Out-of-Core SVD and QR Decompositions*

Eran Rabani [†] ***and Sivan Toledo*** [‡]

1 Introduction

out-of-core singular-value-decomposition algorithm. The algorithm is designed for tall narrow matrices that are too large to fit in main memory and are stored on disks. We have implemented the algorithm and combined it with a larger eigensolver code to obtain the electronic states of a semiconductor nanocrystal. The computational-chemistry application requires finding an orthonormal basis for a subspace spanned by a small set of real vectors. The input typically consists of several hundreds to several thousands vectors whose length is between $200,000$–$2,000,000$. The orthonormal basis vectors are then used to reduce the dimensionality of an operator: given a matrix U of orthonormal basis vectors (U's columns) and a matrix H represented as a matrix-vector-multiplication routine, we compute $\hat{H} = U^T H U$. We report on performance-evaluation runs on random matrices whose size ranges from 8 to 32GB and matches the size of the computational-chemistry application.

Since neither the input matrix A (consisting of the original set of column vectors) nor the basis U fit in 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 A, $A = QR$. The matrix Q is stored on disks but R is small enough to fit in memory.
- An in-core algorithm (from LAPACK) 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 $A = U\Sigma V^*$ is the SVD of A.

*This research was supported by Israel Science Foundation founded by the Israel Academy of Sciences and Humanities (grant number 572/00, grant number 9060/99, and grant number 34/00) 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.

†School of Chemistry, Tel-Aviv University. Email: rabani@tau.ac.il.

‡School of Computer Science, Tel-Aviv University. Email: stoledo@tau.ac.il

- 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 A.
- We apply H to U' by readings blocks of columns of U', applying H to them using a matrix-vector multiplication routine that uses FFT's to quickly apply H, and write the transformed vectors back to disk.
- An out-of-core matrix multiplication computes $\hat{H} = (U')^T(HU')$ to produce the in-core product of two out-of-core matrices.

The most challenging phase in the out-of-core SVD of tall narrow matrices is the out-of-core QR decomposition. Since the input matrix is tall and narrow, 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 [1, 2] in order to achieve a high level of data reuse. The locality 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 Gbytes 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. 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 [6] and for the QR decomposition by Elmroth and Gustavson in [1, 2]. 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 A 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, [4, Chapter 18]).

We prefer to compute the SVD of A 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.

We implemented the new out-of-core QR and SVD algorithms as part of SOLAR [8], a library of out-of-core linear algebra subroutines. Before we started the current project, SOLAR already included sequential and parallel out-of-core codes for matrix multiplication, solution of triangular linear systems, Cholesky factorizations, and LU factorizations with partial pivoting. SOLAR can exploit share-memory parallel processing, distributed-memory parallel processing (or both), par-

allel input-output, and nonblocking input-output. SOLAR can process real and complex matrices, single or double precision.

The main new addition to SOLAR is an out-of-core QR factorization. The new code is optimized for tall narrow matrices. The new code uses existing subroutines extensively to multiply matrices and to solve triangular systems. 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.

We have also improved the I/O layer of SOLAR over the one described in [8]. The changes allow SOLAR to perform non-blocking 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 on large buffers without allocating large auxiliary buffers. These changes improve the performance of SOLAR's I/O, they reduce the amount of I/O, and they 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 [7], 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 14GBytes of main memory, so we simply can't solve very large systems without an out-of-core algorithm. While we 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 this 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.

The remainder of the paper is organized as follows. Section 2 describes the recursive QR and SVD algorithm that we use and their out-of-core implementation. Section 3 describes the computational-chemistry eigensolver for which we developed

the algorithm. Section 4 describes experimental results, and Section 5 describes our conclusions.

2 Out-of-Core Recursive QR and SVD

2.1 Computing R and the Compact-WY Representation of Q

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 [1, 2]. The input of the algorithm is A 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 $QA = (r, 0, \ldots, 0)^T$ (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 A into $[A_1 A_2]$, where A_1 consists of the left $n_1 = \lfloor n/2 \rfloor$ columns of A and A_2 consists of the right $n_2 \lceil n/2 \rceil$ columns.

3. Compute recursively the factorization (Y_1, R_{11}, T_{11}) of A_1.

4. Update $\tilde{A}_2 = Q_1^T A_2 = (I - Y_1 T_{11} Y_1^T) A_2$.

5. Compute recursively the factorization (Y_2, R_{22}, T_{22}) of the last $m - n_1$ rows of $\tilde{A}_2$.

6. Compute $T_{12} = -T_{11}(Y_{11}^T Y_{22}) T_{22}$.

7. Compute R_3 which consists of the first n_1 rows of $\tilde{A}$.

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

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 A 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 A 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 A 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 A fits within main memory, the code reads it from disk, computes (Y, R, T) in core, and writes Y back to disk, overwriting A. 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, because the matrices that this routine must factor are extremely thin, such as $2,000,000$-by-20, and as shown in [1, 2], 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 A to be factored does not fit within main memory, the algorithm splits it into A_1 and A_2 and factors the block recursively. Computing $\tilde{A_2} = Q_1^T A_2 = (I - Y_1 T_1 Y_1^T) A_2$ is done in three out-of-core steps, each of which involves a call to SOLAR's out-of-core matrix-multiply-add routine: $\tilde{A_2} += Y_1(T_1(Y_1^T A_2))$. The first intermediate result $Y_1^T A_2$ is stored in the Z_{12} and the second intermediate result in T_{12} (which is still empty).

Next, the code reads the first n_1 rows of $\tilde{A_2}$ into R_{12}.

The code then writes out a block of zeros into the first n_1 rows of A_2, since Y is lower trapezoidal, and recursively factors the last $m - n_1$ rows of A_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, which are both upper triangular.

2.2 Computing the Q or the SVD

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 A. 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.

3 A Computational-Chemistry Application

One of the most challenging problems of theoretical chemistry is to extend the size of systems that can be studied computationally. Semiconductor nanocrystals fall into the category of large system and their theoretical study requires the development of new computational tools. In the application reported below we have used the solver to obtain the electronic states of a semiconductor nanocrystal that contains thousands of atoms and requires more than 50GB of disk storage. The size of the chemical application is about an order of magnitude larger than the largest size reported using a conventional application of electronic structure calculations within the same physical framework.

Three steps are required to obtain the full electronic information of the nanocrystal. The first is based on a physical approximation, namely that the electronic

structure of the nanocrystal is described within the semiempirical pseudopotential method. In this approximation the electronic state of the nanocrystal can be computed from a single electron picture similar to a density functional approach. We used a screened nonlocal pseudopotential developed recently by Wang and Zunger [9] that produces local-density approximation quality wave functions with experimentally fit bulk band structure and effective masses. For simplicity the spin-orbit interactions were neglected.

We represent the electronic wave function on a three dimensional grid in real-space. In this representation both the nonlocal potential and kinetic operators can be evaluated using linear scaling methods. Since it is not feasible to diagonalize directly the one-electron Hamiltonian to obtain the desired eigenvalues and eigenstates, even for relatively small nanocrystal sizes, we use the filter-diagonalization method and combined it with the out-of-core SVD and QR decompositions. The detailed description of the filter-diagonalization method for semiconductor nanocrystals is described elsewhere [5]. Here we briefly outline the main steps of the eigensolver.

We compute the eigenvalues of a Hamiltonian H that lie in a specific energy range in three steps. First, we compute a set of vectors $A = [a_1 a_2 \cdots a_k]$, not necessarily independent, spanning the desired eigenvectors. Next we compute an orthonormal basis U for the subspace spanned by the columns of A (and hence the desired eigenvectors). We then reduce the order of the Hamiltonian by computing $\hat{H} = U^T H U$. Every eigenvalue λ of $\hat{H}$ is also an eigenvalue of H, and if $\hat{H}w = \lambda w$, then $H(Uw) = \lambda(Uw)$. The last step in our eigensolver is, therefore, the computation of the eigenvalues Λ and eigenvectors W of $\hat{H}$ and the corresponding eigenvectors UW of H.

The columns of A, the matrix whose columns span the desired eigenspace, are generated by a Krylov-like filtering processes that starts from random initial vectors. Each filtering process generates a few columns of A. Since the processes are completely independent, we can run many of them on a cluster of Linux workstations or on multiple processors of a parallel computer (a 112-processor SGI Origin 2000 in our case). Each filtering process stores the columns that it generates in a separate file. The the details of the filtering algorithm are beyond the scope of this paper and are described elsewhere [5].

Once these filtering processes terminate and their output files are ready, our out-of-core SVD code collects the columns of A from these files, in which multiple columns are stored one after the other. All the columns are collected into one SOLAR matrix file which is 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 remote file copying program.

The code now computes the SVD $U\Sigma V^T$ of A. We use the singular values to determine the numerical rank r of A. We then use the first r columns of U, corresponding to the r largest singular values, as an orthonormal basis for A, reduce the order of H, and compute the eigenvalues and eigenvectors of the reduced Hamiltonian $\hat{H}$ and then the eigenvectors of H.

We assume that several matrices of size r fit into main memory, which allows

Table 1. *The performance of our out-of-core QR factorization code including the formation of the explicit Q. 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 A, 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.*

Machine	Mem	m	n	n_0	T	T_{ic}	T_{io}	M	M_{ic}
Pentium III	1.5GB	1e6	1e3	120	39932	12280	27647	100	325
Pentium III	1.5GB	5e5	2e3	260	54339	23373	30960	147	342
Origin 2000	2GB	2e6	2e3	70	122379	69722	52630	261	459

us to compute the eigendecomposition of $\hat{H}$ in main memory (we use LAPACK's `DSYEV` routine).

4 Experimental Results

Table 1 summarizes the results of three performance-evaluation experiments. Two experiments were conducted on a 600MHz dual Pentium III machine running Linux and another on a 400MHz, 112-processor SGI Origin 2000. We used only one processor on both machines. The Linux machine did not have sufficient attached disk space, so we used 4 other similar machines 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 18GB SCSI each, and were connected to the other machine using fast Ethernet (100Mbits/s). The effective I/O rate that we measured on this setup was about 9.8MBytes/s. The Origin had an attached disk array with approximately 300GBytes.

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. We choose these matrix sizes for our performance evaluation runs since they approximate our needs in the computational chemistry application.

We can also see from the table that the code performs better on wider, shorter matrices, because it performs less I/O.

We were unable to produce large-scale results from our production code in time for the submission deadline of this paper, but we did manage to run part of the application. Our production run completed its filtering stage on a 2e6-by-3500 matrix. The filtering stage ran for 4 days on 10 of the Origin's processors. An extrapolation from our measured performance results show that computing the SVD of this matrix would also take about 4 days (100 hours) on the Origin, using

only 1 processors. We conclude that the out-of-core SVD constitutes a significant fraction of the total running time of the application, but does not dominate the running time.

5 Summary

We have presented an out-of-core SVD and QR factorization algorithm and its implementation. The code is intended to be used in a computational-chemistry eigensolver. We have demonstrated that the code is efficient and that it can solve within days problems whose size is much bigger than main memory.

We expect that the algorithm would also be useful in other algorithms that require large-scale orthogonalization.

Bibliography

[1] E. Elmroth and F. Gustavson. New serial and parallel recursive QR factorization algorithms for SMP systems. In B. Kågström et al., editors, *Applied Parallel Computing: Large Scale Scientific and Industrial Problems*, volume 1541 of *Lecture Notes in Computer Science*, pages 120–128. Springer-Verlag, 1998.

[2] E. Elmroth and F. Gustavson. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM Journal of Research and Development*, 44(4):605–624, 2000.

[3] E. Elmroth and F. Gustavson. High-performance library software for QR factorization. In P. Björstad et al., editors, *Applied Parallel Computing: New Paradigms for HPC in Industry and Academia*, Lecture Notes in Computer Science. Springer-Verlag, To appear.

[4] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 1996.

[5] Eran Rabani, Balazs Hetenyi, Bruce J. Berne, and Louis E. Brus. Electronic properties of cdse nanocrystals in the absence and presence of a dielectric medium. *Journal of Chemical Physics*, 110(11):5355–5369, 1999.

[6] Sivan Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[7] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. In James M. Abello and Jeffrey Scott Vitter, editors, *External Memory Algorithms*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 161–179. American Mathematical Society, 1999.

[8] Sivan Toledo and Fred 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*, pages 28–40, Philadelphia, May 1996.

[9] Lin-Wang Wang and Alex Zunger. Local density derived semiempirical pseudopotentials. *Physical Reviews B*, 51(24):17398–17416, 1995.