社団法人 情報処理学会　研究報告
IPSJ SIG Technical Report

2004－HPC－99　(16)
2004／7／31

性能安定型LAPACKの自動生成の試み

今　村　俊　幸†

著者らは, 数値計算ライブラリに登場する典型的なループ構造の性能を安定化するアルゴリズム (C-Stab) を考案してきた. C-Stab の基本的な考え方は各配列のオフセットと整合寸法 (配列の一次元目のサイズ) を制御することで, キャッシュ内における配列データの格納場所を強制的に同周期・異位相の状態に保ち, キャッシュ競合を回避する方法である.

本論文では, 基本線形演算インタフェース BLAS によって構成される LAPACK に対して適用することを考える. 特に, C-Stab を適用することで, 新たに導入される補助配列のための演算をどの様に生成するのか, 補助配列をどの位置に配置すべきか, その他, コードを自動生成する際の知見を得ることを目的とする.

Towards auto code-generation of a stabilized LAPACK

Toshiyuki Imamura†

The C-Stab algorithm, which stabilises typical two types of cache instability was investigated for development of a numerical library. The basic idea of the C-Stab algorithm is dynamic refinement of matrices, particularly their offset and the first dimension, and furthermore to keep the access pattern coherent in order to avoid cache conflict.

This papers covers the review of the C-Stab algorithm, and the application to LAPACK which is originally constructed by BLAS. Through preliminary works, several technical points for the auto code-generation can be obtained, for example conversion rules from an original code to a C-stabbed code, data layout of the auxiliary arrays, and so on.

1. はじめに

数値計算ライブラリの品質を保証する指標の一つとして欠くことの出来ない「速度性能」が存在する. 高価でかつ高性能なシステムを用いて計算する場合にその指標が重要視されることが多い. 一方で, ハードウェアの利用サイクルとソフトウェアのライフサイクルを比較した場合に, ハードウェアが新システムに変わってもソフトウェアは基本的にはそのまま使われ続けるというライフサイクルの極端な差異が生じている. 一般的に, 数値計算ライブラリに代表される高品質ソフトウェアはハードウェアに特化したチューニングを行いハードウェアの最高性能を達成するように開発されている. この極端なライフサイクルの違いは同一ソフトウェア, 特に科学技術計算ソフトウェアの核をなす数値計算ライブラリにとっては常にハードウェアの進展に応じて内部を最適化していかなくてはならない事を意味している.

† 電気通信大学電気通信学部情報工学科
Department of Computer Science, The University of Electro-Communications

この様に早急な最適化の要求に対して, 自動化チューニングもしくは経験的な最適化をパラメータ化してソフトウェアに組み込む適応的ライブラリの形が一般化しつつある[1)~5)].

ソフトウェアの最適化・チューニングを進めるにおいては, その目標関数が実行時間の最小化であり, 基本アルゴリズムを固定した場合はコードの実装によって生じる性能の差異に帰着される. しかしながら, 近年のハードウェアの複雑化と相まって, 高度な最適化技術を併用するとマイナスの相乗効果によって性能の不安定性が生じることが, 一般に知られている. 特に, キャッシュベースのアーキテクチャを利用する場合には, プログラムの実行毎に性能が大きく変動することがある. また, ライブラリ開発の立場から見ると, 積極的なチューニングによって性能の一部が極端に劣化するケースも存在する. この様に, 速度性能という側面だけではなく実質的なソフトウェアの品質として速度安定性という考え方も重要と考えられる. 本論文は先行研究 6)~8) で培われてきた性能安定化技術を一般ソフトウェアに応用するための足がかりとして, LAPACK への適用を行い一般化への知見, プログラ

ムコードを自動的に安定化するための第一ステップとする.

2. 性能安定化について

2.1 キャッシュ競合の判別について

データがキャッシュのどの位置 (セット) に対応するかはアドレスの特定のビット位置から一意に決定される.☆ 連続アドレスが round-robin 方式にキャッシュに割り当てることを考慮して, 次に定義する位相 (phase on cache) を導入する.

$$[\mathrm{I}]_L := \mathrm{mod}(\mathrm{I} + 2^{L-1}, 2^L) - 2^{L-1} \quad (1)$$

$$^*\mathrm{A} := \&\mathrm{A}/2^d \quad (2)$$

ここで, &A は A のアドレスを示し, $L = s+l-d, d = \log_2 \mathrm{sizeof(double)}$ とする.

この位相を利用して文献 8) で, 数値計算に現れる典型的なループにおけるループアンローリング最適化によって生じるキャッシュ競合の判別式を示している.

[セット競合の判別式]

n-way set associative キャッシュを搭載したプロセッサを用いたシステム上でのループ実行を仮定する. 整合寸法が N_a の 2 次元配列 a に対して, 以下の不等式が成立するとき,

$$0 < \exists i < k/n,\ |[i * N_a]_L| < \delta, \quad (3)$$

配列 a の上位インデックスを k 段展開するアンロールは, 実行時にセット競合を生じさせる.

2.2 C-Stab: 安定化安定化アルゴリズム

上記判別式が成立する場合に性能の大きな劣化が予想されるため, それを排除する必要がある. 判別式を支配する基本項は配列の整合寸法 (1 次元目のサイズ) であり, それを制御すればよい. 文献 8) では, ヒューリスティクス法や位相を考慮した手法を組み合わせたハイブリッドアルゴリズムを提案しており, 本論文ではそのアルゴリズムを C-Stab アルゴリズム (以下単に C-Stab) と呼ぶ. 本節では, 文献 8) で示したものに L2,L1 の階層を意識した配列の配置方法を導入した新たな C-Stab アルゴリズムを紹介する.

[C-Stab]

(0) ループ内に登場する配列を, $V_1, V_2, \ldots, V_m$, それぞれの整合寸法を $N_1, N_2, \ldots, N_m$ とする. 但し, $m \leq 2^{L-l+d}$, $p = \lceil \log_2 m \rceil$ とする.

(1) $N'_j := N_j$ とおく.

(2) 位相構成法の基本周期を $f = 2^{p+l}$ とし, 各配列の整合寸法を f の倍数に調整する.

$$N'_j := \lceil N'_j / f \rceil * f \quad (4)$$

(3) 判別式 (3) を満足する場合は, 次式により整合寸法を決定しステップ (2) に戻る.

$$N'_j := N'_j + \lceil (\delta - [i * N'_j]_L)/i \rceil \quad (5)$$

(4) $[^*\mathrm{V_j}]_L = \sigma(j) \cdot (f/2^p)$ となるように, 配列個々のオフセットを決定する. ビット関数 $\sigma(j)$ の構成方法については後述する.

(5) $V_1, V_2, \ldots, V_m$ をそれぞれ整合寸法 $N'_1, N'_2, \ldots, N'_m$ で再構成する. これ以降, もともとの配列 V_a を意味するときは補助配列 V'_a と併せて (V_a, V'_a) の形で表記する. 補助配列が不要な場合には $V'_a = \emptyset$ とする.

(6) $(V_1, V'_1), (V_2, V'_2), \ldots,$ を用いた形に変形したコアループを実行する.

(7) 出力が必要な変数について, $(V_a, V'_a) \rightarrow V_a$ の再々構成を行う.

ここに示したアルゴリズムはライブラリのインターフェイス部分と本体部分との間で機能するものである. また, 安定化アルゴリズムによって補助配列の導入により, ライブラリ本体に補助配列を処理するコードを追加しなくてはならない. 本研究の主眼はこの部分であり, 従来コードが与えられたときに安定化コードを如何に生成するか, コード自動生成のための知見を深めることにある. 次章以下, 実コードに基づく安定化コード生成のポイントを整理していく.

[補足: ビット関数 $\sigma(j)$ について]

関数 $\sigma(j)$ は配列間の開始アドレスを調整するためのオフセットを決める関数である. 各配列の開始アドレスがキャッシュ内でなるべく離れて位置するように $\sigma(j)$ を次の様に定義する.

$$\sigma(j) := \mathrm{rev}(j, s_2) \cdot \mathrm{rev}(j, s_1) \cdot \mathrm{rev}(j, s_0) \quad (6)$$

ここで, rev(a,b) はビット長 b で表現される a の 2 進数表現の上位/下位ビットを反転させてできる数を表す. '·' はビットの結合を意味し, 左側を上位ビットとする. 定数は $s_0 = L - l + d, s_1 = C_1 - L - d, s_2 = C_2 - C_1$ ($2^{C_1}, 2^{C_2}$ はそれぞれ L1,L2 キャッシュのサイズ).

3. LAPACK の性能安定化

3.1 構成要素としての BLAS の安定化について

LAPACK は主に, 線形演算の中心部分を BLAS(Basic Linear Algebra Subprograms)[9] によって構成され, プログラムの可読性, 移植性, 可搬性, そして性能チューニングを含めて保守性を高めることに成功している.

☆ 一般に TLB によるアドレス変換のタイミングによって仮想アドレッシングと物理アドレッシングが考えられるが, ここでは議論を簡単化するため前者を選択する. なお, 後者では厳密な位相の議論はできないが, 連続する大きな物理メモリが仮想ページに割り当てられればその範囲では区分的に位相を議論できる. 現実的にはループ内の数インスタンスのみを考えればよいのでそれで十分である.

本節では, BLAS レベルでの性能安定化についてその可能性を考えてみる.

BLAS は被演算対象となるベクトル, 行列の組み合わせに応じてその演算量が $O(N), O(N^2), O(N^3)$ と変化する. その演算量の次数に対応させて Level 1 から Level 3 までが実装されている. 性能安定化では整合寸法配列の調整を行うので, 行列を被演算対象とする Level 2 以上で対応しなくてはならない. ところで, 整合寸法の調整は全配列要素 (行列要素) の並べ替えを意味するため, そのコストは $O(N^2)$ となる. したがって Level 2 に対して行うことは演算量と同程度のオーバヘッドを生じるので, それ単独に行うのではなく Level 2 を含むより上位のサブルーチン単位で性能安定化を行うことが望ましい. 一方, 演算量が $O(N^3)$ の Level 3 に対して性能安定化を行うことは性能安定化のオーバヘッドが $O(N^2)$ という点からも許容できる. ただし, DSYR2K などの様に一般的には正方行列と正方行列の演算ではないものも含まれるため, 演算量の面から見て適用には注意が必要である.

また, BLAS 本来の目的である保守性を含めて考慮したときに, 既存の極めて信頼性の高い BLAS 実装ライブラリに安定機構を付加する形での実装になるためソースに手を加えることの出来ない一部商用ルーチンでは呼び出し機構に工夫が必要となり, その分のソフトウェアとしての信頼性に問題が生じる可能性がある. したがって, BLAS レベルではなく（可能であれば）上位サブプログラムでの性能安定化つまり配列データの調整を行うべきである.

3.2 DSYTRD の安定化

次に, LAPACK の中に数あるルーチンから, 著者がこれまでに開発の経験があるアルゴリズムのサブプログラムの性能安定化を試みる. 具体的には, 固有値求解の際に用いる三重対角化ルーチン DSYTRD を例にして, LAPACK の性能安定化についてのコード変換実験を行う. なお, この実装方針は BLAS の性能安定化の一結論である BLAS を呼び出すサブプログラムレベルでの性能安定化に対応する.

まず, DSYTRD で使用されている基本アルゴリズムについて説明する. DSYTRD は対称行列をハウスホルダー変換により三重対角行列に変換するものである. 特に計算効率を上げるためにブロックアルゴリズムを採用している. アルゴリズムを大きく二段階に分けると, ハウスホルダー変換のためのリフレクターを計算する**部分ハウスホルダー変換**と M 本のベクトルによる **2M ランク更新**に分けられる. アルゴリズムの詳細は図 1 に従う.

for $j = N, \ldots, 1$ **step** $-M$
  $U \leftarrow \emptyset, V \leftarrow \emptyset, W \leftarrow A_{(*,j-M+1:j)}$
  for $k = 0, \ldots, M-1$
    (1) $W_{(*,j-k)} \leftarrow W_{(*,j-k)} - (UV^T + VU^T)_{(*,j-k)}$
    (2) Householder reflector $u^{(k)} = H(W_{(*,j-k)})$ の計算
    (3) 行列ベクトル積計算 $v^{(k-\frac{2}{3})} \leftarrow A_{(1:j-k-1,1:j-k-1)}u^{(k)}$
    (4) $v^{(k-\frac{1}{3})} \leftarrow v^{(k-\frac{2}{3})} - (UV^T + VU^T)u^{(k)}$
    (5) $v^{(k)} \leftarrow v^{(k-\frac{1}{3})} - ((u^{(k)}, v^{(k-\frac{1}{3})})/2|u^{(k)}|^2)u^{(k)}$
    ここで $U \leftarrow [U, u^{(k)}], V \leftarrow [V, v^{(k)}]$ とブロック化する
  endfor
  $A_{(*,j-M+1:j)} \leftarrow W$
  (6) 2M ランク更新 $A_{(1:j-M,1:j-M)} \leftarrow A_{(j-M:j-M)} - (UV^T + VU^T)_{(j-M,j-M)}$
endfor

図 1 ブロックハウスホルダー変換 (DSYTRD) のアルゴリズム. なお, 図中ではアルゴリズムの理解を容易にするために中間変数 U, V, W を用いているが, DSYTRD では対応するベクトルを作業配列や行列 A の空き領域に重ねて使用している.

アルゴリズムの中心となる部分ハウスホルダー変換は図 1 中の (1) から (5) を含むループ, また 2M ランク更新は (6) に対応し, それぞれ関数 DLATRD, DSYR2K によって実装されている. DSYR2K は Level 3 ルーチンであり, DLATRD 内では Level 2 ルーチンである DSYMV, DGEMV がコールされている.

これら 3 つの BLAS ルーチンに対して, 性能安定化を施す際にまず考慮しなくてはならないのは, 被演算要素行列を $A = (A_1, A_2)$ の様に 2 つの記憶領域によって一つのデータを表現する場合にもともとの演算の内容を (数学的な意味で) 変更しないよう, BLAS ルーチンで再構成する必要がある.

これら 3 つのサブプログラムの BLAS による再構成方法を以下に示す.

3.2.1 DGEMV

行列 A が補助配列導入によって (A_1, A_2) となる場合. ベクトル $\boldsymbol{x}$ も同様のサイズに分割すれば,

$$\boldsymbol{y} := \alpha A\boldsymbol{x} + \beta\boldsymbol{y}$$

$$:= \alpha \overset{n_1 \quad n_2}{\begin{pmatrix} A_1 & A_2 \end{pmatrix}} \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \end{bmatrix} + \beta\boldsymbol{y}$$

$$:= \alpha A_2\boldsymbol{x}_2 + (\alpha A_1\boldsymbol{x}_1 + \beta\boldsymbol{y})$$

元々の演算は上式の様に分けられ, それぞれを BLAS ルーチンで表現することが出来る.

```
call DGEMV(Trans, M, N,  &
                Alpha, A, LDA,  X, INCX, &
                Beta,  Y, INCY)
```

⇓

```
J=N1+1
call DGEMV(Trans, M, N1, &
      Alpha, A1(1,1),LDA1, X(1)        ,INCX, &
      Beta,  Y(1),INCY)
call DGEMV(Trans, M, N2, &
      Alpha, A2(J,1),LDA2, X(1+N1*INCX),INCX, &
      One,   Y(1),INCY)
```

3.2.2 DSYMV

実対称行列 A が補助配列導入によって (A_1, A_2) となる場合. 同様に $\boldsymbol{x}$, $\boldsymbol{y}$ も分割すると,

$$\boldsymbol{y} := \alpha A\boldsymbol{x} + \beta\boldsymbol{y}$$

$$:= \alpha \overset{n_1 \quad n_2}{\begin{pmatrix} A_1 & A_2 \end{pmatrix}} \boldsymbol{x} + \beta\boldsymbol{y}$$

$$:= \alpha \begin{pmatrix} A_{11} & A_{12} \\ A_{12}^T & A_{22} \end{pmatrix} \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \end{bmatrix} + \beta \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \end{bmatrix}$$

と変形できる. ここで, DSY 系の関数では主引数の行列は上ないしは下三角要素しか有効なデータを保持しないため, 分割によって生成される対角要素 A_{11} や A_{22} は DSY 系の関数で処理しなくてはならない. この条件を考慮すると, 以下の様に BLAS での再構成が可能となる.

$$\boldsymbol{y}_1 := \alpha A_{12}\boldsymbol{x}_2 + (\alpha A_{11}\boldsymbol{x}_1 + \beta\boldsymbol{y}_1)$$

$$\boldsymbol{y}_2 := \alpha A_{12}^T\boldsymbol{x}_1 + (\alpha A_{22}\boldsymbol{x}_2 + \beta\boldsymbol{y}_2)$$

```
call DSYMV(UpLo, N,  &
                Alpha, A, LDA,  X, INCX, &
                Beta,  Y, INCY)
```

⇓

```
J=N1+1
call DSYMV(UpLo, N1, &
      Alpha, A1(1,1),LDA1, X(1)        ,INCX, &
      Beta,  Y(1)      ,INCY)
call DGEMV(Trans, M, N1, N2, &
      Alpha, A2(1,J),LDA2, X(1+N1*INCX),INCX, &
      One,   Y(1+N1*INCY),INCY)
call DSYMV(UpLo, N2, &
      Alpha, A2(J,J),LDA2, X(1)        ,INCX, &
      Beta,  Y(1)      ,INCY)
call DGEMV(!Trans, M, N1, N2, &
      Alpha, A2(1,J),LDA2, X(1)        ,INCX, &
      One,   Y(1+N1*INCY),INCY)
```

3.2.3 DSYR2K

DSYR2K の引数 A ないし B は通常数本のベクトルをまとめた上で, 行列として引数にわたることが多い. したがって, A, B は補助配列表現されないと仮定してもよい. 実際数本のベクトル程度であれば, 作業配列を動的に確保し, ベクトルの内容をコピーすることで一つの記憶領域内で処理を進めることが出来る.

行列 C が補助配列導入によって (C_1, C_2) となる場合を想定する.

$$C := \alpha(AB^T + BA^T) + \beta C$$

$$:= \alpha \overset{m}{\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}} \overset{n_1 \quad n_2}{\begin{pmatrix} B_{11}^T & B_{21}^T \end{pmatrix}} + \alpha \begin{pmatrix} B_{11} \\ B_{21} \end{pmatrix} \begin{pmatrix} A_{11}^T & A_{21}^T \end{pmatrix} + \beta C$$

行列 C の対称性を考慮すると以下の様にまとめられる.

$$C_{11} := \alpha(A_{11}B_{11}^T + B_{11}A_{11}^T) + \beta C_{11}$$

$$C_{22} := \alpha(A_{21}B_{21}^T + B_{21}A_{21}^T) + \beta C_{22}$$

$$C_{12} := \alpha B_{11}A_{21}^T + (\alpha A_{11}B_{21}^T + \beta C_{12})$$

```
call DSYR2K(UpLo, Trans, N, M, &
                Alpha, A, LDA, B, LDB, &
                Beta,  C, LDC)
```

⇓

```
J=N1+1
call DSYR2K(UpLo, Trans, N1, M, &
          Alpha, A1(1,1),LDA, B1(1,1),LDB, &
          Beta,  C1(1,1)   ,LDC1)
call DSYR2K(UpLo, Trans, N2, M, &
          Alpha, A1(J,1),LDA, B1(J,1),LDB, &
          Beta,  C2(J,J),LDC2)
call DGEMM(Trans, !Trans, N1, N2, M, &
          Alpha, A1(1,1),LDA, B1(J,1),LDB, &
          Beta,  C2(1,J),LDC2)
call DGEMM(Trans, !Trans, N1, N2, M, &
          Alpha, B1(1,1),LDB, A1(J,1),LDA, &
          One,   C2(1,J),LDC2)
```

3.3 実装上の留意点

コードを変更する際に幾つかの問題点が存在するのでここでそれを整理しておく.

(1) 一般的なライブラリでは行列の一部をベクトルデータとして扱うために, ベクトルが (A_1, A_2) のいずれに属するかによって引数を代えなくてはならない. C 言語での開発の場合はポインタによって処理できるが, Fortran の場合ポインタ属性変数ではこれを十分に扱うことができないので分岐によって対応するコードを追加する必要がある.

(2) 1 つの BLAS ルーチン呼び出しを複数の呼び出しに分けて再構成するためオーバヘッド増加の問題が生じる. できるだけ複数の BLAS ルーチン呼び出しを避けるために, 場合によっては作業配列に複数の部分配列を結合コピーし BLAS 呼び出し回数を減らす必要がある.

(3) Level 2 もしくは Level 3 BLAS は引数のデータ構造に特化した最適化が施されているために, 複数 BLAS を用いて再構成した場合に性能劣化が予想される. 例えば DSYMV を例にとると, DSYMV×2 と DGEMV×2 で再構成されるが, DSYMV の処理性能を α/要素数, DGEMV を β/要素数と仮定すればオリジナルと分割後 (補助配列導入後) のコストはそれぞれ

$$\begin{aligned} C_{\text{オリジナル}} &= \alpha N^2 \\ C_{\text{分割版}} &= \alpha(n_1^2 + (N-n_1)^2) + 2\beta n_1(N-n_1) \\ &= \alpha N^2 + 2(\beta-\alpha)n_1(N-n_1) \end{aligned}$$

となる. 今, $\alpha = \beta$ であれば, 分割してもコストは変化がない. しかし, $\alpha < \beta$ の場合にはどの様な分割をしても, オリジナルよりコストが増加する. したがって, BLAS の実装レベルに応じて補助配列のサイズを制限するなどの考慮が必要となる.

(4) DSYTRD の計算パターンは (1:N,1:N)→(1:N-1,1:N-1)→ ··· のリダクションである. このことから, 補助配列を配列の先頭に持ってくる場合と末尾に配置する場合とで, (2) で示した複数 BLAS 呼び出しの回数が大きく変動する. すなわち末尾に配置すれば, 始めの数回のリダクションで複数 BLAS 呼び出しを行うが, その後は補助配列が一切計算に関与しないので, 元々のコードと等価の振る舞いをする. つまり, 整合寸法を変化させただけとなるためオーバヘッドを無視することが可能となる.

4. 数 値 実 験

C-Stab アルゴリズムで性能安定化した DSYTRD ルーチンを実際に表 1 の計算機環境において実験を行い, 性能安定化の効果を検証した.

表 1 使用計算機諸源

Hardware specification	
CPU	Pentium 4 (Northwood)
Clock	3.0 GHz / FSB800 MHz
Memory	DDR400 dual channel
L1 cache	8KB
L2 cache	512KB
HT technology	On
Software specification	
OS	Linux 2.6.5 SMP (Fedora 2)
Compiler	Intel Fortran Compiler 8.0
BLAS (1)	Intel MKL 5.2 (Pentium 4 用)
BLAS (2)	ATLAS 3.6.0 (SSE2 版)

なお, 補助配列の配置場所の検証として次のパターンについて, 異なる BLAS の実装 (MKL, ATLAS) を用いて計 6 通りを測定した.

(1) オリジナル (Original)
(2) 補助配列を先頭に配置 (C-Stab pos-head)
(3) 補助配列を末尾に配置 (C-Stab pos-tail)

入力行列の次元を 64 次元から 4296 次元まで毎 8 次元その性能 (FLOPS 値) を測定し, プロットしたものが図 2 ならびに 図 3 である. 両グラフからも分かるように, 'C-Stab pos-tail' が最も性能の揺れが少なく, ついで'Original' であり, 'C-Stab pos-head' は規則的で大きな振動をしていることがみてとれる.

さらに, 表 2 にそれぞれのグラフの統計処理を行った結果を示す. ここで, 性能特性曲線は定数や一次関数では近似できないので, $ax/(x+b)$ によって近似している. 表内の平均値は近似関数の漸近値であり a の値である. また, 分散は SQRT(WSSR/n) (重み付残差の二乗和平均の平方根) である. 併せて, 分散/平均の値も示した. この結果からも, 'C-Stab pos-tail' が予想どおり最良の成績を得ることができた.

また, これは期待していなかった結果であるが, ATLAS による実装で C-Stab による安定化ルーチンの方が高性能となる結果が得られた. これは, 行列の一部をベクトルデータとして利用している部分を別途作業配列にコピーし処理したために, キャッシュの再利用がより促進されたためではないかと推測される.

5. ま と め

性能安定化アルゴリズム C-Stab を LAPACK に応用する試みとして, 三重対角化ルーチン DSYTRD の性能安定化を行った. 安定化コード作成の過程では, 安定化コード生成のための自動化に関する数々の知見を得ることができた. 特に, BLAS ルーチンの再構成

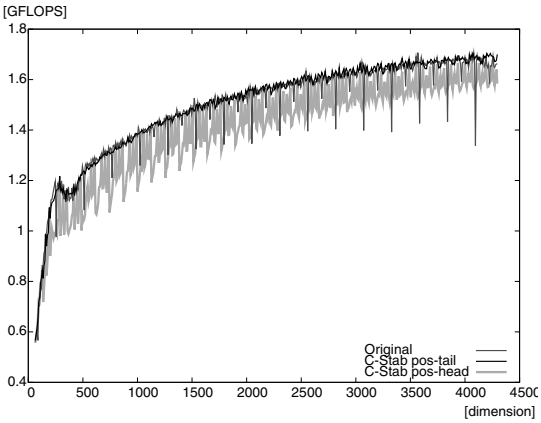

図 2 Intel MKL を用いた場合の C-Stabed DSYTRD の性能

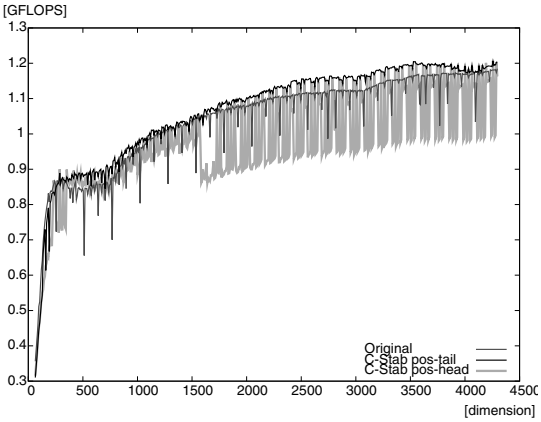

図 3 ATLAS を用いた場合の C-Stabed DSYTRD の性能

表 2 測定結果の統計的解析結果 (単位は GFLOPS)

		平均	分散	分散/平均
MKL	Original	1.6874	0.0620	0.0367
	pos-tail	1.7062	0.0518	0.0303
	pos-head	1.6193	0.0789	0.0487
ATLAS	Original	1.1672	0.0580	0.0496
	pos-tail	1.2082	0.0500	0.0413
	pos-head	1.0715	0.0806	0.0752

規則や実装における種々の注意点は今後の自動コード生成のための研究に役立てていきたい. また, 予備実験において作成された DSYTRD ルーチンは補助配列を末尾に配置することで性能安定化がなされ, 性能安定化しないオリジナル版の性能不安定点を完全に排除しかつ高性能を維持することができた.

今後は, 他の LAPACK ルーチンへの対応を行い自動コード生成を一般化するための知見を深めるとともに, 並列ライブラリ ScaLAPACK やその他 OpenMP をベースとする共有メモリ方計算機環境を志向した数値計算ライブラリへの応用を進めていきたい.

参 考 文 献

1) Whaley R.C., Petitet A., and Dongarra J.J.: Automated Empirical Optimization of Software and the ATLAS Project, LAPACK Working Note 147 (2000).
2) Bilmes J., Ananoic K., Chin C.-W., and Demmel J.: Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology, *Proceedings of International Conference on Supercomputing 97*, pp. 340–347 (1997).
3) 片桐孝洋, 黒田久泰, 大澤清, 工藤誠, 金田康正: 自動チューニング機構が並列数値計算ライブラリに及ぼす効果, 情報処理学会誌：ハイパフォーマンスコンピューティングシステム, Vol.42, No.SIG12(HPS 4), pp. 60–76 (2001).
4) Katagiri T., Kise K., Honda H., and Yuba T.: FIBER: A Generalized Framework for Auto-tuning Software, *Proceedings of 5th International Symposium, ISHPC2003*, LNCS 2858, pp. 146–159 (2003).
5) Dongarra J., and Eijkhout V.: Self-adapting Numerical Software for Next Generation Applications, *The International Journal of High Performance Computing and Applications*, Vol. 17, No. 2, Summer 2003, pp. 125–131 (2003).
6) 直野健, 今村俊幸: 自動チューニング型の固有値ソルバーについて, 情処研報, Vol.2002, No.91, pp. 49–54 (2002).
7) 今村俊幸, 直野健: 性能安定化を目指した自動チューニング型固有値ソルバーについて, 先進的計算基盤システムシンポジウム SACSIS2003, pp. 145–152 (2003).
8) 今村俊幸, 直野健: キャッシュ競合を制御する性能安定化機構内蔵型数値計算ライブラリについて, 情報処理学会論文誌, コンピューティングシステム, Vol. 45, No.SIG 6(ACS 6), pp. 113–121 (2004).
9) Lawson C., Hanson R., Kincaid D., and Krogh F.: Basic Linear Algebra Subprograms for Fortran Usages, *ACM Transacton on Mathematical Software*, Vol. 5, pp. 308–323 (1979).
10) Gunnel J., Gustavson F., Henry G., and Van de Geijn R.: FLAME: Formal Linear Algebra Methods Environment, *ACM Transaction on Mathematical Software*, Vol. 27, No. 4, pp. 422–455 (2001).
11) Bientiesi P., Quintana-Ortí E., and Van de Geijn R.: Representing Linear Algorithms in Code: The FLAME Application Program Interfaces, *to appear ACM Transaction on Mathematical Software*, `http://www.cs.utexas.edu/users/flame/pubs.html`.