ETC 順序による 3 重対角行列の並列ソルバー

寒　川　　光†

本論文では最近の高性能な算術演算パイプラインを備えたスカラプロセッサに適した，3 重対角行列の並列ソルバーのアルゴリズムを提案する．これは特に，2 重の算術パイプラインを備えたプロセッサで効果が大きい．このアルゴリズムは行列を並べ替えることで，両端（行列を格納した配列では初めと終わり）から真ん中に向かう順序を採用する．これにより演算量を増加することなく 2 ウェイの並列性が生まれる．この順序付けは再帰的に適用することができ，並列性は 2 のべきで増やしてゆけるが，演算量は増加する．インタフェースは現行の 3 重対角ソルバーを変更することなく実装できる．さらに単一の算術演算パイプラインしか持たない機種に対しても，並列性を通分に利用することで除算を減らせるので効果がある．4 ウェイの並列性は SMP 並列計算機向きである．また，この並列 3 重対角ソルバーのアルゴリズムの延長として，対称 3 重対角行列のスツルム列と行列式を並列計算するアルゴリズムを述べる．

A Parallel Tridiagonal Solver Based on ETC Ordering

Hikaru Samukawa†

In this paper we present a new parallel solver for tridiagonal matrices that exploits the potential of scalar pipelines in recent high-performance scalar processors, especially dual-pipeline processors. We use an ordering from both ends toward the center, which yields 2-way parallelism without increasing computational complexity. A k-times application of the ordering yields 2^k-way parallelism, but the complexity becomes greater. This ordering can easily be implemented in current tridiagonal solvers, because the locations of matrix elements in arrays are not modified. Furthermore, the ordering is effective not only for dual-pipeline processors, but also for single-pipeline processors, because it reduces the number of division operations by a common denominator technique. Four-way parallelism is effective for SMP parallel machines. As an extension of a tridiagonal solver algorithm, a parallel algorithm for calculating Sturm sequence and determinant of a symmetric tridiagonal matrix is described.

1. はじめに

高性能 3 重対角ソルバーは，偏微分方程式の数値解法で連立方程式をブロック SOR 法や，FACR 法のような FFT を応用した解法で用いられる．3 重対角行列を係数行列とする連立 1 次方程式 $T\boldsymbol{x} = \boldsymbol{r}$ を解く問題を考える．T を単位下三角行列 L と上三角行列 U の積の形に三角分解する．つまり $T = LU$,

$$\begin{pmatrix} a_1 & b_1 & & \\ c_2 & a_2 & \ddots & \\ & \ddots & \ddots & b_{n-1} \\ & & c_n & a_n \end{pmatrix} \tag{1}$$

$$= \begin{pmatrix} 1 & & & \\ l_2 & 1 & & \\ & \ddots & \ddots & \\ & & l_n & 1 \end{pmatrix} \begin{pmatrix} u_1 & b_1 & & \\ & u_2 & \ddots & \\ & & \ddots & b_{n-1} \\ & & & u_n \end{pmatrix}$$

である．分解アルゴリズムは，式 (1) の右辺の積を左辺と比較して，$u_1 = a_1$，また $i = 2:n$ について

$$l_i = c_i/u_{i-1},\ u_i = a_i - l_i b_{i-1} \tag{2}$$

となる．

係数行列は変更せずに，右辺ベクトル $\boldsymbol{r}$ だけを更新して方程式を繰り返し解く場合は，T を 1 回だけ三角分解し，代入計算だけを繰り返す．この場合は l_i と u_i は T の c_i と a_i を入力した配列に上書きされる．しかし方程式を 1 回だけ解けばよい場合には，三角分解のループと前進代入のループを融合することで，上書きを T の対角項だけにできる．本論文ではこの形を前提とする．またこの形を扱うサブルーチンを **3 重対角ソルバー**と呼ぶことにする．本論文で提案するアルゴリズムは前者に対しても適用可能であるが，その効果は小さくなる．またプログラム例のサブルーチンは，右辺ベクトル $\boldsymbol{r}$ は配列 $x(1:n)$ に与え，ここに解ベクトル $\boldsymbol{x}$ を上書きするインタフェースをとる．また演算は倍精度とする．

3 重対角ソルバーは典型的には次のプログラムで記述できる．この形のプログラムはよく使用されている

† 日本アイ・ビー・エム株式会社東京基礎研究所
IBM Research, Tokyo Research Laboratory

表 1 演算量と重み付き演算量
Table 1 Operation counts and weighted operation counts.

	*dtds*0	*dtds*1	*dtds*2c	*dtds*4
÷	$2n$	n	$0.5n$	n
∗	$3n$	$5n$	$7.5n$	$8n$
+, −	$3n$	$3n$	$3n$	$4.5n$
total	$8n$	$9n$	$11n$	$13.5n$
weight	$36n$	$23n$	$18n$	$27.5n$

（たとえば文献 1)）．本論文ではこれを dtds1 と呼ぶ．

```
subroutine dtds1
 a(1)=1.d0/a(1)
   do i=2,n
    r=c(i)*a(i-1)          ! M c*a
    s=a(i)-r*b(i-1)        ! MS a-r*b
    a(i)=1.d0/s            ! Div 1/s
    x(i)=x(i)-r*x(i-1)     ! ms x-r*x
   enddo
 x(n)=x(n)*a(n)
   do i=n-1,1,-1
    x(i)=(x(i)-x(i+1)*b(i))*a(i)
   enddo
```

はじめのループが三角分解と前進代入，2 番目のループが後進代入を行う．浮動小数点演算量を**表 1** に示した．“total” は合計，“weight” は除算に 15 倍の重みを掛けて評価した合計である．*dtds*1 は u_i の逆数を $a(1:n)$ に格納することで後進代入の除算を乗算に変えている．

プログラムのコメント欄に記した “M”，“MS” および “ms”，“Div” は，IBM の RISC System/6000（以下，RS/6000）の乗算，乗減算，除算命令を表す．なお RS/6000 では乗算と加算または減算を融合した複合演算命令を採用している[2]．コンパイラは XL Fortran の第 5 版を使用した．

このプログラムの 2 つのループはどちらもループ運搬依存性を持つため，連続する反復に現れる算術演算は逐次的に処理する必要がある．またはじめのループの 3 つの算術演算 “M c∗a”，“MS a-r∗b”，“Div 1/s” は，Div が MS に依存し，MS が M に依存するため，これらはパイプライン処理できず，クリティカルパスを形成する．“ms x-r∗x” は r が確定すればいつでも計算できる．このため，ループアンローリングは，算術演算命令に関するかぎり，効果はほとんどない．なお本論文ではこの乗減算のように，先行する命令とオーバラップ処理可能な命令を小文字で示すことにする☆．

ここで M に 2 クロック，MS に 2 クロック，Div に

☆ 生成されたコードの移動を考慮すると，どの命令が他のどの命令に依存するかを厳密に決定しにくいが，ここでは原始プログラムでの算術演算命令で，先行する命令に依存性がないものを小文字で表すことにする．

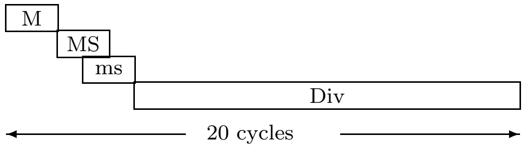

図 1 *dtds*1 のループ計算実行ダイアグラム
Fig. 1 Execution diagram of loop computation in *dtds*1.

15 クロックの遅延時間を仮定して，**図 1** に算術演算命令に対するダイアグラムを示した．なお除算は他の算術演算とオーバラップしない．図からループ反復 1 回に 20 クロックを要することが分かる．

Fortran コンパイラが単一の演算パイプラインを持つ機種に対して予測する実行サイクル数も 20 クロックであった．この場合，ロード，ストア，分岐命令の実行時間の大部分はクリティカルパスの裏に隠れると考えられる．ただしコンパイラの予測は，オペランドがキャッシュに存在する状態（装填キャッシュ状態）を仮定しているので，キャッシュミスをともなう場合は，ロード，ストア命令の実行時間が現れる．図から “このコードを，2 重の演算パイプライン（dual pipeline）を持つ機種で実行しても速くならない” ことが予想される．つまりこのような機種においては，パイプラインの稼動率は低い．本研究はこの低稼動率を改善する目的から出発した．

本論文で提案する方法は，領域分割による節点の順序付けに基礎を置くもので，両端から中央への順序（以下，“ends toward the center” の略として **ETC 順序**と呼ぶ）を使用する．これを 1 回適用すると，演算量を増加することなく 2 ウェイの並列性が生まれ，これを *dtds*1 のインタフェースを変えることなく実装できる．ETC 順序を 2 回適用すると，4 ウェイの並列性が生まれるが，演算量は 1.5 倍に増加する．

本論文では，2 章で既存の並列化アルゴリズムと提案方法の関係について簡単に述べ，3 章で ETC 順序とその代数表現を述べる．4 章で 2 重の演算パイプライン用，単一の演算パイプライン用，SMP 並列計算機用の実装について述べる．5 章で RS/6000 での計測結果について述べる．6 章でこの手法の延長として，3 重対角行列の重要な応用であるスツルム列の計算への応用を述べる．

2. 既存の方法との関係

3 重対角ソルバーの並列化アルゴリズムとして再帰的 2 重化法（recursive doubling method）と巡回縮約法（cyclic reduction method）が知られている．これらのアルゴリズムは “3 重対角行列の奇数行目を消

$$
\left(\begin{array}{cccc|cccc|cccc|cccc}
a_1 & & & g_1 & & & & & & & & & & & & \\
 & a_2 & & g_2 & & & & & & & & & & & & \\
 & & a_3 & b_3 & & & & & & & & & & & & \\
 & & & a_4 & & & & g_4 & & & & & & & & \\
\hline
 & & & c_5 & a_5 & & & g_5 & & & & & & & & \\
 & & & f_6 & & a_6 & & g_6 & & & & & & & & \\
 & & & f_7 & & & a_7 & b_7 & & & & & & & & \\
 & & & f_8 & & & & a_8 & & & & g_8 & & & & \\
\hline
 & & & & & & & c_9 & a_9 & & & g_9 & & & & \\
 & & & & & & & f_{10} & & a_{10} & & g_{10} & & & & \\
 & & & & & & & f_{11} & & & a_{11} & b_{11} & & & & \\
 & & & & & & & f_{12} & & & & a_{12} & & & & g_{12} \\
\hline
 & & & & & & & & & & & c_{13} & a_{13} & & & g_{13} \\
 & & & & & & & & & & & f_{14} & & a_{14} & & g_{14} \\
 & & & & & & & & & & & f_{15} & & & a_{15} & b_{15} \\
 & & & & & & & & & & & f_{16} & & & & a_{16}
\end{array}\right)
$$

図 2 分割法
Fig. 2 Partition method.

去して，未知数を半分にする操作” を基礎としている．この操作は $k = 1, \ldots, n/2$ について次式で表せる．

$$
\begin{aligned}
a'_k &= a_{2k} - c_{2k} a_{2k-1}^{-1} b_{2k-1} - b_{2k} a_{2k+1}^{-1} c_{2k+1} \\
b'_k &= -b_{2k} a_{2k+1}^{-1} b_{2k+1} \\
c'_k &= -c_{2k} a_{2k-1}^{-1} c_{2k-1}
\end{aligned} \qquad (3)
$$

この式は各 k について独立であるが，b'_k と c'_k がフィルインなので演算量は増える．

a'_k，b'_k，c'_k を係数行列とする次数 $n/2$ の 3 重対角行列に対して再び式 (3) を適用する．つまりこの操作を再帰的に $\log_2 n$ 段繰り返すのが再帰的 2 重化法である．最初の段には $n/2$ ウェイの並列性，次の段には $n/4$ ウェイの並列性と，並列性は段ごとに半分になり，記憶域アクセス（ストライド）は 2，4，8，... と倍々に増える．

係数行列を奇数行/列を Red，偶数行/列を Black として Red-Black 順序に並べ替えて分割すると，

$$
A \rightarrow \begin{pmatrix} A_{rr} & A_{rb} \\ A_{br} & A_{bb} \end{pmatrix}
$$

となる．A_{rr} と A_{bb} は対角行列である．これに対しシュール補元 $A'_{bb} = A_{bb} - A_{br} A_{rr}^{-1} A_{rb}$ を求める操作を考えると，これは式 (3) に一致する．この操作を再帰的に行うのが巡回縮約法である．

再帰的 2 重化法は複数のオペレーションを並列に動かすのに適しており，文献 3) に Illiac IV での実装例が論じられている．一方巡回縮約法はベクトル化に適しており，文献 4) に CDC STAR-100 での実装例が論じられている．どちらも問題のサイズを半分，半分にしてゆく再帰的なアルゴリズムであり，浮動小数点演算量は標準的な方法の約 2 倍になる．また n が 2 のべき乗でない場合の扱いが厄介である．

これを改良した方法として，分割法（partition method）が提案された[5)]．p ウェイの並列性を目的とする場合，$k = n/p$ として，係数行列を $k \times k$ の小行列に分割する．各小行列について並列に，まず c_i を消去するとフィルイン f_i が現れ，次に b_i を消去するとフィルイン g_i が現れる．この状態を $n = 16$，$k = p = 4$ の場合について**図 2** に示した．次に $k, 2k, \ldots$ 列の f_i を消去し，最後に g_i を消去する．右辺の代入はそれぞれのステップで処理する．$k = 2$ に選ぶと，g_i は現れず，巡回縮約法の 1 ステップと同じになる．分割法は前述の方法と比べると，並列度を固定でき，再帰的な操作を必要としない利点がある．

これらの既存の方法を 2 重の演算パイプラインを持つ機種のパイプライン稼動率を向上させる目的で使用しても，演算量の増加のために効果は期待できない．

本論文で提案する方法は，分割法の最後の小行列について，計算順序を逆転すると，フィルイン（図 2 では f_{14}，f_{15}，f_{16}，g_{12}，g_{13}，g_{14}）が現れないことを利用するものである．この方法では並列性と演算量がトレードオフの関係になるので，並列性が低い計算機を対象とした場合，計算機に合せた実装を行うことになる．並列性 2 の場合は演算量が増えず，並列性 4 の場合は，演算量は 1.5 倍の増加にとどまるので，分割法よりも有利である．

3. ETC 順序とその定式化

図 3 は，非オーバラップ型の領域分割を 1 次元領域に適用したものと考えられる．ここで左側の部分領域は 1 から m，右側の部分領域は n から $m+2$ に降順に番号付けしている．両者は境界領域である中点 $m+1$ によって分離される．計算を矢印で示したよう

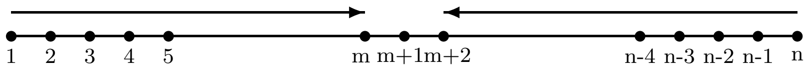

図 3 1 次元領域と ETC 順序
Fig. 3 One dimensional region and ETC ordering.

に両端から開始すれば，両者が中央でぶつかるまで，つまり部分領域の計算には 2 ウェイの並列性が存在する．

3.1 ETC 順序による係数行列

3 重対角ソルバーの定式化を行列を用いて示す．ただしこの節では n は奇数とする．はじめに $n=7$ の場合を示し，これを一般の場合に拡張する．$P_{i,j}$ を $n\times n$ の i 行目と j 行目あるいは i 列目と j 列目を交換する基本変換行列とする．行と列を交換して得られる行列 $R=P_{5,6}P_{4,7}TP^t_{4,7}P^t_{5,6}$ は，最後の行と列だけに連成項を持つ，2 つの 3 重対角行列 $T1$ と $T2$ を主小行列に持つ行列を作り出す．一般的な場合も，$R=P_{m+2,m+3}\cdots P_{2,n-1}\,P_{1,n}TP^t_{1,n}P^t_{2,n-1}\cdots P^t_{m+2,m+3}$ はやはり 2 つの 3 重対角行列 $T1$ と $T2$ を主小行列とする行列を生成する．ただし $m=\lfloor n/2\rfloor$ とする．$T2$ の分解は $T1$ の分解に依存しないので，$T1$ と $T2$ の分解は並列に計算できる．

$$
R=\left(\begin{array}{ccc|ccc|c}
a_1 & b_1 & & & & & \\
c_2 & a_2 & b_2 & & & & \\
 & c_3 & a_3 & & & & b_3 \\
\hline
 & & & a_7 & c_7 & & \\
 & & & b_6 & a_6 & c_6 & \\
 & & & & b_5 & a_5 & c_5 \\
\hline
 & & c_4 & & & b_4 & a_4
\end{array}\right)
=\left(\begin{array}{c|c|c}
T1 & & b_m \\
\hline
 & T2 & c_{m+2} \\
\hline
c_{m+1} & b_{m+1} & a_{m+1}
\end{array}\right) \quad (4)
$$

最後の行と列の非ゼロ要素の位置は，$T1$ と $T2$ の最後の位置に一致しているので，三角分解を行ってもフィルインは発生しない．すなわちこの順序に変更しても，3 重対角ソルバーの演算量は変わらない．

3.2 ETC 順序の繰返し適用

ETC 順序付けを，個々の部分領域に再帰的に適用することを考える．つまり，それぞれの部分領域を全領域と考えて，それらに ETC 順序付けを行うと並列性を 4 ウェイにできる．$n=15$ の場合は，前節に述べた R に $P_{4,7}$，$P_{5,6}$ および $P_{12,15}$，$P_{13,14}$ を適用する．ただしこの交換を行うと，三角分解により最後の行と列に $n/2$ 項のフィルインが発生するために演算量は増加する．**図 4** にこの順序と，対応する係数行列を $n=15$ の場合について示した．ただし n は “4 の倍数 +3” とする．3 重対角小行列は，$T1$，$T2$，$T3$，$T4$ の順に並び，各部分領域では矢印の方向に計算する．これらの分解計算には依存性がないので並列に行える．なお，図に加えた弧がフィルインにつながり，これらは分解された行列では f と d の位置に現れる．これらのフィルインが浮動小数点演算量を $9n$ から $13.5n$ flops にする．しかし表 1 に示したように，除算に 15 倍の重みを掛けて評価すると，増加は 20%程度で，並列化の効果が期待できる．

この操作を k 回再帰的に適用すると，2^k ウェイの並列性が得られるが，$(1-1/2^{k-1})n$ 項のフィルインが発生する．最大の並列性が得られる計算順序は，各 3 重対角主小行列のサイズが 1×1 のときで，式 (3) に一致する．

4. 実 装 方 法

ETC 順序は式の上では単純だが，これを素直にプログラミングしても，コンパイラとの関係から速くならないことが多い．また ETC 順序の効果を正確に評価しようとすると，記憶域アクセスの待ち時間も正確に分析する必要がある．ここでは 2 種類のプロセッサと XL Fortran を用いた性能分析に基づき ETC 順序を評価する．ここはこの孫サブルーチンを示す．

4.1 記憶域のアクセス

図 1 や**図 5** のダイアグラムは算術演算命令だけに着目して描いたものである．記憶域をアクセスするロード命令とストア命令の実行時間は，オペランドがキャッシュに存在すれば，算術演算命令の影に隠れて実行時間に現れないが，キャッシュに存在しなければ，キャッシュミスによる待ち時間が現れる．配列要素を連続参照する場合，たとえば RS/6000 の Power1 機種では，キャッシュラインサイズが 128 バイトで構成され，キャッシュミス 1 回で平均 11 クロック程度の待ち時間なので，$dtds1$ のように 4 つの 1 次元配列を連続アクセスする場合は，$4\times(11/16)=2.8$ クロック周期が，ループ反復 1 回で現れるキャッシュミスによる平均待ち時間である[2)]．この単純な予測では $dtds1$ の場合，20 クロックに対して 2.8 クロックなので，15%程度の影響がでることになる．

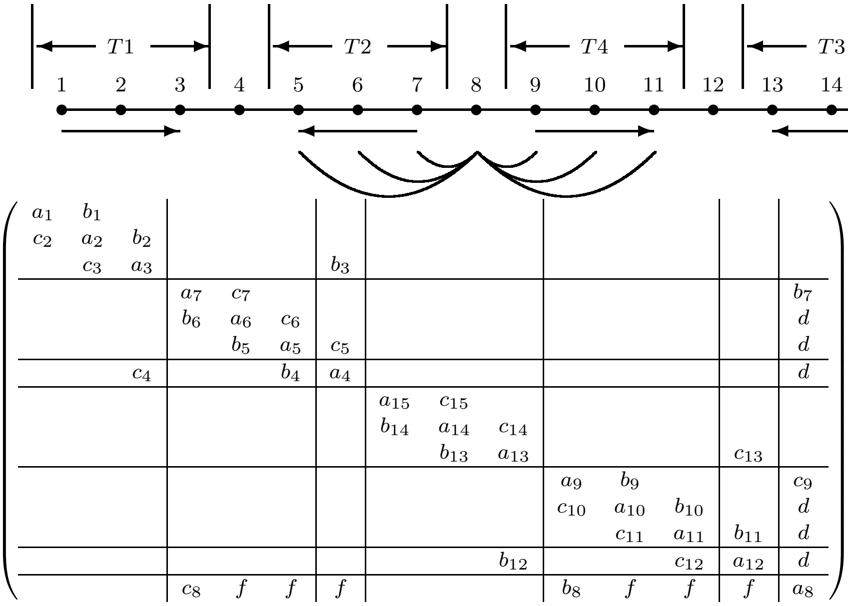

図 4 ETC 順序の 2 回適用
Fig. 4 Two times application of ETC ordering.

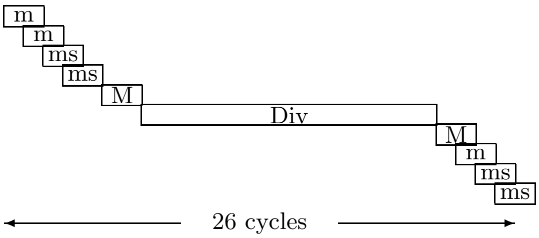

図 5 通分計算の実行ダイアグラム
Fig. 5 Execution diagram of common denominator.

計算に必要になるデータに対するロード命令を，実際に演算器がそれを使用するよりも十分に先行して発行することで待ち時間を消すプログラミング技法を**アルゴリズム・プリフェッチ**という[2)]．この技法は演算ユニットの仕事が，データのロードを行うユニットの仕事よりも多いときに有効である．3 重対角ソルバーは除算のために演算ユニットの仕事が重くなっており，プリフェッチの効果がある．

XL Fortran では，次のループ反復で使用するオペランドに対して，先行してロード命令を発行するためのダミーの実行文を挿入することができる．次の例 *dtds1p* では，ダミー実行文は仮引数に指定した変数 *da* などに配列要素を格納する処理を用いている．この処理はループ内部ではレジスタへのロード命令だけで，ループ外で仮引数へのストア命令が出される．

<u>*subroutine dtds1p*</u> $(\ldots, da, db, dc, dx)$

```
da=a(i+1)      !  dummy
db=b(i)        !  touch for
dc=c(i+1)      !  algorithmic
dx=x(i+1)      !  prefetching
r=c(i)*t
t=a(i)-r*b(i-1)
t=1.d0/t
a(i) =t
s=x(i)-r*s
x(i)=s
```

スーパスカラ計算機でには，ロード命令の実行が待たされても，そのオペランドを使用しない演算命令を追い越し実行（out-of-order execution）するものがある．この場合，追加されたロード命令がキャッシュミスを引き起こしても，その待ち時間は除算と重なるので，実際にその配列要素が必要になる次の反復では，データはキャッシュに装填されているか，あるいはミスしても待ち時間は短縮される．ダミーのロード命令の処理時間を短くする場合には，ループアンローリングと併用して，キャッシュラインに 1 つのロード命令にまで減らすことができる[2)]．*dtds1p* と *dtds1* を最適化レベル 2 でコンパイルして実測してみると，ハードウェアにプリフェッチ機能を持たない PowerPC 機種などでは約 10%の差が現れる．

しかし一時変数 t を使用した <u>*dtds1t*</u> では，最適化レベルを 3 にしてコンパイルすると，この差は現れない．

<u>*subroutine dtds1t*</u>

```
a(1)=1.d0/a(1)
t=a(1)
  do i=2,n
   r=c(i)*t              ! M c*a
   s=a(i)-r*b(i-1)       ! MS a-r*b
   t=1.d0/s              ! Div 1/s
   a(i)=t
   x(i)=x(i)-r*x(i-1)    ! ms x-r*x
  enddo
x(n)=x(n)*t
t=x(n)
  do i=n-1,1,-1
   t=(x(i)-t*b(i))*a(i)
   x(i)=t
  enddo
```

最適化レベル 3 でコンパイルすると，ソフトウェア・パイプライニングと 4 重のループアンローリングが適用され，はじめのループ内のすべてのロード命令と，そのオペランドを使用する演算命令の間に，少なくとも 3 つの除算を挟むコードが生成され，アルゴリズム・プリフェッチ以上に効率の良いコードが生成されるからである．また後進代入のループにも一時変数を適用すると $dtds1t$ 全体の性能を 10%近く向上させる☆．そこで以下，$dtds1t$ に対して ETC 順序を適用することにする．

4.2 擬装引数

ETC 順序で生成された $T1$ と $T2$ の分解と前進代入は 1 つのループで並行して計算できるが，これにはコンパイラの依存性解析を回避することが必要になる．次に示すプログラム例 <u>$dtds2$</u> では，仮引数 $a(1:n)$ に対し，これと同じ番地を持つ別の仮引数 $aa(1:n)$ も使用し，実行文のペア “$r0 =$” と “$r1 =$” が別の配列を参照するように見せかける（擬装する）．このコンパイラの場合，このループ計算は擬装なしでは “$a(i-1)$ と $a(ii+1)$ をロードして，$a(i)$ と $a(ii)$ にストアする” ので，$T1$ と $T2$ に関する 2 つの文脈に依存性があるかもしれないと判定し，逐次的に処理するコードを生成する☆☆．

<u>*subroutine dtds2*</u>

```
m=n/2                ! assume n is odd.
a(1) =1.d0/a(1)
 aa(n)=1.d0/aa(n)
 t0=a(1)
 t1=aa(n)
   do i=2,m
    ii=n-i+1
    r0=c(i) *t0
    r1=b(ii)*t1
    t0=1.d0/(a(i)  -r0*b(i-1))
    t1=1.d0/(aa(ii)-r1*c(ii+1))
    a(i)=t0
    aa(ii)=t1
    x(i) =x(i) -r0*x(i-1)
    x(ii)=x(ii)-r1*x(ii+1)
   enddo
  r0=c(m+1)*a(m)          !  *t0
  r1=b(m+1)*aa(m+2)       !  *t1
  x(m+1)=x(m+1)-r0*x(m)-r1*x(m+2)
  a(m+1)=1.d0/(a(m+1)-r0*b(m)-r1*c(m+2))
```

ループ外の実行文には依存性があるので注意が必要である．たとえば，“r1=b(m+1)*aa(m+2)” の “aa(m+2)” を “a(m+2)” としてはならない．コンパイラはこの場合，ループ計算にはソフトウェアパイプライニングを適用するので，最後の反復の後半，つまり $aa(m+2)$ へのストア命令はループ外へ回っており，これと “r1=...a(m+2)” のロード命令の順序が，最適化のコード移動によって逆転するからである．

4.3 並列性と通分

単一パイプラインのプロセッサに対しても ETC 順序は効果がある．これは依存性のない 2 つの除算は通分によって 5 回の乗算と 1 回の除算に置き換えられるからである[2]．これを $dtds2$ のループ内の 2 つの除算に適用したプログラムを <u>$dtds2c$</u> とする．

<u>*subroutine dtds2c*</u>

```
r0=c(i) *t0               ! m
r1=b(ii)*t1               ! m
s0=a(i)  -r0*b(i-1)       ! ms
s1=aa(ii)-r1*c(ii+1)      ! ms
rd=s0*s1                  ! M
rr=1.d0/rd                ! Div
t0=s1*rr                  ! M
t1=s0*rr                  ! m
a(i)  =t0
aa(ii)=t1
x(i) =x(i) -r0*x(i-1)     ! ms
x(ii)=x(ii)-r1*x(ii+1)    ! ms
```

コメント欄に $dtds1$ の場合と同様に算術演算命令を示した．依存性を持たない命令はパイプライン処理されるので 1 クロック周期で実行される．図 5 に装填キャッシュ状態での実行ダイアグラムを示したが，26 クロック周期で $dtds1$ の反復 2 回分を計算する．コンパイラの予測も 26 クロック周期であり，単純予測では $dtds1$ の $20/13 = 1.54$ 倍が期待できる．

通分を使用すると，最後のビットが不正確になる．

☆ 一時変数を用いると 4 ウェイのアンローリングが適用され，t はレジスタ渡しとなるため，もとのループ 1 回あたり 3 回のロード命令となり，ロード命令とそのオペランドを使用する算術演算命令は十分に離される．しかし一時変数を用いないと依存性の制約が残り，8 ウェイのアンローリングが適用されるが，これは 4×2 ウェイで，ロード命令は 3.25 回となる．冗長なロード命令は $x(i+1)$ に対するもので，このロード命令の直後にこれを使用する乗減算命令が置かれる．

☆☆ $a(ii)$ のストア命令の後で，次の反復の $a(i-1)$ のロード命令を置く．

この影響を調べるために，式 (1) に忠実な 2 回の除算を行う方法 <u>*dtds*0</u>，*dtds*1，ETC 順序に通分を適用した *dtds*2*c* を比較した．乱数を用いて係数行列と右辺ベクトルを生成して比較したところ，*dtds*0 と *dtds*1 では解ベクトルの要素のうち 27%に差異が現れ，*dtds*2*c* は 42%に差異が現れた．ブロック SOR 法のような緩和法の中で行う連立方程式の解法には，個々の連立 1 次方程式の解法が，より大きな近似計算の一部になっている場合が多い．このような場合は，通分によるこの程度の誤差は無視してもかまわないと考えられる．なお，表 1 に，*dtds*0，*dtds*2*c* の演算量を示した．

4.4 SMP 向き 4 ウェイ並列

ETC 順序を 2 回適用して，SMP 並列計算機用に 4 ウェイ並列性を持つプログラム <u>*dtds*4</u> を作る．ここではフィルインを計算するので，後進代入に図 4 の行列の最後の列の d を渡すための配列が必要になる．これには b，c の後進代入で使用されない要素（図 4 では b_4，b_5，b_6 と c_{10}，c_{11}，c_{12}）を利用するか，別途作業配列を用いる．ここでの説明は後者の方法を用いる（これを $d(1:n)$ とする）．プログラム例は *dtds*2 のループの最後に次の 4 行を追加したものが，$T1$ と $T2$ の分解計算に使用できる．

<u>*subroutine dtds4*</u>
```
d(ii)=-r1*d(ii+1)
f=-f*c(ii+1)*aa(ii)
sum=sum+f*d(ii)
sub=sub+f*x(ii)
```

同様のループプログラムが，$T3$ と $T4$ の分解計算に使用できる．これら 2 つのループを OpenMP の parallel sections 機能を用いて並列計算する．

*dtds*4 は 2 重のパイプラインのプロセッサを 2 つ持つ SMP 計算機に向いており，単一のパイプラインのプロセッサを 2 つ持つ SMP 計算機に対しては，通分を併用した *dtds*2*c* に対して同様の方法で並列化することができる．

5. 性能の測定

数値実験を単一パイプラインのプロセッサとして，PowerPC 機種の 604e（以下 ppc604e，332 MHz，最大性能 664 Mflop/s），2 重パイプラインのプロセッサとして，Power3 機種の RS/6000 の 43P 型モデル 260（200 MHz，最大性能 800 Mflop/s）を使用して行った．Power3 はハードウェアによるプリフェッチ機能を備えている．これはキャッシュミスが連続する（昇順または降順）キャッシュラインで発生する場合は，次のキャッシュラインにプリフェッチ要求を出す（プリフェッチバッファにロードする）．そして後続のキャッシュミスのアドレスがプリフェッチバッファ内のキャッシュラインにあればこれを使用するとともに，さらにその次のキャッシュラインにプリフェッチ要求を出す．キャッシュミスを発生する複数のアドレスから，プリフェッチストリームを生成するのは，フィルタリングによる．これにより最大 4 つのストリームに対してプリフェッチが機能する[6]．

表 2 3 重対角ソルバーの性能（Mflop/s）と比
Table 2 CPU time in seconds (The ratio to the time for the *dtds* program is shown in parentheses).

	ppc604e	Power3
*dtds*0	22.0 (0.72)	32.7 (0.85)
*dtds*1	30.4 (1.00)	38.1 (1.00)
*dtds*1*t*	35.8 (1.18)	52.2 (1.36)
*dtds*2	32.5 (1.07)	77.6 (2.02)
*dtds*2*c*	37.7 (1.24)	71.8 (1.87)

5.1 逐次計算

測定結果を**表 2** に示す．行列のサイズは 500 から 2048 の間で 9 通りを使用しその平均速度を，演算量を $9n$ として Mflop/s で求めた．コンパイラは XL Fortran 第 5.1 版を，最適化レベル 3 で使用した．

表 2 の Power3 の *dtds*1 と *dtds*2 の比から，ETC 順序は 2 重パイプラインの機種では，2 倍を超える効果を示すことが分かる．しかし *dtds*1*t* と *dtds*2 の比で見ると効果は 1.5 倍（=2.02/1.36 倍）に圧縮されている．これは *dtds*1*t* は記憶域アクセスが単純で，コンパイラの最適化やハードウェアのプリフェッチ機能が奏効しているのに対し，*dtds*2 ではプログラムの複雑化によりレジスタが不足して最適化が不十分であり，またプリフェッチ機能も 4 つのプリフェッチストリームでは不足するため，キャッシュミスによる遅れが現れているためである．

図 6 に，*dtds*1，*dtds*1*t*，*dtds*2，*dtds*4 の Power3 機種の 43P モデル 260（2 ウェイ SMP モデル）による測定結果を示した．*dtds*4 は並列計算であるが，他は逐次計算による．図は横軸に n を $2^6 = 64$ から $2^{16} = 65536$ について対数スケールで，縦軸に Mflop/s 性能値を（$9n$ を経過時間で割った値をリニアスケールで）示した．縦軸に添えた数値はそれぞれの方法で計測された最高性能値である．また *dtds*2 の上の小さな黒丸は，*dtds*2 にプリフェッチ命令を挿入したプログラムによる値である．

*dtds*1 と *dtds*1*t* の性能は n にほとんど依存せず，それぞれ 38 Mflop/s と 52 Mflop/s である．*dtds*2 は n によって変化し，最高値は 81 Mflop/s である．性能が低下する理由は，$n = 2500$ までは 1 次キャッシュ（サイズ 64 KB）に係数行列と右辺ベクトルが収まる

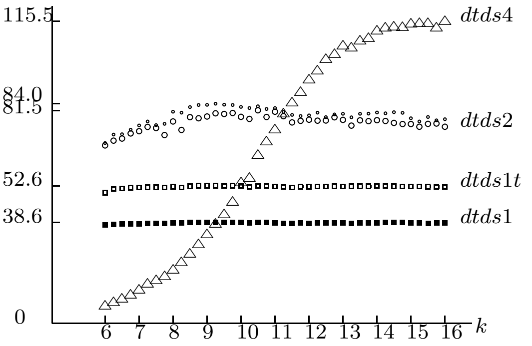

図 6 $dtds1$，$dtds1t$，$dtds2$，$dtds4$ の行列のサイズ $n = 2^k$ に対する性能（Mflop/s）

Fig. 6 Performance of $dtds1, dtds1t, dtds2, dtds4$ vs. matrix size $n = 2^k$ (Mflop/s).

ため，キャッシュミスを発生せずに後進代入を完了できるが，これを超えると後進代入の最終段階でキャッシュミスが始まるためである．$dtds1$ や $dtds1t$ ではハードウェアのプリフェッチ機能でこの影響は現れないが，$dtds2$ では後進代入に 6 つのプリフェッチストリームが必要となり，ハードウェアの機能ではカバーしきれない．

キャッシュミスの影響をどの程度小さくできるかを調べるために，$dtds2$ サブルーチンを，コンパイラの生成したアセンブラ・ソースにプリフェッチ命令（dcbt: data cache block touch 命令）を挿入してアセンブルしたプログラムによって置き換えて測定した結果を合わせて示した．この結果から，コンパイラの機能の充実によってさらに 7%程度，つまり $dtds1t$ の 1.6 倍程度の性能が期待される．

また単一のパイプラインの機種に対しては，表 2 の ppc604e の結果から，ETC 順序は通分と組み合せなければ効果がないことが分かる．

5.2 SMP 並列計算

図 6 の $dtds4$ の最高性能は 115 Mflop/s に達するが，これは SMP 並列化のオーバヘッドのために，n が 8000 という大きな値でである．また $dtds2$ とのクロスオーバも n が 2400 の近辺にある．$dtds4$ の $dtds2$ に対する性能は，最高性能値で比べても 1.4 倍程度と低いが，これは Mflop/s 値を $9n$ と固定した分子で計算していることの影響が最も大きい．除算の重みを 15 とした演算量によれば，両者の仕事には 20%程度の差があるので，これを考慮すると 1.7 倍程度，つまり並列化効率は 0.85 程度になり，必ずしも悪い値ではない．

6. スツルム列の計算

ETC 順序を，3 重対角行列の重要な応用である，対称 3 重対角行列の主小行列の行列式を計算する問題，すなわちスツルム列を数えることで，この行列の負の固有値の数を求める計算に適用する．

式 (1) の c_i を b_{i-1} によって置き換えると，対称行列用の三角分解 $u_1 = a_1$，$l_i = b_i/u_i$，$u_i = a_i - b_{i-1}^2/u_{i-1}$ が得られる．T_i を T の $i \times i$ の主小行列とする．$det(T_i) = u_1 u_2 \cdots u_i$ なので，行列式は次の漸化式から得られる[7]：

$$det(T_i) = a_i \cdot det(T_{i-1}) - b_{i-1}^2 \cdot det(T_{i-2}). \quad (5)$$

行列式は相似変換 $P_k T P_k^t$ によって不変なので，変換 $R = P_{m-1} \cdots P_2 P_1 T P_1^t P_2^t \cdots P_{m-1}^t$ によって作り出される式 (4) の 2 つの 3 重対角行列 $T1$ と $T2$ の行列式に着目する．$u_{m+1} = a_{m+1} - b_m^2/u_m - b_{m+1}^2/u_{m+2}$ を $det(T) = det(T1)det(T2)u_{m+1}$ に代入する：

$$\begin{aligned} det(T) &= det(T1)det(T2)a_{m+1} \\ &\quad - det(T2)det(T1_{m-1})b_m^2 \\ &\quad - det(T1)det(T2_{m-1})b_{m+1}^2. \end{aligned} \quad (6)$$

これから，除算なしに対称 3 重対角行列の行列式を並列計算するアルゴリズムが得られる．

このアルゴリズムから，T の負の固有値の数を，$det(T1_i)$ と $det(T2_i)$ の符号変化の回数を並列に数えることで得ることができる．R_{n-1} の負の固有値の数は，$T1$ と $T2$ の負の固有値の数の和である．したがって，$det(R)$ の符号が $det(R_{n-1})$ の符号と異なれば，符号変化の回数に 1 を加え，同じなら加えない．この最後の調整は u_{m+1} の符号を調べて行う．

対称 3 重対角行列の固有値を求める 2 分法では，対象区間に固有値が複数存在する場合は，主小行列の符号変化の回数を数えなくてはならないが，区間に 1 つしか固有値が存在しない状態では，行列式だけ計算すればよい．これは行列式の正/負でスツルム列の 0/1 が判定でき，また，行列式の値を利用した行列式法を 2 分法と組み合わせることで，収束を加速できるからである[8]．多項式 $d_i(w) = det(T_i - wI)$ に対する漸化式は $d_i(w) = (a_i - w)d_{i-1}(w) - b_{i-1}^2 d_{i-2}(w)$ である．この式に従って漸化式を反復ごとに 1 つずつ計算するプログラムを <u>det1</u> とする．ここでは行列式の値の桁あふれを毎回チェックする．$det1$ に 8 重のループアンローリングを適用した <u>det8</u> を示す．

```
do i=i1,n,8
 d3=(a(i)  -w)*d2-bb(i)  *d1  ! s, m, MS
 d4=(a(i+1)-w)*d3-bb(i+1)*d2
 d5=(a(i+2)-w)*d4-bb(i+2)*d3
 ............................
 d1=(a(i+6)-w)*d8-bb(i+6)*d7
 d2=(a(i+7)-w)*d1-bb(i+7)*d8
  if(abs(d2).gt.2.**256) then
    ---( Normalize)---
```

表 3 行列式の性能（Mflop/s）と比
Table 3 CPU time in seconds (The ratio to the time for the *dtds* program is shown in parentheses).

	ppc604e	Power3
$det1$	25.1 (1.00)	30 (1.00)
$det8$	37.6 (1.50)	132 (4.40)
$det82$	38.0 (1.51)	181 (6.03)
$det8p$	56.0 (2.23)	204 (6.80)

```
elseif(abs(d2).gt.2.**(-256)) then
  ---( Normalize)---
```

ここで $bb(1:n)$ には b_i^2 が，w には多項式 $d_i(w) = det(T_i - wI)$ の変数 w が格納されている．こうすると算術パイプラインの稼働率が向上し，また桁あふれのチェックも 8 分の 1 に減らすことができる．ETC 順序を適用して $T1$ と $T2$ に対して 1 つのループで 8 ウェイのアンローリングで計算するプログラムを $\underline{det82}$，OpenMP の parallel sections 機能を用いて 2 つのループでそれぞれ 8 ウェイのアンローリングで計算するプログラムを $\underline{det8p}$ とする．これらのプログラムを 604e と Power3 プロセッサで，2 ウェイの SMP 環境で実測した．なお $n = 10^6$ と十分大きくして計測した（**表 3**）．ETC 順序は 2 重パイプラインの機種では逐次処理から効果を発揮する．しかし単一パイプラインの機種では効果がなく，ETC 順序は SMP 並列環境で使用すべきである．

7. おわりに

本論文では 3 重対角行列に対して，ETC 順序に基づくソルバーと行列式の計算法を提案した．この方法は並列性が 2 の場合演算量を増加させない．既存の巡回縮約や分割法では，演算量は並列性をいくつに設定しても，演算量は約 2 倍になり，並列度が 2 とか 4 という小さな値の計算機環境に対しては有効でない．ETC 順序によるアルゴリズムの効果を評価するためには，計算機環境に特有の要因を排除して評価する必要がある．この要因としては，コンパイラの最適化，キャッシュミスによる遅れ，演算パイプラインの構成，除算に要する時間などがあげられる．本論文では RS/6000 の 2 つの機種を用いて，これらの要因の分析を行い，その結果として，ETC 順序の効果を確かめた．現実には $dtds1$ に類似したコードが広く使用されていると考えられるが，ETC 順序によると，2 つのパイプラインを持つ単一プロセッサでは，2 倍を超える性能が，またこのプロセッサを 2 つ備えた SMP 並列計算機では 3 倍の性能が確認された．また，ETC 順序を行列式の計算に応用する方法にも効果があることを確認した．

参考文献

1) 荒川忠一：数値流体工学，東京大学出版会 (1994).
2) 寒川　光：RISC 超高速計算法，共立出版 (1995).
3) Stone, H.S.: Parallel Tridiagonal Equation Solvers, *ACM Trans. Math. Softw.*, Vol.1, No.4, pp.289–307 (1975).
4) Lambiotte, J.J. and Voigt, R.G.: The Solution of Tridiagonal linear systems on CDC STAR-100 Computer, *ACM Trans.Math.Softw.*, Vol.1, No.4, pp.308–329 (1975).
5) Wang, H.H.: A Parallel Method for Tridiagonal Equations, *ACM Trans.Math.Softw.*, Vol.7, No.2, pp.170–183 (1981).
6) Andersson, S., Bell, R., Hague, J., Holger, H., Mayes, P, Nakano, J., Shieh, D. and Tuccilo, J.: *RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide*, IBM Corp., SG24-5155 (1998).
7) Golub, G.H. and Van Loan, C.F.: *Matrix Computations*, Third edition, Johns Hopkins Univ. Press (1996).
8) 寒川　光：RISC 計算機に適した 3 重対角行列の固有値の計算法，情報処理学会ハイパフォーマンスコンピューティング研究会，No.59, pp.37–42 (1995).

(平成 12 年 8 月 29 日受付)
(平成 13 年 1 月 11 日採録)

寒川　光（正会員）
1972 年早稲田大学理工学部機械工学科卒業．同年日本ユニバック（株）入社．1984 年日本アイ・ビー・エム（株）入社．現在東京基礎研究所勤務，金沢工業大学連携大学院客員教授兼任．数値解析，数値計算法，計算機アーキテクチャ，専用計算機に関する仕事に従事．工学博士．計算工学会，日本シミュレーション学会各会員．