Jacobi-Davidson 法の並列前処理とその性能評価

西田 晃[†] 小柳義夫[†]

大規模疎行列向きの固有値解法である Jacobi-Davidson 法は, 従来の Lanczos/Arnoldi 系の解法に比べて高精度な計算が可能であり, 様々な研究がなされている. 本稿では, Jacobi-Davidson 法において計算量の大部分を占める修正方程式の計算に対して, 共有メモリ型並列計算機向きの解法である Jacobi 前処理を適用, 実装し, その評価結果について報告する. Jacobi 前処理を用いることにより, Jacobi-Davidson 法の計算時間に関して約 2 倍の性能向上が得られることが分かった.

Parallel Preconditioning of the Jacobi-Davidson Method and its Evaluation

Akira Nishida[†] and Yoshio Oyanagi[†]

The Jacobi-Davidson method is a promising alternative to the Lanczos/Arnoldi approach. In fact, the Lanczos/Arnoldi approach is suitable for computing extreme eigenvalues of general sparse matrices, whereas the Jacobi-Davidson method does not have such restrictions. In this presentation, the Jacobi preconditioner for the iterative solvers, which is used for the computation of the correction equations which arrise in the Jacobi-Davidson method, will be surveyed and evaluated on some shared memory architectures. The results show that the computational cost of the Jacobi-Davidson method is reduced by about half with the preconditioner.

1. はじめに

大規模疎行列の固有値計算アルゴリズムとしては, 従来 Lanczos/Arnoldi 系の解法[1)] を用いるのが一般的であった. 比較的小規模な行列においては, 全固有値を求める QR 法を用いることができるが, 問題の大きさ n に対して $\mathcal{O}(n^3)$ の計算量を要するため, この方法では規模の大きな問題を扱うことができない. このため, リスタートを用いた反復 Lanczos/Arnoldi 法は, 特に疎行列を扱う場合に最も実際的な解法であるといえるが, 固有値が近接している場合, 正確な計算が難しいことが知られている. Jacobi-Davidson 法[10)] 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[5)] をもとに, Jacobi 法[7)] の考え方を用いて構成された解法であるが, 比較的条件の悪い場合にも正確に固有値を計算できる[6),10)] ことから, Lanczos/Arnoldi 法に代わる有力なアルゴリズムとして注目されている. 本稿ではこの手法の概要を紹介するとともに, 前処理付 Jacobi-Davidson 法の実装結果と数値特性について述べる.

† 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

2. Jacobi-Davidson 法

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.

これに対して, 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

[☆] 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 1　JD 法による最大固有値の計算

$$A = A_P + u_k u_k^* A + A u_k u_k^* - \theta_k u_k u_k^* \quad (4)$$

と書き直すことができる．修正ベクトル z は

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

を満たすので, ここに 式 (4) を代入すれば

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

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, 式 (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 1 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．

3. 前　処　理

Jacobi-Davidson 法においては, 反復法による (8) の計算を効率的に行なう必要がある．そこで以下では Jacobi-Davidson 法の前処理について考える[11]．式 (8) の近似解を $\tilde{z}$ とする．このとき, $\tilde{z} \perp u_k$ より,

$$(A - \theta_k I)\tilde{z} - \alpha u_k = -r \quad (9)$$

が成り立つので, $A - \theta_k I$ を M_k で近似すれば,

$$\tilde{z} = -M_k^{-1} r + \alpha M_k^{-1} u_k \quad (10)$$

と表すことができる．この場合にも近似解は u_k と直交する空間に限定されるので, 実際には近似演算子として

$$\tilde{M}_k = (I - u_k u_k^*) M_k (I - u_k u_k^*) \quad (11)$$

を用いる必要がある．左前処理では, 演算子として $\tilde{M}_k^{-1}\tilde{A}$ $(\tilde{A} = (I - u_k u_k^*)(A - \theta_k I)(I - u_k u_k^*))$ を用いる．この場合, 反復ベクトル y に対して $\tilde{y} = (A - \theta_k I)y$ と置くと, $y \perp u_k$ より

solve $\bar{u}$ from $M_k \bar{u} = u$;
compute $\tilde{r} \equiv \tilde{M}_k^{-1} r$ as:
  solve x from $M_k x = r$;
  $\tilde{r} = x - \frac{u^* x}{u^* \bar{u}} \bar{u}$;
solve approximately $\tilde{M}_k^{-1}\tilde{A}z = -\tilde{r}$
with $z_0 = 0$;

図 2　左前処理を用いた修正方程式の計算部分

$$\tilde{A}y = (I - u_k u_k^*)(A - \theta_k I)(I - u_k u_k^*)y \quad (12)$$
$$= (I - u_k u_k^*)(A - \theta_k I)y \quad (13)$$
$$= (I - u_k u_k^*)\tilde{y} \quad (14)$$

となるので, $\tilde{M}_k^{-1}\tilde{A}y = \hat{y}$ は

$$\tilde{M}_k \hat{y} = (I - u_k u_k^*)\tilde{y} \quad (15)$$

を解くことによって求めることができる．実際には, $\hat{y}$ と u_k との直交性から

$$M_k \hat{y} = \tilde{y} - \alpha u_k \quad (16)$$

と書けるので, $M_k \bar{y} = \tilde{y}$, $M_k \bar{u} = u_k$ とすれば

$$\hat{y} = \bar{y} - \alpha \bar{u} \quad (17)$$

より $\hat{y}$ の直交条件を用いて

$$\alpha = \frac{u_k^* \bar{y}}{u_k^* \bar{u}} \quad (18)$$

を得る．以上をまとめたものを図 2 に示す．

式 (10) において, 前処理行列 M_k の取り方により, いくつかの方法が考えられる．$M_k = I$ と取る場合は,

$$\tilde{z} = -r + \alpha u_k = -Au_k + (\theta_k + \alpha)u_k \quad (19)$$

であることから, 右辺第 1 項より Arnoldi 法と同値である．また, $M_k = A - \theta_k I$ と取る場合は,

$$\tilde{z} = -u_k + \alpha(A - \theta_k I)^{-1} u_k \quad (20)$$

であることから, 右辺第 2 項よりシフト付逆反復法と同値である．$M_k = \mathrm{diag}(A)$ とする場合もこれに含まれる．

一方, 式 (8) を近似的に解く場合は, 係数行列 $B = (I - u_k u_k^*)(A - \theta_k I)(I - u_k u_k^*)$ に関して方程式 $B\tilde{z} = -r$ を近似的に解くことになる．実装方式については, QMR 法やブロック ILU 分解による前処理の実装例が報告されているが[3),9)], 一般行列に対する Jacobi-Davidson 法の前処理としては,

(1)　問題の物理的性質を仮定しないこと
(2)　演算量が小さいこと
(3)　並列性の高い解法であること

の各条件を満たす必要がある．この場合のアルゴリズムは以下のように構成される．

4. 実装と性能評価

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

solve $\bar{u}$ from $M_k\bar{u} = u$;
compute $\tilde{r} \equiv \tilde{M}_k^{-1} r$ as:
　　solve x approximately from $M_k x = r$
　　by the preconditioner;
　　$\tilde{r} = x - \frac{u^* x}{u^* \bar{u}} \bar{u}$;
apply BiCGSTAB to solve $\tilde{M}_k^{-1}\tilde{A}z = -\tilde{r}$
approximately with $z_0 = 0$ as:
　　solve x' from $M_k x' = (A - \theta_k I)z$;
　　$y' \equiv \tilde{M}_k^{-1}\tilde{A}z = x' - \frac{u^* x}{u^* \bar{u}} \bar{u}$;

図 3　前処理を用いた修正方程式の計算部分

般に演算量は $\mathcal{O}(n^2)$ 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[4] 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[12] 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.

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. 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.

そこで, 本アルゴリズムの並列化に当たっては, 上記の関数の外側ループを OpenMP API を用いてブロック化し, 並列に処理することとした. 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.

5. Jacobi-Davidson 法の数値特性

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, LAPACK 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

$$A = \begin{pmatrix} T_N & -I & & O \\ -I & \ddots & \ddots & \\ & \ddots & \ddots & -I \\ O & & -I & T_N \end{pmatrix}, \qquad (21)$$

$$T_N = \begin{pmatrix} 4 & -1 & & O \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ O & & -1 & 4 \end{pmatrix} \qquad (22)$$

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 15 までとし, 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), また基底数が 10 以下の場合には GMRES を用いた.

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は内訳は表 1の通りである.

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,
```

表 1 $n = 256^2$ での実行結果
Table 1 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
...			

表 2 $n = 256^2$ での並列実行結果
Table 2 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
...			

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`
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.

並列化に当たっては, まず, `zgemv`, `zdotc`, `zaxpy`, `zxpay` の 4 関数の外側ループを 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.

まず, 4-way SMP 上での並列化後の実行結果を表 2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. また, 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.

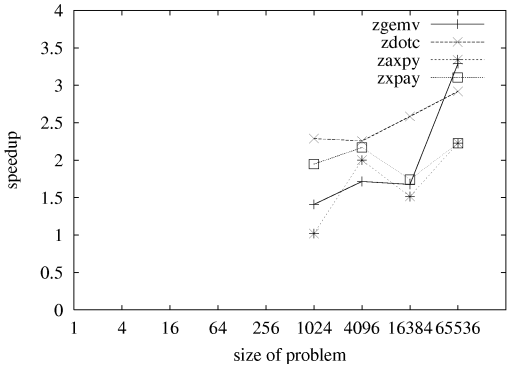

図 4 問題サイズと各ルーチンの速度向上率
Fig. 4 Size of problems and speedups.

6. Enterprise 10000 上への並列実装

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

$$y(1:m) = y(1:m) + M(1:m, j)x(j), \quad (23)$$
$$j = 1, 2, ..., n \quad (24)$$

として計算している.

```
*
* 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

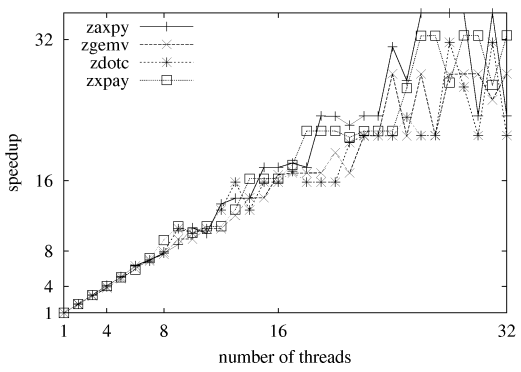

図 5 $n = 256^2$ での各ルーチンの速度向上率
Fig. 5 Speedups of subroutines for $n = 256^2$.

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{pmatrix} + \begin{pmatrix} M_1 \\ M_2 \\ \vdots \\ M_k \end{pmatrix} \times x \quad (25)$$

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+ FIRSTPRIVATE(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, 問題サイズ 256^2 で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. Enterprise 上では, Level 1, Level 2 BLAS ともほぼ線形な性能向上が得られている.

7. Jacobi 前処理

前節の例から, Jacobi-Davidson 法において修正方程式の求解に要する計算量が大きいことが分かるが, 適当な前処理を行なうことにより, この計算量を削減することができる.

ここでは行列形式のみを利用する解法として, 高い並列性を持つことが知られている Jacobi 反復法を適用することを考える. Jacobi 法では, 係数行列は対角優位性を満たせばよく, 修正方程式の計算においても高い収束性を示すと予想される.

まず, 問題サイズ 128^2 での適用結果を図 6に, また収束特性を図 7に示す. 横軸は修正方程式の計算回数, 縦軸は修正方程式の計算に要する BiCGSTAB 法の反復回数を示す. 前処理には, 並列化した Jacobi 法を用いた. ここでは, Jacobi 法の反復回数は総計算時間が最小となる 150 回とした.

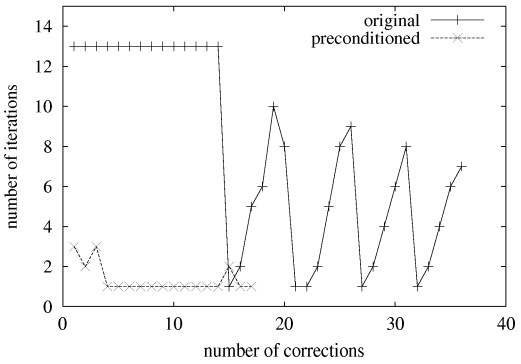

図 6 $n = 128^2$ での Jacobi 前処理の効果
Fig. 6 Performance enhancement with Jacobi preconditioner for $n = 128^2$.

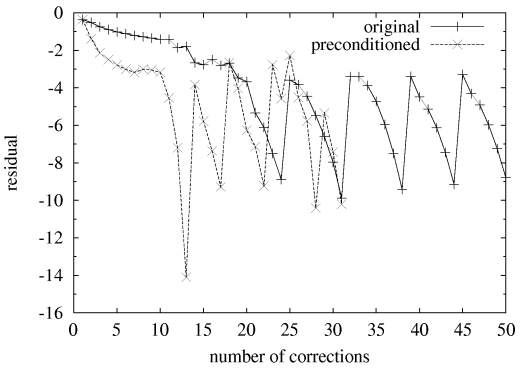

図 7 収束特性
Fig. 7 Convergence behavior.

この例では, 前処理を用いた場合の修正回数は, 前処理を行なわない場合の修正回数の約 50% であり, 計算時間についても約 50% である. 解法全体において反復解法の占める割合は大きく, またこの部分の並列化効率は, アルゴリズムが複雑であるためそれほど高くない. Jacobi 前処理の並列化効率がほぼ 100% であることを考慮すると, 前処理を行なうことにより, Jacobi-Davidson 法の

並列化効率は大きく向上するものと期待される.

次に図 8に Sun Enterprise 10000 上での速度向上率を示す. ここでは問題サイズは 256^2 とし, 最大固有値 1 個を計算した. Jacobi 前処理, 係数行列・ベクトル積, `zgemv`, `zdotc`, `zaxpy`, `zxpay` の 4 ルーチンを並列化している. Jacobi 法の反復回数は 20 回とした.

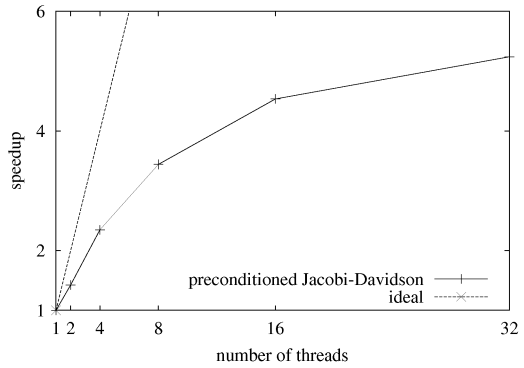

図 8 $n = 256^2$ での Jacobi 前処理付 JD 法の速度向上率
Fig. 8 Speedup of Jacobi preconditioned JD for $n = 256^2$.

8. まとめと今後の課題

本稿では, Jacobi-Davidson 法において計算量の大部分を占める修正方程式の計算に対して, 共有メモリ型並列計算機向きの解法である Jacobi 前処理を適用, 実装し, その評価結果について報告した. 特に, Jacobi 前処理を用いることにより, Jacobi-Davidson 法の計算時間に関して約 2 倍の性能向上が得られること, また, OpenMP を用いたループ並列化により, 前処理付 Jacobi-Davidson 法において計算量の大部分を占める Level 1, Level 2 BLAS 演算においても, 大規模 SMP 上で効率的に並列実行できることを示した.

本手法は比較的新しい解法であるため, 特性については明らかになっていない点も多い. 今後, 大規模固有値解法に対する有力な解法の一つとして様々な評価を行なっていく必要がある. 疎行列アルゴリズムにおいて必要となる Level 1, 2 BLAS 演算についても, プロセッサレベルでの最適化と合わせて, 今後実験・評価を通じてより効率的な実装方法を明らかにしていきたい.

参 考 文 献

1) Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, eds., *Templates for the Solution of Algebraic Eigenvalue Problems : A Practical Guide*, SIAM, 2000.
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*, SIAM, 1994.
3) A. Basermann, *Parallel Jacobi-Davidson Methods with Iterative Preconditioning for the Solution of Large Sparse Hermitian Eigenproblems*, in Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, CD-ROM, SIAM, Philadelphia, 1999.
4) 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.
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, *Über 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) M. Nool and A. van der Ploeg, *A parallel Jacobi-Davidson-Type method for solving large generalized eigenvalue problems in magnetohydrodynamics*, SIAM J. Sci. Comput., 22 (2000), pp. 95–112.
10) 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.
11) G. L. G. Sleijpen, H. A. van der Vorst, and E. Meijerink, *Efficient expansion of subspaces in the Jacobi-Davidson method for standard and generalized eigenproblems*, Tech. Rep. 1047, Department of Mathematics, Utrecht University, 1998.
12) B. C. Whaley and J. Dongarra, *Automatically Tuned Linear Algebra Software*, in Proceedings of SC98, 1998.