T2K スパコンにおける固有値ソルバの開発

今村 俊幸
電気通信大学情報工学科
JST/CREST

1 はじめに

固有値計算は諸工学, 計算科学領域において欠くことのできない重要なツールである. 分野によって要求されるものが異なるが, 我々が現在取り組んでいる超電導材料設計分野では, 個々のエネルギーモードによる直交基底分解が必要となり

$$AX = X\Lambda \quad \text{または} \quad AX = BX\Lambda$$

で与えられる系全ての（または一部分）の固有モードの計算が重要となる. ここで A, B はエルミートもしくは実対称行列 (B については正定値性が課せられる), Λ は固有値を要素とする対角行列, X は固有値に対応する固有ベクトルを並べた行列を表している. 数値シミュレーションに必要な問題サイズは, ここ近年の計算機パワーの増大に伴い, 数万から数十万次元にも達することがある. また, 固有値計算は単に 1 回行えば十分なものではなく, 計算した固有値を元に行列を再度作成し直し計算を繰り返すこともある. その場合, 固有値計算に多大なコストをかけることになる. したがって, 使用するシステムの能力をほぼ使い切るような高速でしかもある程度精度の高い固有値ソルバでなければ, 高並列計算機の様なシステムではその高い性能の恩恵を受けることができない. 如何に固有値計算を安価に済ませられるかが高性能アプリケーションの必要条件となる. もちろん, それは研究室にある安価なデスクトップサーバから T2K オープンスパコンのようなスーパーコンピュータまでをシームレスかつスケーラブルにカバーしていて欲しいのが一般利用者の考えである. 我々は, ハードウェア要件に現在神戸に開発が進行中の次世代スパコンを想定し, 1 から数十万コアまでを使用した時にスケーラブルな性能を発揮する固有値ソルバの開発を実施している. 東大の T2K スパコン (HA8000) は国内にも有数のコア数を誇るクラスタ型スパコンであり, 次世代計算機に向けたソフトウェア開発環境として活用している.

本稿では我々が開発を進めている固有値ソルバ `eingen_s` の概要と, その開発状況ならびに, 現時点での 64 ノード (1024 コア) を使用した性能測定結果を報告する. 本稿には T2K スパコンでの開発の TIPS を記しているが, T2K スパコンだけではなく幾つかのマルチコアプロセッサシステムに共通した問題点や解決方法を述べた部分もある. 本稿の読者に有用な情報提供ができればと考えている.

2　固有値計算の概要

本章は固有値計算の数学的背景を扱う．性能測定結果に興味がある方は読み飛ばされても結構です．

実対称行列の固有値計算の算法について簡単に説明したい．数値計算の名著である「Numerical Recipes」[1] や「Matrix Computation」[2] には各種の固有値計算アルゴリズムが紹介されている．今回の固有値ソルバ開発では，これらの書籍や邦書の中であまり触れられていないアルゴリズムを使う．わずかではあるが高性能計算のための分析も含めて紙面を割いておきたい．

2.1　ハウスホルダー三重対角化

いま，入力 (解くべき行列) が密で正方の実対称行列とし A と表記する．行列 A の固有値計算の基本は，相似変換によりデータ量が少なくかつ数学的に扱いやすい行列への変換が第一ステップとなる．特に対称行列の場合は，三重対角行列への変換を行うことで扱いやすい形となる．この三重対角化にハウスホルダー変換を用いる．

ハウスホルダー変換は行列 A を以下の様にブロック表現したときに

$$A = \begin{bmatrix} a & x^T \\ x & B \end{bmatrix}, \quad P = I - \beta \begin{bmatrix} 0 \\ u \end{bmatrix} [0\ u^T], \quad \tilde{P} = I - \beta u u^T, \quad \beta = 2/\|u\|^2$$

の形をした鏡像変換 P(直交変換でもある) により，$\tilde{P}x = \|x\|e_1$ となるようにベクトル u を選び行列の特定部分ベクトル x を消去 (厳密には特定要素以外を 0 にする) する処理を再帰的に実施するものである．つまり，PAP^T の相似変換により

$$PAP^T = \begin{bmatrix} a & \|x\|e_1^T \\ \|x\|e_1 & \tilde{P}B\tilde{P}^T \end{bmatrix}$$

と変換されるが，ベクトル e_1 は第一要素のみ 1 でそれ以外 0 の単位ベクトルであり，第一列ならびに第一行の成分は第二要素以降が 0 になっている．この操作を同様にして $\tilde{P}B\tilde{P}^T$ について行っていく．この直交変換操作を続けることにより，最終的に行列は三重対角行列 T になる．直交変換 (相似変換) により固有値は変化しないため三重対角行列 T の固有値を求めれば，もとの行列 A の固有値を求めたことと同じとなる．また，A に作用させた直交変換 P を逆順に三重対角行列 T の固有ベクトルに作用させると A の固有ベクトルになる．これは

$$Tx = PAP^T x = \lambda x \quad \text{より}, \qquad A(P^T x) = \lambda(P^T x)$$

の変形からも $y = P^T x$ が固有ベクトルであることが確認できる．この一連の計算の中で $\tilde{P}B\tilde{P}^T$ はさらに次の様に計算せよと多くの数値計算のテキストにある．

$$\tilde{P}B\tilde{P} = (I - \beta u u^T)B(I - \beta u u^T) = B - uv^T - vu^T, \quad v = \beta(Bu - \beta/2(uBu^T)u)$$

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) Householder block reflector: $(\beta, u^{(k)}) := H(W_{(*,j-k)})$
    (2) Matrix-Vectors multiplication
      $v^{(k-\frac{2}{3})} \leftarrow A_{(1:j-k-1,1:j-k-1)}u^{(k)}$
    (3) $v^{(k-\frac{1}{3})} \leftarrow v^{(k-\frac{2}{3})} - (UV^T + VU^T)u^{(k)}$
    (4) $v^{(k)} \leftarrow \beta(v^{(k-\frac{1}{3})} - su^{(k)}), s = \frac{1}{2}\beta {u^{(k)}}^T v^{(k-\frac{1}{3})}$
      $U \leftarrow [U, u^{(k)}], V \leftarrow [V, v^{(k)}]$.
    (5) $W_{(*,j-k:j)} \leftarrow W_{(*,j-k:j)} - (u^{(k)}{v^{(k)}}^T + v^{(k)}{u^{(k)}}^T)_{(*,j-k:j)}$
  endfor
  $A_{(*,j-M+1:j)} \leftarrow W$
  (6) $2M$ rank-update (BLAS3)
    $A_{(1:j-M,1:j-M)} \leftarrow A_{(1:j-M,1:j-M)} - (UV^T + VU^T)_{(1:j-M,1:j-M)}$
endfor

図 1: Dongarra–Sorensen のブロックハウスホルダー三重対角化アルゴリズム

この変形により行列–ベクトル, ベクトル–ベクトルの単純な演算によって構成される. この実装は極めて単純であるが, キャッシュの利用効率が悪く性能を出すことができないことが知られている. これは, その他の線形代数計算にもいえることであるが, 行列とベクトル, またはベクトルとベクトルの乗算でのみ構成されるアルゴリズムは演算に対して必要なデータがほぼ 1 対 1 となるためである. 行列ベクトル積は主記憶とプロセッサ間でのデータ転送によって演算性能を決定されることは高性能計算分野では知られている.

性能の上限がメモリバンド幅によって決定される事実は, 現代のマイクロプロセッサを用いる計算シミュレーションにとっては致命的である. 一般的にメモリ性能はプロセッサのクロックよりも 1 桁程度劣ることが普通であり, メモリバンド幅でアルゴリズムが律速される場合は性能が理論ピーク値の 10%も出せないことを意味している. これを解消するために, Dongarra–Sorensen は図 1 に示すようなブロックズムアルゴリズムを考案している [3]. このアルゴリズムは分散メモリ型並列計算機を意識したものであるが, 実際は図中の (6) での計算を行列—行列積によって構成することができる. 行列–行列積は BLAS(Basic Linear Algebra Subprograms) の中で DGEMM として知られる関数によって実装できる. 関数 DGEMM は HPC Challenge ベンチマーク [4] の結果や Linpack ベンチマーク [5] の結果でも知られるように, 理論ピーク性能の 70%以上の性能が期待できる. このアルゴリズムの登場により (6) の部分の計算コストはほぼ 0 とみなせるようになり, アルゴリズム全体の計算時間を従来型の約 1/2 程度にまで削減することができる. このアルゴリズムの出現は非常に驚くべきものであり, 現在でもハウスホルダー変換関数で利用される基本中

の基本のアルゴリズムである.

2.2 分割統治法 (Cuppen's divide and conquer method)

ハウスホルダー変換によって三重対角行列に変換された後は, 三重対角行列の固有値と対応する固有ベクトルの計算を行う. 邦書の多くはこの部分にスツルム列に基づく 2 分法と逆反復法, 同時逆反復法, または QR 法系統を利用することを推奨している. しかしながら, 近代的な数値計算ライブラリでは邦書に書かれていないアルゴリズムが用いられている. 精度と安定性の面で優れているものとして, 我々は Cuppen によって考案された分割統治法 [6] を利用する. 「分割統治法」という用語は並列処理の一つの処理形態を指すものであるが, 固有値計算の分野で分割統治法といえば Cuppen のアルゴリズムを指す. 分割統治法の他に, Dhillon らによる MRRR(Multiple Relatively Robust Representations)[7] が LAPACK3 に収納されている. 中村らによる I-SVD[8] は特異値分解のアルゴリズムであるが, 固有値計算にも展開が可能である.

Cuppen の分割統治法は次の 1 階摂動による基本定理を利用する.

基本定理 1 *今, 2 つの対角行列 $L = diag(\lambda_1, \lambda_2, \ldots, \lambda_n)$, $M = diag(\mu_1, \mu_2, \ldots, \mu_n)$ に対して直交変換 S により, 次の関係が成り立つとするようなベクトル a が存在すると仮定する.*

$$S^T M S = L + aa^T$$

このとき, a, λ, μ は次の等式 (secular 方程式) を満足する.

$$1 + \sum_{j=1}^{n} \frac{a_j^2}{\lambda_j - \mu_k} = 0$$

この基本定理は, L ならびに a が定まれば同時に M が定まることを主張している. これは, 対角行列に 1 階の摂動 (aa^T) が加われば, 摂動が加わった行列の固有値が上記の等式により定まることを意味している. Cuppen の分割統治法はこの基本定理を用いて次のように計算を進める.

1. まず三重対角行列 T を 2 つの三重対角行列 T_1 と T_2 の直和と 1 階摂動の和に分解する.
$$T = \begin{bmatrix} T_1 & \\ & T_2 \end{bmatrix} + uu^T$$
2. 部分問題 T_1, T_2 の固有値と固有ベクトルが何らかの方法で求められたとする. つまり, $Q_1^T T_1 Q_1 = D_1$, $Q_2^T T_2 Q_2 = D_2$ の変換がなされたとする (ただし D_1, D_2 は対角行列). このとき $Q = \mathrm{diag}(Q_1, Q_2)$ なる行列を作り, T に作用させれば,
$$Q^T T Q = \begin{bmatrix} D_1 & \\ & D_2 \end{bmatrix} + (Q^T u)(Q^T u)^T$$

となる．この固有値を基本定理により求めることができる．固有値 λ に対応する右辺の固有ベクトルは $(D-\lambda I)^{-1}Q^T u$ で定まるが, 行列 T の固有ベクトルはこれに Q を乗じた $Q(D-\lambda I)^{-1}Q^T u$ で求められる．

3. 分割統治法は分割した部分問題の答えを上位の問題に統合していく．そのために, 下位の問題に含まれる誤差が伝播しやすい．特に固有ベクトルはその影響を受けやすく, 直交性が即座に破たんすることが多い．それを補正する方法として Löwner の定理を利用する．基本定理において, 対角要素 μ_i, λ_i が与えられたとき, 1 階摂動の第 j 要素 a_j とそれらは次の式で関係づけられる．

$$a_j^2 = (\mu_j - \lambda_j)\prod_{i\neq j}\frac{\mu_i - \lambda_j}{\lambda_i - \lambda_j}$$

これにより, 1 階摂動ベクトル (上記ステップ 2. の式展開では $Q^T u$) を修正したうえで, 固有ベクトル計算に進むのである．(他の解法を使ってでも) 固有値は十分な精度で計算できるため, 下位の問題の結果から算出される摂動項を上位問題のレベルで修正し誤差をさらに上位の問題に伝播させないようにすることができる．実際, Löwner の定理のおかげで分割統治法で求めた固有ベクトルの直交性は極めて高い．この誤差軽減のテクニックが LAPACK3 に取り込まれて以来, 分割統治法は三重対角行列の固有値計算で重要な地位を占めるようになっている．

上記ステップは部分問題 T_1, T_2 への再帰的な適用が可能である．つまり, 図 2 に示すように問題を順次分割しながら固有値計算を計算し, 上位の問題に集約することができる．なお, アルゴリズムでコストが高い部分は secular 方程式の求解と固有ベクトルに行列 Q を乗じる部分である．実際複数のベクトルをまとめて Q に乗じるため行列–行列積となる．したがって, ハウスホルダー変換の部分でも述べたように関数 DGEMM を利用することで高速な処理が可能となる．また, 部分問題や個々の固有値の求解が独立である．極めて高い並列性を有するため, 並列処理に向いていることも忘れてはならない．

2.3 ハウスホルダー逆変換

ハウスホルダー三重対角化が鏡像変換 P_j を $j = 1, 2, 3\ldots$ の順に連続して作用させることであったので, 逆変換はその鏡像変換を逆順に固有ベクトルに作用させればよいことになる．この変換を式で書けば $P^T = P_n P_{n-1}\cdots P_2 P_1$ と書くこともできる．個々の鏡像変換へのベクトル x の作用は次のように書ける．

$$P_j x = (I - \beta_j u_j u_j^T)x = x - \beta_j (u_j^T x)u_j$$

この計算は, dot(内積) と axpy(スカラ積和) タイプの計算で構成されている．複数のベクトルの逆変換を行う場合は, ベクトル x の部分を複数のベクトルをまとめて X としても

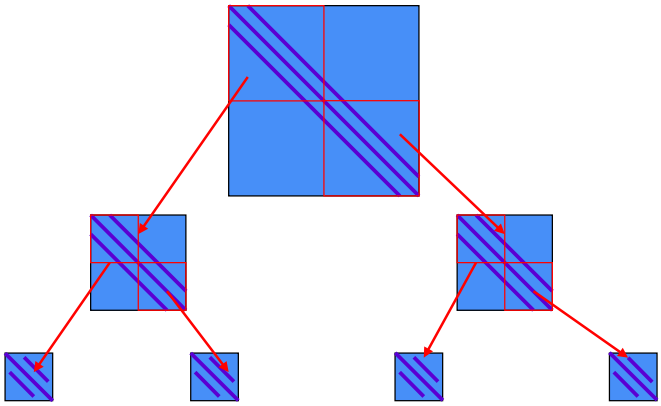

図 2: 分割統治法の部分問題への分割フェーズの概念図

よい. その際, ベクトル–行列積で構成できる.

いま, 複数鏡像変換を一度にまとめて実施するブロッキング技法を適用する. このとき, 複数の鏡像変換の積を $I - UCU^T$ の形で表せると仮定して U と C を定める. 以下の 2 式が一致するので, 両式を比較して U と C を定めることができる.

$$(I - \beta_j u_j u_j^T)(I - UC_{j-1}U^T) = I - \beta_j u_j u_j^T - UC_{j-1}U^T + \beta_j u_j u_j^T UC_{j-1}U^T$$

$$I - [U, u_j] \begin{bmatrix} C_{j-1} & c_2 \\ c_1 & c_3 \end{bmatrix} [U, u_j]^T = I - UC_{j-1}U^T - u_j c_1 U^T - Uc_2 u_j^T - u_j c_3 u_j^T$$

$$C_j = \left[\begin{array}{c|c} C_{j-1} & 0 \\ \hline -\beta_j u_j^T UC_{j-1} & \beta_j \end{array}\right], \ U := [U, u_j]$$

このブロッキング手法をコンパクト WY ブロッキング表現手法 [9] という. 本ブロッキングを用いれば, 複数ベクトルの変換式は次の様に書けるから, 全ての演算が行列–行列積で構成できるようになる. 従って, ハウスホルダー逆変換ではマイクロプロセッサのほぼ最高性能を達成することができる.

$$(I - UCU^T)X = X - (UC)(U^T X)$$

2.1 節から 2.3 節までの説明ををまとめると, 固有値の計算は図 3 に示すような 3 段階ステップで計算される. それぞれ, 前処理, 固有値計算, 後処理という分類もできる. 図にも示しているように, 計算コストは前処理と後処理の $O(N^3)$ が支配的である. アルゴリズム部分で説明したブロックアルゴリズムの適用によって, $O(N^3)$ のコストを適切な演算性能で計算できるようになる. 次章以下で数値計算ライブラリの性能を示していく.

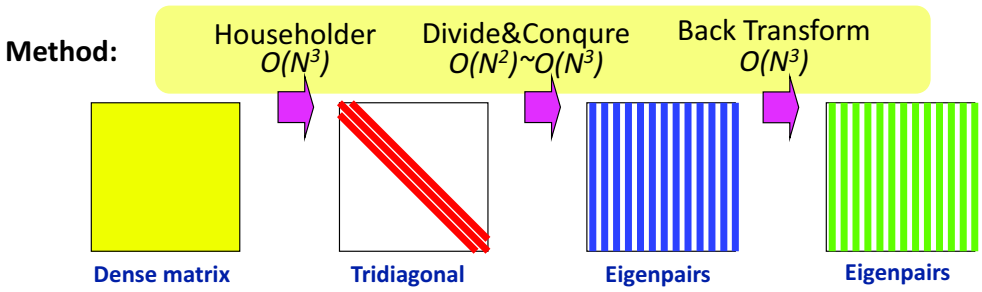

図 3: 固有値計算の 3 段階ステップ

3 数値計算ライブラリの性能

3.1 ScaLAPACK の T2K 上での性能

数計算ライブラリ, 特に, 線形計算のフリーでかつ信頼性の高いライブラリとして ScaLAPACK[10] が存在する. 多くの商用ライブラリがある中でその存在は業界標準 API として広く知られている. T2K オープンスパコン HA8000 にもインストールされている. ScaLAPACK は線形方程式, 固有値計算などのライブラリなどを含む LAPACK の分散メモリ型並列計算機向けの数値計算ライブラリとして 1995 年に version1.0 がリリースされた. 現在は 2007 年 4 月にリリースされた version1.8 が最新のものである. テネシー大学の netlib サイトから 1 次配布されているが, 各ベンダーからの独自の最適化がなされたバージョンも存在する.

ScaLAPACK は業界標準として広く使われてはいるが, 一方で各種の問題点を持っている. 著者が考えるものを 2 つ挙げておく.

- 逐次版 LAPACK に比べ内容が 1 世代古い
- 基本設計が 1990 年代の分散並列計算機を想定

逐次版の LAPACK は現在 version3.2.1(2009 年 4 月現在) である. 一方, ScaLAPACK のベースとなった LAPACK が version2 相当であることからも解るように, 内部で利用できるアルゴリズムに違いがある. そのあたりの注意については文献 [11, 12] を読むとよい. また, リリースが 1995 年, つまり 1990 年代頭ということで, 利用が想定された計算機は当時主流の並列計算機であるシングルコアでかつ 1000 からせいぜい 4000 コアを搭載するものである. T2K スパコンでそれを対応させると, ScaLAPACK のスコープは 64 ノードから 256 ノードまでカバーすることになる. もちろん当時はマルチコアではないのでスレッドによる並列処理は考えない. 一見, T2K スパコンがカバーされているように見えるのであるが, 使用されているハードウェア技術は 15 年で飛躍的な進化を遂げており, ソフトウェアの作りそのものは追いついていない状況にあると考えてよいはずである. 表 1 に ScaLAPACK の固有値ドライバルーチン pdsyevd の中から呼ばれる, 固有値計算の 3 ス

テップに対応する関数 pdsytrd, pdstedc, pdormtr の実行時間を記した．実際に使用したサンプルプログラムを図 4 に示す．使用したコンパイラは Intel コンパイラ version 11, BLAS は Intel MKL 11 を同様に使用した．Intel MKL の提供するスレッド並列機能を利用するために, 実行時環境変数 `OMP_NUM_THREADS` を 4 に設定している．また, `numactl` コマンドを使って, 1 ソケットに 1 プロセスが対応して動作するように設定している．

表は実行時間を示しているだけであるが, 実効性能 (FLOPS 値) を示せば, 32 ノード 20480 次元で PDSYTRD が 161GFLOPS, PDORMTR が 1354GFLOPS を記録する．固有値計算全体では 3029GFLOPS である． また, 64 ノードを用いれば, それぞれ 329GFLOPS, 2235GFLOPS, 580GFLOPS となる．T2K スパコンの単体ノードのピーク性能が 2.3*4*16 =147.2GFLOPS であるので, システム利用効率が如何に低いかが判る．この原因は問題サイズが小さすぎることと使用するアルゴリズムの限界 (メモリバンド幅による性能上限) もあるのだが, ScaLAPACK の実装方法にも問題がある．より大規模な問題 81920 次元では, 64 ノードで 2313 秒を要する．これは実効性能で 792GFLOPS である．一方, 後述する我々が開発している `eigen_s` では 1146 秒で計算できる (図 5)．これは 1598GFLOPS に相当し, ScaLAPACK の半分の時間で 81920 次元の行列の対角化ができるのである．小規模問題で大きな性能の差は見えにくいが, 大規模問題に ScaLAPACK を利用するのはパワーユーザであれば避けた方がよいかもしれない．

ScaLAPACK では行列データは 2 次元のブロックサイクリック分割をすることが要請される．分割方法は図 4 のサンプルプログラムにも記しているが, パラメタ変数 NB によって制御されている．この変数が先に示したブロック化アルゴリズムのブロック化係数と一致しているため (おそらくそうした実装が最も楽であることから), 個々の変数の性能が NB の値におおじて大きく変動することがある．表 2 は PDSYTRD と PDORMTR の 2 つについて検証したものであり, NB=1 からの性能比を示している．赤でマークした部分が最も高性能を示しており, 使用するノード数によって最適な NB 値が異なることが分かる．アルゴリズムレベルでも異なる．固定ノード数で利用されるユーザは自身が最適な NB 値を使用しているかを, 一度確認して見るとよい．場合によっては数%の計算時間削減が可能かもしれない．

表 1: ScaLAPACK の T2K での実効性能 (実行時間「秒」)

(表 1.1 32 ノードでの測定, 128 プロセス, 4 スレッド/プロセス)

行列次元	PDSYTRD	PDSTEDC	PDOMTR
1000	0.340497	5.81E-02	2.14E-02
2000	0.597057	0.137764	6.03E-02
3000	1.040523	0.260074	0.116418
10240	10.19719	2.123313	1.955015
20480	70.95069	8.780391	12.68503

(表 1.2 64 ノードでの測定, 256 プロセス, 4 スレッド/プロセス)

行列次元	PDSYTRD	PDSTEDC	PDOMTR
1000	0.38724	5.54E-02	2.20E-02
2000	0.616548	0.136502	5.85E-02
3000	0.988244	0.261725	0.102517
10240	6.017904	1.919496	1.343884
20480	34.79985	6.81047	7.684606
81920	1978. 29939	111.904	222.94

表 2: 分割方法での性能の違い

(表 2.1 PDSYTRD で分割ブロック幅 NB を変化させて)

Nodes	1	8	16	24	32	48	64	96	128
1	1	0.553	0.52	0.515	0.516	0.342	0.524	–	–
2	1	0.506	0.491	0.487	0.488	0.328	0.51	0.362	0.593
4	1	0.436	0.408	0.413	0.413	0.428	0.424	0.471	0.538
8	1	0.437	0.438	0.414	0.417	0.424	0.419	0.483	0.553
16	1	0.409	0.392	0.403	0.42	0.405	0.447	0.457	0.516
32	1	0.427	0.409	0.389	0.417	0.414	0.428	0.483	0.548
64	1	0.387	0.372	0.381	0.372	0.408	0.425	0.475	0.528

(表 2.2 PDORMTR で分割ブロック幅 NB を変化させて)

Nodes	1	8	16	24	32	48	64	96	128
1	1	0.137	0.075	0054	0.045	0.034	0.035	–	–
2	1	0.134	0.074	0.055	0.045	0.035	0.036	0.032	0.038
4	1	0.130	0.073	0.054	0.046	0.038	0.036	0.034	0.036
8	1	0.132	0.074	0.056	0.048	0.040	0.038	0.037	0.040
16	1	0.135	0.077	0.059	0.051	0.043	0.041	0.040	0.040
32	1	0.138	0.082	0.063	0.055	0.050	0.047	0.045	0.047
64	1	0.149	0.089	0.076	0.071	0.055	0.053	0.052	0.054

```
      subroutine EIGEN(n, NB)
      integer                       :: n, NB

      integer                       :: lda1,lda2
      real(8), pointer              :: d(:),e(:),tau(:)
      real(8), pointer              :: w(:), a(:), z(:)

      INTEGER, PARAMETER            :: DLEN_  = 9
      integer                       :: DESCA(DLEN_), DESCZ(DLEN_)

      integer                       :: world_size, my_rank
      integer                       :: LWORK, LIWORK, TRILWMIN

      real(8), pointer              :: work(:)
      integer, pointer              :: iwork(:)

      include 'mpif.h'
      real(8) :: t1,t2, z1,z2

      call MPI_COMM_SIZE( MPI_COMM_WORLD, world_size, ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, my_rank, ierr )

! BLACS/PBLAS/SCALAPACK initialization
      CALL BLACS_PINFO( IAM, NPROCS )
      if ( NPROCS < 1 ) then
!  MPI group setup
         NPROCS = world_size
         IAM = my_rank
         CALL BLACS_SETUP( IAM, NPROCS )
      end if
      CALL BLACS_GET( -1, 0, ICTXT )

      NPROW = INT(SQRT(DBLE(NPROCS)))
      DO
         IF(MOD(NPROCS,NPROW)==0)THEN
            EXIT
         ENDIF
         NPROW=NPROW-1
      ENDDO
      NPCOL = NPROCS/NPROW

      CALL BLACS_GET( 0, 0, ICTXT )
      CALL BLACS_GRIDINIT( ICTXT, 'Row-major', NPROW, NPCOL )
      call BLACS_GRIDINFO( ICTXT, NPROW, NPCOL, MYROW, MYCOL )

! BLACS array registration
      NP = NUMROC( n, NB, MYROW, 0, NPROW )
      NQ = NUMROC( n, NB, MYCOL, 0, NPCOL )
      lda1 = ((NP-1)/16+1)*16; lda2 = NQ
      lda = lda1; ldz = lda1
      call DESCINIT( DESCA, n, n, NB, NB, 0, 0, ICTXT, lda, INFO )
      allocate(w(n+1), a(lda1*lda2), z(lda1*lda2))
      allocate(d(n+1), e(n+1), tau(n+1) )

! preparing working arrays
      TRILWMIN = 3*N + MAX( NB*( NP+1 ), 3*NB )
      LWORK  = MAX( 1+6*N+2*NP*NQ, TRILWMIN ) + 2*N + 2*NB*NB
      LIWORK = 2+7*n+8*NPCOL
      allocate(work(lwork+16), iwork(liwork+16), stat=istat)
      if(istat.ne.0) then
           print*,"Memory exhausted"
           call flush(6)
           call MPI_Abort( MPI_COMM_WORLD, 1, ierr )
      end if

!$OMP PARALLEL DO PRIVATE(k1,i0,j0,i1,j1,i,j)
      do k2=1,lda2
         do k1=1,lda1
            i0=(k1-1)/NB; j0=(k2-1)/NB
            i1=MOD(k1-1,NB); j1=MOD(k2-1,NB)
            i=(i0*NPROW+MYROW)*NB+i1+1
            j=(j0*NPCOL+MYCOL)*NB+j1+1
            if(i<=n.AND.j<=n)then
               a(k1+(k2-1)*lda)=(n+1)-MAX(i,j)
            else
               a(k1+(k2-1)*lda)=0.0D+00
            endif
         end do
      end do
!$OMP END PARALLEL DO
      z = 0.0D+00

      if(my_rank==0)then
         print*,"N=",n,"NB=",NB
      endif
      z1 = MPI_Wtime()

      t1=MPI_Wtime()
      CALL PDSYTRD( 'U', n,
     $              a(1), 1, 1, DESCA(1),
     $              d(1), e(1), tau(1),
     $              WORK(1), LWORK, INFO )
      t2=MPI_Wtime()
      if(my_rank==0)then
         print*,"PDSYTRD",t2-t1,DBLE(n)**3*(4./3.)/(t2-t1)*1D-9
      endif

      CALL PDLARED1D( n, 1, 1, DESCA(1), d(1), w(1),
     $                WORK(1), LWORK )
      CALL PDLARED1D( n, 1, 1, DESCA(1), e(1), d(1),
     $                WORK(1), LWORK )
      w(n+1)=0.0D+00
      d(n+1)=0.0D+00
*
      t1=MPI_Wtime()
      CALL PDSTEDC( 'I', n,
     $              w(1), d(1+1),
     $              z(1), 1, 1, DESCA(1),
     $              WORK(1), LWORK, IWORK(1), LIWORK, INFO )
      t2=MPI_Wtime()
      if(my_rank==0)then
         print*,"PDSTEDC",t2-t1
      endif
*
      t1=MPI_Wtime()
      CALL PDORMTR( 'L', 'U', 'N', n, n,
     $              a(1), 1, 1, DESCA(1),
     $              z(1), 1, 1, DESCA(1),
     $              WORK(1+16), LWORK, IINFO )
      t2=MPI_Wtime()
      if(my_rank==0)then
         print*,"PDORMTR",t2-t1,DBLE(n)**3*(4./2.)/(t2-t1)*1D-9
      endif
*
      z2 = MPI_Wtime()
      if(my_rank==0)then
         print*,"Total=",z2-z1,DBLE(n)**3*(10./3.)/(z2-z1)*1D-9
      endif

      deallocate(work,iwork,w,a,z,d,e,tau)

! BLACS/PBLAS/SCALAPACK finalize
      call BLACS_GRIDEXIT( ICTXT )

      return
      end subroutine
```

図 4: ScaLAPACK サンプルプログラム. 本コードは各ルーチンの実行時間を測定する目的のものである. 実際固有値計算をするだけの目的の場合は PDSYTRD から PDORMTR までをまとめてドライバ関数 PDSYEVD のみを呼び出せばよい.

3.2 eigen_s について

3.1 節で述べたように、我々は eigen_s という固有値ソルバーを開発している．これは, 地球シミュレータを利用していた頃, 地球シミュレータで十分な性能を発揮できるベクトル並列計算機に適度にチューンアップされた固有値ソルバーが存在しなかったことから, 開発が始まったものである．その当時の開発成果については SC06 のゴードンベルファイナリストペーパーにまとめてあるので, そちらから参照することができる [13]．地球シミュレータ上ではそのリッチなハードウェアを利用して, 一部ルーチンで最大ピーク性能の 70%をたたき出すことができている．これは, 当時の LINPACK ベンチマークの性能が 80%以上であったので, それに次ぐものと考えてよい．

現在、eigen_s は量子シミュレーションの中核となるハミルトニアン行列の全固有値固有ベクトル求解ルーチンとして, デスクトップコンピュータから次世代スパコンに至るまでの計算プラットフォーム上で高性能を発揮するような, スケーラブルかつ高性能な固有値ソルバとして T2K スパコン上で開発が継続されている．T2K スパコンが 2008 年に運用され始めてすぐから利用しているので, T2K スパコンでの開発はすでに 1 年半以上が経過している．幸いなことに, 地球シミュレータでの開発経験を評価され HPC 特別プロジェクトにも採用され 512 ノードでの試験実行も実施することができた．初期の各種苦労話のようなものは東大スパコンニュース (2009 年特集号 2[14]) にあるのでそれを参考にされたい．

eigen_s は ScaLAPACK 同様のブロックアルゴリズムを採用し, 3 ステップそれぞれ, TRED1, DC, TRBAKWY と名付けている．開発ではコストが最も高いハウスホルダー変換と逆変換部分を地球シミュレータ版を元にマルチコアプロセッサ向けの最適化を施したものである．固有値計算については ScaLAPACK の pdstedc をスレッド並列化する改良を加えている．MPI による一様なメッセージパッシング並列化 (所謂 flat–MPI), MPI と OpenMP を併用するハイブリッド並列化のいずれにも対応する．eigen_s は, 行列データを 2 次元サイクリック–サイクリック分割で固定するため, 分割幅 NB とブロックアルゴリズムが持つブロック幅を分離する設計となっている．そのため, 個々の関数に適したブロック幅を選択することで, ブロックアルゴリズムの持つ最高性能を出すことができる．この報告では, 個々の関数でほぼ best の性能を出すブロック値を使用して計測している．

図 5 は 10240 から 81920 次元までの全固有値固有ベクトルの計算時間を示したものである．図からわかるように, 10000 次元程度の対角化には 10 秒とかかっていない．T2K スパコンの 64 ノードの理論ピーク性能は 2.3*4*16*64=9421GFLOPS であるので, システムの 4.2%しか出せていない．しかしながら, シミュレーション研究者から見たときに対角化に 10 秒を切るインパクトは大きいのではないだろうか．また, 81920 次元の場合は, 理論ピーク性能に対して 16.9%である．20 万次元の対角化を行うと, 総計 14286 秒要する

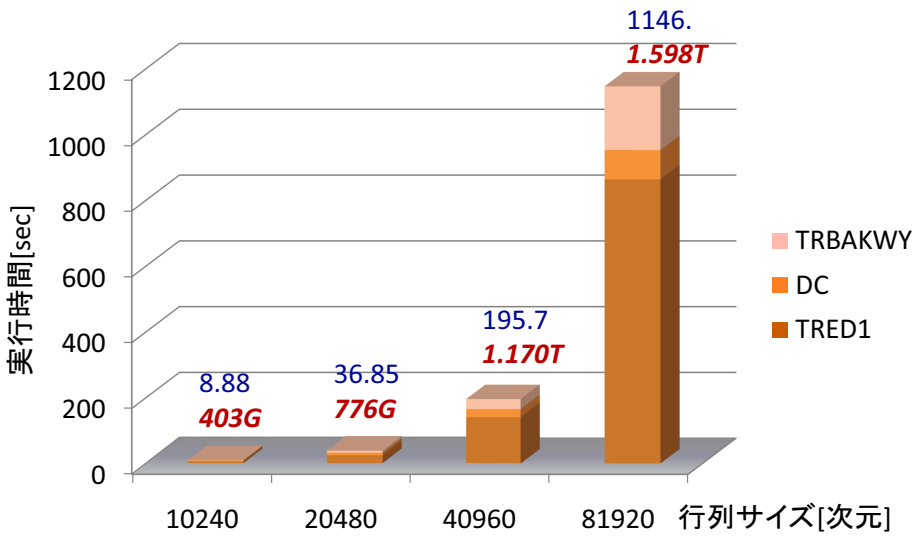

図 5: `eigen_s` の性能評価 (T2K スパコン 64 ノード, 4 スレッド/プロセス) バーの上の斜体が実効性能 (FLOPS), その上の数字が実行時間を示す.

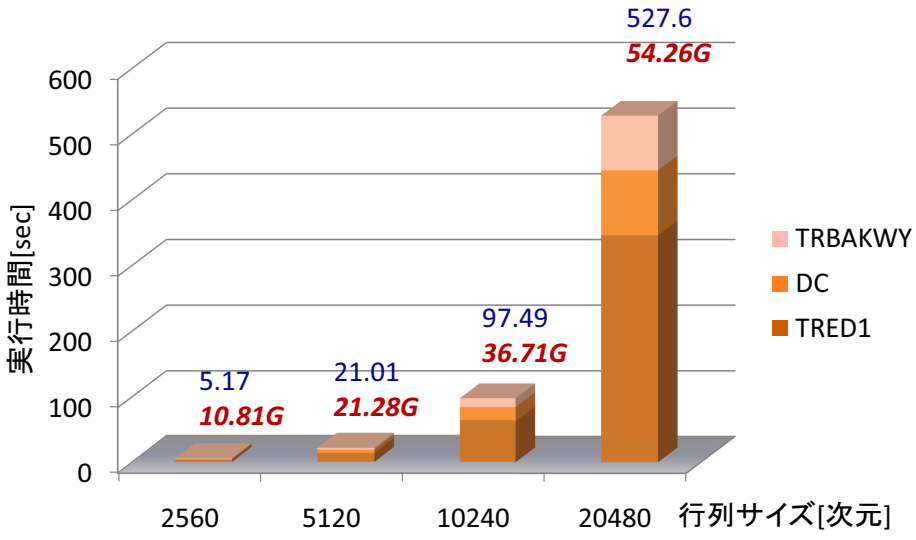

図 6: `eigen_s` の性能評価 (Xeon Quadcore クラスタ 8 ノード, 4 スレッド/プロセス) バーの上の斜体が実効性能 (FLOPS), その上の数字が実行時間を示す.

が, 2004GFLOPS(論理ピーク性能の 21.2%) にまで性能が向上する. 比較参考のために図 6 に, 研究室にあるような安価なサーバ機を用いて `eigen_s` の性能を測定したものもつけておく. 著者の研究室には Quadcore Xeon X3330(2.66GHz) の 8 ノードからなるクラスタを有するが, T2K スパコンのように, マルチソケットではなくネットワークもギガビットイーサーのため小規模問題では並列化の効果は望めない. しかしながら、研究室規模で行える大規模計算では十分な高性能を記録していることが判る. 現時点の開発版のおいても, T2K スパコン以外の PC クラスタでの利用という開発目的は十分果たすことができる.

図 7, 8 がスケーラビリティを評価するための Strong Scaling と Weak Scaling とで性能を測定したものである. グスタフソンのスケーリング則に合うように, 行列の問題サイズが大きくなれば通信などの並列化オーバヘッドが相対的に減少して, 性能がスケーラブルになる. 38400 次元では性能向上がかなりリニアに近いことが分かる. 図 8 の weak scaling では個々のプロセスごとに保持する行列サイズを一定にして, プロセス数を変動させている (厳密には問題サイズの 3 乗に応じて計算量が増加するので, weak scaling とは言い難いのではあるが). 若干の性能劣化は見られるがほぼリニアな性能向上が確認できる. N=9600*(ノード数) としているので, かなり大規模問題である. T2K スパコンの比較的リッチなネットワーク環境の下で上記の数値 9600 を下げていき, 許容可能な weak scaling が達成できれば, 適度な問題サイズかつ適切なシステム利用率でのライブラリ使用を保証することができるようなる. これはまだ測定がなされていないので, 今後の課題である.

次に, ハイブリッド並列処理化での性能がどのように変化するのかを測定したものがあるので, 最後に紹介しておく. 図 9 に, T2K スパコン 1 ノード 16 コア上で, 様々なプロセス形態で `eigen_s` を実行した場合の実行性能 (GFLOPS) を示す. ハウスホルダー三重対角化 (TRED1) と逆変換 (TRBAKWY) の二つについて, 100 から 1 万次元まで 100 次元刻みに, 「1 スレッド＊ 16 プロセス」, 「2 スレッド＊ 8 プロセス」, 「4 スレッド＊ 4 プロセス」,「8 スレッド＊ 2 プロセス」, 「16 スレッド＊ 1 プロセス」の 5 つの組み合わせを選択した. HA8000 システムのマルチコアプロセッサ性能を最大限に引き出せる, 4 スレッド＊ 4 プロセスが最も性能が高い結果となった. 続いて, 2 スレッド＊ 8 プロセスとなる. この結果は, できるだけソケット内でスレッドグループを閉じることにより無駄なソケット間トラフィックの発生を抑えられるからと考えられる. 例えば, 8 スレッド＊ 2 プロセスでは 1 プロセスが 2 ソケットに跨るため, 異なるソケットに接続されたメモリへのアクセスが発生する. 逆変換 TRDBAKWY(表中で PDORTRM と表記) は 4 スレッド＊ 4 プロセスでほぼ 100GFLOPS に達していたのであるが, 8 スレッド＊ 2 プロセスでは 83GFLOPS にまで劣化している. 複数の HT(Hyper Transport) 経由でのスループットの低下は無視できないということである. また, 100 次元刻みの測定ではあるが性能は極めて安定している. これは, `eigen_s` 内部でキャッシュ性能安定化 (C-Stab アルゴリズム [15]) を施しているためであり, 8192 次元周辺で僅かな揺れはあるものの, 特定の問題サイズでの性能劣

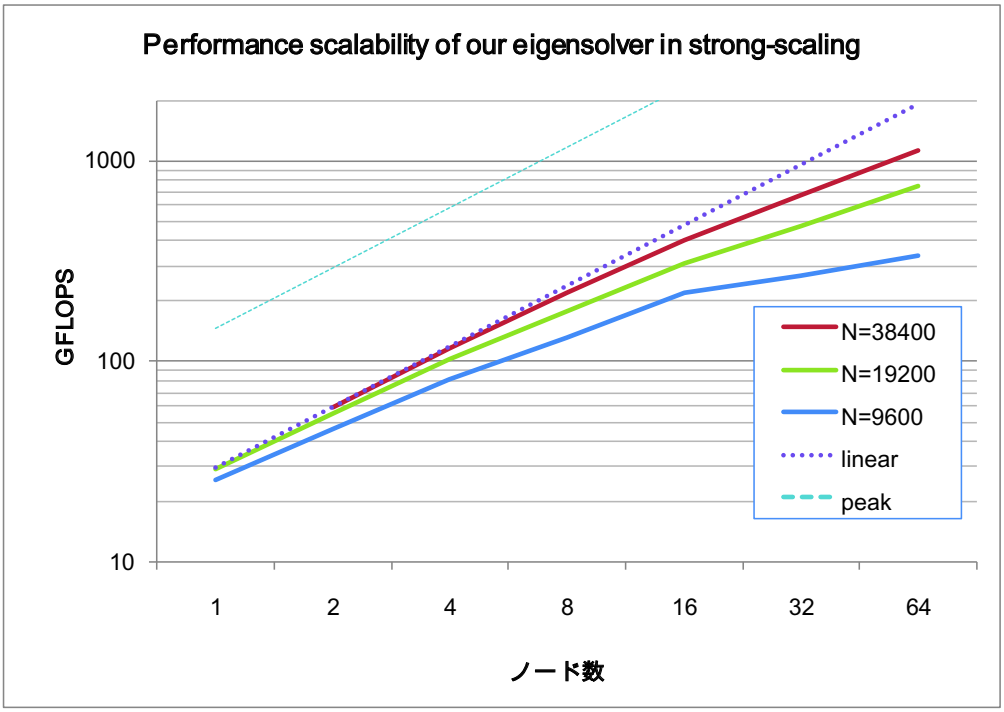

図 7: Strong Scaling on a T2K clutser system at Univ. of Tokyo

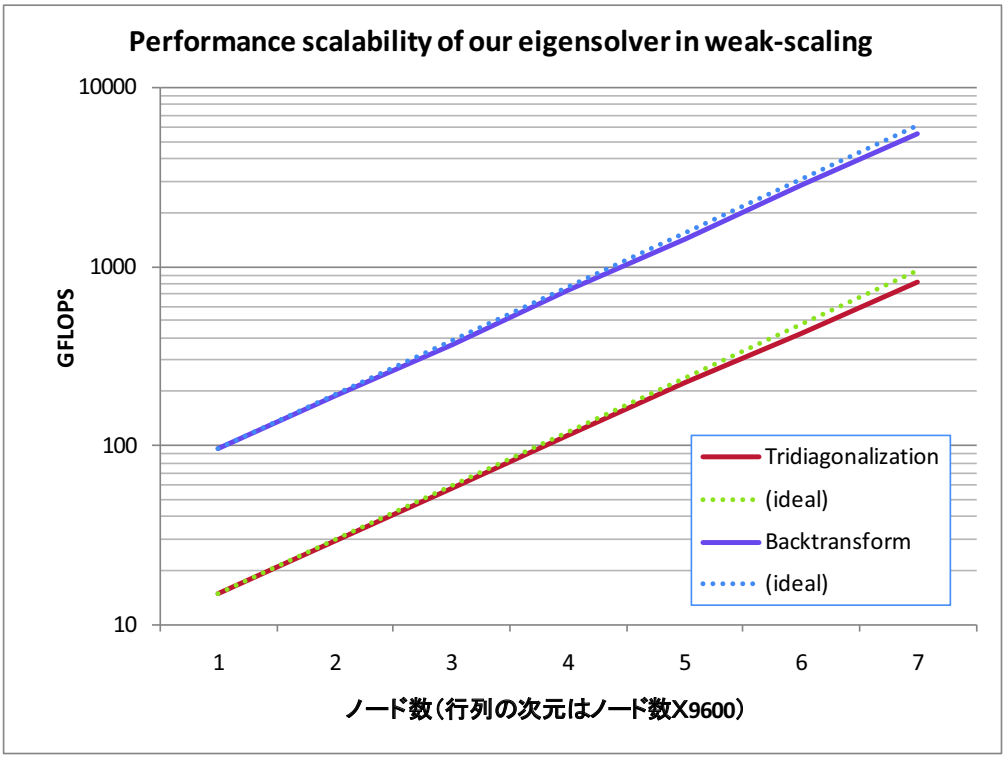

図 8: Weak Scaling on a T2K clutser system at Univ. of Tokyo

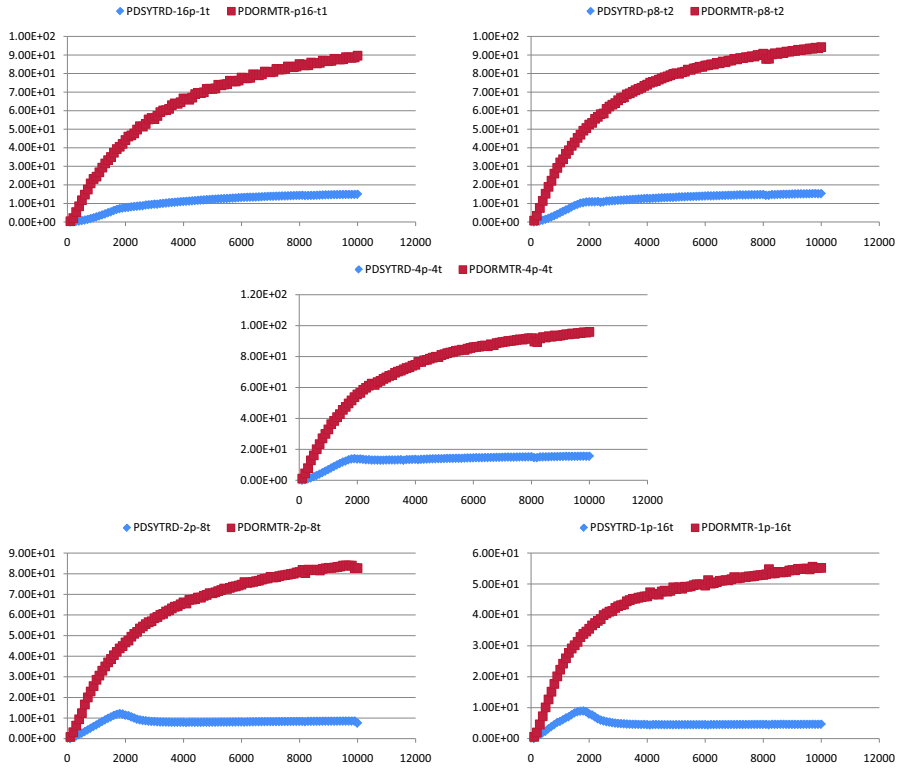

図 9: `eigen_s` の 1 ノードでの性能．TRED1 ならびに TRBAKWY を測定 (左上から 1 ＊ 16, 2 ＊ 8, 4 ＊ 4, 8 ＊ 2, 16 スレッド＊ 1 プロセス)．

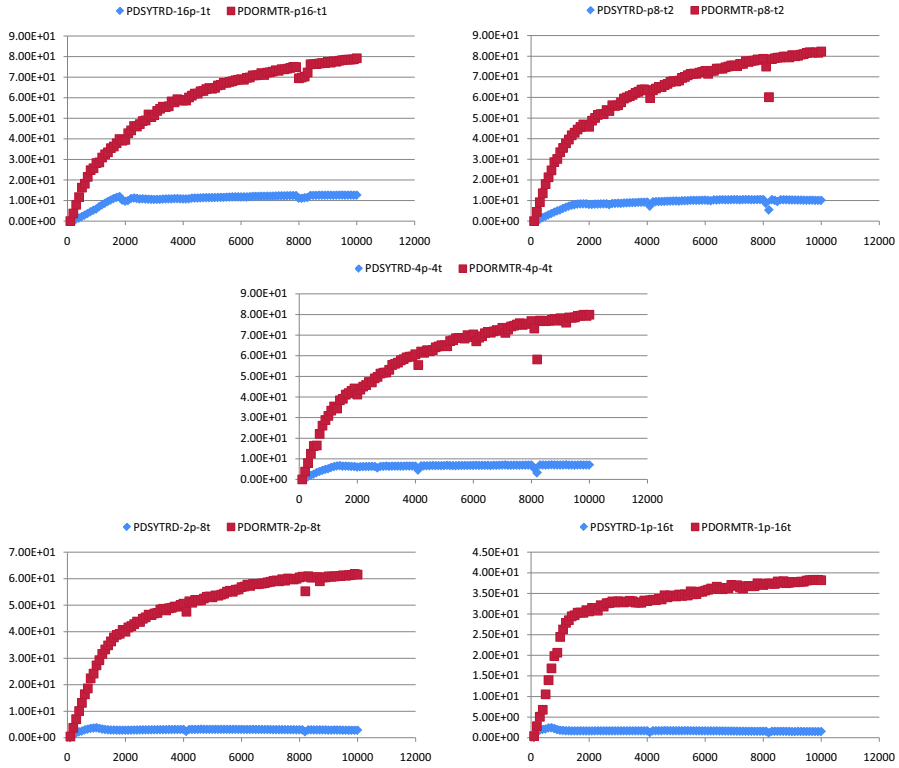

図 10: ScaLAPACK の 1 ノードでの性能．PDSYTRD ならびに PDORMTR を測定

化は殆どないと考えてよい. 参考のために, ScaLAPACK でも同様の計算を行った. 図 10 から, PDSYTRD は「1 スレッド＊ 16 プロセス」が最も高速で, プロセスあたりのスレッド数が増加すると性能が劣化する. PDORMTR は「2 スレッド＊ 8 プロセス」が最良パターンである. また, 4096 と 8192 次元で大きな性能劣化が見られる. ScaLAPACK は下位プログラム群の BLAS でスレッド並列処理を行う. 一方, `eigen_s` は上位レベルでスレッド並列処理を始めるため, メモリの帯域を使いきらない限り個々のスレッドから呼び出される BLAS の性能をフルに使いきれる構造になっている.

数値計算ライブラリ開発の立場からすると, 任意のスレッド数＊プロセス数で最高性能を出す関数を開発したい. しかしながら, ここで示したように現状は容易に高い性能はでない. 特に, 利用者がどのように配列データをメモリ (ソケット) に割り当てているかが判らない状況下では, 複雑なメモリ階層で高速かつ安定的なアクセスは難しい. マルチコア環境下での数値計算ライブラリには, ループの最適化以外にこれまでにはなかったデータ配置の制御と安定的なデータアクセス技術が必要となる. これらの課題を T2K スパコンをテストベッドとして克服していきたい.

4 開発の TIPS(OpenMP スレッド並列化)

`eigen_s` は MPI と OpenMP のハイブリッドプログラミングを実施していることをすでに述べた. プログラム開発の元となったコードはベクトル計算機地球シミュレータで MPI 並列版として作成されていたものであり, ベクトル長を長くするようなベクトル化の戦略を捨て, キャッシュアーキテクチャ向けの修正を行った後, OpenMP でのスレッド並列化は段階的に進めることで対応ができる. 一方, スケーラブルかつ高性能な数値計算ライブラリの立場からは次の様なスレッド並列化が求められる.

- スレッドの立ち上げにかかるコストはできるだけ抑える
- 指定した任意数のスレッドで動作する
- スレッド数で結果が大きく変わらない

などを如何に解決するか. OpenMP でのスレッド並列化に焦点を当てて, プログラミングの TIPS であるが簡単な例を紹介しよう.

4.1 スレッドの立ち上げコストの削減

OpenMP のスレッドは OMP PARALLEL〜OMP END PARALLEL の間 (OMP PARALLEL リージョンと呼ぶ) に生成される (処理系によってはタイミングが異なる). スレッド生成と破壊のコストは大きく, できるだけそのコストを減らすことが求められる. 一般的な OpenMP のテキストには段階を追った並列化がよく記されている.

1. まず, 内側のループから OMP PARALLEL DO で並列化を進める.

```
      SUBROUTINE SUB(Z,V,SS)
      DO I=1,N
         S=SS(I)
!$OMP PARALLEL DO
         DO J=1,K
            Z(J,I)=Z(J,I)+S*V(J)
         ENDDO
!$OMP END PARALLEL DO
      ENDDO
```

2. 次ステップとして OMP PARALLEL リージョンをできるだけループの外側に出す. ここで, 変数の排他制御が必要となり, プライベートであるべきものについては **PRIVATE 節で適切に指定しなくては正しい動作が期待できない**.

```
      SUBROUTINE SUB(Z,V,SS)
!$OMP PARALLEL PRIVATE(S,J)
      DO I=1,N
         S=SS(I)
!$OMP DO
         DO J=1,K
            Z(J,I)=Z(J,I)+S*V(J)
         ENDDO
!$OMP ENDDO
      ENDDO
!$OMP END PARALLEL
```

3. 更に, 関数 SUB 自身がライブラリ関数から呼ばれる子関数であるとき, これを ORPHAN という形で並列化できる. このとき、関数 SUB 内で宣言された変数や配列はプライベートであり, 引数は上位の関数からの属性を引き継ぐ. SAVE や COMMON 文で宣言された変数は SHARED となる. 場合によっては, 適当な個所にバリア同期を入れる必要がある. また, この並列化では関数 SUB を処理するスレッド間での通信手段はなく SHARED 属性のデータを用いることになる. しかしながら, スレッド間での認識がなされない状況下では, SHARED 属性の配列データの担当スレッド以外による領域侵害には気をつけなくてはならない. なお, 場合によっては, OMP DO のワークシェアリング構文すらも自前で書き換え, ループの分割をすることもあり得る.

```
!$OMP PARALLEL
       …
      CALL SUB(Z,V,SS)
       …
!$OMP END PARALLEL
```

```
      SUBROUTINE SUB(Z,V,SS)
      DO I=1,N
!$OMP BARRIER
         S=SS(I)
!$OMP DO
         DO J=1,K
            Z(J,I)=Z(J,I)+S*V(J)
         ENDDO
!$OMP ENDDO
      ENDDO
```

4.2 指定した任意数のスレッドで動作

OMP PARALLEL リージョンに num_threads 句をつけることで, 動作スレッド数を指定することができる. また, nested parallel 指示子を使えば, 複雑なスレッドチームを複製作成することも可能である. 例えば, 1 スレッドは全く別の処理をし, 5 スレッドが 1 チームとして共同動作をするなどは次の様なプログラムでできるようである.

```
     CALL OMP_SET_NESTED(.TRUE.)
!$OMP PARALLEL num_threads(2)
     IF(omp_get_thread_num()==0)THEN
        CALL SINGLE_THREAD_TASK()
     ELSE
!$OMP PARALLEL num_threads(5)
        CALL FIVE_THREAD_TASK()
!$OMP END PARALLEL
     ENDIF
!$OMP END PARALLEL
```

ただし, このとき後発で生成されたスレッドが numactl などのコマンドでどのプロセッサコアに割りあたるのかは今のところ調査が至っていない.

4.3 スレッド数で結果を大きく変えない

数値計算ライブラリを作成する上で, 計算結果に再現性が保証できないことは致命的なエラーとして考えられることがある. OpenMP では REDUCTION 句や CRITICAL, ATOMIC の指定により, 動作が決定的でないプログラムが作成できる. 少なくとも eigen_s では REDUCTION 句による総和計算を使う可能性があるため, 計算順序の違いによる計算結果の違いに苦慮した. 結果的に, 我々は REDUCTION 句を使わずに自前でスレッド間の総和計算を実装した. もし計算結果にシビアな読者がいたら参考にして欲しい.

```
    PSI=ZERO
!$OMP PARALLEL DO REDUCTION(+:PSI)
    DO I=1,N
       PSI=PSI+Z(J)**2
    ENDDO
!$OMP END PARALLEL DO
```

上記の様な単純な総和計算であるが, 以下のように変更することで REDUCTION 句で総和の結果が毎回異なるという事態は避けられる．なお SHARED 配列 TMP はあらかじめ準備が必要である．

```
      PSI=ZERO
!$    TMP(1)=PSI
!$OMP PARALLEL PRIVATE(PSI)
!$    PSI=0.0
!$OMP MASTER
!$    PSI=TMP(1)
!$OMP END MASTER
!$OMP DO
    DO I=1,N
       PSI=PSI+Z(J)**2
    ENDDO
!$OMP ENDDO
!$  TMP(OMP_GET_THREAD_NUM()+1)=PSI
!$OMP BARRIER
!$OMP MASTER
!$  PSI=0.0
!$  DO I=1,OMP_GET_NUM_THREADS()
!$     PSI=PSI+TMP(I)
!$  ENDDO
!$  TMP(1)=PSI
!$OMP END MASTER
!$OMP END PARALLEL
!$  PSI=TMP(1)
```

5　まとめ

以上が T2K スパコンで進めてきた固有値ソルバ開発の一端である．これまで地球シミュレータなどのベクトル計算機を中心に固有値ソルバーの開発を行ってきたため, ベクトル最適化がなされたコードを元にキャッシュ最適化を施す作業をしてきた．T2K スパコンはマルチコアクラスタでもあり, スレッド並列化においてこれまでにないプログラミング技法を積極的に使うことになった．ブロックアルゴリズムの採用によりメモリ帯域不足の解消が基本的な最適化テクニックであるが, 固有値計算にはまだまだブロック化を推し進められる部分がある．現在, もう一つ進行中の `eigen_sx` 固有値ソルバ開発プロジェクトがある．本プロジェクトでは, 従来のアルゴリズムよりも高速でメニイコアにも対応できる

ソルバの開発を目指している. 同ソルバが完成した際にはまた寄稿できればと思います.

最後に, 今回の固有値ソルバ開発は日本原子力研究開発機構の町田昌彦氏, 山田進氏との共同研究のもとになされたものである. また, 本稿執筆の機会を頂きました東京大学情報基盤センターの皆様にも感謝致します.

参考文献

[1] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.O. Fkannery, Numerical Recipes: the Art of Scientific Computing, third edition, Cambridge University Press, 2007.

[2] G.H. Golub, C.F.van Loan, Matrix Computations, third edition, the John Hopkins University Press, 1996.

[3] J. Dongarra, S. Hammarling, and D. Sorensen, Block reduction of matrices to condensed forms for eigenvalue computation. J. Comput. Appl. Math. Vol.27. 215–227, 1989.

[4] HPC Challenge benchmark, `http://icl.cs.utk.edu/hpcc/`

[5] See Top500 benchmark or High Performance Linpack benchmark, `http://www.top500.org/` or `http://www.netlib.org/benchmark/hpl/`, respectively.

[6] J.J. Cuppen, A Divide-and-Conquer Method for the Symmetric Tridiagonal Eigenproblem, Numerische Mathematik 36, 177–195, 1981.

[7] I.S. Dhillon, B.N. Parlett, and C. Vömel, The design and implementation of the MRRR algorithm, ACM Trans. Math. Softw., Vol.32, No.4, 533–560, 2006.

[8] S. Tsujimoto, Y. Nakamura, and M. Iwasaki, Discrete Lotka-Volterra system computes singular values, Inverse Problems, Vol.17, 53–58, 2001.

[9] C. Bischof, and C. van Loan, The WY representation for products of householder matrices, SIAM J. Sci. Stat. Comput. Vol.8, No.1, 2–13, 1987.

[10] ScaLAPACK, `http://www.netlib.org/scalapack/`

[11] 片桐孝洋, ペタスケール計算機環境に向けた固有値ソルバの開発, 東京大学情報基盤センタースーパーコンピューティングニュース, Vol.11, No.1, 2009.

[12] 今村俊幸, マルチコア環境における固有値ソルバ, 計算工学会講演論文集, Vol.14, 2009.

[13] S. Yamada, T. Imamura, T. Kano, and M. Machida, High-Performance Computing for Exact Numerical Approaches to Quantum Many-Body Problems on the Earth Simulator, ACM&IEEE Proceedings of SC|06, 2006.

[14] 町田昌彦, 山田進, 奥村雅彦, 今村俊幸, 密度行列繰り込み群法と行列対角化による強相関量子系のシミュレーション, 東京大学情報基盤センタースーパーコンピューティングニュース, Vol.11, 特集 2, 2009.

[15] 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), 113-121 , 2004.