and that *not all* eigenvalues and eigenvectors are required, only a small set of them.

The OOC–FD method is based on the following three steps.

1. First, *nonorthogonal* states in a desired energy range are generated using the filter-diagonalization method [6–8].
2. Next, these states are orthogonalized using the singular-value decomposition method.
3. Finally, the matrix $H$ is diagonalized within the subspace spanned by the *orthogonal* states generated in the second step.

While the first filtering step has been applied in the past for a wide variety of problems, including the study of electronic properties of nanocrystals [9], the last two steps involve a new out-of-core algorithm. Thus, we briefly describe all three steps in this section and provide a detailed discussion of the orthogonalization step in the next section.

[1] By sparse we mean here that one can operate with the matrix $H$ on an arbitrary vector in the relevant space quickly.

### 2.1. The Filtering Step

The first step in the OOC–FD method is the filtering step. We start with $n_i$ arbitrary initial states $C_i$ that contain all the desired eigenstates. In the application described in Section 5 we use a random initial state $C_i$ with values uniformly distributed between $-1$ and $+1$ and then normalize the random states to unit norm. This choice ensures that the initial states contain contributions from *all* eigenstates. We then apply $n_e$ filters to each initial state of the form

$$C_f = f_e(H)C_i, \tag{3}$$

thus generating $n_f = n_i \times n_e$ filtered states. Each filter, $f_e(H)$, is designed to filter out eigenstates with eigenvalues far from some target value, $E_e$. The choice of the filter function is not unique; in the application described below we use a Gaussian filter [10]

$$f_e(H) = \exp\left\{-\frac{1}{2}\left(\frac{H - E_e}{\sigma_e}\right)^2\right\}, \tag{4}$$

but other filters can be used as well [11]. The filters in Eq. (4) are implemented using a power series in $H$. In this work we have used the Newton interpolation polynomial scheme [12], where the filters were approximated by the interpolation polynomial

$$f_e(H) \approx P_N^e(H) = \sum_{j=0}^{N} a_j(E_e)R_j(H), \tag{5}$$

where

$$R_j(H) = \prod_{k=0}^{j-1}(H - h_k) \tag{6}$$

and the coefficients are given by

$$\begin{aligned} a_0(E_e) &= f_e(h_0), \\ a_1(E_e) &= \frac{f_e(h_1) - f_e(h_0)}{h_1 - h_0}, \\ a_{j>1}(E_e) &= \frac{f_e(h_j) - P_{j-1}^e(h_j)}{R_j(h_j)}. \end{aligned} \tag{7}$$

In the above equations, $h_k$ are the support points taken to be the zeros of the $N+1$ Chebyshev polynomial [13]. This choice defines the points on the interval $[-2, +2]$, and the matrix $H$ is rescaled so that its spectrum of eigenvalues lies in the desired interval

$$H_s = 4\frac{H - E_{min}}{E_{max} - E_{min}} - 2. \tag{8}$$

$E_{min}$ and $E_{max}$ are the lowest and highest eigenvalues of the matrix $H$, respectively. The final result for the filtered states is given by

$$C_f = \sum_{j=0}^{N-1} a_j(E_e)C_i^j, \tag{9}$$

and we use the recursion relation to generate the $C_i^j$ ($C_i^0 = C_i$),

$$C_i^{j+1} = (H_s - h_j)C_i^j. \tag{10}$$

Note that in Eq. (5) only the coefficients $a_j(E_e)$ depend on the target value $E_e$. Hence, it is possible to use the same interpolation polynomial (with different expansion coefficients) to obtain many states simultaneously, and therefore to reduce significantly the computational effort needed to generated the power series in $H$. In addition, since many initial-guess states are required to generate the filtered states, this step can be carried out in parallel by simply filtering out each initial guess state on a different CPU. Finally, to reduce the computational effort in the second step, each set of the filtered states that are generated from a single Newton interpolation polynomial on a random initial-guess state is orthogonalized following the same strategy described below for the second step; however, since there are very few states involved, an in-core algorithm is used.

### 2.2. Out-of-Core Orthogonalization

The second step in the OOC–FD method is the out-of-core orthogonalization step using the SVD decomposition method. Since both the input matrix $C_f$ generated in the filtering step[2] and the orthonormal basis $U$ that is used to reduce the dimensionality of $H$ in the third step are too large to fit in main memory, we must store them on disks. We therefore use the following out-of-core strategy.

- An out-of-core algorithm computes the $QR$ decomposition of $C_f$, $C_f = QR$, using the Householder transformation. The matrix $Q$ whose columns include an orthogonal basis for the column space of $C_f$ is stored on disks since its dimensions are similar to that of the matrix $C_f$. But the matrix $R$, which is a square matrix with dimensions equal to the small dimension of $C_f$, is small enough to fit in main memory.
- An in-core algorithm (from LAPACK [14]) computes the SVD of $R$, $R = U_1 \Sigma V^T$.
- An out-of-core matrix multiplication algorithm computes $U = QU_1$, where $Q$ is stored on disks, usually not explicitly, and $U$ is written to disks. Now $C_f = U \Sigma V^T$ is the SVD of $C_f$.
- We prune from $U$ singular vectors that correspond to zero (or numerically insignificant) singular values. The remaining vectors form $U'$, an orthonormal basis for the columns of $C_f$.

Since this step is the most challenging phase in the OOC–FD method we return to it in Section 3 and describe it in great detail.

### 2.3. Diagonalization

The last step in the OOC–FD method is the diagonalization of the matrix $H$ within the subspace spanned by the orthonormal vectors obtained in the second step. Since the orthonormal basis $U'$ that is used to reduce the dimensionality of $H$ does not fit in main memory, we use the following out-of-core scheme.

- We apply $H$ to $U'$ by reading blocks of columns of $U'$, applying $H$ to them using a matrix–vector multiplication routine that quickly applies $H$, and write the transformed vectors back to disk.

[2] Typically $C_f$ consists of several hundred to several thousands of vectors whose length is between 200,000 and 2,000,000.

- An out-of-core matrix multiplication computes $\hat{H} = (U')^T(HU')$ to produce the in-core product of two out-of-core matrices.
- An in-core algorithm (from LAPACK [14]) diagonalizes $\hat{H}$ to obtain the diagonal matrix $E$ in Eq. (2).
- An out-of-core matrix multiplication algorithm computes $C = U'\hat{U}$, where $\hat{U}$ is the matrix that diagonalized $\hat{H}$, and $C$ are the coefficients of the eigenstates of $H$ (cf. Eq. (2)) in the given energy range.

## 3. THE OUT-OF-CORE *QR* DECOMPOSITION

We now describe in detail the most challenging phase in the out-of-core SVD of tall narrow matrices, namely the out-of-core $QR$ decomposition. Readers who are not interested in these details may proceed to the next section.

Since the input matrix is tall and narrow, as shown in Fig. 1, we cannot use a conventional block-column approach for the $QR$ phase. We use instead a recursive block-Householder $QR$ algorithm due to Elmroth and Gustavson [15, 16] in order to achieve a high level of data reuse. The locality of reference in block-column approaches depends on the ability to store a fairly large number of columns in main memory. In our case, we often cannot store more than 10 columns in main memory, even on machines with several gigabytes of main memory. Recursive formulations of decomposition algorithms that must operate on full columns, such as $QR$ and $LU$ with partial pivoting, enhance locality of reference over block-column formulations for matrices of all shapes. As a result, recursive formulations perform better because they perform fewer cache misses and because they require less I/O in out-of-core codes [15–17]. But while the benefit of recursive formulations is small when processing square matrices, the benefit is enormous for tall narrow matrices, as shown for the $LU$ decomposition by Toledo in [17] and for the $QR$ decomposition by Elmroth and Gustavson in [15, 16]. As a result, our algorithm performs the $QR$ decomposition at rates that are not much slower than in-core computational rates.

We use a block-Householder $QR$ algorithm rather than the cheaper modified Gram–Schmidt $QR$ algorithm since the columns of $C_f$ in our application are often linearly dependent, and in such cases neither classical nor modified Gram–Schimdt is guaranteed to return an orthogonal $Q$ due to rounding errors (see, for example, [18], Chap. 18). Classical and modified Gram–Schimdt perform approximately $2mn^2$ floating-point operations (flops) when $C_f$ is $m$-by-$n$ and $m \gg n$, whereas Householder performs approximately $4mn^2$, but the extra cost is essentially unavoidable when $C_f$ is rank deficient.

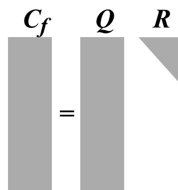


**FIG. 1.** A tall and narrow $QR$ factorization of a matrix $C_f$. $C_f$ is a general rectangular matrix, $Q$ has orthonormal columns, and $R$ is upper triangular. In our application, the rows/columns ratio of $C_f$ is approximately 500/1.

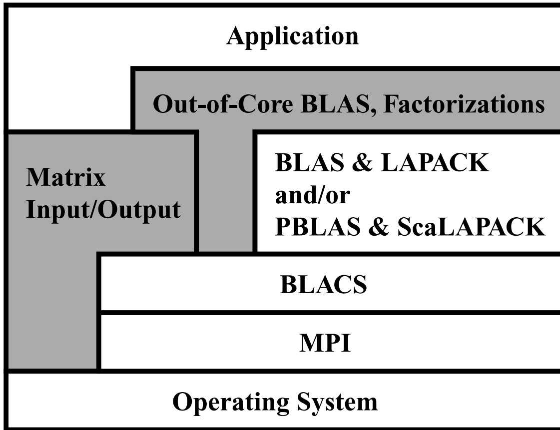


**FIG. 2.** The software structure of SOLAR. The gray areas represent components of SOLAR; the white areas represent other software components. The BLACS are the communication routines of ScaLAPACK; SOLAR uses them as well. The BLACS use, in turn, MPI, a message-passing interface, which uses the operating systems. Solar's matrix input–output routines use the operating system to perform input–output.

We prefer to compute the SVD of $C_f$ rather than a rank-revealing $QR$ factorization because the extra expense of computing the SVD of $R$ is insignificant in our application, since the input matrix is tall and thin. In addition, we are not aware of an efficient column-pivoting scheme for out-of-core matrices. In other words, the column-pivoting actions of a rank-revealing $QR$ factorization are likely to increase the amount of I/O in an out-of-core factorization, but the savings in floating-point arithmetic over the SVD are insignificant when the matrix is thin and tall. More specifically, a rank-revealing $QR$ factorization of $C_f$ performs about $4mn^2 - 2n^3/3$ flops, whereas the SVD performs about $4mn^2 + 11n^3$ (see [19], Section 5.5.9). For $m \gg n$, the difference is negligible.

We implemented the new out-of-core $QR$ and SVD algorithms as part of SOLAR [20], a library of out-of-core linear algebra subroutines, whose overall structure is shown in Fig. 2. Before we started the current project, SOLAR already included sequential and parallel out-of-core codes for matrix multiplication, solutions of triangular linear systems, Cholesky factorizations, and $LU$ factorizations with partial pivoting. SOLAR can exploit shared-memory parallel processing, distributed-memory parallel processing (or both), parallel input–output, and nonblocking input–output. SOLAR exploits distributed-memory parallel processing using explicit calls to ScaLAPACK, the PBLAS (ScaLAPACK's parallel BLAS routines), and the BLACS (ScaLAPACK's communication routines). ScaLAPACK and the PBLAS perform communication only through the BLACS, which use MPI internally. Due to the use of ScaLAPACK, the PBLAS, and the BLACS, the code is highly portable, at least among Unix and Linux platforms. SOLAR can process real and complex matrices, single or double precision.

The main new addition to SOLAR is the out-of-core $QR$ factorization. The new code is optimized for tall narrow matrices and uses existing subroutines extensively to multiply matrices and to solve triangular systems. The focus on tall narrow matrices allows us to simplify the code in two ways, which would not be possible if the code were to work effectively on square or nearly square matrices. First, focusing on tall narrow matrices allowed us to use the SVD rather than a rank-revealing $QR$ factorization without significant

performance implications. Second, focusing on tall narrow matrices allows us to assume that $n$-by-$n$ matrices fit in main memory.

One unique feature of SOLAR was particularly valuable in the implementation of the $QR$ solver. Most SOLAR routines, such as the matrix multiplication routine (out-of-core GEMM), can process any mix of in-core and out-of-core arguments. For example, SOLAR can multiply an out-of-core matrix by an in-core matrix and add the product to an out-of-core matrix. During the recursive $QR$ factorization of a tall narrow matrix we often multiply a large matrix, which we must store out-of-core, by a small matrix that we prefer to leave in-core, so this feature of SOLAR is helpful. On the other hand, SOLAR still lacks some subroutines that could have been useful, such as a triangular matrix multiplication routine (TRMM). Consequently, we had to use instead the more general GEMM routine, which causes the code to perform more floating-point operations than necessary. This overhead is relatively small, however.

We have also modified the I/O layer of SOLAR over the one described in [20]. The changes allow SOLAR to perform nonblocking I/O without relying on operating-system support (which sometimes performs poorly), they allow SOLAR to perform I/O in distributed-memory environments without a data-redistribution phase, and they allow SOLAR to perform I/O without allocating large auxiliary buffers. These changes allow the algorithms to control main memory usage more accurately and more easily than before.

As in many other applications of out-of-core numerical software [21], our primary goal was to be able to solve very large systems, not necessarily to solve them quickly. The largest computer currently at our disposal has only 14 GB of main memory, so we simply cannot solve very large systems without an out-of-core algorithm. While we would like to solve large systems quickly, a running time of a day or two, perhaps up to a week, is entirely acceptable to us, mainly because the SVD code is part of a larger application and it is not the most time-consuming part, only the most memory consuming. We therefore used the following rule of thumb while developing the code: keep the amount of I/O low to achieve acceptable performance, but do not try to eliminate small amounts of I/O if doing so requires a significant programming effort.

As a consequence of this design decision we were able to design and implement the algorithm relatively quickly using existing SOLAR subroutines. The resulting algorithm often achieves over 60% of the peak performance of the computer, but it could probably achieve more if more I/O is optimized away. I/O could be eliminated by implementing out-of-core triangular matrix multiplication routines in SOLAR (which currently only has a routine for general rectangular matrices) and by avoiding the storage and retrieval of blocks of explicit zeros. The number of floating-point operations would also be reduced by applying these optimizations.

When developing algorithms and codes for very large systems, one must consider parallel algorithms and implementations. Designing and implementing a parallel algorithm requires more effort than a sequential algorithm, but the performance of the parallel code typically scales better given a sufficient number of processors. We have decided to implement a sequential out-of-core $QR$ decomposition algorithm, rather than a parallel out-of-core algorithm. In our application, the filtering stage, which has been parallelized, requires about 10 times more CPU time than the orthogonalization step. Therefore, parallelizing the orthogonalization step can only improve the scalability of the overall application when a fairly large number of processors is used. Since we designed the application for small clusters, parallelizing the orthogonalization step did not seem urgent and was left open for future study.

We use a recursive out-of-core algorithm for computing the compact-$WY$ representation of $Q$, $Q = I - YTY^T$. The basic in-core formulation of this algorithm is due to Elmroth and Gustavson [15, 16]. The input of the algorithm is $C_f$ and its output is the triplet $(Y, R, T)$. The algorithm factors an $m$-by-$n$ matrix as follows.

1. If $n = 1$, then compute the Householder transformation $Q = I - tyy^T$ such that $QC_f = (r, 0, \ldots, 0)^T$ ($t$ and $r$ are scalars, $y$ is a column vector). Return the triplet $(y, r, t)$. We have $Y = y$, $T = t$, and $R = r$.
2. Otherwise, split $C_f$ into $[C_1 C_2]$, where $C_1$ consists of the left $n_1 = \lfloor n/2 \rfloor$ columns of $C_f$ and $C_2$ consists of the right $n_2 = n - n_1$ columns.
3. Compute recursively the factorization $(Y_1, R_{11}, T_{11})$ of $C_1$.
4. Update $\tilde{C}_2 = Q_1^T C_2 = (I - Y_1 T_{11} Y_1^T) C_2$.
5. Compute recursively the factorization $(Y_2, R_{22}, T_{22})$ of the last $m - n_1$ rows of $\tilde{C}_2$.
6. Compute $T_{12} = -T_{11}(Y_{11}^T Y_{22}) T_{22}$.
7. Form $R_3$, which consists of the first $n_1$ rows of $\tilde{C}_f$.
8. Return

$$\left( [\, Y_1 \quad Y_2 \,], \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}, \begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{bmatrix} \right).$$

Memory management, both in- and out-of-core, is an important aspect of out-of-core algorithms. Our recursive $QR$ code works with one $m$-by-$n$ out-of-core matrix and three in-core $n$-by-$n$ matrices. The out-of-core matrix initially stores $C_f$ and is overwritten by $Y$. One of the small in-core matrices is passed in by the user as an argument to receive $R$. The code allocated internally two more matrices of the same size, one to store $T$ and the other, denoted $Z$, as temporary storage. The remaining main memory is used by the algorithm to hold blocks of $C_f$ and $Y$ that are operated upon.

The out-of-core implementation of the recursive $QR$ algorithm does not stop the recursion when $n = 1$ but when $n$ is small enough so that the block of $C_f$ to be factored fits within the remaining main memory (taking into account the memory already consumed by $R$, $T$, and $Z$). If the block of $C_f$ fits within main memory, the code reads it from disk, computes $(Y, R, T)$ in core, and writes $Y$ back to disk, overwriting columns of $C_f$. The in-core factorization algorithm is, in fact, an implementation of the same recursive algorithm. We use this recursive algorithm rather than an existing subroutine from, say, LAPACK [14], because the matrices that this routine must factor are extremely thin, such as 2,000,000 by 20, and as shown in [15, 16], the recursive algorithm outperforms LAPACK's blocked algorithm by a large factor in such cases. (The in-core $QR$ factorizations of narrow panels constitute a small fraction of the total work in this algorithm, however, so this optimization is unlikely to significantly impact the total running time.)

If the block of $C_f$ to be factored does not fit within main memory, the algorithm splits it into $C_1$ and $C_2$ and factors the block recursively. Computing $\tilde{C}_2 = Q_1^T C_2 = (I - Y_1 T_{11} Y_1^T) C_2$ is done in three out-of-core steps, each of which involves a call to SOLAR's out-of-core matrix-multiply-add routine: $\tilde{C}_2 = \tilde{C}_2 + Y_1(T_{11}(Y_1^T C_2))$. This process is shown in Fig. 3. The first intermediate result $Y_1^T C_2$ is stored in the $Z_{12}$ block and the second intermediate result in $T_{12}$ (which is still empty). Next, the code reads the first $n_1$ rows of $\tilde{C}_2$ into $R_{12}$. The code then writes out a block of zeros into the first $n_1$ rows of $C_2$, since $Y$ is lower trapezoidal, and recursively factors the last $m - n_1$ rows of $C_2$.

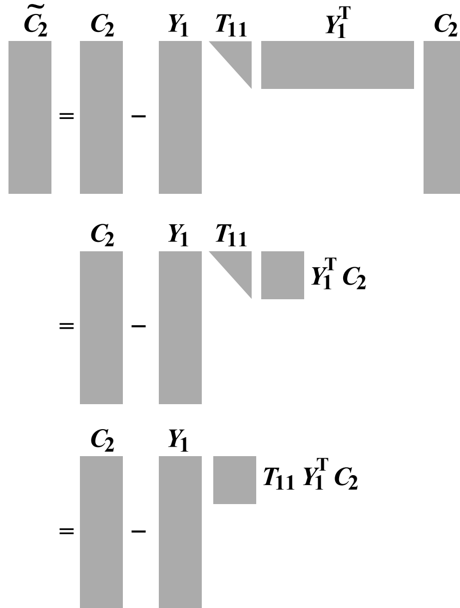


**FIG. 3.** Updating $\tilde{C}_2$.

We compute $T_{12} = (-T_{11}(Y_{11}^T Y_{22}))T_{22}$ in three steps, using $T_{12}$ for the first intermediate result $(Y_{11}^T Y_{22})$, and $Z_{12}$ for the second intermediate result. The first multiplication multiplies two out-of-core matrices, the remaining two multiply in-core matrices. We then zero $T_{21}$ and $R_{21}$, since both $T$ and $R$ are upper triangular.

Following the computation of the compact-$WY$ representation of $Q$ our code actually proceeds to compute $Q$ or the SVD, depending on the routine called by the user. If the user requested a $QR$ decomposition, the code uses SOLAR's out-of-core matrix multiplication to compute the first $n$ columns of $Q = I - I - YTY^T$. If the user requested an SVD, the code first computes the SVD $U_1 \Sigma V^T$ of $R$ in-core and then applies $Q$ to $U_1$ to get the left singular vectors $U$ of $C_f$. The best way to apply $Q$ is to use the compact-$WY$ representation directly and apply $I - YTY^T$ directly to $U_1$. Our code currently uses a slightly less efficient method but we plan to improve it.

## 4. PERFORMANCE OF THE OUT-OF-CORE *QR* FACTORIZATION

Table I summarizes the results of three performance-evaluation experiments that were designed to assess the performance of the out-of-core $QR$ factorization code. The experiments used our $QR$ code to factor double-precision-real $m$-by-$n$ matrices. In three of the experiments we used matrices with random elements (i.e., the matrices were not produced by filter diagonalization); the exception is the experiment on Pentium B, which is part of the filter-diagonalization run described in the next section. We chose the sizes for the random matrices so they approximate our needs in a realistic physical application.

Two experiments were conducted on a 600 MHz dual Pentium III machine running Linux (denoted Pentium A), another on a similar machine with a different disk configuration (denote Pentium B), and a fourth and another on the 400-MHz, 112-bit-processor SGI Origin 2000. We used only one processor on Pentium A, the Origin, two on Pentium B. Pentium A did not have sufficient attached disk space, so we used four other similar machines

**TABLE I**
**The Performance of Our Out-of-Core *QR* Factorization Code, Including the Formation of the Explicit *Q***

| Machine | Mem. | $m$ | $n$ | $n_0$ | $T$ | $T_{ic}$ | $T_{io}$ | $M$ | $M_{ic}$ |
|---|---|---|---|---|---|---|---|---|---|
| Pentium A | 1.5 GB | $10^6$ | $10^3$ | 120 | 39,932 | 12,280 | 27,647 | 100 | 325 |
| Pentium A | 1.5 GB | $5 \times 10^5$ | $2 \times 10^3$ | 260 | 54,339 | 23,373 | 30,960 | 147 | 342 |
| Pentium B | 1.5 GB | $2 \times 10^6$ | 4288 | 60 | 401,739 | 222,172 | 179,360 | 384 | 694 |
| Origin 2000 | 2 GB | $2 \times 10^6$ | $2 \times 10^3$ | 70 | 122,379 | 69,722 | 52,630 | 261 | 459 |

*Note.* The table shows the machine used (one processor was used in all cases), the amount of main memory that was actually used, the number of rows $m$ and columns $n$ in $C_f$, the number $n_0$ of columns that the code was able to process in-core, the total factorization time (in seconds), the time spent on in-core computations, and the time spent on I/O. The last two columns show the computational rate $M$ of the entire factorization in millions of floating-point operations per second (Mflops), and the computational rate of the in-core computations alone.

as I/O servers. Communication between the machine running the code and the I/O servers was done using a remote I/O module that is part of SOLAR. The I/O servers used one 18-GB SCSI each, and were connected to the other machine using fast Ethernet (100 Mbits/sec). The effective I/O rate that we measured on this setup was about 9.8 MB/sec. Pentium B had sufficient local disk space, with a transfer rate of approximately 20 MB/sec. The Origin had an attached disk array with approximately 300 GB.

The purpose of the random-matrix experiments was to assess the impact of I/O on the overall performance of the orthogonalization code. I/O impacts most the $10^6$-by-$10^3$ Pentium run, in which I/O takes 69% of the total running time. The next Pentium run, which orthogonalized a wider and shorter matrix, performed better, with only 57% of the total running time devoted to I/O. The difference is a result of the recursive nature of the algorithm, which recurses only on the columns of the matrix, not on the rows. On a wide matrix, the recursion is deeper and a large fraction of the total work is performed on at the top levels, where out-of-core performance is good. On a narrow matrix a significant amount of work is performed at the bottom levels of the recursion, where the level of reuse of data in main memory is low. The Origin experiment confirms our expectation that faster I/O reduces the fraction of the running time devoted to I/O.

The main conclusion from these results is that on these machines, the code runs at 30−55% of the effective peak performance of the machine and is hence highly usable. Clearly, on faster machines or machines with slower I/O or on even narrower problems, I/O would become a bottleneck. On the other hand, wider problems should lead to better performance.

## 5. APPLICATION TO THE ELECTRONIC STRUCTURE OF CDSE QUANTUM DOTS

In this section we describe the use of the OOC-FD method to study the electronic structure of a realistic system, namely a large semiconducting CdSe quantum dot. The electronic structure of the CdSe quantum dot was described within the framework of the empirical pseudopotential method [22]. In this approximation the electronic states of the quantum dot were computed from a single-electron picture similar to a density functional approach. We used a screened nonlocal pseudopotential developed recently by Wang and Zunger

[23] for the cadmium and selenium atoms. These pseudopotentials produce local-density approximation-quality wavefunctions with experimentally fit bulk band structure and effective masses. The full-scale version of the pseudopotentials without including uncontrolled approximations that result in a reduction of the energy range of the Hamiltonian were used. For simplicity the spin–orbit interactions were neglected.

We represent the electronic wavefunction on a three-dimensional grid in real space, rather than the traditional plane-wave basis, with grid spacing of approximately 0.5 atomic units. In this representation both the nonlocal potential and the kinetic operators can be evaluated using linear-scaling methods (finite difference), or alternatively one can use the more accurate fast Fourier transform (FFT) method which has $O(N \log N)$ scaling, where $N$ is the number of grid points. Both choices ensure that the Hamiltonian matrix is sparse, and the OOC-FD method can be applied to obtain the desired eigenvalues and eigenvectors.

The columns of $C_f$, the matrix whose columns span the desired eigenspace, were generated by the filtering processes that start from random initial vectors, as described in Section 2. Each filtering process generates approximately 10 orthogonal columns of $C_f$ in our desired energy range ($-25-0$ eV) for a Newton interpolation length of 1024. Since the filtering processes are completely independent, we ran many of them on a cluster of Linux workstations or on multiple processors of a parallel computer (the 112-bit-processor SGI Origin 2000 in our case). At the end of each filtering process we stored the columns that were generated in a separate file. The total computational effort to generate the filtered states was approximately 1000 CPU hours.

Once the filtering processes was terminated and the output files were ready, our out-of-core $QR$ code collected the columns of $C_f$ from these files, where multiple columns were stored one after the other. All the columns were collected into one SOLAR matrix file, which was stored by block to optimize disk accesses. Our code can collect filter output files from files stored on locally accessible file systems (typically local disks or NFS mounted file systems) or on remote machines. The code collects columns from remotely stored files using `scp`, a secure remote file copying program.

Next, we computed the SVD, $U \Sigma V^T$, of $C_f$. We used the singular values to determine the numerical rank $r$ of $C_f$. We then used the first $r$ columns of $U$, corresponding to the $r$ largest singular values, as an orthonormal basis for $C_f$, to reduce the order of $H$, computed the eigenvalues and eigenvectors of the reduced Hamiltonian $\hat{H}$ and then the eigenvectors of $H$. We assumed that three $r$-by-$r$ matrices of size $r$ fit in the main memory, which allowed us to compute the eigendecomposition of $\hat{H}$ in the main memory (we used LAPACK's `DSYEV` routine).

The results for the electronic density of states are shown in Fig. 4 for the largest system studied so far. The total number of atoms in the quantum dot is 1277, with 648 cadmium atoms and 629 selenium atoms. The dangling bonds of the cadmium and selenium surface atoms were terminated with ligand potentials to eliminate all surface states from the band gap [9, 24]. The total number of states that were generated in the filtering process is 4288, somewhat larger than the number of occupied states (2515), to ensure that all occupied states are recovered in the process. The number of grid points in each dimension was 128, resulting in a $QR$ decomposition of a matrix of size 2,097,152 by 4,288, which amounts to 67 GB of memory! The resulting HOMO–LUMO band gap is 2.19 eV, in agreement with other approximate methods [9, 24].

The out-of-core steps (steps two and three) were obtained on a 600-MHz dual Pentium III machine with 2 GB of main memory and four 75-GB, 7200-rpm IDE disks, running Linux.

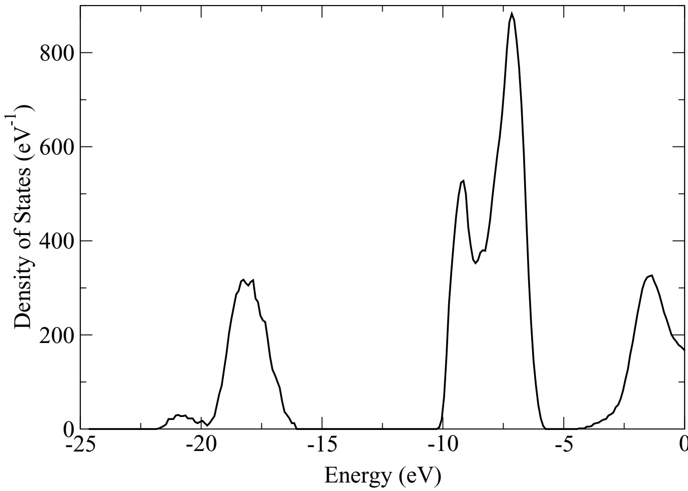


**FIG. 4.** The electronic density of states of a large CdSe quantum dot computed within the framework of the semiempirical pseudopotential method using the out-of-core filter-diagonalization method.

The disks were controlled by an ATA-33 controller. The transfer rate of a single disk was approximately 20 MB/s. The orthogonalization step took 401,739 s, consisting of 222,172 s in actual numerical computations and 179,360 s in I/O. These numbers are also reported in Table I. We then performed the diagonalization step which consists of three out-of-core matrix multiplications, which took 137,341, 118,313, and 134,336 s, respectively. The main conclusion that can be drawn from these results is that the I/O in the out-of-core orthogonalization step slows the computation by less than a factor of two!

## 6. SUMMARY

We have presented an out-of-core filter-diagonalization method that computes the eigenvalues and eigenvectors of a large sparse matrix within a desired range of eigenvalues. The method is based on the following three steps. The first filtering step produces nonorthogonal states in a desired range of eigenvalues. These states are then orthogonalized using the out-of-core SVD decomposition method. Finally, the Hamiltonian is diagonalized within the subspace spanned by the orthogonal states generated in the second step. We have demonstrated that the code is efficient and that it can be used to solve problems whose size is much bigger than main memory. This has been shown both for a random matrix model, and for a more realistic physical system.

We believe that the OOC-FD method would be useful for more realistic electronic structure theories, such as the density functional theory. The main advantage of the OOC-FD approach over other existing methods is that a *single* orthogonalization of all states is required. This is the key point for an out-of-core algorithm to succeed. We also expect that the out-of-core algorithm would be useful in other situations that require large-scale orthogonalization.

## ACKNOWLEDGMENTS

This research was supported by the Israel Science Foundation, founded by the Israel Academy of Sciences and Humanities (grant 572/00, grant 9060/99, grant 34/00, and grant 9048/00) and by the University Research Fund

of Tel Aviv University. Access to the SGI Origin 2000 was provided by Israel's High-Performance Computing Unit.

## REFERENCES


1. M. C. Payne, M. P. Teter, D. C. Allan, T. A. Arias, and J. D. Joannopoulos, Iterative minimization techniques for ab initio total-energy calculations: Molecular dynamics and conjugate gradients, *Rev. Mod. Phys.* **64**, 1045 (1992).
2. S. Goedecker, Linear scaling electronic structure methods, *Rev. Mod. Phys.* **71**, 1085 (1999).
3. A. Szabo and N. S. Ostlund, *Modern Quantum Chemistry* (Dover, New York, 1996).
4. P. Honenberg and W. Kohn, Inhomogeneous electron gas, *Phys. Rev.* **136**, 864B (1964).
5. W. Kohn and L. J. Sham, Self-consistent equations inclusing exchange and correlation effects, *Phys. Rev.* **140**, 1133A (1965).
6. D. Neuhauser, Bound state eigenfunctions from wave packets: Time → energy resolution, *J. Chem. Phys.* **93**, 2611 (1990).
7. D. Neuhauser, Time-dependent reactive scattering in the presence of narrow resonances: Avoiding long propagation times, *J. Chem. Phys.* **95**, 4927 (1991).
8. M. R. Wall and D. Neuhauser, Extraction, through filter-diagonalization, of general quantum eigenvalues or classical normal mode frequencies from a small number of residues or a short-time segment of a signal. 1. Theory and application to a quantum-dynamics model, *J. Chem. Phys.* **102**, 8011 (1995).
9. E. Rabani, B. Hetenyi, B. J. Berne, and L. E. Brus, Electronic properties of CdSe nanocrystals in the absence and presence of a dielectric medium, *J. Chem. Phys.* **110**, 5355 (1999).
10. A. D. Hammerich, J. G. Muga, and R. Kosloff, Time-dependent quantum mechanical approaches to the continuous spectrum: Scattering resonances in a finite box, *Israel J. Chem.* **29**, 461 (1989).
11. V. A. Mandelshtam and H. S. Taylor, A low-storage filter diagonalization method for quantum eigenenergy calculation or for spectral analysis of time signals, *J. Chem. Phys.* **106**, 5085 (1997).
12. R. Kosloff, Propagation methods for quantum molecular dynamics, *Annu. Rev. Phys. Chem.* **45**, 145 (1994).
13. M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions* (Dover, New York, 1972).
14. E. Anderson *et al.*, *LAPACK Users' Guide*, 3rd ed. (Soc. for Industr. & Appl. Math., Philadelphia, PA, 1999).
15. E. Elmroth and F. Gustavson, New serial and parallel recursive QR factorization algorithms for SMP systems, in *Applied Parallel Computing: Large Scale Scientific and Industrial Problems*, Lecture Notes in Computer Science, edited by B. K. Gström *et al.* (Springer-Verlag, Berlin/New York, 1998), Vol. 1541, p. 120.
16. E. Elmroth and F. Gustavson, Applying recursion to serial and parallel QR factorization leads to better performance, *IBM J. Res. Dev.* **44**, 605 (2000).
17. S. Toledo, Locality of reference in LU decomposition with partial pivoting, *SIAM J. Matrix Anal. Appl.* **18**, 1065 (1997).
18. N. J. Higham, *Accuracy and Stability of Numerical Algorithms* (Soc. for Industr. & Appl. Math., Philadelphia, 1996).
19. G. H. Golub and C. F. V. Loan, *Matrix Computations*, 3rd ed. (Johns Hopkins Univ. Press, Baltimore, 1996).
20. S. Toledo and F. G. Gustavson, The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations, in *Proceedings of the 4th Annual Workshop on I/O in Parallel and Distributed Systems* (The Association for Computing Machinery, Philadelphia, 1996), p. 28.
21. S. Toledo, A survey of out-of-core algorithms in numerical linear algebra, in *External Memory Algorithms*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, edited by J. M. Abello and J. S. Vitter (Am. Math. Soc., Providence 1999), p. 161.
22. M. L. Cohen and J. R. Chelikowsky, *Electronic Structure and Optical Properties of Semiconductors* (Springer-Verlag, Berlin, 1988).
23. L. W. Wang and A. Zunger, Local-density-derived semiempirical pseudopotentials, *Phys. Rev. B* **51**, 17398 (1995).
24. L. W. Wang and A. Zunger, Pseudopotential calculations of nanoscale CdSe quantum dots, *Phys. Rev. B* **53**, 9579 (1996).