Effects of Block Size on the Block Lanczos Algorithm

by

Christopher Hsu

2003

Advisor
Professor James Demmel

Contents

Acknowledgements

First, I would like to thank Professor Demmel and Professor Yelick for supporting me as a new member in a research group among members already familiar with the material I had to spend weeks to learn. Their patience has allowed me to catch up and complete my research for the semester. I would like to thank Professor Demmel especially for his advice specific to my project, and for editing this report. I would also like to thank Rich Vuduc for all his help in starting everything and with programming issues. I owe much of my results to Benjamin Lee, who provided me with the information I needed to run all my tests. I am grateful for the correspondence of Osni Marques, who provided suggestions to point me in the right direction of obtaining the desired results. Finally, I thank the BEBOP group members for all their support and insight.

1. Introduction

The problem of computing eigenvalues occurs in countless physical applications. For a matrix A, an eigenvalue is a scalar λ such that for some nonzero vector $\mathbf{x}$, $A\mathbf{x} = \lambda\mathbf{x}$. The vector $\mathbf{x}$ is called the corresponding eigenvector. We will refer to the eigenvalue and eigenvector collectively as an eigenpair. The set of all eigenvalues is called the spectrum. Such a problem is an example of a standard eigenvalue problem (SEP). In different situations such as structural problems, we are given two matrices, a mass matrix M and a stiffness matrix K. Then an eigenvalue is a scalar λ such that $K\mathbf{x} = \lambda M\mathbf{x}$. This type of problem is known as a generalized eigenvalue problem (GEP). In my experiments I deal with large, sparse real symmetric matrices. Real symmetric matrices have applications in clustering analysis (power systems), physics (arrangements of atoms in a disordered material), chemistry (calculation of bond energies in molecules), and much more. The matrices I work with are of dimension ranging from 8070 to 54870.

For different eigenproblems, there are many possible algorithms to choose from when deciding how to approach solving for eigenvalues. *Templates for the Solution of Algebraic Eigenvalue Problems* [3] classifies eigenproblems by three characteristics. The first is the mathematical properties of the matrix being examined. This includes whether the matrix is Hermitian (or real symmetric), and whether it is a standard or generalized problem. The second characteristic to consider is the desired spectral properties of the problem. This includes the required accuracy of the eigenvalues, whether we want the associated eigenvectors, and which and how many of the eigenvalues from the spectrum of the matrix we desire. The third characteristic is the available operations and their costs. Implementation of a specific algorithm may depend on a particular data structure and the cost of operations on such a structure; restrictions on how data is represented may restrict the options of available algorithms.

Depending on the properties of the problem we are solving, *Templates* [3] provides a recommended algorithm to find eigenvalues. Classes of algorithms include direct methods, Jacobi-Davidson methods, and Arnoldi methods. My research will examine a particular variant of the Lanczos method, namely the block Lanczos method. BLZPACK [8] (Block Lanczos Package) is a Fortran 77 implementation of the block Lanczos algorithm, written by Osni Marques. My project consists of integrating BLZPACK with SPARSITY [7], a toolkit that generates efficient matrix-vector multiplication routines for matrices stored in a sparse format. I will give a description of SPARSITY as well as BLZPACK and its interface in later sections. BLZPACK is capable of solving both the standard and the generalized eigenvalue problems; however, my research focuses on the standard problem.

The purpose of this project in particular is to examine the effects of changing the block size (explained later) when applying the block Lanczos algorithm. My goal is to decrease the overall execution time of the algorithm by increasing the block size. This involves observing the time spent in different sections of the algorithm, as well as carefully choosing optimal matrix subroutines. This work complements that of the BEBOP group at Berkeley, whose interests include software performance tuning. In particular, BEBOP explores SPARSITY performance in great detail. The relevance to BLZPACK lies in matrix-vector multiplication unrolled across multiple vectors. Depending on the matrix, the time spent for the operation $A*[\mathbf{x}_1,...,\mathbf{x}_k]$ (i.e. a matrix A times k vectors) may be much less than k times the time spent on $A\mathbf{x}$ (i.e. a matrix A times a vector, done k times). BLZPACK can be used with $A*[\mathbf{x}_1,...,\mathbf{x}_k]$ for any k (known as block size); my work explores whether we can accelerate eigenvalue computation by using block size greater than 1. The work performed by BLZPACK depends in a complicated fashion on the matrix, block size, number of desired eigenvalues, and other parameters; hence the answer to that question is not immediately obvious. As I will show, there are cases when using a block size of 1 performs best, and also cases where a greater block size is better.

A related problem of using a blocked procedure for solving linear systems of equations prompted the question of the usefulness of the analogous problem of computing eigenvalues. GMRES [2] is a method for solving $A\mathbf{x} = \mathbf{b}$, where A is large, sparse, and non-symmetric. A blocked version LGMRES instead solves $AX = B$, incorporating fast matrix-multivector-multiply routines. BLZPACK uses a similar approach involving Krylov subspaces, which will be explained in greater detail in Section 2.

The first part of this report deals with the tools I work with: the Lanczos method (with a description of the algorithm), BLZPACK, and SPARSITY. The second part is a collection of the data gathered from my experiments and the results and conclusions drawn from them.

2. Single-Vector Lanczos Method

The class of Lanczos methods refers to procedures for solving for eigenvalues by relying on Lanczos recursion. Lanczos recursion (or tridiagonalization) was introduced by Cornelius Lanczos in 1950. In this section I will describe the algorithm for the single-vector Lanczos method[1].

The simplest Lanczos method uses single-vector Lanczos recursion. Given a real symmetric (square) matrix A of dimension n and an initial unit vector $\mathbf{v}_1$ (usually generated randomly), for j = 1,2,…,m the Lanczos matrices T_j are defined recursively as follows:

```
Let $\mathbf{z} := A\mathbf{v}_i$
$\alpha_i := \mathbf{v}_i^T\mathbf{z}$
$\mathbf{z} := \mathbf{z} - \alpha_i\mathbf{v}_i - \beta_{i-1}\mathbf{v}_{i-1}$
$\beta_i := ||\mathbf{z}||_2$
$\mathbf{v}_{i+1} := \mathbf{z}/\beta_i$
```

The Lanczos matrix T_j is defined to be the real symmetric, tridiagonal matrix with diagonal entries α_i for i = 1,2,...,j and subdiagonal and superdiagonal entries β_i for i = 1,2,...,j.

The vectors $\alpha_i\mathbf{v}_i$ and $\beta_i\mathbf{v}_{i-1}$ are the orthogonal projections of the vector $A\mathbf{v}_i$ onto $\mathbf{v}_i$ and $\mathbf{v}_{i-1}$ respectively. For each i, the Lanczos vector $\mathbf{v}_{i+1}$ is determined by orthogonalizing $A\mathbf{v}_i$ with respect to $\mathbf{v}_i$ and $\mathbf{v}_{i-1}$. The Lanczos matrices are determined by the scalar coefficients α_i and β_{i+1} obtained in these orthogonalizations.

In all Lanczos methods, solving for an eigenvalue of a matrix A is simplified by replacing A with one or more of the Lanczos matrices T_j's. Real symmetric tridiagonal matrices

[1] The details here are a summary of more complete explanations from Chapter 2 of *Lanczos Algorithms for Large Symmetric Eigenvalue Computations* [4].

have small storage requirements and algorithms for their eigenpair computations are efficient [4].

The basic Lanczos procedure follows these steps:

1. Given a real symmetric matrix A, construct (using Lanczos recursion) a family of real symmetric tridiagonal matrices T_j for $j = 1,2,...,M$.
2. For some $m \leq M$ compute the relevant eigenvalues of the Lanczos matrix T_m. (Relevance refers to which eigenvalues from the spectrum are desired, e.g. the eigenvalue with highest absolute value.)
3. Select some or all of these eigenvalues as approximations to eigenvalues of the given matrix A.
4. For each eigenvalue μ of A for which an eigenvector is required, compute a unit eigenvector $\mathbf{u}$ such that $T_m\mathbf{u} = \mu\mathbf{u}$. Map $\mathbf{u}$ into a vector $\mathbf{y} \equiv V_m\mathbf{u}$ (where V_m is the matrix whose k^{th} column is the k^{th} Lanczos vector), which is used as an approximation to an eigenvector of A. Such a vector $\mathbf{y}$ is known as a Ritz vector; eigenvalues of Lanczos matrices are called Ritz values of A.

In the Lanczos recursion formulas, the original matrix A is used only for the product $A\mathbf{v}_i$; hence it is never modified. For large sparse matrices, this enables storage optimizations, since a user only needs a subroutine which computes $A\mathbf{x}$ for any vector $\mathbf{x}$, which can be done in space linear in the dimension of the matrix. Furthermore, the number of arithmetic operations required to generate a Lanczos matrix is proportional to the number of nonzero entries of A, as opposed to $O(n^3)$ (where n is the size of A) for procedures which completely transform A into a real symmetric tridiagonal matrix by performing an orthogonal similarity transformation.

That the Lanczos matrices possess eigenvalues that can reasonably approximate those of A is not immediately clear. The key facts (which I state without proof[2]) are that the Lanczos vectors form an orthonormal set of vectors, and that the eigenvalues of the Lanczos matrices are the eigenvalues of A restricted to the family of subspaces $\mathcal{K}^j \equiv$

[2] The proof is in Chapter 2 of *Lanczos Algorithms* [4].

sp{$\mathbf{v}_1$,A$\mathbf{v}_1$,$A^2\mathbf{v}_1$,...,$A^{j-1}\mathbf{v}_1$}, known as Krylov subspaces. If j is sufficiently large, the eigenvalues of T_n should be good approximations to the eigenvalues of A. Continuing the Lanczos recursion until j = n (where n is the size of A), T_n is an orthogonal similarity transformation of A, and therefore has the same eigenvalues as A. A Ritz vector $V_j\mathbf{u}$ obtained from an eigenvector **u** of a given T_j is an approximation to a corresponding eigenvector of A.

Error analysis is given in terms of the angle between a vector and a subspace, as explained in Chapter 2 of *Lanczos Algorithms* [4]. It turns out that Krylov subspaces are very good subspaces on which to compute eigenpair approximations. An important result is that the error bound increases as we proceed into the spectrum; that is, extreme eigenvalues and corresponding eigenvectors have higher expected accuracy than eigenvalues in the interior of the spectrum. (Depending on the particular implementation, however, this may not be the case.) For this reason the basic Lanczos method is often a good algorithm to use when we desire a few extreme eigenvalues.

The above description of the Lanczos algorithm assumes exact arithmetic; in practice this is usually not the case. Indeed, in computer implementations, finite precision causes roundoff errors at every arithmetic operation. Computed quantities will obviously differ from the theoretical values. When constructing the Lanczos matrices T_j, the Lanczos vectors lose their orthogonality (and even linear independence) as j increases. The Lanczos matrices are no longer orthogonal projections of A onto the subspaces sp{V_j}. The theoretical relationship between T_j and A and error estimates are no longer applicable. It was originally assumed that a total reorthogonalization of the Lanczos vectors was required. There are Lanczos methods which in fact use no reorthogonalization and employ modified recursion formulas. However, the variant I will be working with uses selective orthogonalization and modified partial reorthogonalization to preserve orthogonality of the Lanczos vectors, as will be described later.

3. Block Lanczos Method

Many eigenvalue algorithms have variants in which blocks of vectors are used instead of single vectors. When multiplying by vectors, these blocks can be considered matrices themselves. As a result, instead of performing matrix-vector operations (Level 2 BLAS routines [5]), matrix-matrix operations (Level 3 BLAS operations [5]) are used. This allows for possible optimizations in the execution of the algorithm. I/O costs are decreased by essentially a factor of the block size [1].

Lanczos methods in particular benefit from a blocked algorithm variant in another way. Depending on the implementation, single-vector methods sometimes have difficulty computing the multiplicities of eigenvalues, and for a multiple eigenvalue a complete basis for the subspace may not be directly computed. A block Lanczos method may be better for computing multiplicities and bases for invariant subspaces corresponding to eigenvalues.

As an alternative to the recurrence relations for the single-vector variant, we use the following approach[3]. Define matrices $B_1 \equiv 0$ and $Q_0 \equiv 0$. Let Q_1 be an nxq matrix whose columns are orthonormalized, randomly generated vectors, where n is the size of the original matrix A and q is the block size. For i = 1,2,...,s define Lanczos blocks Q_i according to the following recursive equations:

```
    Let $Z := AQ_i$
    $A_i := Q_i^T Z$
    $Z := Z - Q_i A_i - Q_{i-1} B_{i-1}$
    Factor $Z = Q_{i+1} B_i$ by Gram-Schmidt
```

The blocks Q_j for j = 1,2,...,s form an orthonormal basis for the Krylov subspace $K^s(Q_1,A) \equiv sp\{Q_1,AQ_1,...,A^{s-1}Q_1\}$ corresponding to the first block. The Lanczos matrices T_s are defined as the block tridiagonal matrices with $A_1,A_2,...,A_s$ along the diagonal, and $B_1,B_2,...,B_s$ along the subdiagonal, and $B_1^T,B_2^T,...,B_s^T$ along the superdiagonal.

[3] The Block Lanczos algorithm described here is a summary of the steps described in Chapter 7 of *Lanczos Algorithms* [4]. The proof of its correctness is there as well; I have not restated it here.

Analogously to the single-vector case, we approximate eigenvalues of A by computing eigenvalues of the T_s matrices.

When A is large enough, the cost of this algorithm should be dominated by the multiplication AQ_i, which is the operation for which we have specially tuned routines to evaluate.

4. BLZPACK[4]

BLZPACK [8] (Block Lanczos Package) is a Fortran 77 implementation of the Block Lanczos algorithm, written by Osni Marques. It is designed to solve both the standard and generalized eigenvalue problems. I work with the standard problem, in which we solve for eigenvalues λ and eigenvectors $\mathbf{x}$ that satisfy $A\mathbf{x} = \lambda\mathbf{x}$, where A is a real sparse symmetric matrix. There are single precision and a double precision versions; I use the double precision version. The main subroutine is BLZDRD, to which the user passes parameters and data I will describe in this section. The only computations involving A are matrix-multiple-vector multiplications done outside the BLZDRD subroutine. In this way, the representation for A and the implementation of matrix operations on A are completely decided by the user.

The BLZDRD interface expects numerous arguments; here I will briefly run through the parameters relevant to my project. Many parameters are ignored because they are meaningless for the standard problem.

- NI: The number of active rows of temporary arrays in the block Lanczos algorithm on the current process. BLZPACK can be run in sequential or parallel mode; for my project only the sequential mode is relevant, so NI is set to the dimension of A.
- LNI: Dimension of temporary arrays; LNI and NI have the same value in all my experiments.
- NREIG: The number of desired eigenpairs. I run my experiments with NREIG = 1, 10, and 50. The eigenvalues returned are those with the greatest magnitude.
- LEIG: Dimension of the array in which to store converged eigenvalues (this may be greater than NREIG). I used LEIG = (NREIG*2)+10.
- NVBSET: The number of vectors in a block. This is the focus of the project; we try to see when an increase in the block size leads to an increase in overall performance. With a block size of 1, the algorithm essentially works as a single-vector Lanczos method. I run the algorithm with block sizes from 1 to 9.

[4] The information in this section can be found in greater detail in the BLZPACK *User's Guide* [9].

- NSTART: Number of starting vectors given to BLZPACK; I use 0 for this value, which causes BLZPACK to generate random starting vectors.
- NGEIG: Number of eigenpairs given as input; 0 is used.
- LISTOR/LRSTOR: Amount of workspace to allocate. For all experiments 10^7 was used for both values.
- THRSH: The threshold for convergence. The default value $||A||\sqrt{\varepsilon}$ is used, where ε is the machine precision 2.2204×10^{-16}, and $||A||$ is estimated by means of the eigenvalue distribution computed by BLZPACK. A computed eigenpair $(\lambda, \mathbf{x})$ is considered converged iff $||A\mathbf{x} - \lambda\mathbf{x}|| \leq$ THRSH.
- NSTEPS: The maximum number of steps to be performed per run. The default value is used, which is determined by BLZPACK based on allocated workspace (see LISTOR and LRSTOR). If not enough eigenpairs have converged after NSTEPS, a restart is performed, using as initial vectors some linear combination of the unconverged eigenvectors.

The block Lanczos algorithm is implemented as follows. Lanczos vectors are denoted by Q_j, and $\mathbf{Q}_j$ is defined to be the basis of Lanczos vectors, $[Q_1\ Q_2\ ...\ Q_j]$. T_j is the jth Lanczos matrix (block tridiagonal) as described in the previous section. At initialization, set $Q_0 = 0$. Set $R_0 \neq 0$ randomly and factorize R_0 as Q_1B_1 where $Q_1^TQ_1$ is the identity. On the jth Lanczos step:

1. Compute $R_j = AQ_j$
2. $R_j := R_j - Q_{j-1}B_j^T$
3. $A_j := Q_j^TR_j$
4. $R_j := R_j - Q_jA_j$
5. Factorize R_j as $Q_{j+1}B_{j+1}$ where $Q_{j+1}^TQ_{j+1}$ is the identity
6. If required, orthogonalize Q_j and Q_{j+1} against the vectors in $\mathbf{Q}_{j-1}$
7. Insert Q_j into $\mathbf{Q}_j$ and A_j, B_j into T_j
8. Solve the reduced problem T_j

There are further steps to test convergence of the computed eigenpairs. If after some number of steps (set either by the user or by default) not enough eigenpairs have converged, Q_j and Q_{j+1} are orthogonalized against specific previously computed vectors; this is referred to as selective orthogonalization. The process is then restarted. The orthogonalization in step 6 above is a modified partial orthogonalization. These two orthogonalization strategies are employed as an alternative to total reorthogonalization for preserving the orthogonality of the Lanczos vectors.

5. Integrating SPARSITY Into BLZPACK

SPARSITY[5] [7] is a toolkit for generating optimized sparse matrix-vector multiplication routines, developed by Eun-Jin Im and Katherine Yelick. SPARSITY employs register blocking, which reorganizes the data structure representing the matrix by identifying small blocks of nonzero elements and storing these blocks contiguously. Further optimization is made possible generating code that unrolls across multiple vectors, so that the operation becomes more similar to a matrix-matrix multiplication, where one matrix is sparse and the other is dense. The result of unrolling is often a speedup significant enough that the time for (A * k vectors) is much lower than k times the time for (A * a single vector). SPARSITY examines a given matrix, and depending on the architecture, generates a suitable routine. Ongoing research examines the effects of taking into account symmetry when generating code. For the purposes of my project, I will refer to a particular SPARSITY-generated routine as a "rxcxv (symmetric or non-symmetric) implementation", where rxc is the block size by which the matrix is blocked, and v is the number of right-hand-sides (vectors) to unroll across. This project was completed on an UltraSPARC processor with a clock rate of 333 MHz, using f77 to compile BLZPACK and cc to compile the BLZPACK driver as well as SPARSITY-generated routines.

To use BLZPACK, I wrote a driver in C which calls the BLZDRD routine, and between calls computes V := A*U using a SPARSITY-generated routine. The particular routine is determined beforehand according to the matrix A and the block size of the block Lanczos algorithm. Optimal implementations given the number of right-hand-sides were provided to me by Rich Vuduc and Benjamin Lee. For the matrices, I ran the block Lanczos algorithm with NREIG = 1, 10, and 50, and with NVBSET = 1 to 9. The goal was to find an example where a block size of greater than 1 gave a better performance than using a block size of 1 (which is the single-vector algorithm). To see any improved performance, the speedup from using a multiple-vector matrix-vector-multiply routine over a single-vector routine must at least outweigh the increase in the number of matrix-vector operations required by the algorithm. Furthermore, that speedup should also dominate

[5] The information on SPARSITY was taken from Eun-Jin's Ph.D. thesis [5].

increases in other operations required by the block Lanczos algorithm, such as vector generation and reorthogonalization.

The remainder of this report deals with the data given by BLZPACK, and the results drawn from it.

6. Nasasrb Matrix

Nasasrb [6] is the matrix on which I performed the most extensive tests. It has dimension 54870, with 2677324 nonzeros, a density of 0.09%. It is used in shuttle rocket booster applications. I ran it through BLZPACK with 1, 10, and 50 required eigenpairs, with block sizes ranging from NVBSET = 1 to 9. The results are shown in Figures 6.1-6.3 below. The matrix-vector-multiply implementations (see the previous section on SPARSITY for an explanation about my notation) for particular right-hand-sides are as follows:

- 1 RHS: 3x3x1 symmetric
- 2 RHS: 3x2x2 symmetric
- 3 RHS: 2x1x3 symmetric
- 4 RHS: 2x1x4 symmetric
- 5 RHS: 3x2x5 non-symmetric
- 6 RHS: 3x2x6 non-symmetric
- 7 RHS: 2x3x7 non-symmetric
- 8 RHS: 2x1x8 non-symmetric
- 9 RHS: 2x1x9 non-symmetric

From the results, we can see that increasing the block size from 1 does reduce the time spent doing matrix-vector operations (specifically, the A*U operation between calls to BLZDRD), at least up to block size 4 before increasing again. Even with the increase in the number of matrix-vector operations required to satisfy BLZPACK's stopping criterion based on convergence (see column labeled "# MVM Ops"), the speedup in the implementation (see column labeled "Time per MVM") is enough to decrease the total time spent doing matrix-vector multiplies (see column labeled "Time for MVM Ops"). However, we can see that for all numbers of required eigenpairs, the total execution time (see column labeled "Total Time") is always lowest with the single-vector case (block size 1). Therefore, the single-vector procedure is the most time-efficient algorithm to use. Note that overall time is not necessarily an increasing function of block size. For 1 and 10 required eigenpairs, the overall time with block size 4 is actually lower than with block size 3. Nevertheless, it is still fastest with block size 1 in both cases.

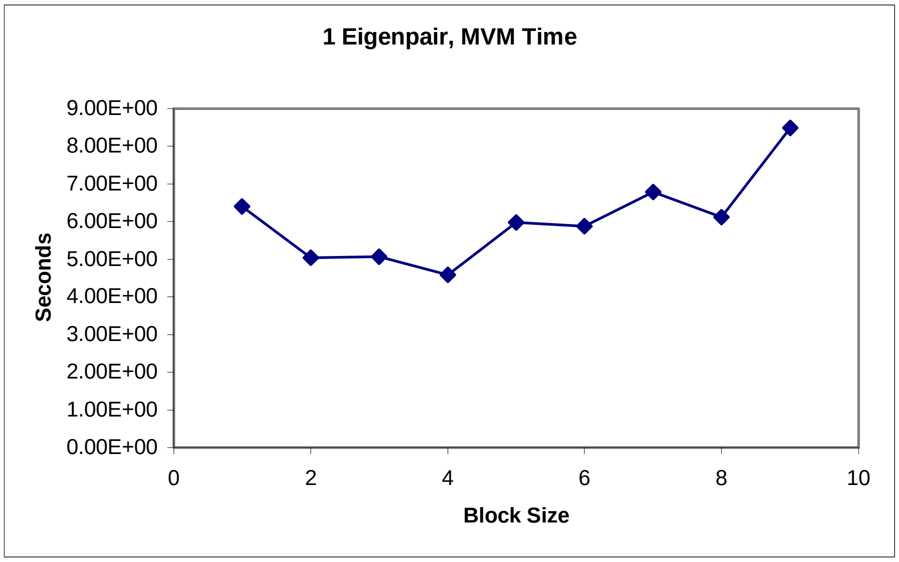
The percentage of time spent in A*U operations is about 30-40% for small block sizes, to 10% or less for higher block sizes (see column "% Time Spent for MVM"). Examining the data more closely reveals that the bottleneck is specifically in the reorthogonalization (see column "Time for Reorth"). Therefore, any speedup gained by increasing block size is dominated by the increase in reorthogonalization costs, thus accounting for the increase in overall time. Note that there is a speedup in the time it takes per matrix-vector multiply as we increase block size (see column "Time per MVM"); this is what we expect from SPARSITY, that multiplying multiple vectors is faster (per vector) than multiplying a single vector. Still, it is not enough to outweigh other costs. Aside from matrix-vector operations and reorthogonalization, BLZPACK spends its time in vector generation, solving the reduced problem, and computing Ritz vectors.

Consulting with Osni Marques offered two insights on possible situations where using a block size greater than 1 might be advantageous. First, the subroutines used to perform reorthogonalizations change when moving from 1 to multiple vectors. With a single vector, BLAS-2 operations are used. Using a bigger block size employs BLAS-3 operations instead. It is possible that continuing to use BLAS-2 operations on multiple vectors may change the performance. However, Osni commented that while this may work for small matrices, it would almost surely not work for large ones. The second idea was that while for this particular matrix (nasasrb) the single-vector procedure worked best, there may be matrices for which the A*U operations dominate the reorthogonalizations. (Indeed, I will show later an example of matrices where this is the case.) Then in those cases, a block implementation would improve performance.

In the next sections, I discuss the results of running the same tests on different matrices. However, I chose to show results for block size up to 5 since anything higher generally did not help; in fact, calculating megaflop rates indicate that the matrix-vector-multiply routines generated by SPARSITY usually show decreased performance with right-hand-sides greater than 5 or 6.

Figure 6.1 nasasrb – 1 required eigenpair

# Vectors in Block	# MVM Ops	Time for MVM Ops	Time per MVM	Time for Reorth	Total Time	% Time Spent for MVM	% Time Spent Reorth	% Time Spent Other
1	84	6.40E+00	7.62E-02	5.38E+00	1.44E+01	44%	37%	18%
2	110	5.04E+00	4.58E-02	9.59E+00	2.17E+01	23%	44%	33%
3	120	5.07E+00	4.23E-02	1.34E+01	2.45E+01	21%	55%	25%
4	132	4.59E+00	3.48E-02	1.19E+01	2.36E+01	19%	50%	30%
5	150	5.97E+00	3.98E-02	1.61E+01	3.16E+01	19%	51%	30%
6	168	5.88E+00	3.50E-02	2.14E+01	4.14E+01	14%	52%	34%
7	182	6.79E+00	3.73E-02	2.28E+01	4.95E+01	14%	46%	40%
8	184	6.12E+00	3.33E-02	2.53E+01	5.33E+01	11%	47%	41%
9	234	8.49E+00	3.63E-02	2.99E+01	6.80E+01	12%	44%	44%



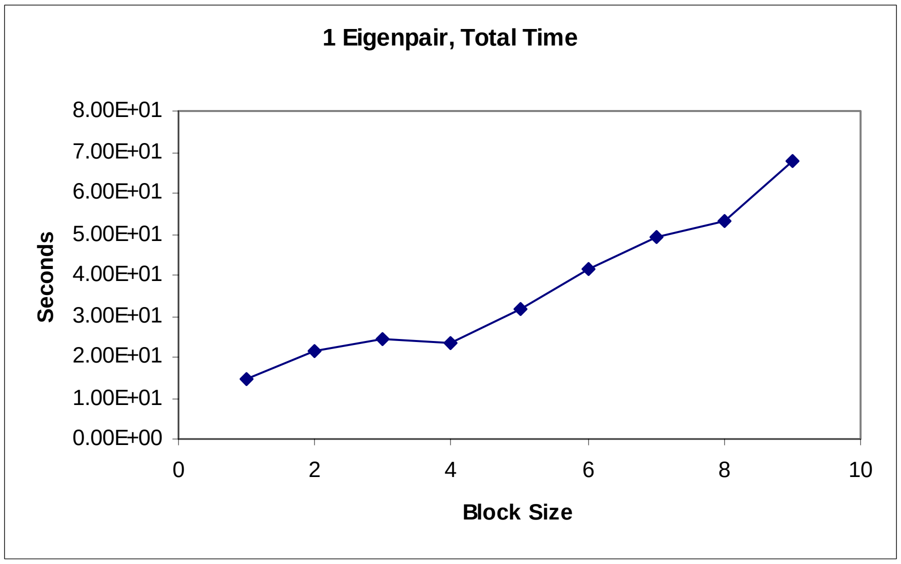
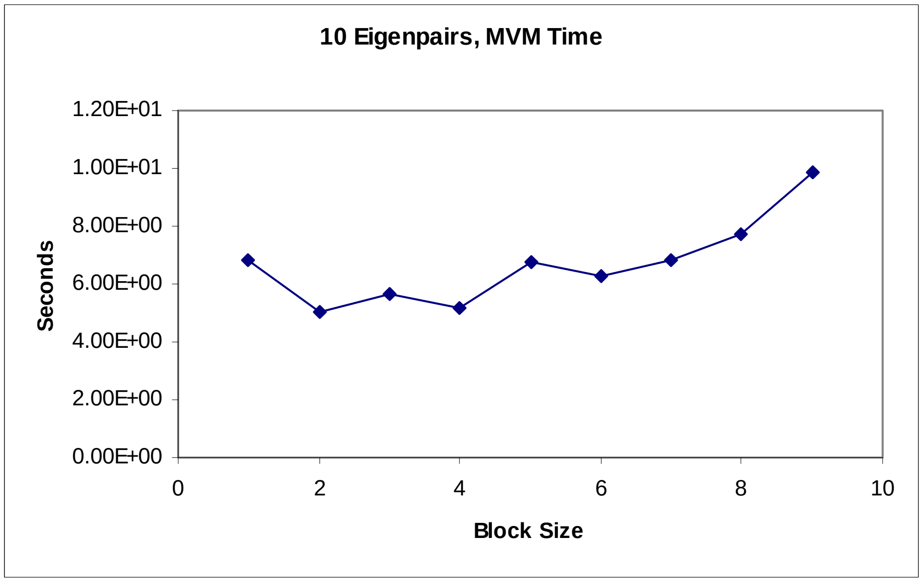


Figure 6.2 nasasrb – 10 required eigenpairs

# Vectors in Block	# MVM Ops	Time for MVM Ops	Time per MVM	Time for Reorth	Total Time	% Time Spent for MVM	% Time Spent Reorth	% Time Spent Other
1	90	6.84E+00	7.60E-02	6.23E+00	1.62E+01	42%	38%	19%
2	110	5.04E+00	4.58E-02	1.16E+01	2.17E+01	23%	53%	23%
3	135	5.68E+00	4.21E-02	1.65E+01	2.97E+01	19%	56%	25%
4	148	5.14E+00	3.47E-02	1.49E+01	2.86E+01	18%	52%	30%
5	170	6.75E+00	3.97E-02	2.07E+01	4.09E+01	17%	51%	33%
6	180	6.31E+00	3.51E-02	2.17E+01	4.50E+01	14%	48%	38%
7	182	6.81E+00	3.74E-02	2.29E+01	4.94E+01	14%	46%	40%
8	232	7.69E+00	3.31E-02	3.07E+01	6.84E+01	11%	45%	44%
9	270	9.83E+00	3.64E-02	3.44E+01	7.79E+01	13%	44%	43%



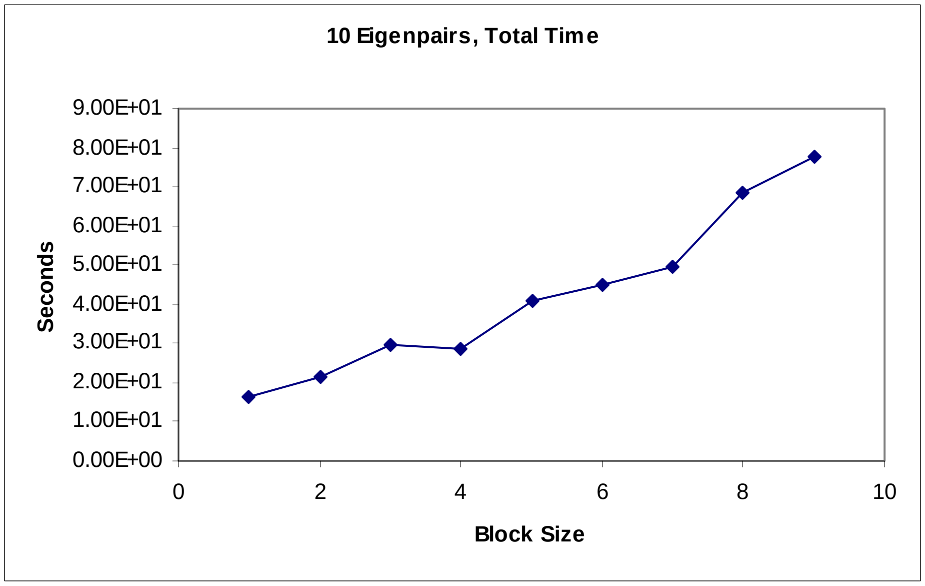
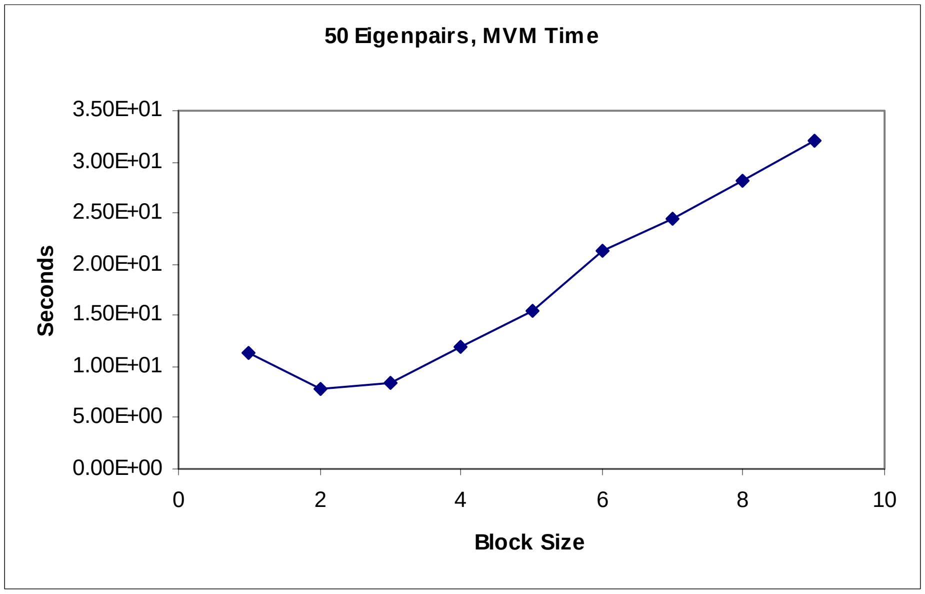
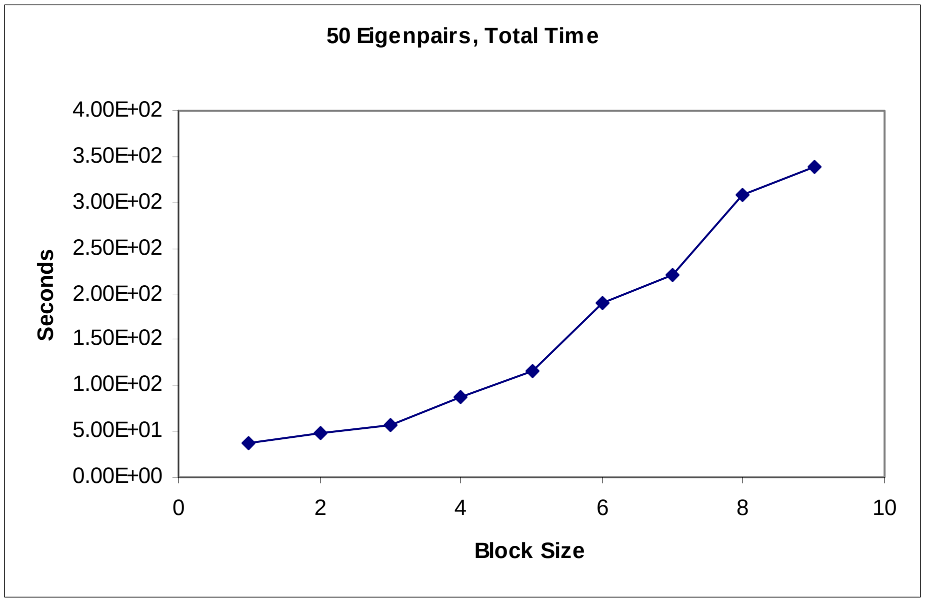


Figure 6.3 nasasrb – 50 required eigenpairs

# Vectors in Block	# MVM Ops	Time for MVM Ops	Time per MVM	Time for Reorth	Total Time	% Time Spent for MVM	% Time Spent Reorth	% Time Spent Other
1	151	1.14E+01	7.55E-02	1.90E+01	3.73E+01	31%	51%	18%
2	170	7.80E+00	4.59E-02	2.96E+01	4.78E+01	16%	62%	22%
3	201	8.46E+00	4.21E-02	1.09E+01	5.64E+01	15%	55%	30%
4	344	1.19E+01	3.46E-02	4.98E+01	8.79E+01	14%	57%	30%
5	390	1.54E+01	3.95E-02	6.13E+01	1.15E+02	13%	53%	33%
6	612	2.14E+01	3.50E-02	1.05E+02	1.90E+02	11%	55%	33%
7	651	2.44E+01	3.75E-02	1.16E+02	2.21E+02	11%	52%	36%
8	848	2.82E+01	3.33E-02	1.64E+02	3.08E+02	9%	53%	38%
9	882	3.21E+01	3.64E-02	1.77E+02	3.39E+02	9%	52%	38%





7. Bcsstk, Crystk, and Vibrobox Matrices

The bcsstk [6] matrix, used in an automobile frame application, has dimension 30237 and 1450163 nonzeros, a density of 0.16%. As before, the single-vector procedure is optimal for all numbers of required eigenvalues, even though execution time does not necessarily increase with block size. The matrix-vector-multiply implementations used are as follows:

- 1 RHS: 3x3x1 symmetric
- 2 RHS: 3x2x2 symmetric
- 3 RHS: 3x1x3 symmetric
- 4 RHS: 2x1x4 symmetric
- 5 RHS: 3x2x5 non-symmetric

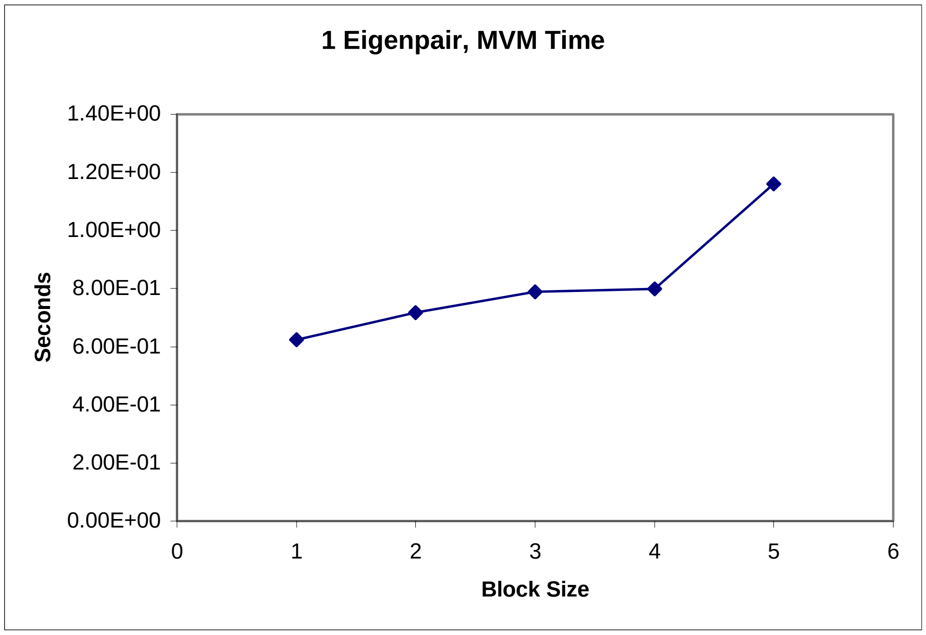
The crystk [6] matrix is used for crystal free vibration applications. It has dimension 24696 and 1751178 nonzeros, a density of 0.29%. The single-vector procedure is optimal. The matrix-vector-multiply implementations are:

- 1 RHS: 3x3x1 symmetric
- 2 RHS: 3x2x2 symmetric
- 3 RHS: 3x1x3 symmetric
- 4 RHS: 2x1x4 symmetric
- 5 RHS: 3x1x5 non-symmetric

The vibrobox [6] matrix is used for the structure of the vibroacoustic problem. It has dimension 12328 with 34828 nonzeros, a density of 0.23%. Once again, the single-vector procedure is optimal. Blocking of this matrix was not helpful in optimizing matrix-vector-multiply routines, so the 1x1xv (with v equal to the block size for the Lanczos algorithm) routine, using the symmetric version for block sizes 1 to 4 and non-symmetric for block size 5.

Figure 7.1 bcsstk – 1 required eigenpair

# Vectors in Block	# MVM Ops	Time for MVM Ops	Time per MVM	Time for Reorth	Total Time	% MVM	% Reorth	% Other
1	16	6.24E-01	3.90E-02	1.59E-01	1.30E+00	48%	12%	40%
2	28	7.17E-01	2.56E-02	6.51E-01	2.22E+00	32%	29%	38%
3	36	7.89E-01	2.19E-02	1.13E+00	3.10E+00	25%	36%	38%
4	44	7.99E-01	1.82E-02	1.23E+00	3.52E+00	23%	35%	42%
5	55	1.16E+00	2.11E-02	1.94E+00	5.08E+00	23%	38%	39%



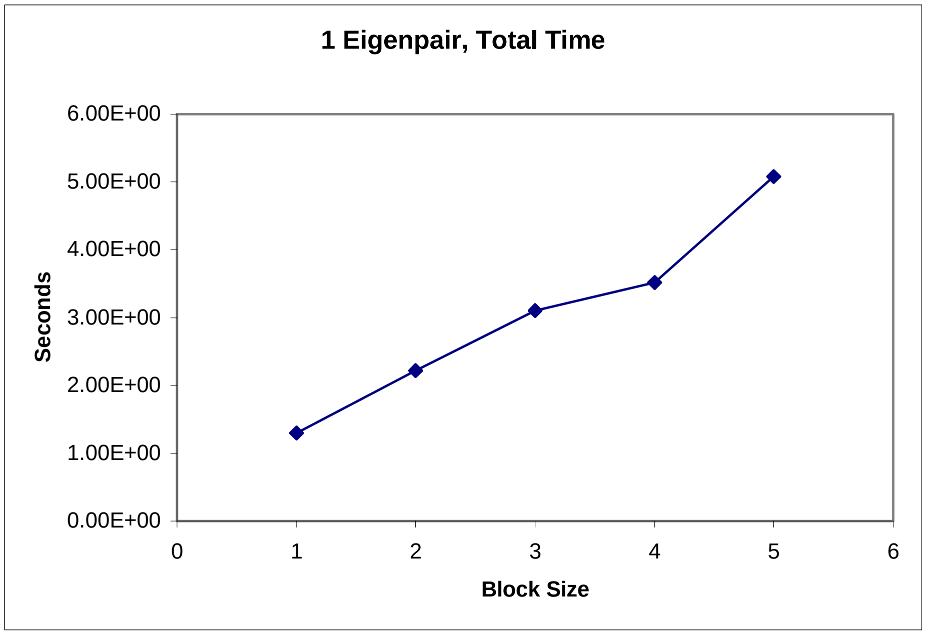
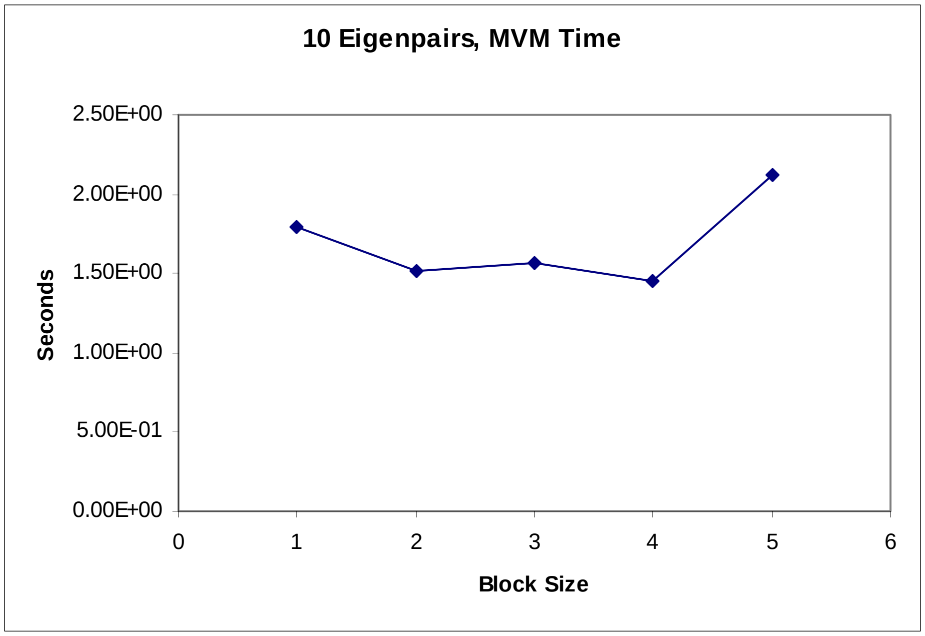


Figure 7.2 bcsstk – 10 required eigenpairs

# Vectors in Block	# MVM Ops	Time for MVM Ops	Time per MVM	Time for Reorth	Total Time	% MVM	% Reorth	% Other
1	45	1.79E+00	3.98E-02	1.34E+00	4.06E+00	44%	33%	23%
2	58	1.51E+00	2.60E-02	2.69E+00	5.69E+00	27%	47%	26%
3	72	1.57E+00	2.18E-02	3.86E+00	7.53E+00	21%	51%	28%
4	80	1.45E+00	1.81E-02	3.30E+00	7.28E+00	20%	45%	35%
5	100	2.12E+00	2.12E-02	5.18E+00	1.08E+01	20%	48%	32%



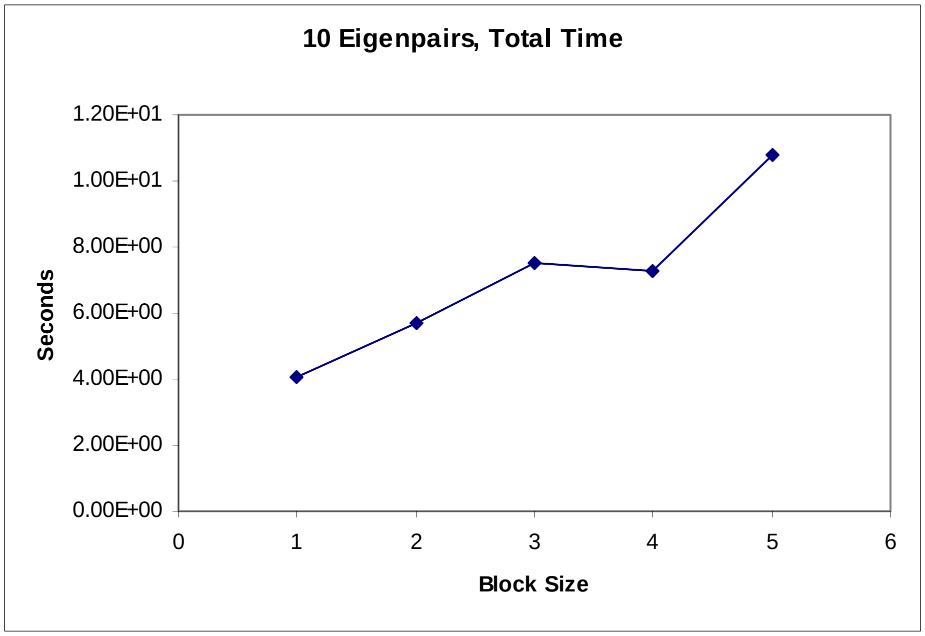
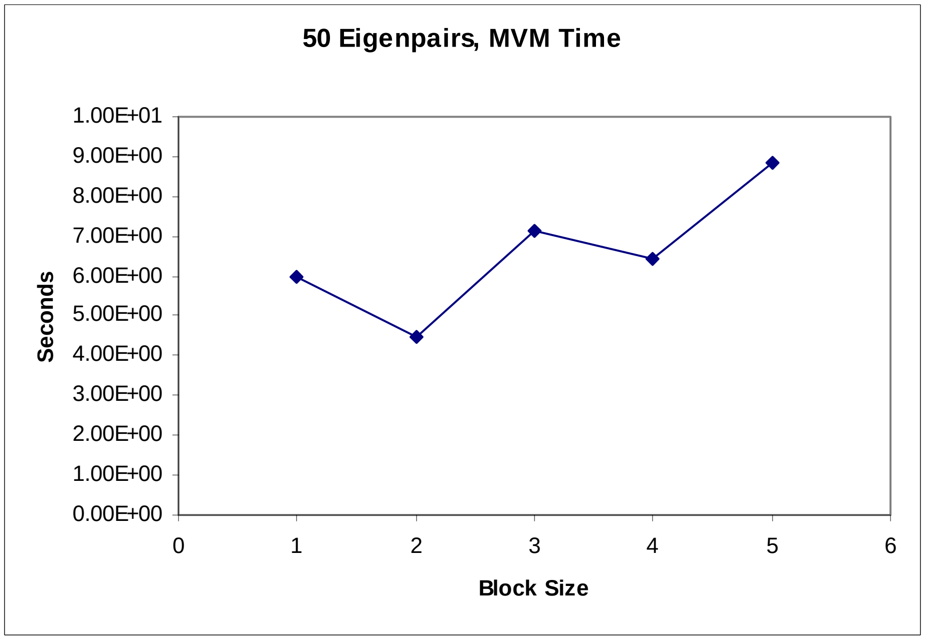


Figure 7.3 bcsstk – 50 required eigenpairs

# Vectors in Block	# MVM Ops	Time for MVM Ops	Time per MVM	Time for Reorth	Total Time	% MVM	% Reorth	% Other
1	150	5.99E+00	3.99E-02	1.68E+01	2.66E+01	23%	63%	14%
2	172	4.45E+00	2.59E-02	2.17E+01	3.16E+01	14%	69%	17%
3	324	7.12E+00	2.20E-02	3.87E+01	5.68E+01	13%	68%	19%
4	356	6.41E+00	1.80E-02	3.12E+01	5.02E+01	13%	62%	25%
5	420	8.86E+00	2.11E-02	3.69E+01	6.23E+01	14%	59%	27%



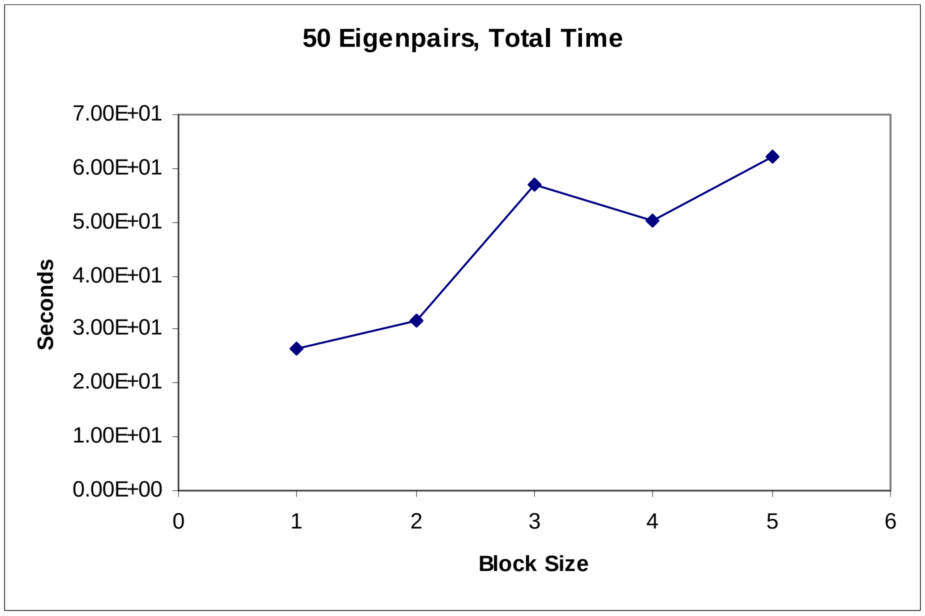
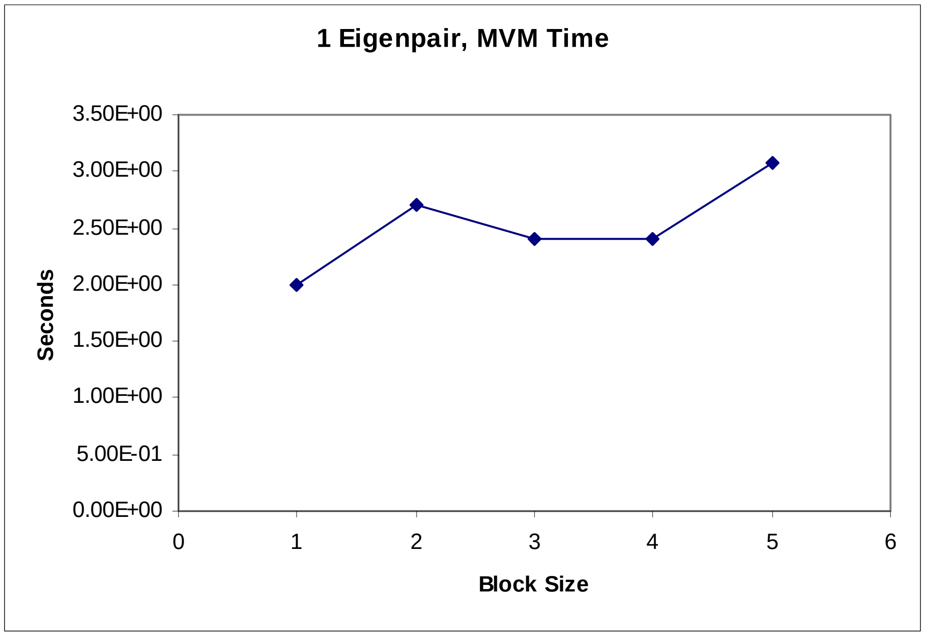


Figure 7.4 crystk – 1 required eigenpair

Vectors in Block	# MVM Ops	Time for MVM Ops	Time per MVM	Time for Reorth	Total Time	% MVM	% Reorth	% Other
1	45	2.00E+00	4.44E-02	5.79E-01	3.27E+00	61%	18%	21%
2	96	2.71E+00	2.82E-02	2.97E+00	7.31E+00	37%	41%	22%
3	99	2.41E+00	2.43E-02	3.26E+00	7.63E+00	32%	43%	26%
4	108	2.41E+00	2.23E-02	2.66E+00	7.30E+00	33%	36%	31%
5	130	3.07E+00	2.36E-02	4.07E+00	1.03E+01	30%	40%	31%



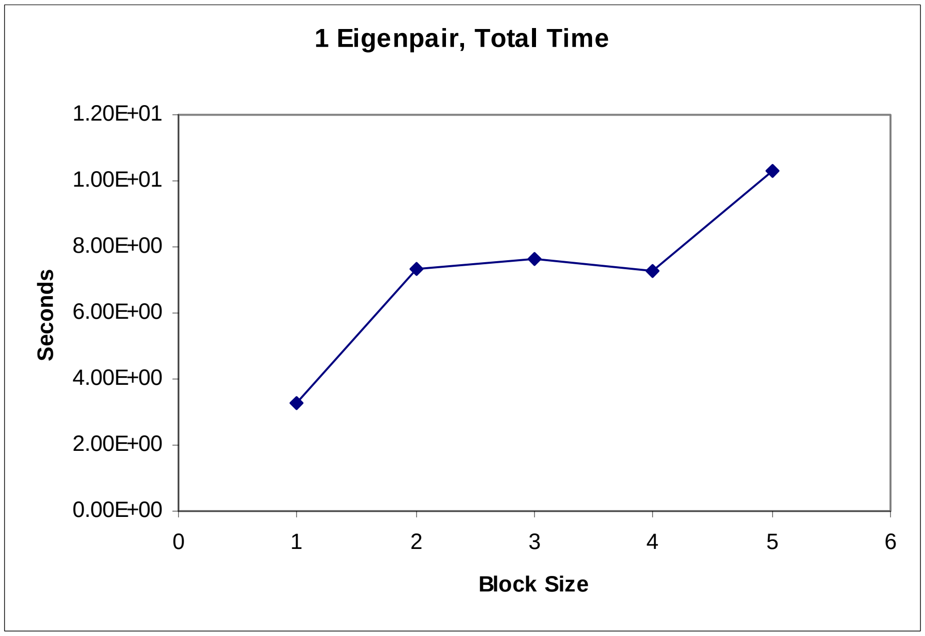
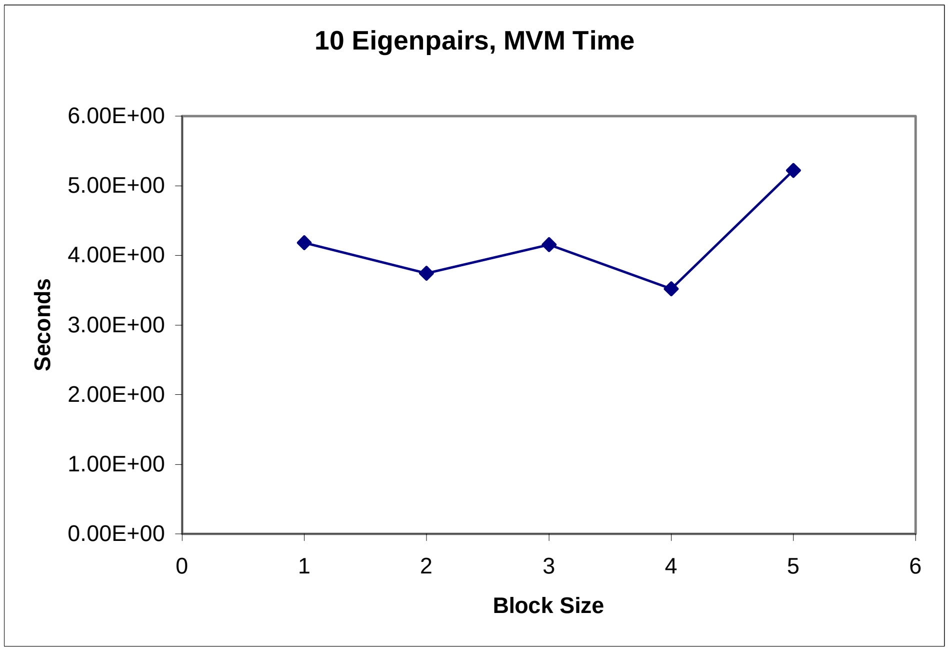


Figure 7.5 crystk – 10 required eigenpairs

Vectors in Block	# MVM Ops	Time for MVM Ops	Time per MVM	Time for Reorth	Total Time	% MVM	% Reorth	% Other
1	94	4.18E+00	4.45E-02	2.36E+00	7.86E+00	53%	30%	17%
2	132	3.74E+00	2.83E-02	5.33E+00	1.15E+01	33%	46%	21%
3	171	4.15E+00	2.43E-02	8.48E+00	1.64E+01	25%	52%	23%
4	156	3.52E+00	2.26E-02	4.91E+00	1.20E+01	29%	41%	30%
5	220	5.22E+00	2.37E-02	7.44E+00	1.85E+01	28%	40%	32%



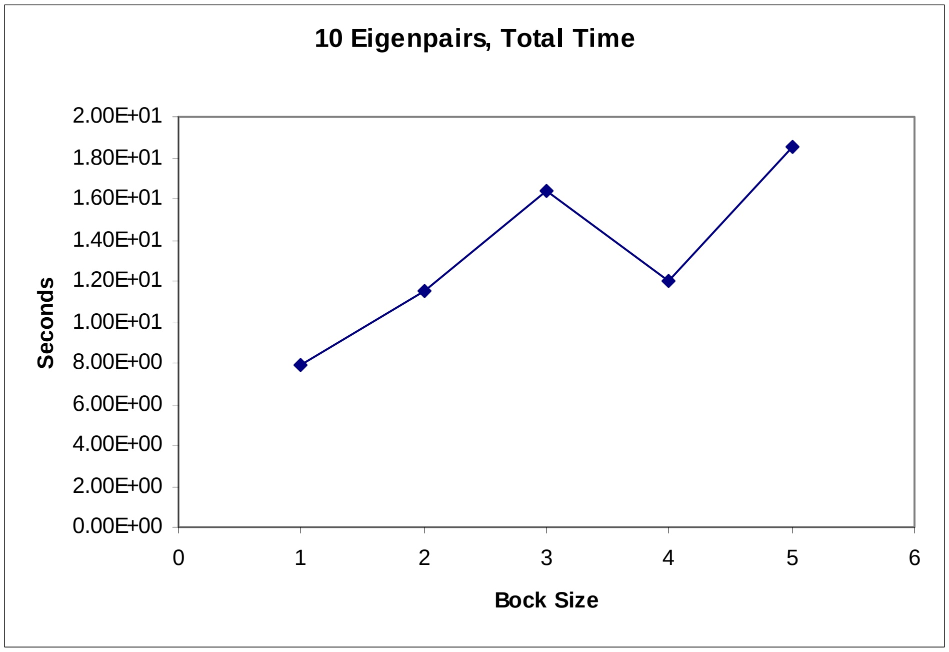
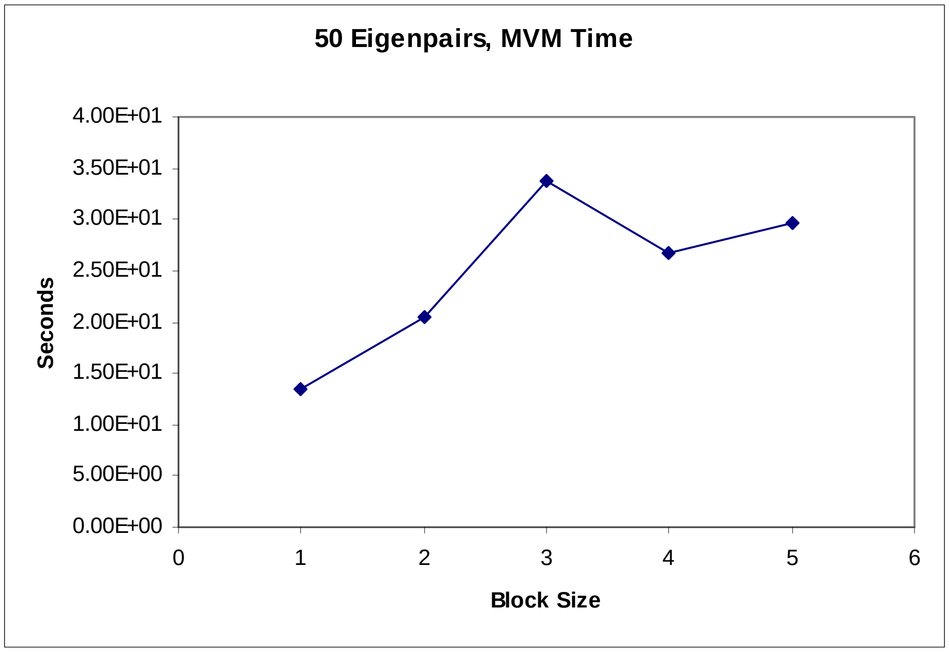


Figure 7.6 crystk – 50 required eigenpairs

# Vectors in Block	# MVM Ops	Time for MVM Ops	Time per MVM	Time for Reorth	Total Time	% MVM	% Reorth	% Other
1	301	1.35E+01	4.49E-02	2.13E+01	4.02E+01	34%	53%	13%
2	714	2.04E+01	2.86E-02	6.93E+01	1.04E+02	20%	67%	14%
3	1383	3.37E+01	2.44E-02	1.33E+02	1.98E+02	17%	67%	16%
4	1192	2.67E+01	2.24E-02	9.65E+01	1.51E+02	18%	64%	18%
5	1260	2.97E+01	2.36E-02	1.03E+02	1.67E+02	18%	62%	21%



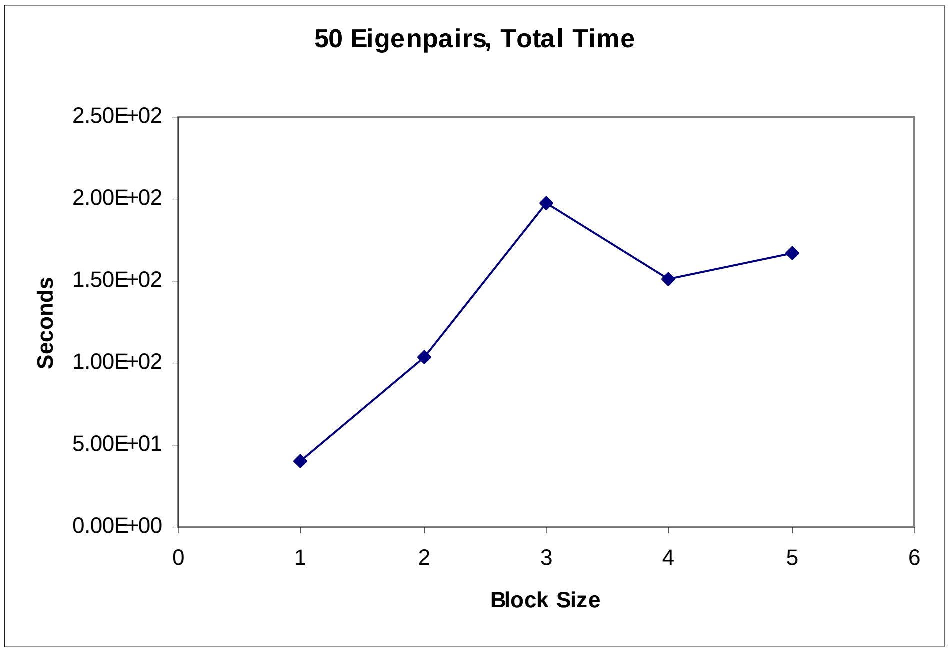
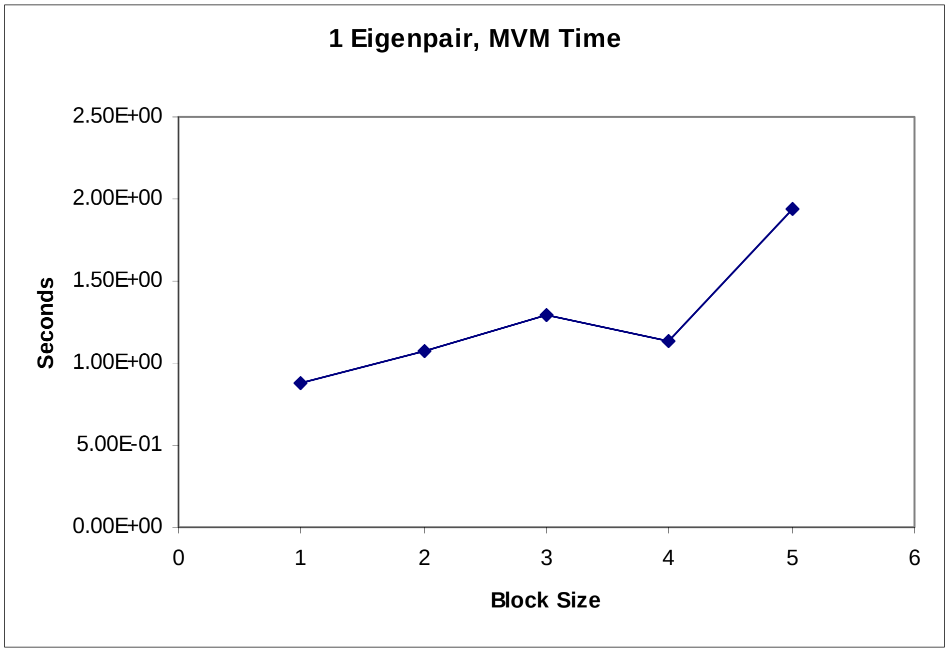


Figure 7.7 vibrobox – 1 required eigenpair

# Vectors in Block	# MVM Ops	Time for MVM Ops	Time per MVM	Time for Reorth	Total Time	% MVM	% Reorth	% Other
1	53	8.76E-01	1.65E-02	3.58E-01	1.62E+00	54%	22%	24%
2	108	1.07E+00	9.91E-03	1.65E+00	3.70E+00	29%	45%	26%
3	153	1.29E+00	8.43E-03	3.31E+00	6.41E+00	20%	52%	28%
4	156	1.13E+00	7.24E-03	2.18E+00	5.27E+00	21%	41%	37%
5	240	1.94E+00	8.08E-03	3.96E+00	9.21E+00	21%	43%	36%



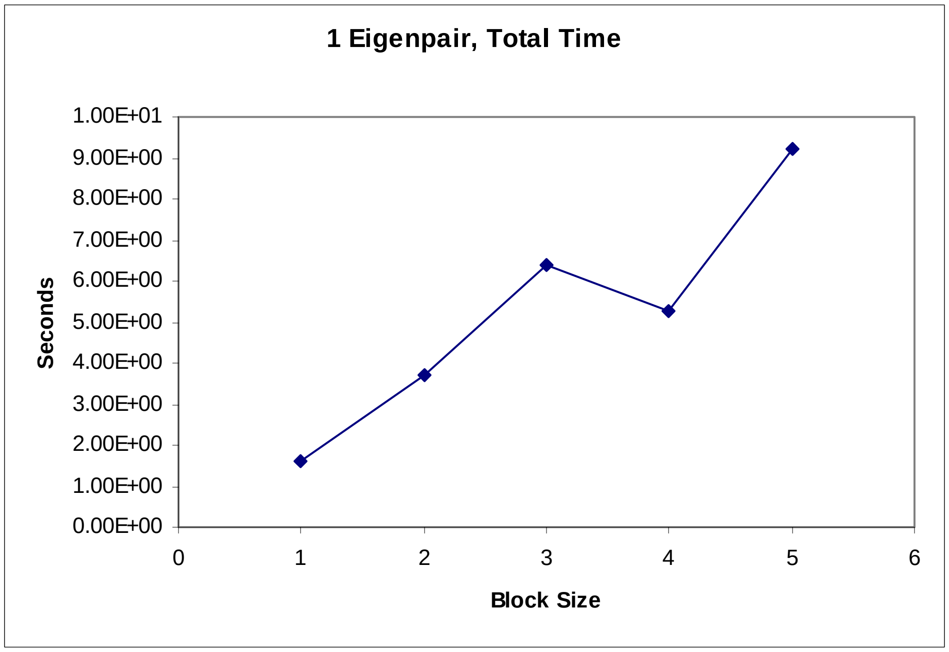
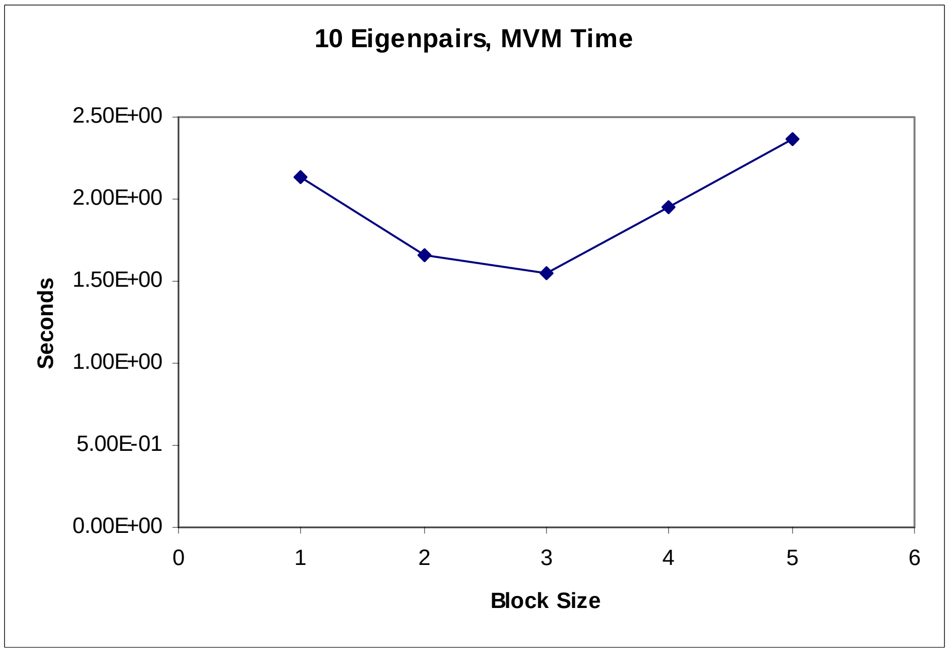


Figure 7.8 vibrobox – 10 required eigenpairs

# Vectors in Block	# MVM Ops	Time for MVM Ops	Time per MVM	Time for Reorth	Total Time	% MVM	% Reorth	% Other
1	123	2.13E+00	1.73E-02	1.83E+00	4.92E+00	43%	37%	20%
2	156	1.66E+00	1.06E-02	3.33E+00	6.74E+00	25%	49%	26%
3	180	1.55E+00	8.61E-03	4.10E+00	8.04E+00	19%	51%	30%
4	260	1.95E+00	7.50E-03	3.60E+00	8.90E+00	22%	40%	38%
5	300	2.37E+00	7.90E-03	4.93E+00	1.14E+01	21%	43%	36%



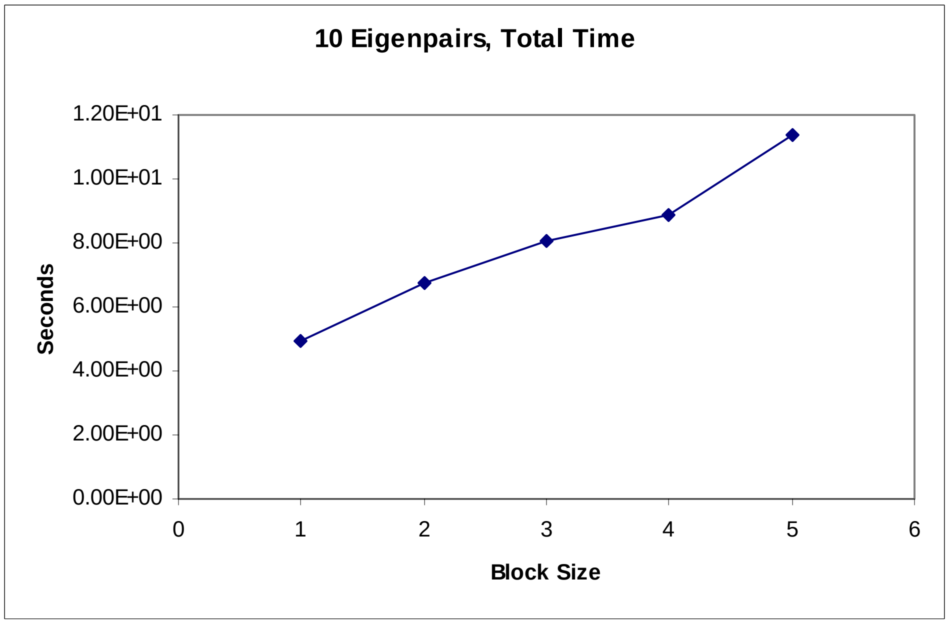
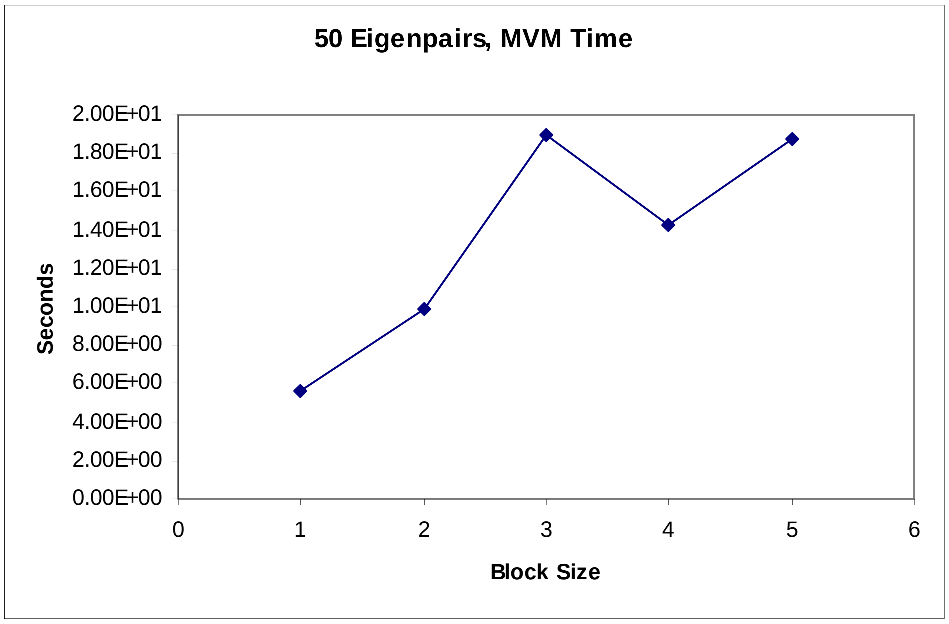
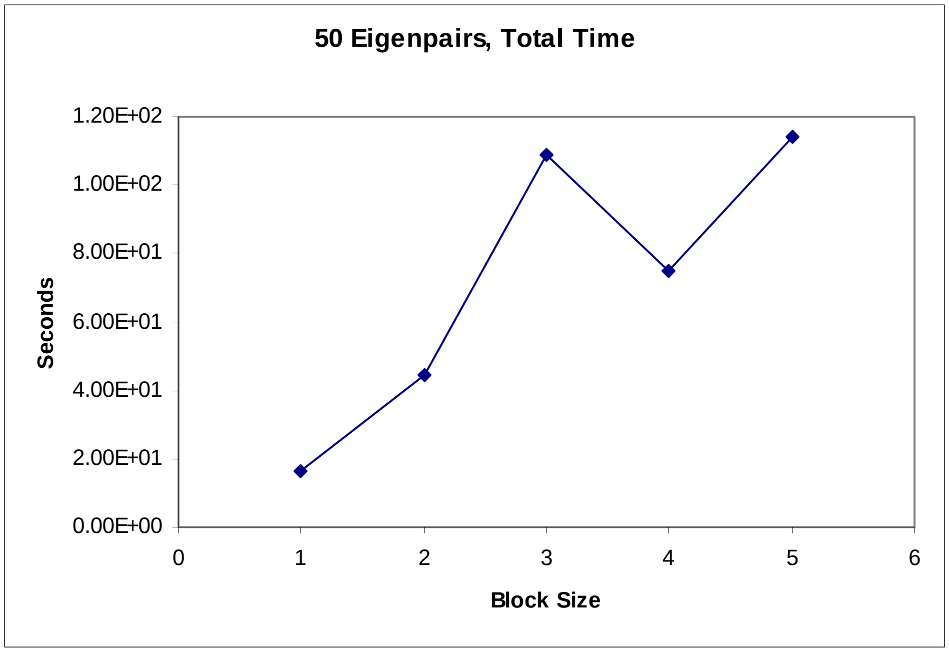


Figure 7.9 vibrobox – 50 required eigenpairs

# Vectors in Block	# MVM Ops	Time for MVM Ops	Time per MVM	Time for Reorth	Total Time	% MVM	% Reorth	% Other
1	336	5.64E+00	1.68E-02	7.68E+00	1.66E+01	34%	46%	20%
2	900	9.86E+00	1.10E-02	2.44E+01	4.45E+01	22%	55%	23%
3	2094	1.90E+01	9.07E-03	6.30E+01	1.09E+02	17%	58%	25%
4	1748	1.43E+01	8.18E-03	3.67E+01	7.47E+01	19%	49%	32%
5	2410	1.87E+01	7.76E-03	5.78E+01	1.14E+02	16%	51%	33%





8. Protein Matrices

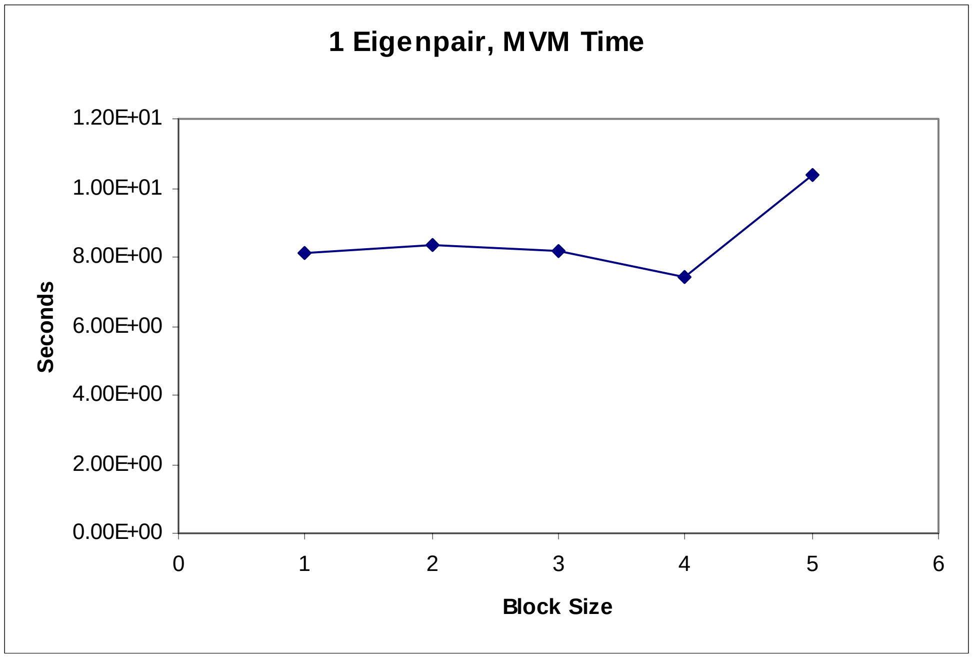
The tests run on the previous matrices were repeated on two more, both used in protein applications. The smaller one, pdb1FXK, has dimension 8070 and 1088905 nonzeros, a density of 1.67%. The larger of the two protein matrices, pdb1TUP, gave more interesting results. It has dimension 16323 and 2028247 nonzeros, a density of 7.61%. Blocking these matrices did not help, so the 1x1xv implementation was used for v =1,...,5, all with the symmetric version.

As indicated by Figures 8.1-8.3, for the FXK protein matrix, the single-vector procedure was optimal. From Figures 8.4 and 8.5, we can see that this is the case for the TUP matrix as well, when we require 1 or 10 eigenpairs. However, Figure 8.6 shows that when we require 50 eigenpairs, using a block size of 2, 3, 4, or 5 performs better than the single-vector procedure, with 2 being the optimal.

Note that for both matrices, the time spent performing A*U operations is around 80-90% of the total execution time for a block size of 1, and around 50-60% for higher block sizes. In contrast to previous matrices, the algorithm is always spending more than half the total time on A*U operations. In particular, reorthogonalization costs no longer dominate matrix-vector operation costs. In accordance with Osni Marques' suggestion mentioned earlier, we begin to see improved performance with increased block size when matrix-vector operations dominate the algorithm.

Figure 8.1 pdb1FXK – 1 required eigenpair

# Vectors in Block	# MVM Ops	Time for MVM Ops	Time per MVM	Time for Reorth	Total Time	% MVM	% Reorth	% Other
1	82	8.12E+00	9.90E-02	4.70E-01	8.98E+00	90%	5%	4%
2	144	8.37E+00	5.81E-02	1.61E+00	1.11E+01	75%	15%	10%
3	180	8.16E+00	4.53E-02	2.14E+00	1.22E+01	67%	19%	14%
4	220	7.42E+00	3.37E-02	1.70E+00	1.12E+01	66%	15%	19%
5	320	1.04E+01	3.25E-02	2.61E+00	1.61E+01	65%	16%	19%



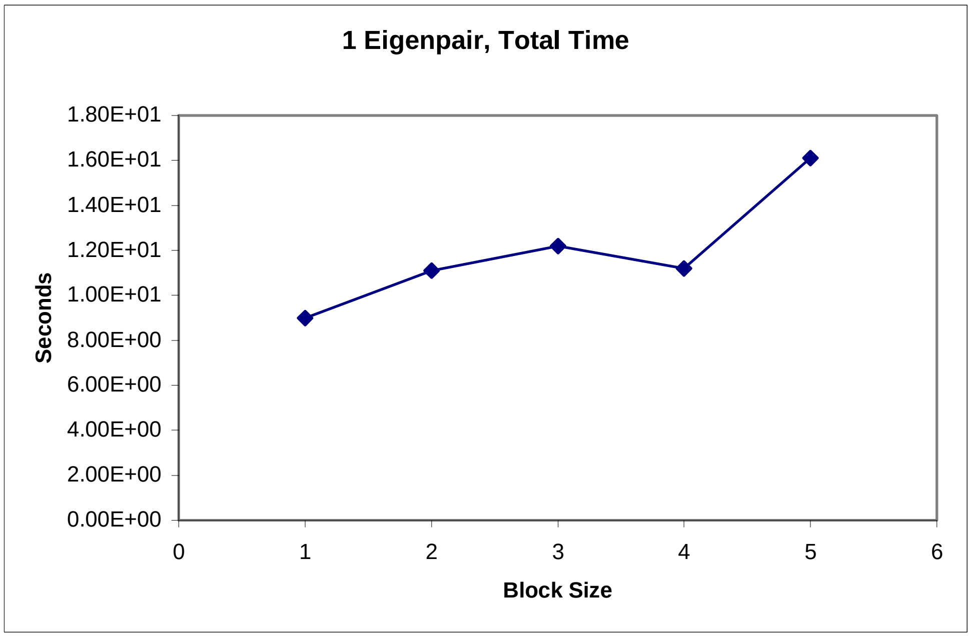
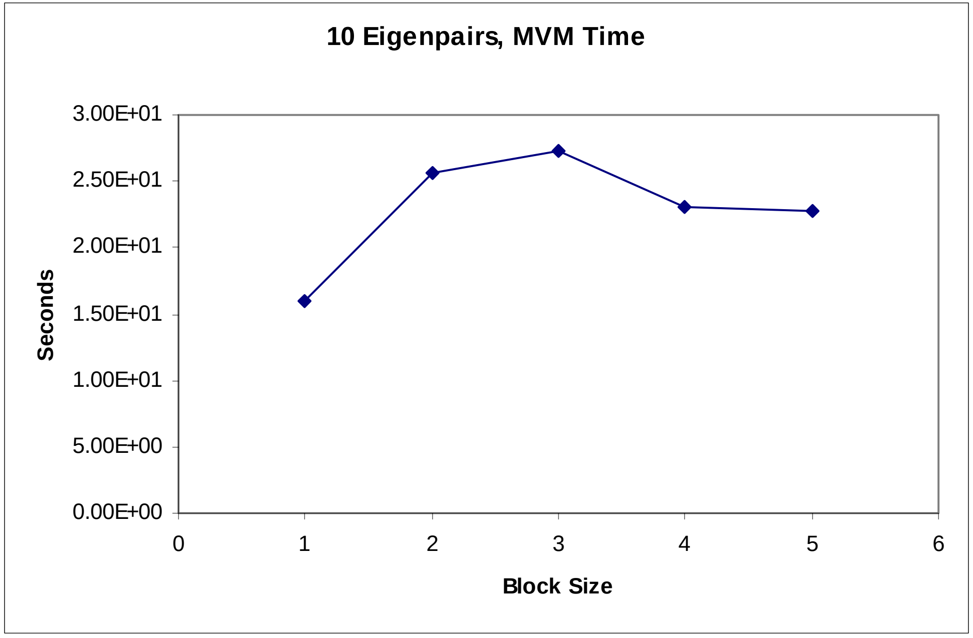


Figure 8.2 pdb1FXK – 10 required eigenpairs

# Vectors in Block	# MVM Ops	Time for MVM Ops	Time per MVM	Time for Reorth	Total Time	% MVM	% Reorth	% Other
1	162	1.60E+01	9.88E-02	1.81E+00	1.89E+01	85%	10%	6%
2	442	2.56E+01	5.79E-02	4.00E+00	3.47E+01	74%	16%	11%
3	609	2.73E+01	4.48E-02	8.92E+00	4.21E+01	65%	21%	14%
4	684	2.30E+01	3.36E-02	6.54E+00	3.61E+01	64%	18%	18%
5	675	2.27E+01	3.36E-02	6.90E+00	3.66E+01	62%	19%	19%



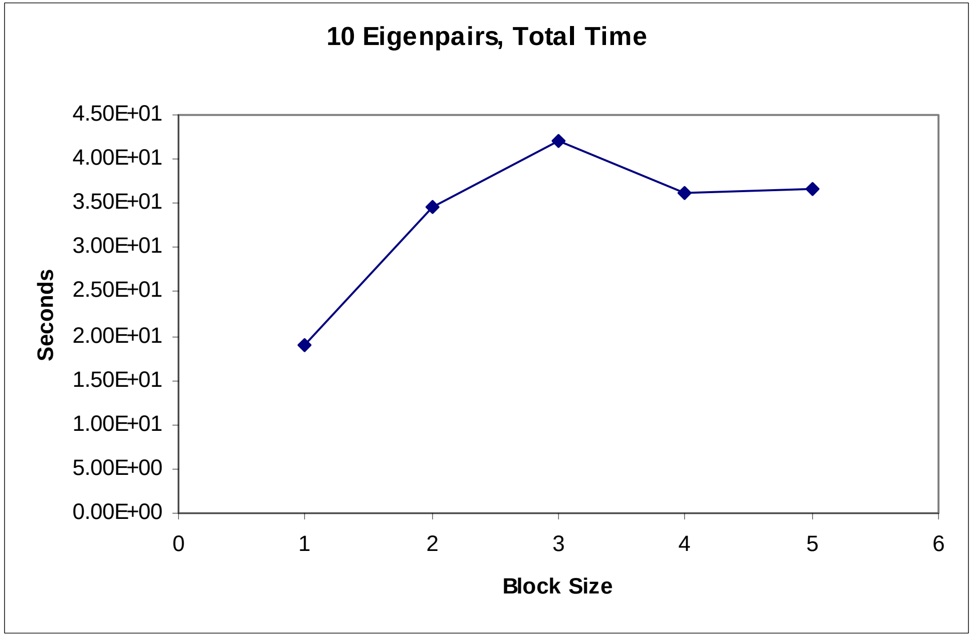
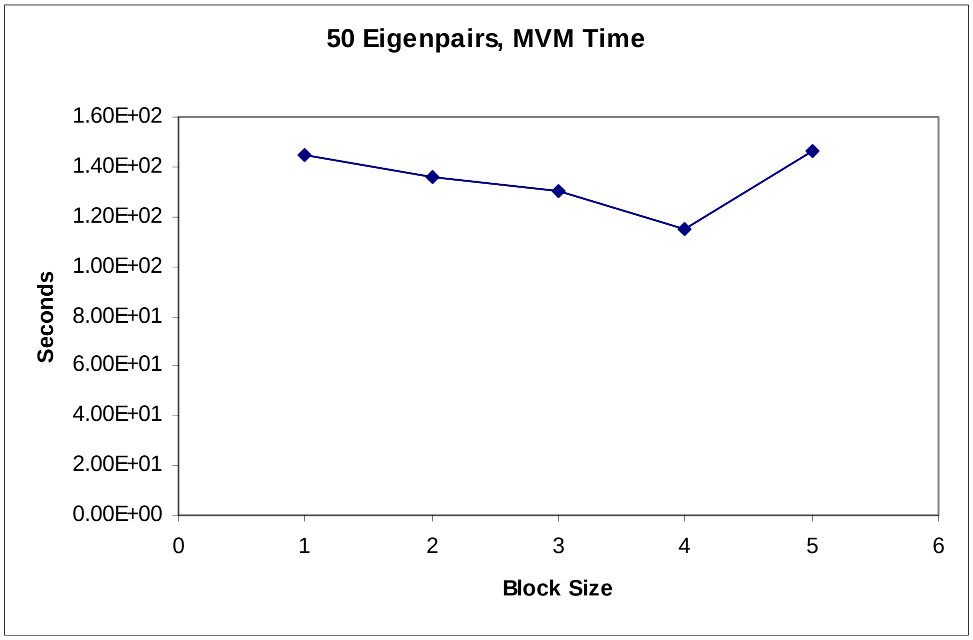


Figure 8.3 pdb1FXK – 50 required eigenpairs

# Vectors in Block	# MVM Ops	Time for MVM Ops	Time per MVM	Time for Reorth	Total Time	% MVM	% Reorth	% Other
1	1468	1.45E+02	9.88E-02	2.91E+01	1.85E+02	78%	16%	6%
2	2340	1.36E+02	5.81E-02	3.95E+01	1.96E+02	69%	20%	10%
3	2880	1.30E+02	4.51E-02	5.87E+01	2.18E+02	60%	27%	13%
4	3396	1.15E+02	3.39E-02	5.84E+01	2.07E+02	56%	28%	16%
5	4335	1.46E+02	3.37E-02	7.40E+01	2.67E+02	55%	28%	18%



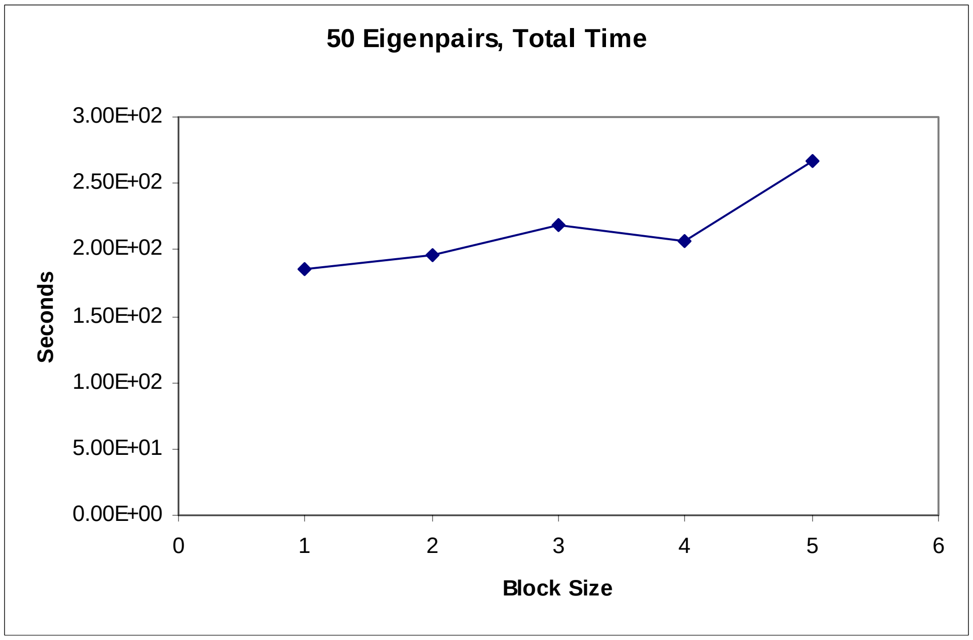
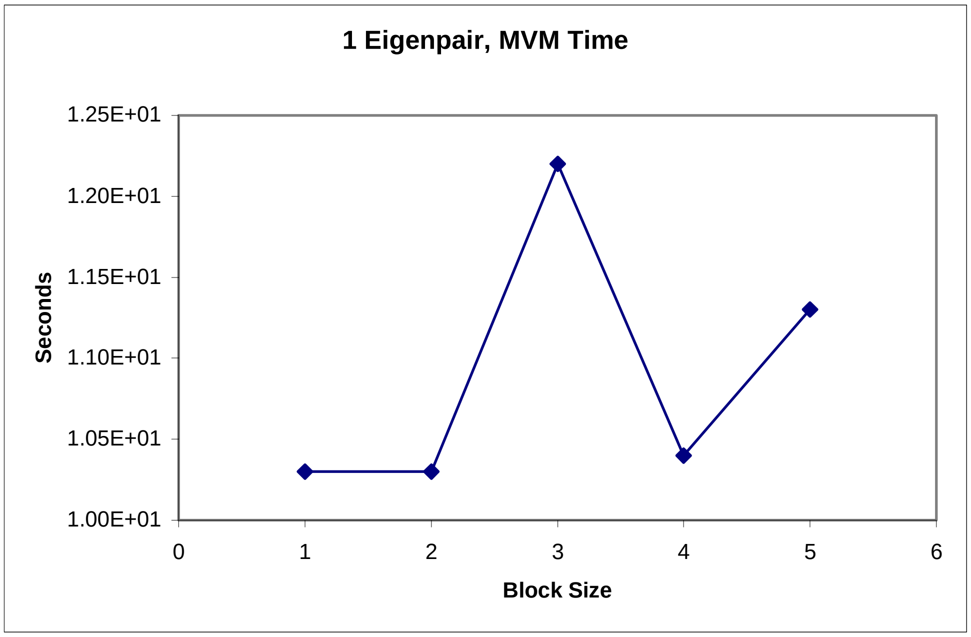


Figure 8.4 pdb1TUP – 1 required eigenpair

# Vectors in Block	# MVM Ops	Time for MVM Ops	Time per MVM	Time for Reorth	Total Time	% MVM	% Reorth	% Other
1	54	1.03E+01	1.91E-01	4.92E-01	1.14E+01	90%	4%	5%
2	88	1.03E+01	1.17E-01	1.55E+00	1.29E+01	80%	12%	8%
3	132	1.22E+01	9.24E-02	3.09E+00	1.73E+01	71%	18%	12%
4	156	1.04E+01	6.67E-02	2.97E+00	1.58E+01	66%	19%	15%
5	180	1.13E+01	6.28E-02	3.72E+00	1.83E+01	62%	20%	18%



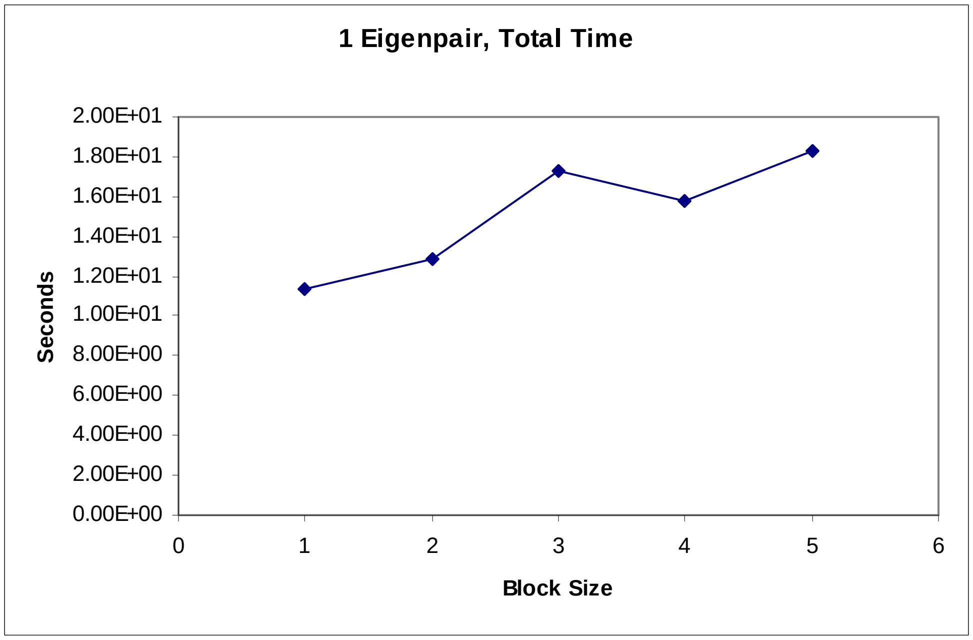
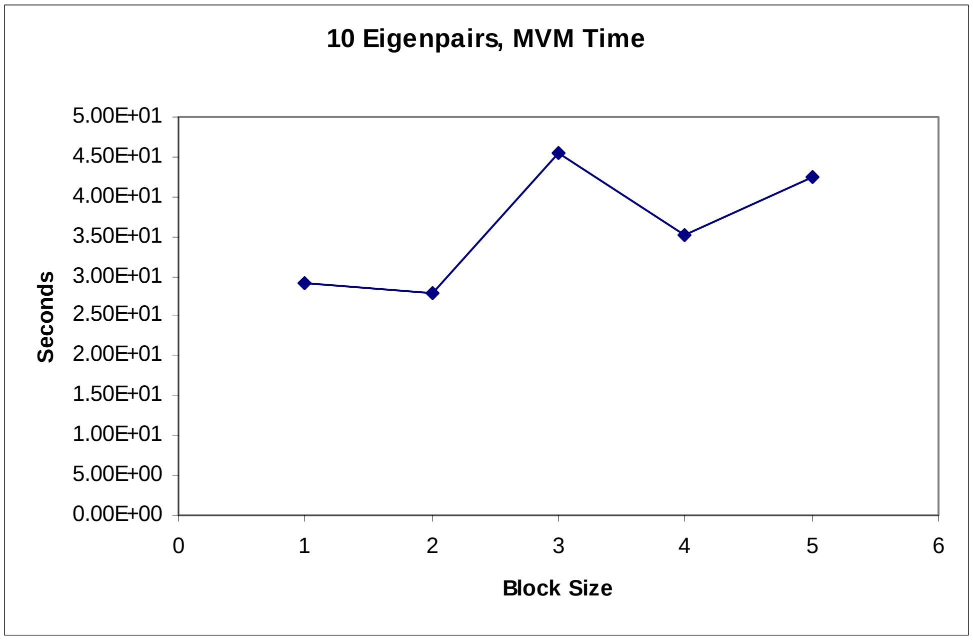


Figure 8.5 pdb1TUP – 10 required eigenpairs

# Vectors in Block	# MVM Ops	Time for MVM Ops	Time per MVM	Time for Reorth	Total Time	% MVM	% Reorth	% Other
1	152	2.91E+01	1.91E-01	3.74E+00	3.45E+01	84%	11%	5%
2	242	2.80E+01	1.16E-01	6.13E+00	3.76E+01	74%	16%	9%
3	486	4.56E+01	9.38E-02	1.49E+00	6.83E+01	67%	22%	11%
4	528	3.52E+01	6.67E-02	1.25E+00	5.66E+01	62%	22%	16%
5	670	4.24E+01	6.33E-02	1.66E+00	7.15E+01	59%	23%	17%



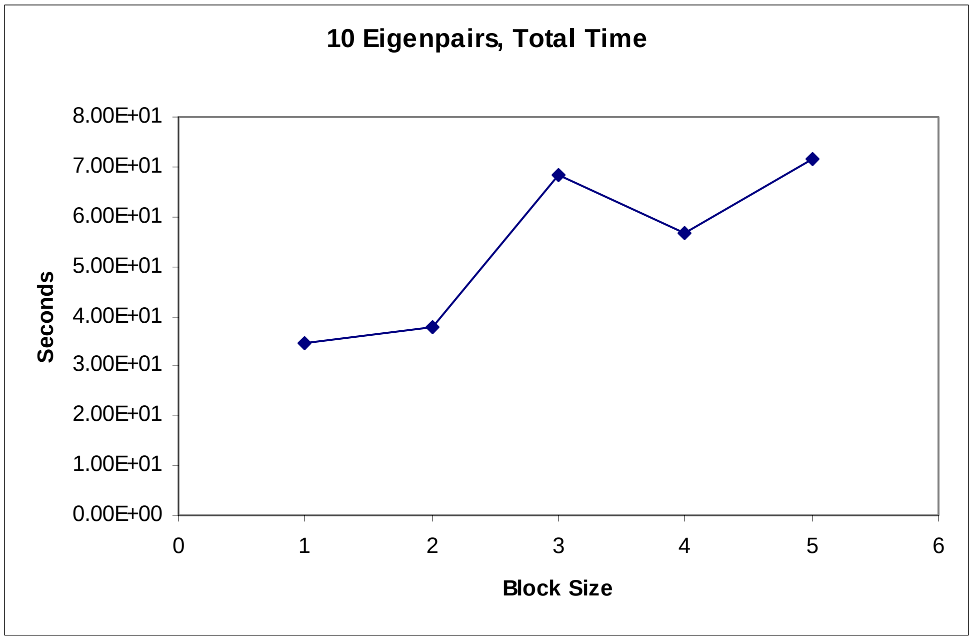
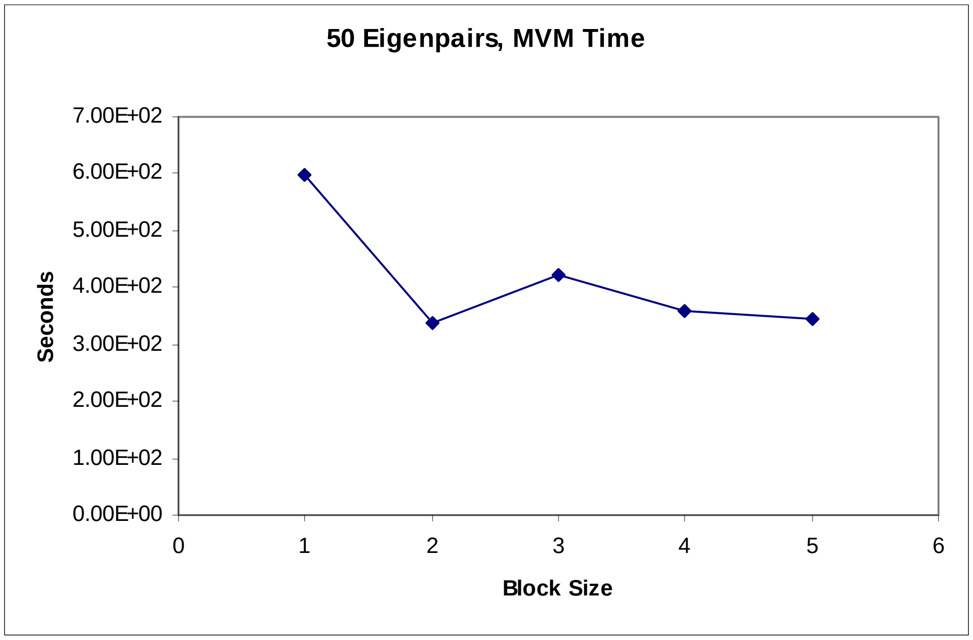
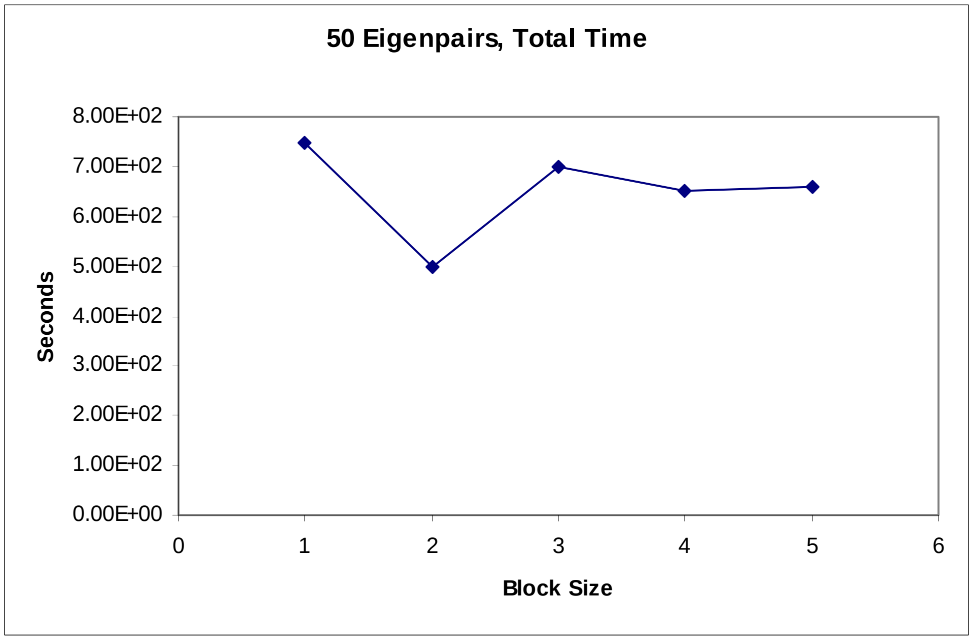


Figure 8.6 pdb1TUP – 50 required eigenpairs

# Vectors in Block	# MVM Ops	Time for MVM Ops	Time per MVM	Time for Reorth	Total Time	% MVM	% Reorth	% Other
1	3122	5.98E+02	1.92E-01	1.14E+02	7.47E+02	80%	15%	5%
2	2898	3.36E+02	1.16E-01	1.19E+02	4.97E+02	68%	24%	8%
3	4494	4.21E+02	9.37E-02	2.02E+02	6.98E+02	60%	29%	11%
4	5336	3.59E+02	6.73E-02	2.00E+02	6.50E+02	55%	31%	14%
5	5390	3.45E+02	6.40E-02	2.10E+02	6.59E+02	52%	32%	16%





9. Conclusion

Figure 9.1 Comparing all the matrices

Matrix	# Required Eigenpairs	% MVM*	% Reorth*	% Other*	Increase in Overall Time**
nasasrb	1	44.44%	37.36%	18.19%	50.69%
density:	10	42.22%	38.46%	19.32%	33.95%
0.09%	50	30.56%	50.94%	18.50%	28.15%
bcsstk	1	48.00%	12.23%	39.77%	70.77%
density:	10	44.09%	33.00%	22.91%	40.15%
0.16%	50	22.52%	63.16%	14.32%	18.80%
crystk	1	61.16%	17.71%	21.13%	123.24%
density:	10	53.18%	30.03%	16.79%	46.31%
0.29%	50	33.58%	52.99%	14.43%	158.71%
vibrobox	1	54.07%	22.10%	23.83%	128.40%
density:	10	43.29%	37.20%	19.51%	37.00%
0.23%	50	33.98%	46.27%	19.76%	168.07%
pdb1FXK	1	90.42%	5.23%	4.34%	23.61%
density:	10	84.66%	9.58%	5.77%	83.60%
1.67%	50	78.38%	15.73%	5.89%	5.95%
pdb1TUP	1	90.35%	4.32%	5.33%	13.16%
density:	10	84.35%	10.84%	4.81%	8.99%
7.61%	50	80.05%	15.26%	4.69%	-33.47%

* Percentage of total execution time spent in the indicated operation, for block size = 1.

** Increase in execution time when changing from block size 1 to the next best block size. In all cases it is 2, except for crystk, 1 required eigenpair, where it is 4. A negative amount indicates speedup.

Figure 9.1 puts some of the statistics of all the matrices together. No patterns seem immediately obvious. One thing to note is that the only speedup comes with the matrix that is most dense (i.e. pdb1TUP). Furthermore, the number of required eigenpairs has a significant effect on the difference in performance between block sizes 1 and 2. Finally, as noted before, the matrices in which the increases in overall time are generally the lowest are those whose time spent in matrix-vector-multiply operations is greatest relative to total time (although there is an exception for pdb1FXK, 10 required eigenpairs). However, not much can be said for those that spend less than half the total time on A*U operations; the behavior varies greatly between matrices.

One question that may warrant further exploration is the behavior of the algorithm with respect to total number of matrix-vector multiplies are required before termination. In the tables shown in the previous sections, I have indicated the number of such operations. When increasing the block size, this number changes unpredictably, even when restricted to a single matrix and varying the number of required eigenpairs. In fact, the number of matrix-vector multiplies can more than double (vibrobox, 50 required eigenpairs), or stay relatively the same (bcsstk, 50 required eigenpairs), or even decrease (pdb1TUP, 50 required eigenpairs). If there were a way to modify the criterion for termination such that the number of matrix-vector multiplies are relatively equal (so that there is not a significant increase in the number of operations needed), results of the algorithm running time may be different. Figure 9.2 shows the results of taking the number of matrix-vector multiplies in the single-vector procedure and estimating the running time of a blocked procedure (column "New Total Time") by multiplying the blocked algorithm's running time (column "Original Total Time") by the number of matrix-vector multiplies in the single-vector procedure (column "New # MVM") and dividing by the number of matrix-vector multiplies in the blocked procedure (column "Original MVM"). The block size I use is that which gave the best performance out of the sizes greater than 1. In all cases it is 2 except for crystk, 1 required eigenpair, where it is 4.

Figure 9.2 Performance with modified number of matrix-vector multiplies

Matrix	# Req EP	Original # MVM	New # MVM	Original Total Time	New Total Time	Old Single-Vector Time	Time Difference
nasasrb	1	110	84	2.17E+01	1.66E+01	1.44E+01	-2.17E+00
	10	110	90	2.17E+01	1.78E+01	1.62E+01	-1.55E+00
	50	170	151	4.78E+01	4.25E+01	3.73E+01	-5.16E+00
bcsstk	1	28	16	2.22E+00	1.27E+00	1.30E+00	3.14E-02
	10	58	45	5.69E+00	4.41E+00	4.06E+00	-3.55E-01
	50	172	150	3.16E+01	2.76E+01	2.66E+01	-9.58E-01
crystk	1	99	45	7.30E+00	3.32E+00	3.27E+00	-4.82E-02
	10	132	94	1.15E+01	8.19E+00	7.86E+00	-3.29E-01
	50	714	301	1.04E+02	4.38E+01	1.30E+00	-4.25E+01
vibrobox	1	108	53	3.70E+00	1.82E+00	1.62E+00	-1.96E-01
	10	156	123	6.74E+00	5.31E+00	4.92E+00	-3.94E-01
	50	900	336	4.45E+01	1.66E+01	1.66E+01	-1.33E-02
pdb1FXK	1	144	82	1.11E+01	6.32E+00	8.98E+00	2.66E+00
	10	442	162	3.47E+01	1.27E+01	1.89E+01	6.18E+00
	50	2340	1468	1.96E+02	1.23E+02	1.85E+02	6.20E+01
pdb1TUP	1	88	54	1.29E+01	7.92E+00	1.14E+01	3.48E+00
	10	242	152	3.76E+01	2.36E+01	3.45E+01	1.09E+01
	50	2898	*				

* No entry for this row because the single-vector procedure used more matrix-vector multiplies than with block size 2.

For the first four matrices there is a negative time difference (i.e. the blocked algorithm is still slower than the unblocked algorithm), with one exception. However, for bcsstk, 1 required eigenpair, and for the two protein matrices, the difference is positive. That is, if we were to perform the same number of matrix-vector multiplies in the blocked version as in the single-vector version of the algorithm on those matrices, we would see a speedup in the overall running time. The time saved by using a more efficient (unrolled) matrix-vector-multiply routine is apparently enough to outweigh the increase in other costs in those examples.

The behavior of the block Lanczos method is not well understood. We have yet to determine why some particular problems require such varying numbers of matrix-vector multiplies when changing the block size. Aside from the possible optimization by modifying the stopping criterion as just discussed, the results of this project seem to agree with the concluding suggestions of *Lanczos Algorithms* [4], which state that single-vector Lanczos procedures are cheaper, and generally recommended over, block Lanczos procedures.

10. Bibliography

[1] Bai, Z., and Day, D. (2000). *Block Lanczos Methods*. Chapter 7 in [3]. http://www.cs.utk.edu/~dongarra/etemplates/node250.html SIAM, Philadelphia.

[2] Baker, A., Dennis, J., and Jessup, E. (2002). *Toward Memory-Efficient Linear Solvers*. http://amath.colorado.edu/student/allisonb/cse03_slides.pdf Springer, Porto.

[3] Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., and van der Vorst, H. (2000). *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. http://www.cs.utk.edu/~dongarra/etemplates/index.html SIAM, Philadelphia.

[4] Cullum, J., and Willoughby, R. (2002). *Lanczos Algorithms for Large Symmetric Eigenvalue Computations Vol.1: Theory*. SIAM, Philadelphia.

[5] Im, E. (2000). Optimizing the Performance of Sparse Matrix-Vector Multiplication. http://www.cs.berkeley.edu/~ejim/publication/thesis-0.ps *Ph.D. Thesis*, UC Berkeley.

[6] Im, E. (2002). Sparse Matrices. http://www.cs.berkeley.edu/~ejim/matrices/

[7] Im, E. and Yelick, K. (2000) SPARSITY software. http://www.cs.berkeley.edu/~ejim/sparsity/

[8] Marques, O. (2000). BLZPACK. http://www.nersc.gov/~osni/CODES/blzpack.shar

[9] Marques, O. (1999). BLZPACK User's Guide. http://www.nersc.gov/~osni/CODES/blzpack.shar

[10] Ruhe, A. (2000) *Lanczos Method*. Chapter 5 in [3]. http://www.cs.utk.edu/~dongarra/etemplates/node103.html SIAM, Philadelphia.