CS240a: Final Project Report
A Parallel Reusable Implementation
of an Eigensolver for Large Eigenproblems

Victor Amelkin
Department of Computer Science
University of California, Santa Barbara

June 12, 2012
1 Introduction

The problem of finding eigenvalues and eigenvectors of a matrix is probably the most notorious problem of linear algebra. Its essence is, given a square matrix $A$, to find such vectors $x_i$ that do not change their direction when multiplied by $A$ but only scale with the scaling factors of $\lambda_i$:

$$Ax_i = \lambda_i x_i$$

Vectors $x_i$ meeting this requirement are called eigenvectors of the matrix $A$ and $\lambda_i$ are its eigenvalues.

Even in a case of a relatively small matrix size, the analytical solution for the eigenproblem is unfeasible. This creates a necessity to solve the problem approximately using numerical methods, which are in abundance. In this work, we focus on one of these numerical methods – Lanczos algorithm or Lanczos iteration [5]. This method allows to find a few specific eigenpairs of a square symmetric\(^1\) matrix, which makes this method convenient for dealing with large matrices, for which it is problematic to solve the full eigenproblem. Additionally, one of the most computationally intensive operations used by Lanczos algorithm is matrix-vector multiplication, which can be performed efficiently for sparse matrices. Thus, Lanczos iteration is an especially promising method for dealing with large sparse symmetric eigenproblems. In this project, we will restrict ourselves to the case of real-valued symmetric matrices.

At the moment of writing of this report, the most well-known implementation of Lanczos iteration along with Arnoldi iteration – a similar method for handling non-symmetric matrices – is in ARPACK package [6]. ARPACK is a library written in Fortran77 that provides serial algorithms for Lanczos iteration. These algorithms are written in terms of basic linear algebra operations and primitives from BLAS and LAPACK. ARPACK’s parallel version, P_ARPACK [7] features a parallel implementation of Lanczos iteration for distributed-memory architectures (the supported message passing layers are BLACS and MPI).

In this project, we propose a parallel implementation of Lanczos iteration with selective orthogonalization for shared-memory architectures. The project’s main goal is to implement Lanczos iteration as fast as possible and analyze how much the performance of Lanczos algorithm depends on the performance of the underlying parallelizable linear algebra operations.

The rest of this report is organized as follows. Section 2 describes the theory underlying

---

\(^1\) According to Boley and Golub [1], in his original paper [5], Lanczos described how to deal with both symmetric and non-symmetric matrices. However, non-symmetric block Lanczos iteration has problems with stability, necessity to restart, and, overall, is much less studied. Thus, we are referring to Lanczos iteration as applicable to symmetric eigenproblems.
Lanczos method, section 3 provides details of serial and parallel implementations, section 4 is dedicated to the evaluation of correctness and performance of the implemented algorithm and comparison of its performance with existing implementations, and the last Section 5 summarizes the results of this work and provides a few pointers for future investigation.

2 Lanczos Method

Lanczos method includes three major algorithms. The first one is Lanczos iteration itself, the second one is the QR algorithm for the solution of a full tridiagonal eigenproblem, and the last is QR decomposition based on Givens rotations. The following three subsections describe each of these algorithms.

2.1 Lanczos iteration

The general idea for Lanczos iteration is to tridiagonalize a symmetric square matrix and then compute the eigenpairs of the original matrix based on the eigenpairs of the tridiagonal matrix.

The algorithm starts with choosing a random vector \( q_1 \in \mathbb{R}^n \), where \( n \) is the size of \( A \). Then, it constructs an orthogonal basis of the Krylov subspace \( \text{span}\{Q_i\} = K(A, q_1, k), \ i = 1 \ldots k \), and computes components of the tridiagonal matrix \( T_i \) related to \( A \) as \( A = Q_i^T A Q_i \). The columns of \( Q_i \) are called Lanczos vectors. The following pseudocode is a modified version of Lanczos algorithm provided in [2].

\[
q_1 = q_1 / \|q_1\|_2, \ \beta_0 = 0, \ q_0 = 0
\]

for \( i = 1 \) to \( k \)

\[
z = Aq_i \\
\alpha_i = q_i z \\
z = z - \alpha_i q_i - \beta_{i-1} q_{i-1} \quad \text{(Step Z)}
\]

Orthogonalization of \( z \) (discussed later) \quad \text{(Step O)}

\[
\beta_i = \|z\|_2 \\
\text{if } \beta_i = 0, \text{ terminate}
\]

\[
q_{i+1} = z / \beta_i \\
\text{Compute eigenpairs and error bounds for } T_i
\]

end for
Some identifiable eigenvalues of $T_i$ are equal to some\(^2\) eigenvalues of the original matrix $A$, while the $j'$th eigenvector $V_j^A$ of $A$ can be obtained from the corresponding eigenvector $V_j^{(i)}$ of $T_i$ as $V_j^A = Q_i^T V_j^{(i)}$.

The original Lanczos algorithm without orthogonalization (Step O in the algorithm above) suffers from the Lanczos vectors' loss of orthogonality, which considerably affects the quality of the obtained eigenpairs of $T_i$. Accumulated round-off errors seem a plausible cause for this problem. However, Golub and van Loan [3, pp.481-482], using the results of Paige’s analysis, describe that the primary cause for the loss of orthogonality in $q_j$ is a numerical cancellation that appears at Step Z in the algorithm when the previously computed Lanczos vector $q_{i-1}$ has a significant (and unwanted) component in the direction of every already converged Ritz vector $y_{j,i} = Q_i V_j$.

The problem of the loss of orthogonality can be alleviated by periodically orthogonalizing the already computed columns of $Q_k$. Orthogonalization of every newly computed vector $q$ against all the previously computed $q_j$ is very costly and, according to the mentioned results of Paige’s analysis, not necessary. It should suffice to orthogonalize only against those $q_j$ that correspond to the already converged Ritz vectors. This version of Lanczos iteration is known as Lanczos algorithm with selective orthogonalization. In this version, Step O of Lanczos algorithm will be expanded into the following.

\[
\text{for all } j < i \text{ such that } \beta_i |V_j^{(i)}| \leq \sqrt{\epsilon} \|T_i\| \\
z = z - (y_j^T z) y_{j,i} \\
\text{end for}
\]

At this point, the only unexplained step of Lanczos algorithm is how to compute eigenpairs of a tridiagonal $T_i$ on each iteration. This will be performed using QR algorithm, described below.

\(^2\)In the original Lanczos iteration without shifting and inverting, the first eigenvalues to converge are those on the opposite ends of the spectrum.
2.2 QR Algorithm

QR algorithm (or QR iteration) solves the full eigenproblem for a tridiagonal matrix. The original QR algorithm by Francis and KUBLANOVSKAYA works as follows.

\[ A_0 \leftarrow \text{tridiagonal input matrix} \]

\[
\begin{align*}
\text{do until convergence} & \\
A_j &= Q_j R_j \quad // \text{QR decomposition} \\
A_{j+1} &= R_j Q_j (= Q_j^{-1} A_j Q_j)
\end{align*}
\]

The process converges to a upper-triangular matrix

\[
A_m = U^{-1} Q_m^{-1} Q_{m-1}^{-1} \cdots Q_0^{-1} A_0 Q_0 \cdots Q_{m-1} Q_m,
\]

where \(U\) is unitary (or, equivalently, orthogonal in our case), thereby, delivering a Schur decomposition or the original matrix \(A_0\). Thus, eigenvalues of \(A_0\) and \(A_m\), which are equal to the entries on the main diagonal of \(A_m\), are the same, and eigenvectors of \(A_0\) are the columns of \(U\).

QR algorithm applied to a tridiagonal \(T_i\) may converge slowly if eigenvalues of \(T_i\) are not well-separated. To fight this slow convergence, instead of decomposing \(T_i\) (or the result of its decomposition on the previous steps of QR algorithm), a shifted matrix \((T_i - \mu I)\) can be decomposed, where \(\mu\) is a shift. A properly chosen shift \(\mu\) increases the separation of eigenvalues and increases the convergence. For example, a Wilkinson’s shift delivers a cubic rate of convergence. However, implementing the shifting strategy for QR algorithm will not be pursued in this project.

2.3 QR Decomposition

On each iteration, QR algorithm performs a QR decomposition of matrix \(A_j\) – it decomposes \(A_j\) into a product of an orthogonal matrix \(Q\) and an upper-triangular matrix \(R\). Such factoring can be performed by finding a series of orthogonal transformations of the original matrix \(A_j\) which would nullify the elements under its main diagonal, thereby, reducing \(A_j\) to an upper triangular form. The resulting transformed matrix will constitute \(R\), while the product of all the applied nullifying orthogonal transformations will constitute \(Q\).

If we deal with an arbitrary matrix, the first step is usually to use Householder reflections capable of nullifying several matrix elements per one reflection. The result of application of multiple Householder reflections is a tridiagonalized matrix. However, we are starting with an already tridiagonal matrix \(T_i\) and only need to nullify its subdiagonal elements.
The common way to nullify a few elements in a matrix is to use Givens rotations $G(i, j, \theta)$ – a rotation by angle $\theta$ in the plane containing axes $(i, j)$.

$$G(i, j, \theta) = \begin{pmatrix}
1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\
\vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
0 & \cdots & \cos \theta & \cdots & -\sin \theta & \cdots & 0 \\
\vdots & \vdots & \ddots & \ddots & \vdots & \ddots & \vdots \\
0 & \cdots & \sin \theta & \cdots & \cos \theta & \cdots & 0 \\
\vdots & \vdots & \ddots & \ddots & \vdots & \ddots & \vdots \\
0 & \cdots & 0 & \cdots & 0 & \cdots & 1
\end{pmatrix}$$

The angle $\theta$ can be chosen so that the rotation applied to a matrix nullifies its particular element. Thus, having used a Givens rotation to nullify every subdiagonal element of $T_i$, we are getting an upper triangular matrix, $R$. The product of used Givens rotations becomes $Q$.

3 Implementation

3.1 Serial Implementation

The serial code for Lanczos iteration, QR algorithm, and QR decomposition is implemented in C++. The language was chosen due to its suitability for writing high-performance code and the ability for the code to be subsequently easily ported to Intel’s Cilk. All the algorithms are implemented from scratch without using any libraries, so that it is easy to modify any part of the whole system for the purposes of parallelization or performance tuning.

To support the main computational algorithms, we wrote a compact C++ library that provides fundamental linear algebra data structures and subroutines used by Lanczos iteration and QR algorithm. The two main entities in this library are a vector and a matrix classes. The implementations of both use dense representation of elements, which was chosen over sparse representation since the used QR algorithm heavily employs dense matrices. However, this choice in no way affects our performance analysis\(^3\), and all the computational algorithms are written independently of the inner representation of used vectors and matrices.

\(^3\)The running times for dense and sparse data will be different, but the scaling properties should be similar.
3.2 Parallelization

To parallelize Lanczos iteration, we decided to use Cilk, since using it is much easier and less error prone than dealing with threads directly. Intel’s Cilk++ was the first choice, but it failed to compile C++ templated classes due to the peculiarities of linkage between C++ and Cilk++ functions. Our final choice is an open-source variation of Cilk, the “CilkPlus” branch of GCC4.8 maintained by Intel. This compiler requires almost no modifications to the original C++ code in order for it to be ported to CilkPlus. The only problem with the recent version of CilkPlus is that it cannot parse some complex templated code due to a bug. However, this problem was solved by extracting computationally intensive parallelizable regions of our code from class methods to global functions, which do not confuse the compiler.

Approaching the question of parallelization, we have not found many good targets for parallelization close to the top level of the main algorithms. For example, the main iteration in Lanczos process cannot be parallelized, since every such iteration depends on the results of all the previous iterations, i.e. this code is inherently sequential. Since parallelizing too small functions is not efficient since it creates a considerable overhead, the only two good targets for parallelization left are matrix-vector multiplication, which is the core of Lanczos iteration, and matrix-matrix multiplication, used by our implementation of QR algorithm.

Since in our experiments, dense matrices are in use, it is easy to parallelize matrix-vector and matrix-matrix multiplication by dividing matrices across Cilk workers by rows. A good property of such parallelization is that every thread exclusively writes to the memory region assigned to it, and the regions corresponding to different workers do not overlap. In other words, we cannot get race conditions.

3.3 Performance Optimization

The initial experiments with our code showed poor performance primarily due to an inefficient implementation of QR decomposition. The maximum problem size that our algorithm could solve in a reasonable amount of time was less than 1000. In this first implementation, the Givens rotations used for nullifying subdiagonal elements of a tridiagonal $T_i$ were factored by using the general matrix-matrix multiplication in order to form the unitary matrix $U$ in the Schur decomposition of $T_i$. This approach takes cubic time for every multiplication, which resulted in the factoring of Givens rotations’ comprising about 95% of the whole execution time of the Lanczos algorithm. However, Givens rotations have a very specific form, and multiplying a matrix by $G(i, j, \theta)$ essentially affects only two rows (or columns, depending on which side $G$ is applied from) of the matrix. Thus, the first performance optimization was
to write an efficient algorithm that would multiply a matrix by a Givens rotation in a linear
time. Having completed this optimization, the time spent on multiplying Givens rotations
now accounts for only fractions of a percent of the whole execution time.

The second optimization was to remove all the redundant computations from the code regions
being parallelized. This includes optimizing those functions who create copies of large objects
when operations can be performed inplace.

4 Evaluation

4.1 Correctness Evaluation

The correctness of serial code has been verified by a number of tests. The first tests checks
normality of the resulting eigenvectors.

∀xi ∈ V- eigenvector : ∥xi∥ = 1.0 ± ϵ = 10^{−6}.

The second test checks orthogonality of the resulting eigenvectors.

∥VT V − I∥ = 0 ± ϵ = 10^{−6}.

Finally, we check how close are Rayleigh quotients\(^4\) to the corresponding eigenvalues.

∀xi ∈ V- eigenvector : R(A, xi) = \frac{x_i^T A x_i}{x_i^T x_i} = \lambda_i ± ϵ = 10^{−6}\(^4\)

and

∥VT AV − Λ∥ = 0 ± ϵ = 10^{−6}.

In our experience, the first normality test has never failed, which agrees with the statement by
Golub and van Loan in [3] that the normality of eigenvectors should not be lost. The second
and the third tests usually successfully pass if QR algorithm completes enough iterations.
However, for the current classic implementation of QR algorithm, the number of iterations
to converge may be very large. For example, for a test matrix of size 100 × 100, 100 QR
iterations may be already not enough. Thus, doing parallel performance analysis, we sacrifice

\(^4\)In the formula for Rayleigh quotient, the denominator may be omitted in case of normalized eigenvectors,
whose scalar product against themselves is 1.0. However, even though our normality tests always pass, we do
not get ideally normal eigenvectors and, thus, leave the denominator to preserve precision.
precision and fix the number of iterations in QR algorithm to 100. In such a case, on a test matrix of size $10000 \times 10000$, the Rayleigh quotient test produces the following error

$$\|V^T AV - \Lambda\| = 1.910236 > \epsilon = 10^{-6},$$

which is large enough for the algorithm to finish in an acceptable amount of time and small enough for the resulting eigenvectors to be close to the exact ones.

Trying to evaluate parallel correctness of our code, we have not been able to do it automatically using Intel’s cilkscreen, since the latter simply does not recognize the parallel code produced by the CilkPlus compiler. However, as it has been already said, parallel correctness of our code is of no concern due to the fact that all the Cilk threads work on different non-overlapping regions of memory (if these regions erroneously overlapped, the serial correctness tests would necessarily fail).

### 4.2 Performance Evaluation

We have run our performance tests on Mirasol - a shared-memory machine with 40 physical Intel Xeon E7-8870 @ 2.40GHz clock / 30,720Kb cache cores, 80 logical (SMT) cores, and 256Gb of memory.

The tests are performed with 1, 4, 8, 16, 32, and 64 Cilk workers on test matrices of sizes\(^5\) 10, 100, 1000, 5000, 10000, 16000, and 32000. The test matrices are square symmetric 5-diagonal. Each test computes $k = 5$ eigenpairs of a matrix having set the number of Lanczos vectors (the number of iterations in the Lanczos process) to $m = 20k = 100$, which is usually more than enough (in MATLAB, $m = 2k$ by default).

The data collected from our performance experiments are presented on the following four figures. They display dependency of the total running time on the size of the problem and the number of employed Cilk workers as well as the dependency of parallel efficiency on the same parameters.

---

\(^5\)By the size of a square matrix we mean one of its dimensions. Thus, a square matrix of size 32000 is a $32000 \times 32000$ matrix.
Figure 1: Dependency of the total running time of the algorithm on the matrix size. Each plot corresponds to a run on a different number of Cilk workers.

Figure 2: Dependency of the total running time of the algorithm on the number of Cilk workers. Each plot corresponds to a run on a matrix of a different size.
Figure 3: Dependency of parallel efficiency of the algorithm on the matrix size. Each plot corresponds to a run on a different number of Cilk workers.

Figure 4: Dependency of parallel efficiency of the algorithm on the number of Cilk workers. Each plot corresponds to a run on a matrix of a different size.
Having the plots of experimental data, several conclusions can be made:

- It makes no sense to use more than 16 Cilk workers with our implementation of Lanczos algorithm. Using 8 workers may be justified for the problems of size larger than $32000 \times 32000$. On smaller problems, 4 workers are usually enough, and 8 workers are acceptable. Overall, using 4 workers improves the performance for every problem in test and is always recommended.

- Larger problems expectedly scale better than the smaller ones. This can be clearly seen on the Figure 4 – parallel efficiency noticeably increases when the problem size surges from 16000 to 32000.

- Performance on 4 and 8 workers is not very much different, while moving to 16 workers, performance improves for the large problems. This is unlikely to be an accident, since similar results were obtained on the same machine using differently structured matrices of the same size. However, it is not clear what causes this anomaly, which is most likely related to the architectural idiosyncrasies of Mirasol.

Our scaling results are similar to the results obtained by some other researchers, while somewhat different from the others. The scalability analysis of packages PLANSO and P_ARPACK solvers, both MPI-based, by Wu and Simon in [8] shows a better parallel efficiency when looking for 5 eigenpairs of a matrix of size $\approx 56000 \times 56000$ – their code for both PLANSO and P_ARPACK scales well until the number of processing units reaches 32 (and, possibly, even further). On the other hand, scalability analysis of the package HPEC tested on the data similar to ours in a more recent paper on parallelization of Lanczos method [4] shows scaling similar to ours – their algorithm scales acceptably until the number of processing units overgrows 8.

## 5 Summary

To summarize, we have outlined theory underlying Lanczos iteration along with QR algorithm and QR decomposition, implemented the algorithms serially in C++ and parallelized them via CilkPlus, optimized the code, and evaluated both correctness and performance on data of different sizes on different numbers of Cilk workers. Overall, the project may be considered successful. A practical parallel implementation of Lanczos iteration has been obtained that scales until the number of Cilk workers overgrows 8.

There are a few interesting directions for future investigation. One is to investigate how fast can we run if the fastest known QR algorithm in used on every Lanczos iteration.
References


