実対称固有値問題に対する多分割の分割統治法の分散メモリ型並列計算機への一実装

田村 純一　桑島 豊　重原 孝臣

埼玉大学大学院 理工学研究科

坪谷 怜

コンピュートロン株式会社

1　はじめに

実対称固有値問題は量子物理や金融工学などの分野で現れ，その高速な解法が必要とされている。実対称固有値問題を解く際には，前処理として実対称行列を三重対角化してから計算するのが一般的である。この処理には，数値的に安定な *Householder* **法**がよく用いられる。

実対称三重対角行列の固有値問題に対する解法は，今まで多くのアルゴリズムが提案されている。古典的な手法である**二分法・逆反復法**(以下 BII) や，行列の性質により高速化できる高精度な**二分割の分割統治法**(以下 DC2)[1] などがある。また近年，桑島らにより多分割の分割統治法 (以下 DCK)[3, 4] が提案された。DCK は DC2 を拡張し，$k > 2$ の分割数をとることが可能な手法で，DC2 では遅延が生じる行列においても固有分解を高速に計算することができる。他にも，二分法を改良した *Multiple Relatively Robust Representations(MRRR)* **法**[2] もあるが，並列化に難点があると言われている。

DCK の逐次実装および共有メモリ型並列計算機での実装は既に行われている [6]。しかし，分散メモリ型並列計算機での実装は未だ行われていない。そこで本稿では，DCK の分散並列化手法の提案と，HITACHI HA8000 と HITACHI SR11000 を用いて行った数値実験の結果を述べる。DCK のアルゴリズムには，最大で n 次行列の積をとるステップ，また必要であれば計算した固有ベクトルの再直交化を行うステップがあり，これらの計算ステップの負荷が大きくなることがある。本稿で提案する並列化手法は，これらのステップの負荷を低減することに主眼を置いている。とくに行列積は *deflation* が少ない場合に負荷が大きくなるため，そのような場合に負荷をうまく低減することを目的とした並列化手法を提案する。

以下では，2 章で多分割の分割統治法のアルゴリズムを簡単に説明し，3 章で分散並列化の手法を述べる。4 章では数値実験について述べ，5 章でまとめを述べる。

2　DCK のアルゴリズム

多分割の分割統治法 (DCK) は，n 次実対称三重対角行列 T と分割数 $k \ll n$ を入力として，T の固有分解 $T = Q\Lambda Q^T$ を与える対角行列 $\Lambda \equiv \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$ と直交行列 $Q \equiv (\boldsymbol{q}_1, \ldots, \boldsymbol{q}_n)$ を出力する。

DCK のアルゴリズムは大雑把に次の 3 ステップからなる。まず，入力行列 T を，直交行列 Q_p を用いて実対角行列 D と低階数摂動 UU^T の和 (U は $n \times (k-1)$ 行列) へ相似変換する: すなわち $T = Q_p(D + UU^T)Q_p^T$。次に，この $D + UU^T$ の固有分解 $D + UU^T = Q_u \Lambda Q_u^T$ を計算する。最後に行列積 $Q_p Q_u$ を計算することで，T の全固有ベクトルを得ることができる。各ステップの詳細は以下に示す。いくつかのステップには，数値実験で参照するために `Tj` や `group` といった名前をつけている。

1. T を $T = Q_p(D + UU^T)Q_p^T$ と相似変換する。$D = \mathrm{diag}(d_1, \ldots, d_n)$ は対角行列，
   U は $n \times (k-1)$ 行列，Q_p は k 個の直交行列の直和。
   (a) T を三重対角行列の直和と摂動の和 $T = \bigoplus T_j + VV^T$ $(j = 1, \ldots, k)$ に分割する。
   (b) T_j の固有分解 $T_j = Q_j \Lambda_j Q_j^T$ を (再帰的に) 計算する (`Tj`)。
   (c) $Q_p \equiv \bigoplus Q_j$ を用いて $T = Q_p(D + UU^T)Q_p^T$ と相似変換する。
2. 固有分解 $D + UU^T = Q_u \Lambda Q_u^T$ を計算する。
   (a) U の第 j 行が全て 0 のとき，deflation により $D + UU^T$ の自明な固有対 $(d_j, \boldsymbol{e}_j)$ を除去する。ただし n 次単位行列の第 j 列ベクトルを $\boldsymbol{e}_j$ とする。
   ここで $n - r$ の自明解が得られたとして，以降のステップでは自明でない r 個の固有対を計算する。
   (b) $D + UU^T$ の非自明な固有値 $\lambda'_1, \ldots, \lambda'_r$ を求める (`eval`)。これらは T の固有値に一致する。
   (c) $D + UU^T$ の非自明な固有ベクトル $\boldsymbol{q}'_1, \ldots, \boldsymbol{q}'_r$ を求める (`evec`)。
   (d) 必要ならば固有ベクトルを再直交化する。
      i. 固有ベクトル同士の直交性を判定する (`group`)。
      ii. 直交しないベクトル同士をグループ化する (`group`)。
      iii. グループ毎に直交化する (`reortho`)。
3. 行列積 $Q_p Q_u = Q$ を計算する (`prod`)。

DCK は，分割数 k を大きくすることで直交行列 Q_p の非零要素数を減少させることができる。これによりステップ 3. の行列積の演算量を削減できる。DCK のステップの中で常に $O(n^3)$ かかり，かつ最も大きい演算量となるのはこの行列積のみである。よって，分割数 k を大きくして行列積の演算量を削減することで，全体として高速化できるという点が DCK の特徴である。

ステップ 2a. の deflation は，自明な固有対を事前に取り除き，解くべき固有値問題のサイズを小さくする操作である。deflation で得た固有対を用いると，ステップ 3. の行列積を計算せずに T の固有対が得られる。deflation が多く起こると，高速に T の固有分解が計算できる。

ステップ 2b. では，問題を対角行列と階数 1 の摂動の和 $D + uu$ の固有分解に帰着させる。これを $k - 1$ 回逐次的に解くことで $D + UU^T$ の固有値を得る。$D + uu$ の固有分解の計算は DC2 の内部計算に現れ，並列性の高さが知られている。ステップ 2c. も同様に，$k - 1 + \alpha_\lambda$ 次 (通常 α_λ は n に対して十分小さい) の行列 $\tilde{F}(\lambda)$ の核の計算に帰着させる [4] が，この計算は λ ごとに独立に行うことができる。核の計算には，逆反復法を用いる。

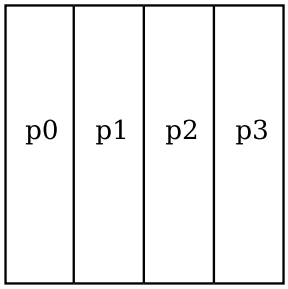

図 1　列ブロック分割の例 ($p = 4$)

ステップ 3. は，DCK のアルゴリズム中で唯一 $O(n^3)$ かかるステップである。しかし Q_p が直和型行列でありゼロ要素が多く存在することを利用して，効率良く計算できる。

以上のアルゴリズムの中で，プロセス間の通信が頻発するのはステップ 2d. およびステップ 3. であり，両ステップにおける通信負担の低減が重要になる。

3　分散並列化

一般に，プログラムの並列化には，プロセス内の複数のスレッドで処理を並行に実行する方式と，複数プロセスが処理を並行に実行する方式がある。それぞれの方式は OpenMP[8] と MPI[9] という規格で標準化されている。前者は共有メモリ型並列計算機で用いられ，並列化したい処理にコンパイラに対する指示を記述するだけで並列化できる。ただし効率はコンパイラに依存する。後者は主に分散メモリ型並列計算機向けであり，プロセス間の通信を明示的に記述する必要があり繁雑さが増すが，他方詳細なチューニングが可能である。一般に通信はネットワークを経由するため，CPU がメモリにアクセスする時間と比較してコストが高い。そこで，MPI を用いて並列化する際はできるだけ通信を避けることが望ましい。

分散メモリ型並列計算機向けに並列化するにあたり，今回は MPI のみを使用し，OpenMP によるスレッドを利用した並列化は行っていない。以降では，使用するプロセス数を p とし，各プロセスを $p_0, p_1, \ldots, p_{p-1}$ と呼ぶ。また，本実装では，分割数 k は p の倍数であることを仮定する。

分散並列化を行うにあたり，データ分散の手法はスケーラビリティや効率に大きく関わるため重要である。今回は，アルゴリズム中で演算量最大のステップ 3.(行列積) と並列化手法が自明ではないステップ 2d.(再直交化) を分散並列化するため，行列の列ブロック分割を選択した。

列ブロック分割とは，行列をベクトルの並びと見て，あるプロセスがいくつかの隣接するベクトルをまとめて保持することである。図 1 のように，n 次行列を p 個のプロセスが分散して持つ場合，各プロセスは n/p 本の隣り合うベクトルを持つ。DCK では，各プロセスは出力する直交行列 Q を列ブロック分割で保持する。このデータ分散方式に合うよう，プロセスグリッドは $1 \times p$ としている。

入力の T と出力の Λ は $O(n)$ 領域であり，全プロセスが保持する。

DCK のアルゴリズムにおいて，1b.(T_j の固有分解)，2b.($D+UU^T$ の固有値)，2c.($D+UU^T$ の固有ベクトル)，2d.(固有ベクトルの再直交化)，3.(行列積 Q_pQ_u) の 5 つが主要な計算ステップである。そこで，以下では各ステップにどのような分散並列化を行うかを述べる。

■T_j の固有分解　ステップ 1b.(T_j の固有分解) を T_j ごとにプロセスに割り振ることで並列化する。各プロセスは k/p 個の T_j の固有分解を行うが，各固有分解は逐次版 LAPACK の DC2 を用いて計算する。

入力行列 T を全プロセスが保持しているため，このステップ中に通信は必要ない。T_j の固有分解が全て終わると，通信を行い D,U を全プロセスで共有する。

■$D+UU^T$ の固有値　ステップ 2b. は実対角行列と階数 1 の摂動の和 $D+uu^T$ の固有分解を繰り返し逐次的に解くことで計算できる。DC2 の内部計算に現れる $D+uu^T$ の固有分解は，並列性が高いことが知られており，この計算を並列化することで，本ステップを並列化している。

$D+UU^T$ の全固有値を求めたら，固有値を昇順に整列して全体に放送する。これは，固有ベクトル計算後の再直交化の効率を上げるための準備である。

■$D+UU^T$ の固有ベクトル　ステップ 2c. では，$D+UU^T$ の固有値 λ に属する固有ベクトルを $\tilde{F}(\lambda)$ の核を利用して計算する。核の計算は逆反復法による。各固有ベクトルは固有値ごとに独立に求められるため，各プロセスに割り振ることで並列化する。Q_u を列ブロック分割形式でデータ分散するため，プロセス p_j は $\boldsymbol{q}'_{jn/p+1},\ldots,\boldsymbol{q}'_{(j+1)n/p}$ のみを計算する。

固有ベクトル計算時には正規化前の長さとレイリー商を保存しておき，全固有ベクトル計算後にそれらを全プロセスで共有する。これは固有ベクトルの直交性検査 (後述の簡易検査) に用いられる。

以上の 3 つのステップは，各ステップの持つ並列性を利用して分散並列化しており，手法としては自明なものである。残りの再直交化と行列積計算に関しては，手法が自明ではないため，詳しく述べる。

3.1　再直交化

再直交化はベクトル間の直交性の検査，非直交グループの構成，直交化計算の 3 つのステップからなる。本節では，提案する分散並列化手法をステップごとに述べる。

3.1.1　直交性の検査

直交性の検査は以下の手順で行う。固有値を事前にソートし，また，列ブロック分割を用いたことで，非直交な固有ベクトルはプロセス番号が近い (または同じ) プロセスに局在化しており，その結果，後述の [内積による検査] におけるデータ通信量を抑制することができている。

固有ベクトル同士の直交性検査の結果は 2 値行列 B に保存する。行列 B は Q_u と同様に，列ブロック分割形式で各プロセスにデータを分散させ，プロセス p_j が持つ部分を $B_{:j}$ と表記する。行

列 B の (i,j) 成分 b_{ij} は，ステップ 2c. で求めた固有ベクトル $\boldsymbol{q}'_i, \boldsymbol{q}'_j$ と微小パラメータ ϵ を元に次のように計算する (具体的手順は後述)。

$$b_{ij} = \begin{cases} 0 & (|(\boldsymbol{q}'_i, \boldsymbol{q}'_j)| \leq \epsilon) \\ 1 & (otherwise) \end{cases}$$

固有ベクトルの組み合わせで要素の値が決まるため，B は対称行列となる。また非直交な組み合わせは隣り合った固有ベクトルで多く起こるため，B は対角成分近辺に 1 が多い行列となる。

行列 B の成分を計算するための，固有ベクトル同士の直交性判定は 2 段階で行う。まず，簡易検査を行い，直交性が不十分と判定された場合のみさらに詳細な内積を用いた検査を行う。内積の前に簡易検査を行うことで，出来るだけ通信を避けている。各検査は以下の通り。

[簡易検査] 固有ベクトルを求める際に得たそのベクトルの長さとレイリー商を利用して，固有ベクトルの内積を過大評価する。必要なデータは全プロセスが保持しているため，簡易検査中に通信は不要である。簡易検査の詳細は [4] に譲る。

[内積による検査] 簡易検査で直交性が不十分とされた場合，内積により直交性を検査する。内積を計算するにはベクトルデータが必要であるため，通信を要する。

以下，まず 6 プロセス用いる場合 $(p = 6)$ の例を述べたあとで，アルゴリズムとして直交性検査のステップを示す。図 2 はこの例での計算ステップの進行を示す。以降，特に明記しない限り，ベクトルは $D + UU^T$ の固有ベクトルを意味する。また $D + UU^T$ の固有ベクトルを並べた Q_u のうちプロセス p_j が計算した部分を $Q_{u;j} = (\boldsymbol{q}'_{jn/p+1}, \ldots, \boldsymbol{q}'_{(j+1)n/p})$ と表記する。まず，プロセス p_0 の持つベクトル $Q_{u;0}$ と p_2 の持つベクトル $Q_{u;2}$ との直交性を次のように調べる。

1. p_2 が，$Q_{u;0}$ と $Q_{u;2}$ の間で簡易検査を行い，$Q_{u;2}$ の各ベクトルと非直交となる $Q_{u;0}$ のベクトルの対の候補を求める。候補が無ければ，B の (1,3) ブロックに 0 を代入し，直交性の検査を終了する。候補がある場合，以下の 2.∼ 4. のステップに進む。
2. 1. で求めた非直交なベクトルの対の候補の中で，$Q_{u;0}$ の中の最初と最後のベクトルの番号 $f_{0,2}, \ell_{0,2}$ を p_2 から p_0 へ送信する。
3. p_0 は，$Q_{u;0}$ のうち $f_{0,2}$ 番目から $\ell_{0,2}$ 番目までのベクトルを p_2 に送信する。
4. p_2 は，受信したベクトルのうち非直交の可能性のあるベクトルの対に対して内積計算を行い，2 値行列 B の $(1,3)$ ブロックの値を得る。

これで p_2 は，p_0 の持つベクトルとの直交性検査を終え，2 値行列 B の一部を計算できた。同じ処理を p_0 と p_4 の間でも行うことで，図 2(a) の最上行のうちプロセス名の記載されているブロックの計算ができたことになる。また p_1 と p_0，p_1 と p_3，p_1 と p_5 の間でも同じように行う (いずれも p_1 がベクトルデータを送信する) と，図 2(b) のように，上から 2 行目を計算できる。図 2 の四角の中の文字は，その部分の直交性検査を行うプロセスを示している。上の矢印はベクトルデータの流れを示しており，背景が赤のプロセスから送信することを表す。ここで，上から 2 ブロック分の部分を構成するステップは，通信がほとんど競合せず，各プロセスの行う処理は独立なため並列に実行できると考えられる。

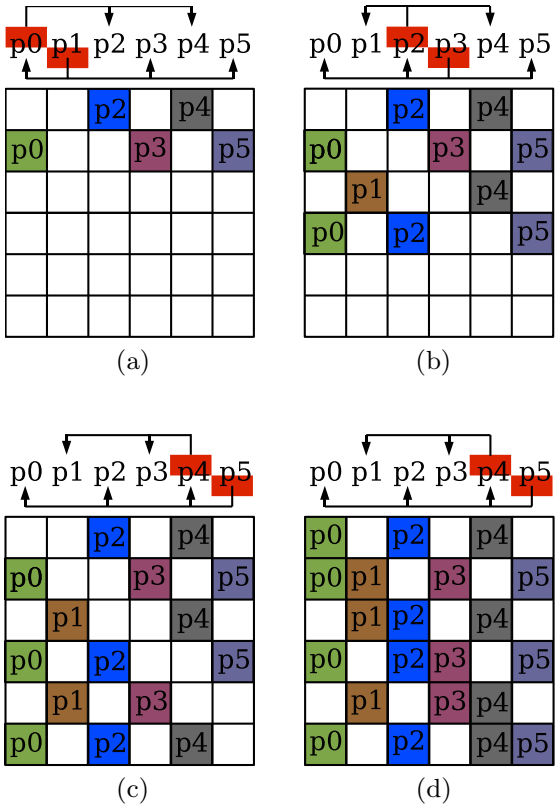

図 2　直交性検査 (行列 B の構成) の進行

2 値対称行列 B の構成は，図 2(a) から (b)，(c) へと進行する。先程述べたように，このステップは行ごとに並列に実行可能である。

最後に各プロセス p_j が持つベクトル $Q_{u;j}$ 同士の直交性を調べる (図 2(d))。固有値の並びと固有ベクトル計算の分配方法から，ある一つのプロセスが持つベクトル同士の直交性検査には内積計算が必要になることが多いと考えられる。この部分を最後に行うのは，図 2 のステップ (a) から (c) のように，通信が必要な時にその部分を計算すると遅延が発生する恐れがあり，これを避けるためである。

一般的には，以下のアルゴリズムに従い直交性検査を行う。ここで，整数 s を自分のプロセス番号とする。

1: **for** $i = 0$ to $p - 1$ **do**
2: **if** $s - i$ が正の偶数 or $i - s$ が正の奇数 **then**
3: $Q_{u;i}$ と $Q_{u;s}$ に対して簡易検査を行う
4: 簡易検査の結果に基づき，$Q_{u;s}$ と非直交な $Q_{u;i}$ の最初と最後のベクトルの番号を $f_{i,s}$, $\ell_{i,s}$ とする
5: 直交性の悪い組合せが無いならば $f_{i,s} = \ell_{i,s} = -1$ とする
6: p_i に $f_{i,s}$, $\ell_{i,s}$ を送信する
7: **if** $Q_{u;i}$ と $Q_{u;s}$ に直交性の悪い組合せがある **then**

8: p_i から $Q_{u;i}$ の $f_{i,s}$ 列目以降 $\ell_{i,s}$ 列目までを受信するまで待つ
9: 受信後，直交性の悪い組合せについてのみ内積計算を行い，直交性を判定する
10: **end if**
11: **else if** $s = i$ **then**
12: **for** $j = 0$ to $p - 1$ **do**
13: **if** $j - s$ が正の偶数 or $s - j$ が正の奇数 **then**
14: p_j から $f_{i,j}$, $\ell_{i,j}$ を受信するまで待つ
15: 受信後，$f_{i,j} \geq 0$ ならば，p_j へ $Q_{u;s}$ の $f_{i,j}$ 列目以降 $\ell_{i,j}$ 列目までを送信する
16: **end if**
17: **end for**
18: **end if**
19: **end for**
20: $Q_{u;s}$ のベクトル同士の直交性を検査する

3.1.2 グループ化

前節で作成した 2 値対称行列 B に基づき，非直交なベクトル同士をまとめたグループを構成する。グループ情報は互いに素な集合 (Disjoint set) を扱うデータ構造 Union-Find[10] を用いて管理する。

まず各プロセスは，2 値対称行列 B のうち自分が計算した部分のみを参照してグループを作る。つまりプロセス p_j が B の部分行列 $B_{:j}$ を参照する。グループ情報としては，各固有ベクトルがどのグループに属するかをグループごとに整数値で表せばよいため，n 次整数ベクトル一つの領域があれば十分である。次に，p_{p-1} は作成したグループ情報を p_{p-2} に送信する。p_{p-2} は，受信した p_{p-1} のグループ情報を自身の持つグループ情報と合併させ，それを p_{p-3} に送信する。これをプロセス番号の降順に繰り返すことで，最終的に全プロセスからのグループ情報が p_0 に集まる。p_0 は最終結果を全プロセスに送信する。

本ステップでプロセスが通信する情報は毎回 $O(n)$ であり，これを $p - 1$ 回繰り返して最後に全体に送信するため，$O(pn)$ の通信が必要となる。

3.1.3 直交化計算

グループごとにレイリー・リッツ法を用いてベクトルを直交化する。直交化は，グループ内のベクトル数があらかじめ設定した閾値以下であれば，単一のプロセスが行う。そうでなければ，全てのプロセスを用いて行う。

グループの直交化の手順を述べる。一般にグループ内のベクトルは異なるプロセスが保持しているため，単一プロセスを用いる場合は，そこにグループ内の全てのベクトルを集めて該当プロセスが直交化計算を行う。全プロセスを用いる場合は，適切なプロセスにベクトルを送信してデータ分散を行い，分散並列ルーチンを用いて直交化計算を行う。最後に，直交化前のベクトルを保持して

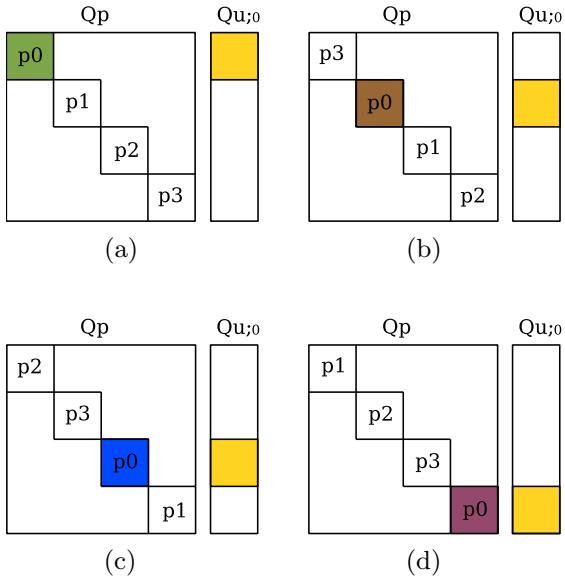

図 3　DGEMM+MPI による行列積の計算の進行 (p_0 の例)

いたプロセスに直交化後のベクトルを送信する。なお直交化計算には，単一プロセスを用いる場合は逐次版 LAPACK の `DSYEV` を，全プロセスの場合は ScaLAPACK の `PDSYEV` を使用する。

3.2　行列積

このステップは，分散メモリ型並列計算機向け線形代数ライブラリ ScaLAPACK の下位ライブラリ PBLAS に含まれる行列積を計算するルーチン `PDGEMM` を用いる方法と，プロセス間で Q_p の対角ブロック行列を MPI により通信し，各プロセスは数値線形代数ライブラリ LAPACK の下位ライブラリ BLAS の `DGEMM` を用いて逐次的に行列積を計算する方法のいずれかにより計算することができる。`PDGEMM` を用いると通信に関する事項をライブラリに任せることができるが，通信を自分で書き `DGEMM` を用いると通信を制御できるため性能を上げられる可能性がある。

まず，`DGEMM+MPI` でどのように通信と計算を行うかを述べる。概要としては，Q_p の対角ブロックを隣接プロセス間で交換し合いながら，行列積自体はプロセス内で `DGEMM` により計算を行うというものである。以下に，プロセス数 $p = 4$ のときに p_0 が対角ブロックを交換しながら行列積を行う様子を示す。図 3(a) の Q_p には対角ブロックを所持しているプロセスを記し，p_0 が所持している $Q_{u;0}$ を図の $Q_{u;0}$ の下にある長方形で示している。まず p_0 は，今所持している Q_p の対角ブロックと $Q_{u;0}$ の色で示す部分の積を取る。次に，所持している Q_p の対角ブロックを p_3 に送信し，p_1 の持つブロックを受け取る。隣接するプロセスへ対角ブロック行列を渡し，逆隣から受け取るというように，バケツリレー的にブロック行列を循環させている。今度は図 3(b) のように，対角部分と，今度は一ブロック下の色付きの部分とで行列積を行う。これを (c)，(d) と進めていき，他のプロセスでも同様に計算することで，Q_pQ_u が計算できる。各プロセスが対角ブロック行列を k 回交換するため通信量は $O(n^2/k)$，計算量は $O(n^3/(kp))$ である。

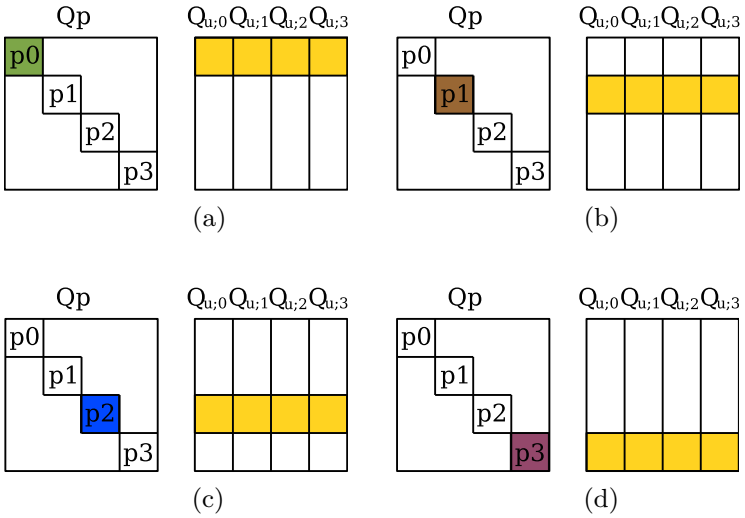

図 4　PDGEMM による行列積の計算の進行

次に，PDGEMM を用いた計算方法を述べる。図 4 に PDGEMM を用いた行列積計算 ($p = 4$ の例) の進行の様子を示す。DGEMM+MPI では対角ブロックをプロセス間で明示的に通信しながら行列積を計算したが，こちらでは通信をルーチン側に任せている。またこちらの方式でも，ブロック行列が対角に並ぶという Q_p の形式を生かすため，Q_p の i 番目の対角ブロック行列と全プロセス上に分散した $Q_{u;j}$ の部分行列 (図 4 の色付き部分で示している) との積を PDGEMM により計算する。この積を対角ブロックごとに行うことで，Q_p と Q_u の積を計算できる。

今回は，HITACHI SR11000 における次の予備実験の結果を元に計算方法を決定した。DCK のアルゴリズムに現れる n/k 次ブロックが k 個対角に並ぶ直和行列 (全体で n 次行列) と n 次密行列の積を，PDGEMM と DGEMM+MPI でベンチマークとして実装し，それぞれの所要時間を計測した。実験結果を図 5 に示す。横軸にプロセス数，縦軸に所用時間の比を取り，行列サイズ n と分割数 k を変えて得たグラフを示している。ここで，グラフは PDGEMM による行列積の計算時間に対する DGEMM+MPI による計算時間の比であり，1 より下にあるほど DGEMM+MPI の方が速いことを表す。図を見ると，どのグラフもほぼ 1 に近いかそれより下にあり，また分割数 k が大きいグラフほど下の方にある。以上より，本実装では，行列積には DGEMM+MPI を用いた。

4　数値実験

分散メモリ型並列計算機 HITACHI HA8000 上で数値実験を行い，今回提案した DCK の分散並列化の効率を確かめる。また，他の実対称三重対角固有値問題の解法と速度を比較する。

数値計算ライブラリとして LAPACK(BLAS), その分散並列版である ScaLAPACK[5](PBLAS) を用いている。ScaLAPACK と PBLAS は，線形代数ライブラリ用メッセージ交換 (通信) ライブラリ BLACS を経由して MPI 通信を行うが，DCK で通信を行う際も同様に BLACS を用いている。MPI による通信には，MPI1.2 通信ライブラリ MPICH-MX を利用している。BLAS に自前

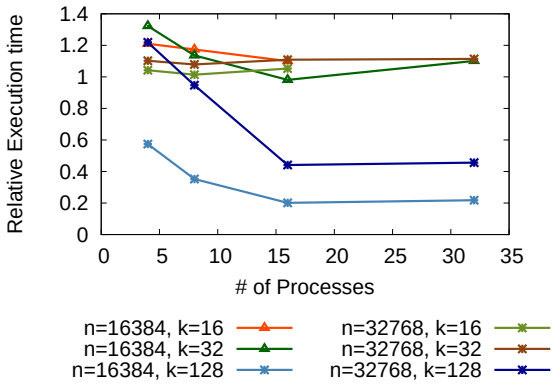

図 5　PDGEMM に対する DGEMM+MPI の相対実行時間

行列 DS	$a_j = j \times 10^{-6}$,　$b_j = 1$。deflation はあまり起こらない。
行列 QC	平均 0，分散 1 の正規乱数を要素とする実対称行列を三重対角化した行列。 量子カオス系のモデルとして用いられる。deflation はあまり起こらない。
行列 DL	$a_j \in (2, 4]$,　$b_j \in (1, 2]$ の一様乱数を持つ実対称三重対角行列。deflation が多く起こる。

表 1　実験に使用する行列

でコンパイルした GotoBLAS 1.26 を用いている以外は，いずれのライブラリも並列計算機に備え付けのものを使用している。

プログラムは C 言語で実装し，HITACHI 最適化 C コンパイラでコンパイルした。MPI を用いて通信するため，コンパイルは `mpicc -64 -Wc,-V -O4 +Op -noparallel` のようにした。

4.1　対象行列

表 1 に示す 3 種類の実対称三重対角行列を用いた。サイズは $n = 10000, 20000, 30000$ の 3 種類である。表 1 内では，主対角成分を a_j，副対角成分を b_j としている。

4.2　実験条件

分散並列化した DCK を用いて，n 次実対称三重対角行列の全固有対 $(\lambda_j, \boldsymbol{q}_j)$ を計算する時間を計測する。解の相対残差 ϵ_r，直交誤差 ϵ_o は次の式で評価する。

$$\epsilon_r = \max_{1 \le j \le n} \frac{||T\boldsymbol{q}_j - \lambda_j \boldsymbol{q}_j||_2}{||T||_2}$$
$$\epsilon_o = \max_{1 \le i \le j \le n} |\boldsymbol{q}_i^T \boldsymbol{q}_j - \delta_{ij}|$$

ただし δ_{ij} はクロネッカーのデルタである。

また BII，DC2 を用いて同様の計算を行い，DCK と速度を比較する。どちらも ScaLAPACK

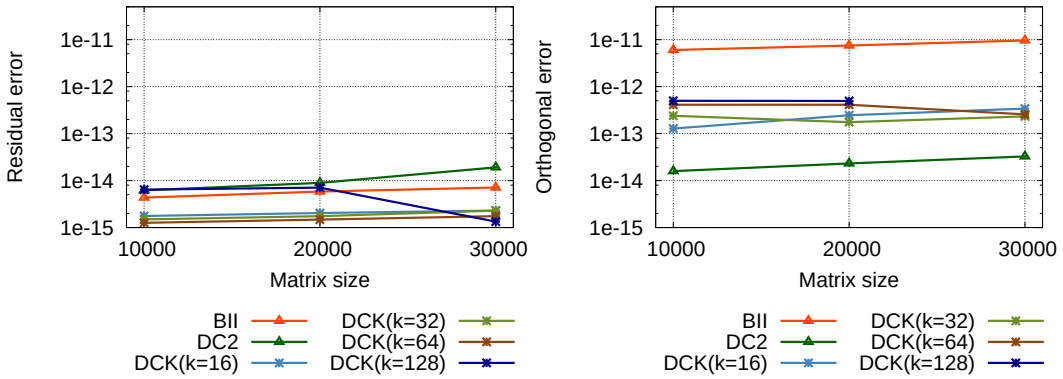

図 6　相対残差 ϵ_r と直交誤差 ϵ_o

に実装されているルーチン (pdstebz/pdstein, pdstedc) を用いている。

なお，実験時は 1 プロセスに CPU コアを 1 つ割り当てている。

4.3　実験結果

まず始めに，精度に関する実験結果を述べる。DCK は精度に関するパラメータを持っており，これが全体の性能にも影響を与えるためである。プロセス数 $p = 16$ として行列 DS を用いた。図 6 に示すように，DCK の計算精度は BII と DC2 の中間程度である。以降の実験では，特に指定のない限り，精度に関するパラメータはここで指定したものと同じである。

次に，分散並列化した DCK の評価を行う。プロセス数を増加させた時に，DCK の各計算ステップにかかる時間を図 7 に示す。30000 次の行列 DS を分割数 $k = 32$ で計算している。なお図 7 から図 12 まで，右の図は左の図を両対数にしたものである。all が全体時間を意味し，他は 2 に記載したアルゴリズムでの名前付けに従う。計算時間が最も大きい prod とそれに次いで大きい eval，evec，Tj がプロセス数の増加に応じて高速化できており，これらの計算ステップを分散並列化できていることが分かる。group と reortho は所用時間の絶対値が小さいことで並列化の効果が見えづらいため，追補実験を次に示す。

再直交化に関する処理の詳しい計測を HA8000 で取りきれなかったため，SR11000 の 1 ノード (16CPU) における結果を代わりに示す。

共有並列化を行った DCK[6] と今回提案する分散並列化を行った DCK とを比較し，グループ化 (group) と直交化計算 (reortho) の分散並列化の効率を見る。一般に，共有並列の方が通信を必要としないため並列効率は高い。表 2 に行列サイズ $n = 20000$，16 並列 (共有並列は 16 スレッド，分散並列は 16 プロセス) の元で行列 QC を用いたときの計算時間を計算ステップごとに分割数 $k = 16, 32$ について示す。項目名は 2 章のアルゴリズムでの名前付けに従う。なお再直交化の負荷を上げるため，直交精度に関するパラメータを本章の冒頭で示した値より 1 桁小さくしている。また共有並列ではスレッド並列化済みの LAPACK(BLAS) を利用している。

表 2 によると，分散の group は共有よりも約 30% 速く計算できており，3.1.1 節に述べた手法

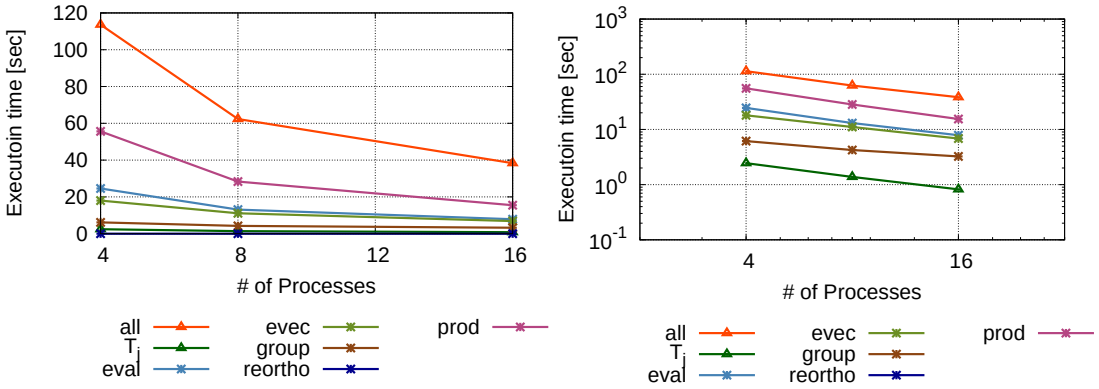

図 7　プロセス数に対する各計算ステップの所要時間 (行列 DS，$n = 30000$，$k = 32$)

k	並列化手法	`all`	`recur`	`eval`	`evec`	`group`	`reortho`	`prod`
16	共有	27.8	6.8	3.8	1.8	3.5	1.6	9.9
	分散	25.0	0.8	3.7	1.8	2.4	2.8	12.1
32	共有	23.8	5.0	4.2	2.2	3.7	3.2	5.2
	分散	21.7	0.3	3.9	2.2	2.6	4.7	6.5

表 2　共有並列と分散並列の比較 (行列 QC，$n = 20000$，単位:秒)

が効果を上げていることを示している。一方，分散の `reortho` のみを見ると，$k = 16$ では 75%，$k = 32$ では 47% の遅れが生じているが，`group` と合わせて再直交化としてまとめると共有並列に対して数パーセントの遅れとなり，十分な速度で計算できていると考えられる。

次に，分割数を変えて行列 DS の固有分解を計算したときの結果を図 8 に示す。凡例の k の値は分割数を表す。どの分割数でも，プロセス数を増やすことで高速化している。また行列 DS は deflation が起こらないため，分割数をある程度大きくした方が性能が良く，この場合には分割数 $k = 32, 64$ の時が最も高速に計算できている。

次に，プロセス数を $p = 16$ とし，行列サイズを変えて実験した結果を図 9 に示す。$k = 16$ は $n = 30000$ で大きく時間がかかっているが，図 9 右の両対数グラフを見ると他はいずれも n^3 よりは傾きが小さい。他誌で報告されている [6] ように，DCK の計算量は $O(n^3)$ であるが，deflation が少ない行列に対しては実際の並列実行時間はそれを下回ることがここでも確認できる。

次に，図 9 に他の解法のグラフを重ねたものを図 10 に示す。行列 DS の場合，分割数を大きくすると有利であり，DCK が DC2 より高速に計算できる。また，適切な分割数であれば BII より高速に計算できていることがわかる。実際の演算量としては，図 10 右で DCK より BII がやや傾きが緩やかであることから，サイズを大きくしていくと BII が有利となる可能性がある。

行列 QC に対して図 10 と同等な実験を行った結果を図 11 に示す。deflation の起こる頻度は行列 DS と似ているため，図 10 と同様の結果が得られた。行列 QC では，$\tilde{F}(\lambda)$ の最大サイズが行

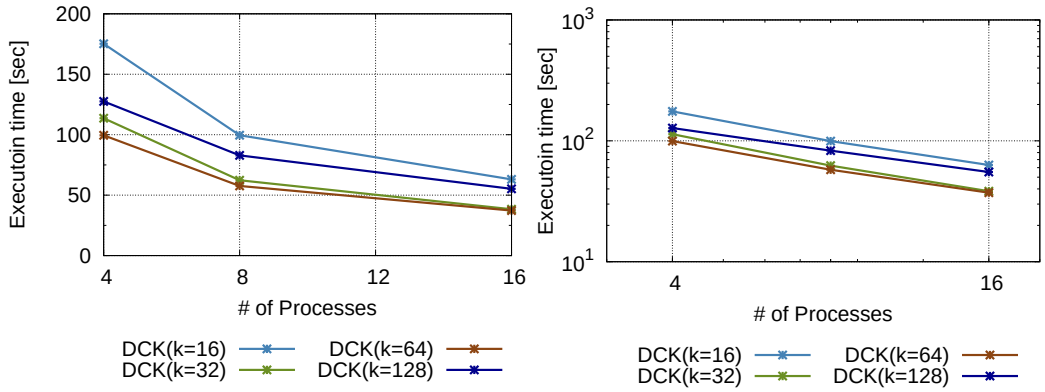

図 8　分割数毎のプロセス数に対する所要時間の変化 (行列 DS，$n = 30000$)

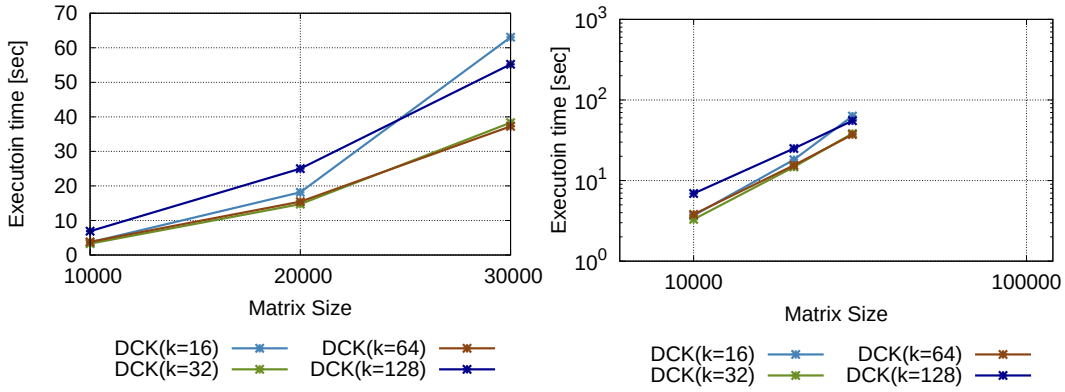

図 9　分割数毎の行列サイズに対する所要時間の変化 (行列 DS，$p = 16$)

列 DS での 1/4 程度になっていることで，固有ベクトル計算にかかる時間が短くなり，DCK で最速となる分割数が行列 DS より大きくなっていた。

行列 DL を用いた結果を図 12 に示す。この行列は上の 2 つの行列とは違い deflation が多く起こり，行列積の計算負荷が削減されるため，DC2 が最も高速となっている。また適切な分割数であれば DCK は BII と同程度の速度で計算可能である。

5　まとめ

今回，多分割の分割統治法 (DCK) の分散並列化手法を提案，実装し，HITACHI HA8000 と HITACHI SR11000 上で評価を行った。deflation が多く起こる行列に対しては二分割の分割統治法 (DC2) が有効だが，deflation が少ない行列においては DCK の方が有利となる。後者の行列では DCK のアルゴリズム中の行列積と再直交化の負荷が大きくなることがあるため，これらの負荷を低減するような手法を提案した。

DCK の主要な計算ステップのうち T_j の固有分解 (`Tj`)，$D + UU^T$ の固有値計算 (`eval`)，$D + UU^T$ の固有ベクトル計算 (`evec`) には高い並列性があり，それらを利用することで分散並列

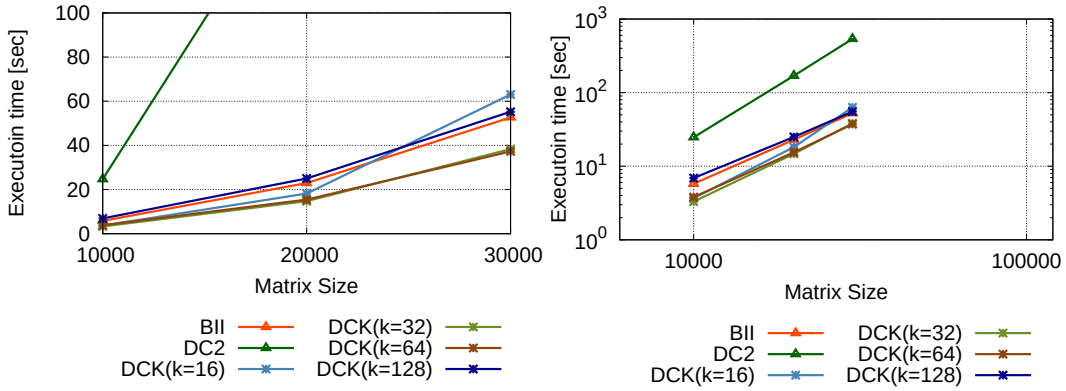

図 10　解法ごとの行列サイズに対する所要時間の変化 (行列 DS，$p = 16$)

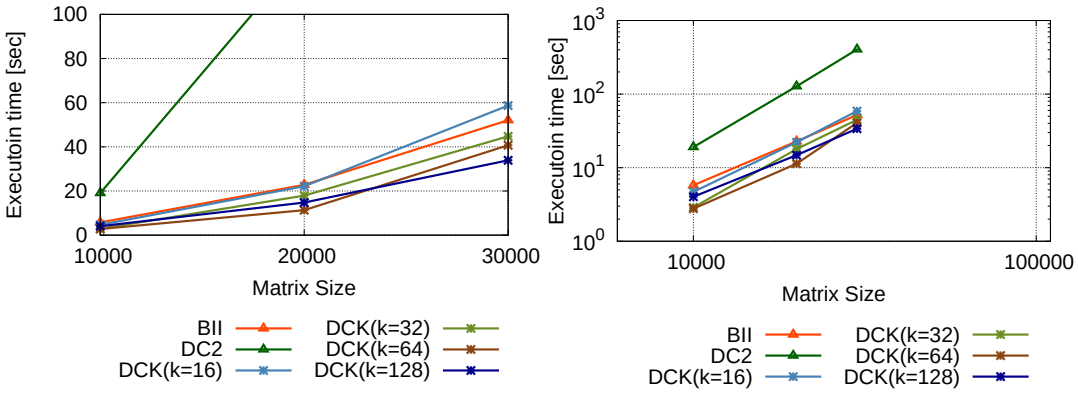

図 11　解法ごとの行列サイズに対する所要時間の変化 (行列 QC，$p = 16$)

化が行えた。一方，行列積 (`prod`) と再直交化 (`group`，`reortho`) の並列化手法は自明ではないが，本稿で提案した手法により良好な結果を得られた。

数値実験によると，deflation が少ない行列において DCK は ScaLAPACK の DC2 と比較して最短で 1/10 程度の時間で計算できており，本稿で提案した並列化手法が有効であることを示している。

なお，数値実験で他の解法と比較する際には適切な分割数を選択して行っているが，現実的にはいくつかの分割数で試してみなければ適切な分割数は分からない。そこで，適切な分割数を事前に求めることは課題の一つである。逐次ではこの問題に対して解決策が提案されている [7]。また，直交化計算 (`reortho`) に関してはまだ改善の余地がある。たとえば，本稿の実装ではグループごとに逐次的に行っている直交化を，プロセスに振り分けて行うことで小さなグループの直交化を全プロセスに分散させることができる。現在この作業は完了している。また再直交化において，グループ情報をまとめる操作を行う際の通信を木構造にすることで高速化できる可能性がある。このような再直交化の分散並列化の改善も今後の課題である。

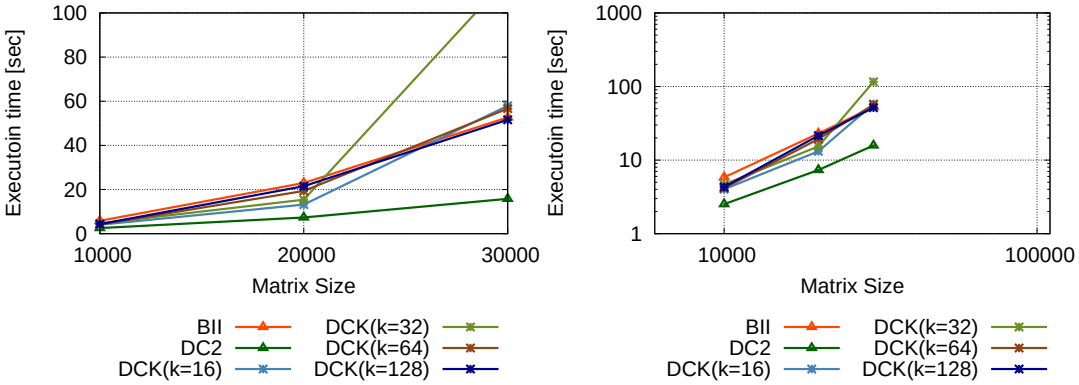

図 12　解法ごとの行列サイズに対する所要時間の変化 (行列 DL，$p = 16$)

6　謝辞

本研究は，東京大学情報基盤センターの若手利用者推薦制度の下で遂行し，HITACHI HA8000 を半年に渡り利用させていただきました。ここに謝意を表します。

7　情報基盤センターへの要望

HITACHI HA8000 では `ls` でファイルリストを表示したり，`vim` で小さなファイルを編集して保存するなどというときに，結果が表示されるまで時間がかかることがあった。そこでファイルの編集や列挙などの性能を改善してほしい。この要望に対しては，既にセンターは増強計画に基づき対策を進行しているため，センターの対処に感謝したい。

参考文献

[1] J. J. M. Cuppen, A Divide and Conquer Method for the Symmetric Tridiagonal Eigenproblem, *Numerische Mathematik*, 36, pp. 177–195 (1981).

[2] I. S. Dhillon, A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem, University of California at Berkeley, Berkeley, CA(1998).

[3] 桑島豊, 重原孝臣, 実対称三重対角固有値問題の分割統治法の拡張, 日本応用数理学会, Vol.15, No.2 (2005), pp. 89–115.

[4] 桑島豊, 重原孝臣, 実対称三重対角固有値問題に対する多分割の分割統治法の改良, 日本応用数理学会論文誌, Vol.16, No.4 (2006), pp. 453–480.

[5] L. S. Blackford et. al., ScaLAPACK Users' Guide, Society for Industrial and Applied Mathematics, Philadelphia, PA (1997).

[6] 田村純一, 坪谷怜, 桑島豊, 重原孝臣, 実対称固有値問題に対する多分割の分割統治法の共有メ

モリ型並列計算機における有効性, HPCS2009 論文集, pp.97–104 (2009).
[7] 石川祐輔, 田村純一, 桑島豊, 重原孝臣, 実対称固有値問題に対する多分割の分割統治法における準最適分割数の自動決定について, 第 38 回数値解析シンポジウム 講演予稿集, pp.17–20 (2009).
[8] OpenMP ARB, http://openmp.org/.
[9] Message Passing Interface Forum, http://www.mpi-forum.org/.
[10] T.H. Cormen, 浅野 哲夫 et.al., アルゴリズムイントロダクション 改訂 2 版 第 2 巻, 近代科学社 (2007).