疎行列アルゴリズムのための共有メモリアーキテクチャ上での BLAS の並列化手法と性能評価

西田 晃[†] 小柳義夫[†]

疎行列アルゴリズムの共有メモリアーキテクチャへの実装において, BLAS レベルでの並列化を行うことにより, 行列 - ベクトル間演算レベルでの並列性を容易に記述することができる. 本稿では, Level 1, 2 BLAS サブルーチン並列化の利点及び問題点について, SMP 上での実装評価をもとに考察する.

Parallel Implementation of the BLAS Library on Shared Memory Architectures for Sparse Matrix Algorithms and its Evaluation

Akira Nishida[†] and Yoshio Oyanagi[†]

Parallel implementation of the BLAS library for sparse matrix algorithms in computational linear algebra is a critical problem, especially on the shared memory architectures with low data access latency. In this paper, we discuss the advantages and disadvantages of the parallelizing methodology of Level 1 and 2 BLAS subroutines using pthreads library, and report its implementation and performance evaluation on shared memory architectures.

1. はじめに

大規模疎行列の反復解法においては, 疎な成分を持つ行列とベクトル間の演算が計算量の大部分を占めるため, この演算の効率的な処理が必要不可欠である. これらは Dongarra らによって開発された BLAS (Basic Linear Algebra Subprograms)[8] を用いて実装することが可能であるが, 行列 - ベクトル間演算においては, 一般に演算量は $\mathcal{O}(\backslash^{\in})$ であり, 十分にキャッシュメモリを活用するのは難しい. したがって, 疎行列アルゴリズムの並列化を行う場合に, BLAS レベルでの並列化が最適な選択肢であるかどうかを決定するためには, 十分な評価が必要である.

BLAS の並列実装に関しては, 1) ScaLAPACK[3] のように, MPI による BLAS と等価な並列化ライブラリを構築するもの, と, 2) ATLAS[10),11)] のように, pthreads により並列化を行うものがあり, 1), 2) とも一部の BLAS ルーチンについては, 効率的な並列実装が実現している. ただし, PBLAS は分散メモリアーキテクチャを主な対象としており, また, ATLAS の Level 3 BLAS ルーチンに関する対称型マルチプロセッサ向けの並列化は, 本アルゴリズムのような疎行列解法への適用を想定したものではない.

しかしながら, メモリアクセス遅延の小さい共有メモリアーキテクチャ上においては, ループレベルでの並列化によって, Level 1, 2 BLAS 演算についても効率的に性能向上を達成できるものと期待される. 本稿では, 大規模疎行列の固有値解法に関する BLAS レベルでの並列化について, 評価の結果を報告するとともに, 疎行列用ライブラリの構築の際の問題点を考察する.

2. 背景

BLAS の並列化については, Whaley ら[3] により BLACS (Basic Linear Algebra Communication Subprograms) を通信ライブラリとする並列化 BLAS (PBLAS, Parallel BLAS)[4] が開発されており, 通信は MPI 及び PVM, 及びベンダ提供のライブラリにより記述されている.

また, LINPACK に関しては, MPI により記述された最適化支援ライブラリである HPL (High Performance LINPACK) が開発されており, アーキテクチャに応じてパラメータを調節することにより, 手動での最適化が可能である.

上記のライブラリは主に分散メモリ型並列計算機もしくはクラスタを対象としたものであるが, メモリ階層を利用したプロセッサレベルでの最適化の自動化を目的とするソフトウェアである ATLAS (Automatically Tuned Linear Algebra Software) においては, 一部のルーチンで対称型マルチプロセッサに対応した最適化が

† 東京大学大学院理学系研究科情報科学専攻
Division of Information Science, School of Science, the University of Tokyo

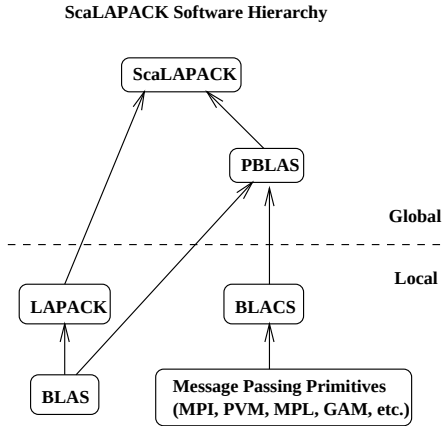

図 1 ScaLAPACK のソフトウェア階層
Fig. 1 ScaLAPACK software hierarchy.

行われるようになっている.

3. PBLAS・BLACS における並列化手法

BLACS は, ScaLAPACK 内での通信操作を記述するために設計された 2 次元配列用の通信ライブラリで, 1 次元ないし 2 次元の格子状に割り当てられたプロセス (Process Grid) が, 行列やベクトルのデータの一部を保持する. プロセス間のデータの送受信操作は PVM, MPI 等のメッセージ通信ライブラリで記述されている. PBLAS は BLACS を用いて BLAS を Process Grid 上での並列化を行うためのライブラリで, Level 1-3 の主要な BLAS ルーチンに対応している. BLACS によるオーバーヘッドは小さく, 十分なサイズの問題に対しては, Level 2, 3 PBLAS ルーチンは高い並列化効率を実現している[3]. 図 1に概念図を示す.

4. ATLAS における最適化化手法

ATLAS では, パラメータ探索の自動化により, レジスタレベルでは浮動小数演算ユニットの利用及びループ最適化を, また, キャッシュレベルではメモリブロッキングによる最適化を実現している.

まず, 行列積におけるキャッシュレベルでの最適化は, 各メモリ階層について以下のように行われる. 命令キャッシュの再利用, ループオーバーヘッドの削減, 命令レベル並列性の抽出に関しては, アンローリングを用いたループ最適化により実現する. L1 キャッシュレベルでは, 行列要素をキャッシュにコピーできる場合 (copy 版) とそうでない場合 (non-copy 版) についてコンパイル時に実行時間を比較し, 高速なものを選択する.

L2 レベルにおいては, サイズの小さな行列を優先してループの内側に割り当てることにより, キャッシュ上で実行可能な演算数を最大化する. 実際には, 可能な場合には実測値から最適なデータサイズの上限値 (CacheEdge) を計算し, 判定に用いる.

Level 3 BLAS の各ルーチンは, 再帰的なブロック分割により, GEMM (general matrix-matrix multiplication) と呼ばれる行列積演算に帰着することができる. このため, 上記の方法で得た最適サイズの行列積をブロック単位として構成することができ, また, 再帰を利用することにより, 最適なブロックサイズでの並列化も可能である. ATLAS 3.2.0 では, Level 3 BLAS についてこの方法での pthreads による並列化を実装している.

Level 2 BLAS についても上記と同様に考えることができ, ATLAS では, L1 サイズの行列 - ベクトル間演算 (L1 matvec) 及び L1 キャッシュの再利用を確実にするためのキャッシュブロッキング (L1 update1) を組み合わせている.

5. 問　　題

本研究では, 疎行列固有値解法の一種である Jacobi-Davidson 法の並列化について, BLAS レベルでの並列化手法を評価する. Jacobi-Davidson 法は, 量子化学において対角化に用いられることの多い Davidson 法をもとに構成された解法であるが, 比較的条件の悪い問題でも正確な固有値を計算することが可能である[6),9)].

5.1 Jacobi-Davidson 法

Jacobi-Davidson 法は, 従来 Davidson 法[5] として提案された手法に, Jacobi 法[7] の考え方を用いて改良を加えたものである.

Davidson 法では, 以下のような手続きで絶対値最大の固有値☆ を求める. 次元 k の部分空間 $\mathcal{K} = \mathrm{span} v_1, ..., v_k$ 上で, 行列 A の近似固有対, すなわち Ritz 対 (θ_k, u_k) を考える. ここで $v_1, ..., v_k$ は正規直交基底とする. u_k を更新するためには $\mathcal{K}$ の次元を拡張する必要があるが, Davidson 法では残差 $r = Au_k - \theta_k u_k$ について修正方程式と呼ばれる以下のような方程式を解く.

$$M_k t = r, \quad M_k = D_A - \theta_k I \tag{1}$$

D_A は A の対角成分である. さらに t を $\mathcal{K}$ と直交化して v_{k+1} を得る. $V_{k+1} = [v_1, ..., v_{k+1}]$ と置けば, 新しい Ritz 対 (θ_{k+1}, u_{k+1}) は行列

$$H_{k+1} = V_{k+1}^* A V_{k+1} \tag{2}$$

の固有対として計算される.

Jacobi-Davidson 法では, u_k の直交補空間から更新のための成分を取り出す. 以下では u_k は正規化されているものと仮定する.

固有値問題 $Ax = \lambda x$ を, 以下のように u_k の直交補空間 $u_k^\perp$ 上に射影する. 行列 A の $u_k^\perp$ への直交射影は

$$A_P = (I - u_k u_k^*) A (I - u_k u_k^*) \tag{3}$$

で表されるが, これは

$$A = A_P + u_k u_k^* A + A u_k u_k^* - \theta_k u_k u_k^* \tag{4}$$

と書き直すことができる. 修正ベクトル z は

$$A(z + u_k) = \lambda(z + u_k), \quad z \perp u_k \tag{5}$$

☆ 以下単に最大固有値と書く.

input a starting vector v and a tolerance ϵ;
compute $u_1 = v_1 = v/ \parallel v \parallel_2$;
$w_1 = Av_1,\ \theta = h_{1,1} = w_1^* v_1,\ r = w_1 - \theta v_1$;
for $k = 2, ...$
  solve approximately a $z \perp u$ from
    $(I - uu^*)(A - \theta I)(I - uu^*)z = -r$;
  for $j = 1, ..., k-1$
    $z = z - (z^* v_j)v_j$;
  $v_k = z/ \parallel z \parallel_2,\ w_k = Av_k$;
  for $j = 1, ..., k$
    $h_{j,k} = w_k^* v_j$;
  compute the largest eigenpair (θ, y)
  of the matrix H_k with $\parallel y \parallel = 1$;
  compute the Ritz vector $u = Vy$
  and $\tilde{u} = Au = Wy$;
  $r = \tilde{u} - \theta u$;
  stop if $\parallel r \parallel_2 \leq \epsilon$;

図 2 JD 法による最大固有値の計算
Fig. 2 Computation of largest eigenvalue by Jacobi-Davidson.

を満たすので, ここに (4) を代入すれば

$$(A_P - \lambda I)z = -r + (\lambda - \theta_k - u_k^* Az)u_k \quad (6)$$

となる. $A_P z \perp u_k$, $z \perp u_k$, $r \perp u_k$ より u_k の係数は 0 でなければならないので, 問題は

$$(A_P - \lambda I)z = -r \quad (7)$$

の計算に帰着されることが分かる. 実際には λ の値を知ることはできないが, (7) は厳密に解く必要がないため, ここでは代わりに θ_k を用いて

$$(I - u_k u_k^*)(A - \theta_k I)(I - u_k u_k^*)z = -r \quad (8)$$

を解く. 得られたベクトルを V_k に対して直交化し, v_{k+1} とする. $H_{k+1} = V_{k+1}^* A V_{k+1}$ の最大固有値が次ステップの Ritz 値 θ_{k+1} となる. 具体的なアルゴリズムを図 2 に示す. 同様の要領で, 減次を用いて複数の固有値を求めることができる.

6. 実装と性能評価

以下では, 上記アルゴリズムの実装, 及び共有メモリアーキテクチャ上での並列化手法について検討する. プログラムに関しては, Fokkema, van Gijzen[6] らによる JDQZ ルーチンをベースとした. 反復解法の記述には Templates[2], また線形演算には BLAS[8], LAPACK[1] を用いている.

評価には, Intel Pentium III Xeon (550MHz, 16KB data, 16KB instruction cache, 512KB L2 cache) の 4-way SMP (Dell PowerEdge 6300, 450NX chipset, 768MB main memory) 及び Sun UltraSPARC II (250MHz, 16KB data, 16KB instruction cache, 1MB L2 cache) の 64-way SMP (Ultra Enterprise 10000) を用いた. OS にはいずれも Solaris 7, またコンパイラにはそれぞれ PGI Fortran 及び Omni OpenMP Compiler を用いた.

評価は, 固有値が既知である以下の実対称行列を用いて行う.

$n = N^2$ 次 5 重対角行列

$$A = \begin{pmatrix} T_N & -I & & O \\ -I & \ddots & \ddots & \\ & \ddots & \ddots & -I \\ O & & -I & T_N \end{pmatrix}, \quad (9)$$

$$T_N = \begin{pmatrix} 4 & -1 & & O \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ O & & -1 & 4 \end{pmatrix} \quad (10)$$

の固有値は, 解析的に $4 - 2(\cos(k\pi/(N+1)) + \cos(j\pi/(N+1))$, $j, k = 1, ..., N$ で与えられる.

ここでは, 残差の許容範囲を 10^{-8} として, A の最大固有値を計算した. 探索空間の基底数は 10–15 の範囲とし, 標準 Petrov 空間上で反復ベクトルを生成する. 修正方程式の計算には BiCGSTAB(4) を用いた.

4-way SMP 上での問題サイズと計算時間の関係は表 1の通りである. なお, A の条件数は

$$\frac{8}{4 - 4\cos\left(\frac{\pi}{N+1}\right)} \approx \frac{4}{\pi^2} N^2 \quad (11)$$

で見積もることができる.

表 1 Jacobi-Davidson 法による最大固有値の計算時間
Table 1 Computation time for the largest eigenvalue.

Size	Time(s)
32^2	1
64^2	12
128^2	60
256^2	396

256^2 次の場合について, 逐次で最大固有値を 5 個まで求めた場合の各ルーチンの実行時間の内訳を表 2に示す.

この結果から, Jacobi-Davidson 法においては修正方程式の計算の高速化が不可欠であることが分かるが, 個々の関数についてみると, 計算時間の大部分を `zgemv`, `zdotc`, `zaxpy` などの Level 1, 2 BLAS ルーチンが占めていることが分かる. なお, `zgemv` は

```
y := alpha*A*x + beta*y,
y := alpha*A'*x + beta*y,
y := alpha*conjg( A' )*x + beta*y,
```

型の Level 2 BLAS, また, `zdotc` は

```
z := conjg(x)*y,
```

`zaxpy`, `zxpay` はそれぞれ

```
y := alpha*x + y,
y := x + alpha*y
```

表 2　$n = 256^2$ での実行結果
Table 2　Result for $n = 256^2$.

Function	Calls	Time(s)	Time(%)
zgemv	10246	319	33.40
zdotc	12615	150	15.77
zaxpy	8163	116	12.14
jdqz	1	109	11.50
jdqzmv	3479	54	5.74
zxpay	4193	54	5.68
dznrm2	5402	52	5.50
amul	3540	47	5.00
bmul	3540	31	3.26
zcgstabl	41	12	1.33
zmgs	369	3	0.32
...			

型の Level 1 BLAS 演算である.

このように, Jacobi-Davidson 法では Level 1, Level 2 BLAS 演算, 特に zgemv 型の演算が計算量の大部分を占める. ただし, 疎行列を対象としているため, 行列 - ベクトル積の演算量はほぼ $\mathcal{O}(n)$ である.

OpenMP Fortran API の実行モデルでは, プログラムの実行はマスタスレッドと呼ばれる単一プロセスとして開始される. マスタスレッドは通常のステートメントを逐次実行し, PARALLEL と END PARALLEL 指示文の対で構成される並列構造が現れると, 1 つ以上のスレッドからなるチームを生成し, チームのメンバのそれぞれについてデータ環境の設定を行なう. 並列構造内のステートメントは, チーム内の各スレッドによって並列に実行され, 並列構造の終了時点でチーム内のスレッドは同期し, マスタースレッドは更新されたデータを用いて計算を続ける.

共有メモリアーキテクチャはメモリアクセス遅延が小さい反面, 分散メモリアーキテクチャと比較してキャッシュミスの発生が多く, また, 同一のキャッシュラインへの書き込みが発生した場合には著しく性能が低下する場合がある. しかしながら, この点についても留意して実装を行うことにより, 高い性能を確保することが可能である.

並列化に当たっては, まず, zgemv, zdotc, zaxpy, zxpay の 4 関数の最内側ループを OpenMP API を用いてブロック化したルーチンを作成し, 並列に処理した. それぞれの演算のループは静的に各スレッドへ割り当てた. また, ループ内での変数の私有化によりメモリアクセスの競合を最小限に抑えるとともに, 可能な限り陰的なバリア同期を行なわないよう指定する.

なお, BLAS, LAPACK については, PGI Fortran では PGI の最適化ライブラリ, Omni Compiler では Dongarra ら[1),8)] の reference implementation を使用し, 必要な関数のみを並列化して置き換えた.

まず, 4-way SMP 上での並列化後の実行結果を表 3に示す. 並列化により, 各ルーチンについてそれぞれスピードアップが得られていることがわかる. 実行時間の

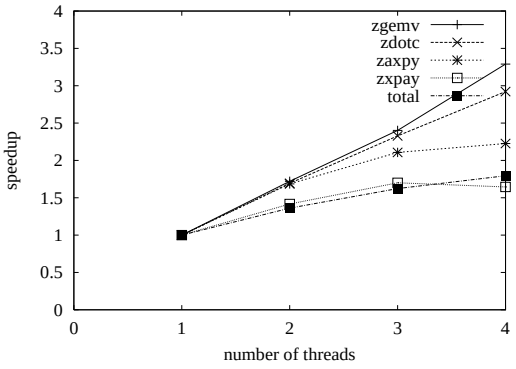

図 3　$n = 256^2$ での各ルーチンの速度向上率
Fig. 3　Speedups of subroutines for $n = 256^2$.

大きな部分を占めている Level 2 BLAS の zgemv についても, 計算のオーダが $\mathcal{O}(n)$ であるため, 上記の並列化により十分高い速度向上率が得られている.

表 3　$n = 256^2$ での並列実行結果
Table 3　Result for $n = 256^2$ in parallel execution.

Function	Calls	Time(s)	Time(%)
jdqz	1	113	19.95
zgemv	9814	96	17.12
zaxpy	7818	54	9.69
zxpay	3954	51	9.12
zdotc	12212	50	8.93
jdqzmv	3290	50	8.85
dznrm2	5296	45	8.09
amul	3352	42	7.56
bmul	3352	42	7.47
zcgstabl	42	11	2.02
zmgs	376	3	0.56
...			

また, スレッド数を変化させた時のスピードアップを図 3, 次元を変えて実行時間をみたものを図 4 に示す. 次元が大きくなるにしたがって速度向上率がスレッド数に近づくが, 複素乗算を含む zgemv, zdotc 型の演算で, より高い効果が得られていることが分かる.

次に, Sun Enterprise 10000 上での実行結果について述べる.

Omni では, Solaris プラットフォーム上でスレッドをプロセッサに固定することが可能である. 今回の評価ではキャッシュの利用が前提となるため, プロセッサに対して LWP を固定している.

Level 2 BLAS である zgemv において, BLAS の reference implementation では, 以下のように行列 - ベクトル積を

$$y(1:m) = y(1:m) + M(1:m, j)x(j), \quad (12)$$
$$j = 1, 2, ..., n \quad (13)$$

として計算している.

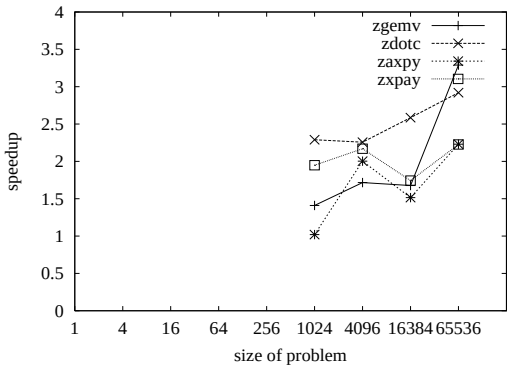

図 4　問題サイズと各ルーチンの速度向上率
Fig. 4　Size of problem and speedup.

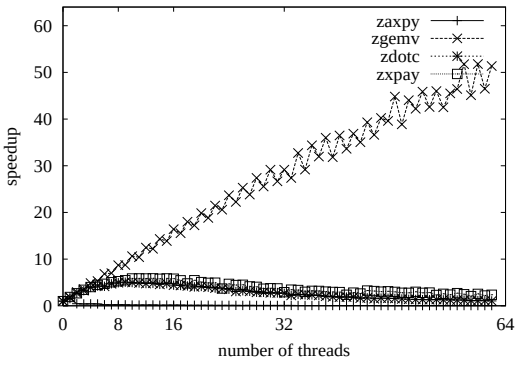

図 5　$n = 64^2$ での各ルーチンの速度向上率
Fig. 5　Speedups of subroutines for $n = 64^2$.

```
*
* Form  y := alpha*A*x + y.
*
      DO 60, J = 1, N
         IF( X( JX ).NE.ZERO )THEN
            TEMP = ALPHA*X( JX )
            DO 50, I = 1, M
               Y( I )
     $          = Y( I ) + TEMP*A( I, J )
   50       CONTINUE
         END IF
         JX = JX + INCX
   60 CONTINUE
```

この演算はレジスタレベルでの最適化を目的としたものであるが, 一般的な Level 2 BLAS 演算においては, ベクトル y と行列 M の行とを

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{pmatrix} + \begin{pmatrix} M_1 \\ M_2 \\ \vdots \\ M_k \end{pmatrix} \times x \quad (14)$$

としてブロック化し, 以下のように各ブロックでのベクトル演算を並列に処理する必要がある.

```
*
* Form y := alpha*A*x + y,
*
!$OMP PARALLEL PRIVATE(JX, TEMP, TEMP_SUM)
!$OMP+ SHARED(KX, INCX, ALPHA)
!$OMP DO
      DO 50, I = 1, M
         TEMP_SUM = (0, 0)
         DO 60, J = 1, N
            JX = KX + J * INCX - INCX
            IF( X( JX ).NE.ZERO )THEN
               TEMP = ALPHA*X( JX )
               TEMP_SUM
     $          = TEMP_SUM + TEMP*A( I, J )
            END IF
   60    CONTINUE
         Y( I ) = Y( I ) + TEMP_SUM
   50 CONTINUE
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

図 5に, スレッド数を変化させた時の関数 zgemv, zdotc, zaxpy, zxpay のスピードアップを示す.

Level 2 BLAS に関してはほぼ線形な性能向上が得られているのに対し, Level 1 については, 一定のプロセッサ数以上で十分な向上率が得られないことが分かる. なお, サイズを変化させてもこの特性はほとんど変化しない. 以上から, Level 2 BLAS での並列化は有効性が高いものの, Level 1 の並列化に関しては, 演算の性質による制約が大きく, 大きなスピードアップは得られないことが分かる.

7. ま と め

本稿では, 並列 BLAS を用いた疎行列アルゴリズムの共有メモリアーキテクチャ上への実装方式について検討を行った.

疎行列アルゴリズムにおいて必要となる Level 1, 2 BLAS 演算について, SMP 上への効率的な並列実装は十分可能であると考えられるが, プロセッサレベルでの最適化☆と合わせて, 今後実験・評価を通じて効率的な実装方法を明らかにしていく必要があると思われる.

☆ Level 1, 2 BLAS において, ATLAS による最適化を同時に行う場合, 上記の並列化手法を前提とすると, 静的にパラメータを決定することができない. 一方, ブロック単位での並列化は動的な負荷分散が容易であるが, オーバーヘッドが大きくなる可能性があるため, 注意が必要である.

参 考 文 献

1) E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, third ed., 1999.

2) R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, Society for Industrial and Applied Mathematics, 1994.

3) 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, *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, 1997.

4) J. Choi, J. Dongarra, and D. Walker, *ParallelMatrix Transpose Algorithms on Distributed Concurrent Computers*, Tech. Rep. UT CS-93-215, LAPACK Working Note #65, University of Tennessee, 1993.

5) E.R. Davidson, *The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real symmetric matrices*, J. Comp. Phys., 17 (1975), pp. 87–94.

6) D. R. Fokkema, G. L. G. Sleijpen, and H. A. van der Vorst, *Jacobi-Davidson style QR and QZ algorithms for the partial reduction of matrix pencils*, Tech. Rep. 941, Department of Mathematics, Utrecht University, 1996.

7) C. G. J. Jacobi, *Ueber ein leichtes Verfahren, die in der Theorie der Säcularstörungen vorkommenden Gleichungen numerisch aufzulösen*, Journal für die reine und angewandte Mathematik, (1846), pp. 51–94.

8) L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, *Basic Linear Algebra Subprograms for FORTRAN usage*, tech. rep.

9) G.L.G. Sleijpen and H.A. van der Vorst, *A Jacobi-Davidson iteration method for linear eigenvalue problems*, SIAM J. Matrix Anal. Appl., 17 (1996), pp. 401–425.

10) B. C. Whaley and J. Dongarra, *Automatically Tuned Linear Algebra Software*, in Proceedings of SC98, 1998.

11) B. C. Whaley, A. Petitet, and J. Dongarra, *Automated Empirical Optimization of Software and the ATLAS Project*, Tech. Rep. UT CS-00-451, LAPACK Working Note #147, University of Tennessee, 2000.