Fast Diagonalzation of Hermitian Matrices for Quantum Mechanical Simulations

Joseph Gonzalez

CS267: Final Project April 29, 2015

Abstract

An efficient and scalable algorithm for obtaining the full eigenvalue spectrum of large Hermitian matrices is presented. This computation is the basis for solving problems in density functional theory simulations (DFT) and is traditionally the bottle neck both in time and space consumption for the convergence to the ground state, with typical matrix dimensions of 10x10 to 1000 x 1000. Typically, an iterative algorithm is implemented, however this provides only the lowest few eigenvalues, while the routine presented here provides the full spectrum. The solution presented is based on reducing the input matrix to tri-diagonal form by the method of successive Householder transformations. This tri-diagonal matrix is then diagonalized to obtain all of the eigenvalues using a parallel QR factorization scheme. Some scaling tests are also presented and analyzed with respect to the performance of the code.

1 Introduction

The problem of accurately predicting the ground state configuration of an atomic system has been an interesting problem for theoretical physicists for roughly the past century, with more and more attention in recent years due to faster computer hardware. In order to determine this low energy configuration, theoretical physicists use a framework known as Density Functional theory (DFT), which defines the energy of a system as a functional of the the electron density, $\rho(\boldsymbol{R})$. To obtain this energy, the central problem is to solve the eigenvalue problem known as the Kohn-Sham [1;2],

$$\mathcal{H}\Psi_i(\boldsymbol{r}_i) = \left[-\nabla^2/2 + v_{eff}(\boldsymbol{R})\right]\Psi_i(\boldsymbol{r}_i) = E_i\Psi_i(\boldsymbol{r}_i) \tag{1}$$

Here, $\mathcal{H}$, is the Hamiltonian operator which contains the kinetic energy of the system and the effective potential felt by the i-th electron. The Hamiltonian has at least $4N^2{}_{electrons}$ entries. The effective potential is determined by integrating $\rho(\boldsymbol{r})$, defined as,

$$\rho(\boldsymbol{R}) = N\int d^3r_2\cdots d^3r_N|\Psi(\boldsymbol{r}_1,\boldsymbol{r}_2,\cdots,\boldsymbol{r}_N)|^2 \tag{2}$$

Clearly, this problem can only be solved iteratively, since the effective potential defines the wave functions, Ψ, which in turn defines the charge density, $\rho(\boldsymbol{R})$. To achieve this, an initial guess of the wave functions is produced, iterating for a number of electronic steps until the current and previous charge densities converged to some tolerance. In order to determine the wave functions for each successive step, the energy of the system is found by diagonalizing the Hamiltonian of the system at each electronic step. For a typical calculation, the Hamiltonian is on the order of 500x500, and depending on the convergence criteria, 15 electronic steps are needed to calculate the energy of a given configuration. To find the minimal energy configuration, a typical calculation will require approximately 30 ionic steps, each of which containing 15 electronic steps.

Depending on the type of system being studied, one may only require the first k eigenvalues, in which case the standard method used in popular DFT codes is the iterative blocked Davidson algorithm [3;4]. However, if one requires a more precise and fine grained calculation, the full eigenspectrum is required and thus this iterative scheme is not applicable. In this work, we present an algorithm for obtaining the full eigenspectrum of a large Hermitian matrix.

2 Theory and Algorithm

In this section, the mathematical framework as well as the algorithmic details of the scheme to obtain the eiegnevalues of a symmetric matrix are presented. The basis for this method begins with a reduction of the input matrix, e.g. the Hamiltonian of a system of particles, to upper triangular form, i.e. Upper Hessenberg, by successive Householder [5] transforms. In the case of symmetric Hermitian matrix, the resulting Hessenberg is symmetric and therefore it is tri-diagonal. The Householder transform is unitary with the following property, $\boldsymbol{\Omega}\ \boldsymbol{\Omega^T} = \boldsymbol{I}$, and is defined as,

$$\boldsymbol{\Omega} = \boldsymbol{I} - 2\boldsymbol{v}\otimes\boldsymbol{v}^T. \tag{3}$$

Given a matrix, $\boldsymbol{\mathcal{H}} \in \mathbb{R}^{nxn}$ such that $\boldsymbol{\mathcal{H}} = \boldsymbol{\mathcal{H}}^\dagger$, the method of Householder deflation is then given by the following,

$$\boldsymbol{\mathcal{H}} \to \boldsymbol{T} = \prod_{k=1}^{n-2}\boldsymbol{\Omega}_k\boldsymbol{\mathcal{H}}_{k-1}\boldsymbol{\Omega}_k \tag{4}$$

$$\boldsymbol{T} = \begin{pmatrix} a_1 & b_1 & 0 & \cdots & 0 \\ b_1 & a_2 & b_2 & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & b_{n-2} & a_{n-1} & b_{n-1} \\ 0 & \cdots & 0 & b_{n-1} & a_n \end{pmatrix}.$$

The next portion of the algorithm proceeds in a divide and conquer fashion, by splitting $\boldsymbol{T}$ into two submatrices, each of which is split again, iteratively until the entire matrix has been divided evenly amongst the requested processors. For example on two processes,

$$\boldsymbol{T} = \begin{pmatrix} \boldsymbol{T_1} & 0 \\ 0 & \boldsymbol{T_2} \end{pmatrix}.$$

Now, $\boldsymbol{T}$ can be diagonalized using an appropriate orthogonal matrix $\boldsymbol{Q}$,

$$\boldsymbol{T_1} = \boldsymbol{Q_1 D_1 Q_1^T} \text{ and } \boldsymbol{T_2} = \boldsymbol{Q_2 D_2 Q_2^T}. \tag{5}$$

We then proceed to obtain the eigenvectors of $\boldsymbol{T}$ in the usual manner by solving the roots of the secular equation,

$$\boldsymbol{y}_i = (\boldsymbol{D} - \lambda_i \boldsymbol{I})^{-1}\boldsymbol{\zeta}. \tag{6}$$

In the above $\boldsymbol{\zeta}$ is defined as follows,

$$\boldsymbol{\zeta} = \begin{pmatrix} \boldsymbol{Q}_1 & 0 \\ 0 & \boldsymbol{Q}_2 \end{pmatrix}^T \mathbf{z},$$

$$\mathbf{z} = \begin{pmatrix} 0 & \cdots & 0 & 1 & 1 & 0 & \cdots & 0 \end{pmatrix}^T$$

Finally, the eigenvectors of the original matrix $\boldsymbol{\mathcal{H}}$ can be obtained by,

$$\mathbf{v}_i = \begin{pmatrix} \boldsymbol{Q}_1 & 0 \\ 0 & \boldsymbol{Q}_2 \end{pmatrix} \boldsymbol{y}_i. \tag{7}$$

This is the case for when we have only 2 processes, however when more processes are requested, the matrices $\mathbf{T}_1$ and $\mathbf{T}_2$ will be divided into halves further, see Fig. (**??**).

Given this set of procedures, we can estimate the computational complexity to be $\mathcal{O}(Cn^3)$, with the majority of the time being spent during the reduction to tri-diagonal form. The complexity is broken down as follows; the construction of $\boldsymbol{\Omega}$, is $\mathcal{O}(n)$, the Householder reduction to tridiagonal form is the most expensive[6], with $\mathcal{O}(\frac{4}{3}n^3 + n^2)$ operations, and finally, the QR factorization of a tridiagonal matrix[7] incurs only $\mathcal{O}(n)$ operations.

As can be seen from Eq. (4), the tri-diagonal reduction step requires $2(n\text{-}2)$ matrix-multiplication operations, which could be quite prohibitive for matrices larger than 100 x 100. In an earlier version of the code, we implemented calls to the subroutine *pdgemm_()*, part of the scaLAPACK library[8]. However this led to a scaling problem with matrices larger than 1000 x 1000, as well as poor load balancing and disproportionate calls to MPI::COMM.Recv versus MPI::COMM.Send(). Therefore, we adopted a new scheme which includes only calls to the Intel MKL BLAS[9] *dgemm_()*. It should be noted that although this implementation is slower than the results when using the scaLAPACK library, it is more robust with no limit on the size of input matrices as well as a more even load balance, see Fig. (4). Also included in the distribution is a vectorized and blocked matrix-multiplication routine, for the case when the user does not have access to the BLAS library.

For the first multiplication step in Eq. (4), we let the root process send the rows of the Householder matrix, $\boldsymbol{\Omega}$, and broadcast the input matrix, $\boldsymbol{\mathcal{H}}$, to all processes in the world, after which each process does makes their own call to *dgemm_()*. For the second multiplication step, the root process behaves in the same way, except now, we overwrite the data in the input matrix to save space.

The next step in the routine is to store and distribute the newly formed tridiagonal matrix, $\boldsymbol{T}$, of Eq. (4) to all processes in the world. Given the sparse nature of this matrix, we store the main diagonal and off diagonal components in 1D vectors, allowing space conservation as well as easier indexing for building the $\boldsymbol{Q}$ matrices.

The details of the algorithm described above are summarized shown below,

Algorithm 1 *TRQR* pseudocode to obtain the eigenvalues of a symmetric matrix.

```
1:  for $i = 0$; $i < N - 2$; do
2:      Construct $\boldsymbol{\Omega}$ on all procs
3:      $\boldsymbol{T} = \boldsymbol{\Omega} * \hat{\boldsymbol{H}}$
4:      $\boldsymbol{T} = \boldsymbol{T} * \boldsymbol{\Omega}$
5:  end for
6:  Broadcast $\boldsymbol{T}$ onto j procs
7:  for $j = 0$; $j < nprocs$; do
8:      Diagonalize $\boldsymbol{T}_j$
9:      Send $\boldsymbol{D}_j$ to root
10: end for
11: Solve secular equation, bi-section method
```

3 Results

In this section, we present the results of the *TRQR* implementation described above. Shown in Fig. (1), are the weak scaling tests on the presented algorithm. For the weak scaling test, the system size is doubled, while simultaneous doubling the number of resources. For this test specifically, we varied the matrix dimensions between $n = 75$, to $n = 2400$. From this plot, we can see that we achieve almost n^2 performance, which is far from the ideal case of a constant time to solution.

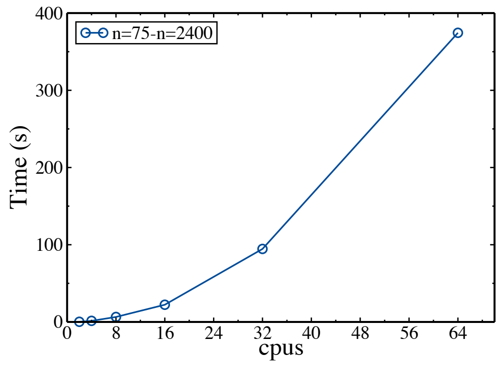

Figure 1: Results of weak scaling test on *trdqr*. Matrices range from $n = 75$, to $n = 2400$.

Another set of tests conducted were strong scaling. In a strong scaling test, the system size is fixed, while the number of resources is increased. From Fig. (2), we can see quite good scaling results when $n = 500$ up to 16 processes. However after 16 processes, the overhead to initiate the MPI tasks becomes less negligible. Also when requesting this many of processes, each process only owns a matrix of approximately 20 which is too small to notice the improvements of the BLAS level 3 subroutine calls.

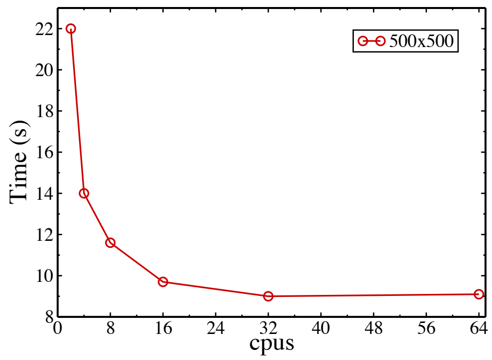

Figure 2: Results of strong scaling test on *trdqr*. Dimension of the test matrix is 500.

Another interesting metric which can be obtained from the strong scaling results is the speed up provided by adding more resources, shown in Fig. (3). Ideally, this data should be linear, however it can clearly be seen that the data showcases an approximately $\log(n)$ trend, providing very little speed up after 16 processes. Worse yet, we can see that at 64 MPI tasks, the speed up decreases, again this is due to the overhead of the MPI paradigm and the inefficient use of BLAS level 3 subroutine calls.

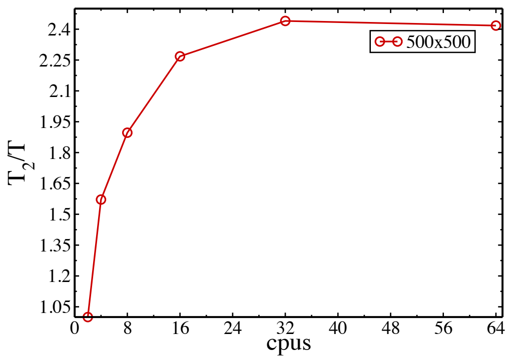

Figure 3: Results of the speed up provided by increasing resources. Dimension of the test matrix is 500.

Aside from time to solution tests, we can further analyze the efficiency of this algorithm by using performance monitors, specifically the Integrated Performance Monitor (IPM)[10], was used. IPM is a very powerful tool which allows very detailed information on the parallel efficiency of a code employing the MPI library. The IPM tool is available on Stampede as a module, and does not require any special build procedures other than including the debug option during compilation. Shown in Fig. (4) is a pie chart detailing the percent usage of the most prominent MPI collectives used in the program. As mentioned in the § 2, in the first implementation using the scaLAPACK library, there was a disproportionate amount of MPI::COMM.Recv calls as compared to Send calls, approximately 70% and 16%, respectively. Now, using the new implementation, the program achieves much better load balancing and an approximately equal proportion of Send/Recv calls, providing better efficiency even though the time to solution is slower.

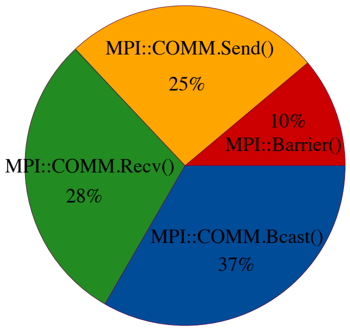

Figure 4: Pie chart representing the most frequently called MPI commands during runtime, as measured by IPM.

Another nice feature provided by the IPM tool, is an analysis of the MPI topography by time, received data size, and sent data size. Shown in Fig. (5), is one such plot showing the communication pattern and timing for each process, darker colors represent more time being spent. From this plot, we can see that the communication pattern itself as we would hope, however, the load balance of computation time is less than ideal.

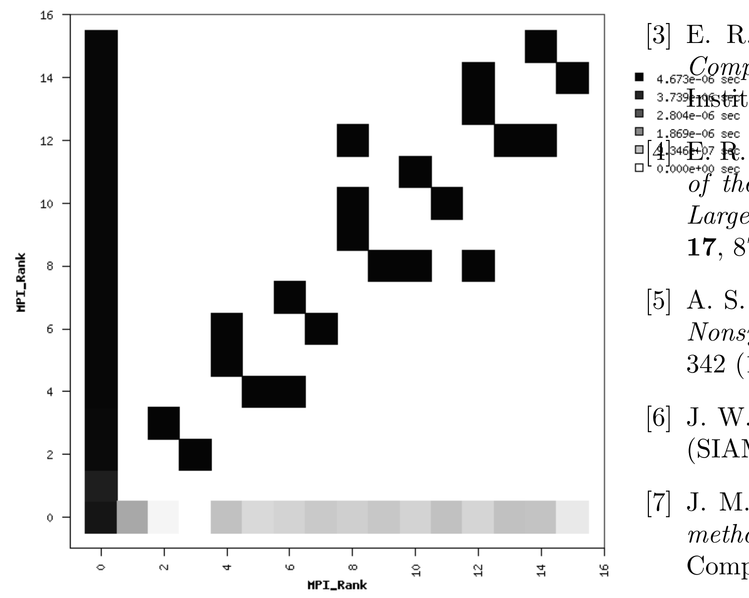

Figure 5: Communication pattern amongst processes and the amount of time for each cpu.

4 Conclusions

We have presented an algorithm for obtaining the full eigenspectrum of a large Hermitian matrix with applications to quantum mechanical optimization problems. The algorithm is based on the fact that the diagonalization of a trid-diagonal matrix is $\mathcal{O}(n)$, and therefore the input matrix is reduced to this form by successive Householder transformations. The time to solution results and scaling are acceptable, however since the number of times the eigenvalues need to be obtained is on the order of 500 for a typical simulation, this method is still prohibitive. In fact, most systems can be simulated with high accuracy and close agreement to experiment by using the iterative method and obtaining only a few eigenvalues at each step. Never the less, this algorithm does showcase a robust method for obtaining the full eigenspectrum, which is advantageous for a certain subset of problems encountered in quantum mechanics.

References

[1] P. Hohenber and W. Kohn, *Inhomogeneous Electron Gas*, Phys. Rev. **136**, B864, (1964).

[2] W. Kohn and L. J. Sham, *Self-Consisten Equations Including Exchange and Correlation Effects*, Phys. Rev. **140**, A1133, (1965).

[3] E. R. Davidson, *Matrix Eigenvector Methods in Computational Molecular Physics*, Advanced Study Institute, Series C. Vol. **113**, (1983).

[4] E. R. Davidson, *The Iterative Calculation of a Few of the Lowest and Corresponding Eigenvectors of Large Real-Symmetric Matrices*, J. Comput. Phys. **17**, 87-94, (1975).

[5] A. S. Householder. *Unitary Triangularization of a Nonsymmetric Matrix*, Journal of the ACM **5**, 339-342 (1958).

[6] J. W. Demmel, *Applied Numerical Linear Algebra* (SIAM,1997).

[7] J. M. Ortega and H. F. Kaiser, *The LL and QR methods for symmetric tridiagonal matrices*, The Computer Journal **6**, 99-101 (1963).

[8] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker and R. C. Whaley. (1997) `http://www.netlib.org/scalapack/`.

[9] `http://www.netlib.org/blas/`.

[10] `http://ipm-hpc.org`