並列 Hessenberg QR 法のための新しいデータ分割法と AP1000+ への効率的実装

須田礼仁、西田晃、小柳義夫
東京大学 理学部 情報科学科

要旨

Hessenberg 行列に対するダブルシフトQR 法は最も重要な固有値解法の一つで、メモリ量の問題から特に高い並列化効率が望まれるものである。本論文では完全なロードバランスと通信待ちのないパイプラインを実現することのできる新しいデータ分割とそのスケジューリングを提案する。我々は提案手法を用いて QR 法を AP1000+ に並列実装し、本手法がアルゴリズムの持つ並列性が効率良く引き出すことを明らかにした。

A New Data Mapping Method for Parallel Hessenberg QR Method and its Efficient Implementation on AP1000+

Abstract

The double shift QR method for real Hessenberg matrices is one of the most reliable eigensolvers for asymmetric real matrices. It requires high parallel efficiency because of a problem of memory requirements. This paper proposes a new data mapping method and schedule on it, where the load balance is perfect, and the pipeline is streamlined without waiting for receiving message. An efficient implementation of the proposed method on AP1000+ is also presented. It is confirmed that the proposed method utilizes the parallelism of the method better than other researches.

§1 はじめに

非対称密行列の全固有値解法として最も重要で信頼できるものは、Householder 変換を用いて行列を Hessenberg 形に変換し、これにダブルシフトQR 法を適用するという方法である。この方法では両方のステップがおよそ $O(n^3)$ の計算量があり、大規模問題に対しては並列計算の必要性があると考えられる。前半の Householder 変換による Hessenberg 形への変換については効率的な並列実装が可能であることがわかっているが、後半のダブルシフトQR 法の効率的な並列実装の実現は現在までなされていないと言える。その根本的な原因は単純に並列性の低さにある。すなわち、ダブルシフトQR 法は $O(n)$ の比較的低い並列性しか有していない (この事実を「QR 法のスケーラブルな並列実装は不可能である」と表現する人もいる)。しかもその少ない並列性すら有効に利用できるデータマッピング手法が比較的最近まで知られていなかった ([4], 1991 年) ことも効率的並列化の研究の遅れにつながっている。

本研究の目的は、ダブルシフトQR 法のアルゴリズムをそのままの形で効率的に並列化することにある。他のアプローチとして、マルチシフトなどの手法によって並列性を増加させるというものもあり、そのような研究も最近行なわれている。しかしこのような手法では対象とする行列によっては収束性の悪化などの問題が生じる可能性があり、速度向上率や並列化効率が問題に依存するため単純に評価できなくなってしまう。むしろダブルシフトQR 法を最高の効率で並列実装したものの方が行列の性質によらず堅実な速度向上を与えるため、マルチシフトよりも多少遅い場合があっても、安心して使用できると考えることもできる。本研究ではこのような立場に立ち、ダブルシフトQR 法に真正面から取り組むことにした。この場合の目標は並列化効率の改善の一点に尽きる。なぜなら、これまでに得られている並列化効率はあまりにも低く、とても使用に耐えないからである。また並列化効率が低いとメモリの問題も生じてくる。並列 QR 法の並列化効率は行列サイズ n とプロセッサ数 p に対して n/p でほぼ決まるので、一定の並列化効率を維持するために $n/p = r$ と固定すると、1 プロセッサあたりのメモリ量は rn となる。並列化効率が低いとこの r が大きくなってしまい、大きな n に対して必要なメモリがとれなくなってしまうという事態が生じてしまうのである。このため並列化効率を特徴付ける $r = n/p$ を小さく抑えることが実用上非常に重要となるのである。

我々は計算効率の改善のために、まずデータマッピングを改良し、良いロードバランスを実現すると同時に、通信レイテンシを完全に隠蔽して計算がパイプライン状に淀みなく流れることを可能とした。さらに AP1000+ の高速通信手法である put や put_stride を最大限に利用してオーバーヘッドを最小限に抑える工夫をし、高い並列化効率を目指した。その結果我々の実装ではこれまでの他の研究に比べてはるかに高い並列化効率を実現し、並列化効率 50% を与える $r = 40$ は [6] で報告されているもののおよそ 1/4 である。本論文ではこの実装について詳細に論ずるが、この節ではその準備としてまず基礎として用いた EISPACK の Hessenberg ダブルシフトQR 法の計算の概要を示し、これまでに提案されているデータマッピング手法を紹介する。

§1.1 Hessenberg QR 法

Hessenberg 行列に対する QR 法にはいくつかの改良版があるが、ここでは代表的なライブラリの一つである EISPACK の HQR を基礎に置く[1]。このルーチンは Hessenberg 行列を入力とし、ダブルシフト QR 法によって固有値を求めるもので、我々の並列プログラムはこのルーチンと完全に同一の計算を行なうように作成してある。Hessenberg 行列 A に対するダブルシフトQR 法の一反復の変換は

$$H = P_{n-1}P_{n-2}\cdots P_2P_1AP_1^TP_2^T\cdots P_{n-2}^TP_{n-1}^T$$

と書くことができる。以下これ全体を「一反復」、P_i を左右から掛ける計算のそれぞれを「1 ステップ」と呼ぶことにする。ここで P_i は Householder 変換行列 $P_i = I - 2w_iw_i^T$ であって、w_i は $w_i^T = (0,0,\cdots,0,\alpha_i,\beta_i,\gamma_i,0,\cdots,0,0)$ のような 3 つの連続した要素以外は 0 であるようなベクトルである。最初の非零要素 α_i はベクトル w_i の第 i 要素にあたる。従って最後の変換 P_{n-1} には γ_i は存在しない。

最初の変換 P_1 の 3 つの非零要素は右下隅の 2×2 小行列と最初の 3×3 小行列の値から決まる。この変換をすると $a_{3,1}$, $a_{4,1}$, $a_{4,2}$ が非零となる。それ以外の変換 P_i では行列の第 $i-1$ 列の対角より下の 3 つの要素 $b_{i,i-1}$, $b_{i+1,i-1}$, $b_{i+2,i-1}$ から決まる。P_i の変換の後、下の 2 つの要素である $b_{i+1,i-1}$ と $b_{i+2,i-1}$ は 0 となるが、そのかわり $b_{i+3,i}$ と $b_{i+3,i+1}$ が非零になる。

1 ステップの変換 $P_iBP_i^T$ での行列の更新は図 1 のようになる。ただし最初と最後は多少計算パターンが異なる。更新前の行列は実線で示すように Hessenberg 行列からすこしコブがでた形である。1 ステップの計算で更新される要素は網を掛けた 3 行 3 列の L 字型の領域である。データの依存関係を四角で囲むと図 1 の右図のようになり、この図の四角のどれかが複数のプロセッサに跨る時に通信が必要となる。更新される要素のうち、行と列がクロスする網掛けの濃い部分を枢軸部分と呼ぶことにする。この枢軸部分はほぼ全対全の依存関係があるので、複数のプロセッサに分割しない方がよいと考えられる。枢軸部分が計算されていなければ次ステップの変換が決まらないので、枢軸部分の計算は他の部分に優先して計算する必要がある。

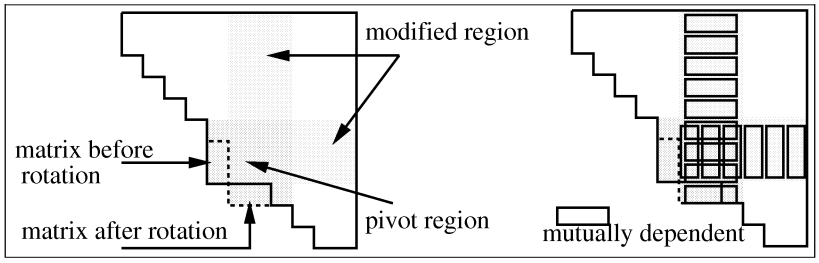

図 1. QR 法の 1 ステップ

§1.2 これまで用いられてきたデータ分割

効率的に並列計算を行なうためにはロードがバランスしており、通信が少ないことが必要である。行列計算の多くは列ごと、行ごと、あるいは二次元にデータを分割することによりこれが実現できる。しかし Hessenberg QR の場合には上三角のみが計算され、しかも一度に 3 行 3 列しか更新されないため、通常の分割では効率が悪い。これに対し、1991 年に van de Geijn [4] は block Hankel-wrapped storage scheme を提案した。これは第 i,j ブロックをプロセッサ $(i+j)/m \bmod p$ に割り当てる (m は正整数)。例えば $p=4, m=2$ の場合図 2 の左の図のようになる。この方法はロードバランス、通信量ともオーダーとしては最適である。Wu と Chu [5] はこれとは逆に主対角方向にデータをならべて図 2 の右の図のようにしたが、これは van de Geijn のものと比べてロードバランスも通信量も少しずつ悪い。

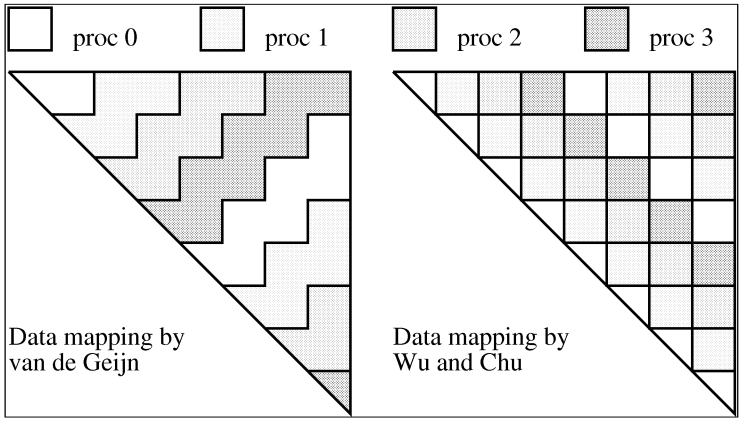

図 2. これまでのデータ分割手法

先にも述べたように、枢軸部分が計算されていれば次ステップの変換が決まる。そこで枢軸部分の計算を優先させて次ステップの変換を先行して決定することによって計算をパイプライン化し、変換情報の通信のレイテンシを隠蔽することが可能となるはずである。しかし実際には単純に枢軸部分の計算を先行させるだけではうまくパイプラインは流れない。それは次ステップの枢軸部分の計算を先行させた分、現在のステップの計算の一部が遅らされているので、それが次々ステップの枢軸部分の計算を遅らせるためで

[1] このプログラムは例えば `http://www.netlib.org/eispack/` などから入手することができる。

ある。この問題を解決するにはこのよ うなパイプラインスケジュールをはじめから考慮に入れたデータ分割をしなければならない。上記のこれまでに提案されてきたデータマッピングの手法はこのよ うなパイプラインスケジュールを考慮していないため、枢軸部分の計算を先行して行な うことができず、枢軸を担当していないプロセッサが遊んでしま う。次の枢軸部分を先行して計算することをも含めてロードバランスを考えたデータ分割でなければ高い並列化効率は出せないのである。

§2 並列 Hessenberg QR 法のための新しいデータ分割

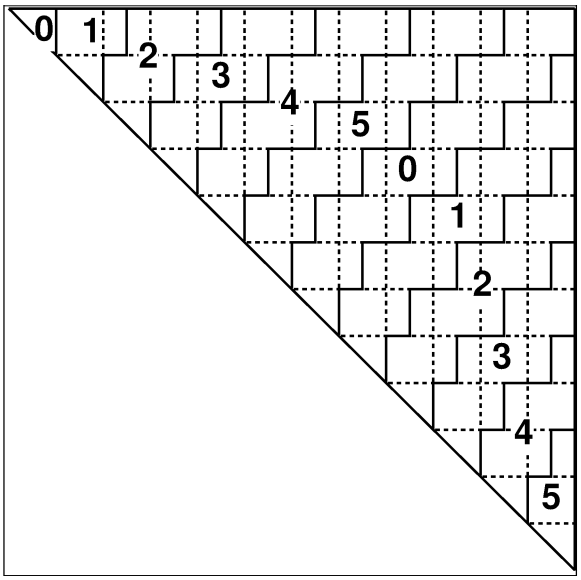

図 3. 提案するデータ分割

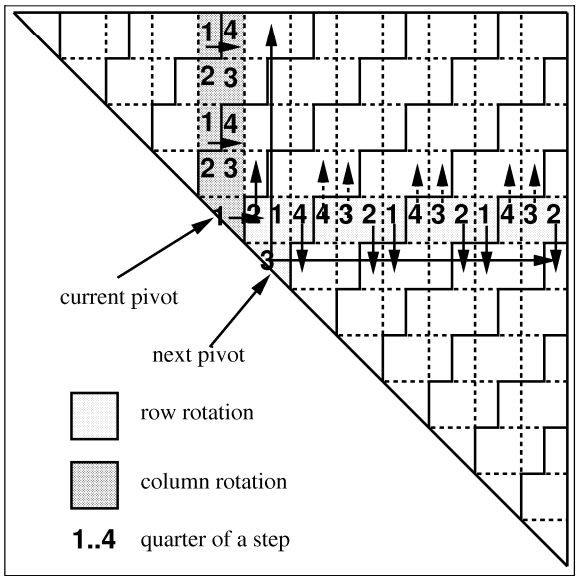

図 4. 提案するスケジューリングと通信の様子

図 3 は本論文が提案するデータ分割を示している。この方法ではプロセッサ数 p に対して行列を $2p \times 2p$ のブロックに分割したものを基礎とする。本論文では 1/2 ブロックの変換を計算量の単位に用いるので、これを 「ハーフブロック」と呼ぶことにする。対角ブロックは要素がおよそ半分しかないから、これだけでハーフブロックとなるが、それ以外ではブロックを縦に 2 分割する。図 1 はプロセッサ数 $p = 6$ の場合を示しており、破線はブロックを、実線はデータ分割の区切りを、数字はそのデータを担当するプロセッサの番号を示している。行列は斜めの帯状に分割され、各プロセッサが p 離れた帯を 2 つづつ扱うが、これは van de Geijn の分割とよく似ている。対角付近を除くと van de Geijn の分割よりも 1.5 ブロック左にずれており、これにより対角付近のロードが下げられて枢軸部分を先行して計算しやすくなっている。

図 4 は図 3 と同じ設定で第 5 ステップの変換を行な う際のスケジューリングを示している。このスケジューリングでは 1 ステップを 4 つの四半期に分け、おのおのの四半期で 1 ハーフブロックの計算をする。図ではこの 4 つの四半期に計算するハーフブロックを 1 から 4 の数字で示している。矢印は計算結果の送信を示している。次の第 6 ステップの枢軸部分はプロセッサ 5 が第 3 四半期に計算する。その際対角に極めて近い要素以外は列変換は行なわないので、この次ステップの変換行列の計算はほぼ 1 ハーフブロックの計算量となる。これが第 3 四半期にスケジュールされることにより変換行列の計算からそれを用いるまで 1 四半期分の余裕ができる　第 3 四半期で計算した結果を次のステップの第 1 四半期に使用するため。以下、これを $3 \to 1^+$ とあらわす) ので、通信遅延を隠蔽することが可能となる。残った列変換はプロセッサ 4 のように、当該のステップの最初に行な う。この結果は次のステップで隣のプロセッサが列変換に用いるが、$1 \to 2^+$ の 1 ステップ分の余裕がある。

行変換は各プロセッサで右から左に実行する。これは右の 2 つのハーフブロックの計算結果を次のステップで隣のプロセッサが使用するからである。この順序によって $2 \to 3^+$ の 1 ステップ分の余裕ができ、通信遅延は完全に隠蔽できる (枢軸の付近ではスケジューリングが異なってきわどくなっているが、それでも $4 \to 2^+$ の 1 四半期分の余裕がある)。一方、左の 2 つのハーフブロックの計算結果は後の列変換の際に必要となるが、これは枢軸付近以外では次々ステップ以降であるし、枢軸付近でも $2 \to 1^+$ のように 2 四半期分の余裕があり、充分通信遅延を隠蔽できる。

列変換は各プロセッサで下から上に実行する。これは最も下のハーフブロックの計算結果を同じステップで隣のプロセッサが使用するからである。このスケジューリングでは $1 \to 4$ の 2 四半期分の余裕がある。このように、すべての通信で送信から受信まで 1 四半期分以上の余裕があるため、オーバーヘッドが極端に大きくない限り受信待ちがまったくない滑らかなパイプラインが実現される。

図 5 は $p = 2$ の場合のスケジューリングの全体である。これからわかるように、最初と最後のステップは事情が若干異なる。最初のステップではプロセッサ 0 が最初の変換行列を計算するだけで、これを他

のプロセッサが待つ以外に方法がない。また、最後のステップではプロセッサ 0 だけは 3 ハーフブロックしかない。これらのオーバヘッドは全体の計算量の $O(1/p)$ なのでプロセッサ数が多くなると余り問題にならなくなる。しかし、ここで生じた空白の時間に次の反復での最初の変換行列を計算することによりパイプラインを完全に詰めることは可能である。これは QR 法のある種の最適化を妨げてしまうので、EISPACK の HQR の並列化を目標としている本論文ではこのパイプライン化についてはこれ以上考えない。

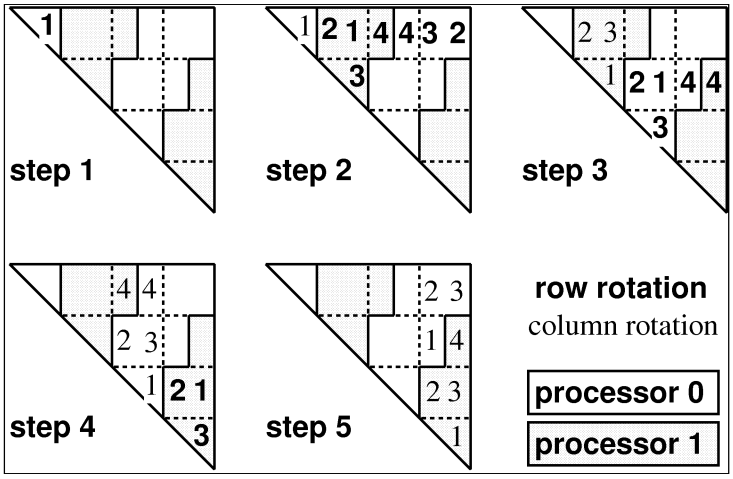

図 5. 2 プロセッサの場合の一反復のスケジューリングの全体

§2.1 データ分割手法の拡張

以上で説明したデータ分割手法は行列を $2p \times 2p$ のブロックに分割したものを基礎にしていたが、これは自然数 k を導入して $2kp \times 2kp$ の細分化されたブロックを基礎とすることが可能である。この場合 k 倍の数の帯をサイクリックまたはブロックに用いるという選択肢があるが、ブロックの方が通信量が少ないので大抵の場合ブロックにした方がよい。

いずれの方法にせよ最初と最後のロードのバランスの悪さが改善される。また通信の余裕も相対的に大きくなる。例えば $k = 2$ とすれば各プロセッサは 1 ステップで 8 ハーフブロックを計算することになるので、変換情報の通信の余裕は $3 \rightarrow 1^+$ で 5 八半期分となる。しかしブロックサイズが小さくなるためループが短くなって計算性能が相対的に低くなったり、雑多なオーバーヘッドで k 倍になるものがあるなど、細分化することが良いとは一概には言えない。

§2.2 HQR における計算の最適化の問題

以上のデータ分割は行列が $n \times n$ であることを仮定している。しかし、実際にはこれほど話は簡単ではない。基礎としている EISPACK の HQR は 3 つの最適化を施しているからである。第一に、固有値が求まるごとに減次を行なっている。これは単に行列の最後の列行をなくすだけであるが、変換する範囲は徐々に小さくなるためそのままではデータ分割が上述のものとは異なってしまう。第二に、各反復において下対角要素で 0 のものがある場合、そこで行列が二つのブロックに分割されてしまう。この場合もデータ分割が理想的なものとは異なってしまう。なお、この場合分割された二つの対角ブロックが独立となるため、それぞれ並列に計算を続行する方法も考えられる。しかし、EISPACK では完全に 0 とならなくても付近の要素に比べ十分小さくなれば良いとしており、行列が更新されることにより一度分割されたブロックが再び一つになることがあるため、このような並列化の手法は適用できない。第三に、下三角要素が 0 でなくても一定の条件を満たせば、そこから上の部分は行変換を省略することができる。これらの最適化により、実際に計算される領域は図 6 に示すような台形の領域に制限されてしまう。以下では EISPACK の変数名にしたがって、図に示すように減次後の行列サイズを n, 第二・第三の最適化の行なわれる位置を l および m とする。

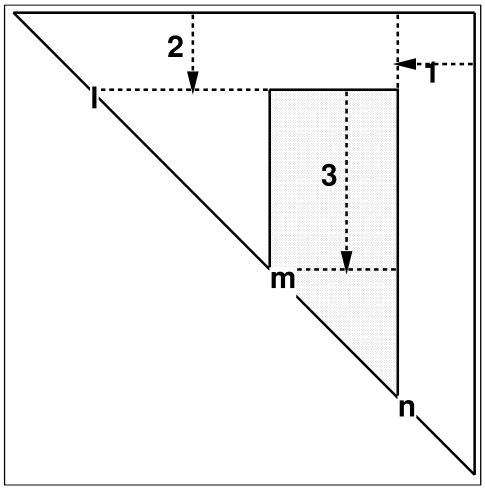

図 6. 最適化による計算領域の縮小

これらの最適化のうち、第三の最適化は計算量が減るだけで並列性に影響しないが、第一・第二の最適化が生じると計算内容が想定していたものと異なり都合が悪い。この問題に対しては、データを再分配する方法とそのままにしておく方法とが考えられる。第二の最適化では l の変化は容易に予測できないので、再分配は慎重に行なわなければならない。第一の最適化では n は一度に高々 1 ずつしか変化しないため再分配も可能であるが、行列が極端に小さくなった場合にはプロセッサ数を減らした方がよいと考えられるので、問題はさほど簡単ではない。一方、前節で説明したようにデータ分割を細かくしてサイクリックにプロセッサに割り当てれば、n が小さくなった時のロードバランスの悪化を低減することが可能である。しかしこの方法では通信のオーバーヘッドが増大し、通信の余裕も絶対時間として短くなるなど問題も多い。最適化に伴うこれらの問題は現在未解決であって、今回の実装の際にはデータの再分配は行なわず、理想的でないデータ分割のまま計算を続行する方法をとった。これに関しては今後研究をさらに進めてゆきたいと考えている。

§3 HQR の AP1000+ への効率的並列実装

さて本節では提案したデータ分割を用いた HQR の AP1000+ 上への効率的実装方式について説明する。実装にあたって 2 つの点に注意した。まず、通信のオーバーヘッドを最小限に抑えることである。AP1000+ には put というDMA を用いた高速の通信ライブラリがある。CPU が put を発行した時にはDMA にパラメタが設定されるだけですぐに処理がプログラムに戻ってくるので、この put を用いれば通信のための CPU 時間の消費が極めて少ない。しかしさらにバッファ処理や put の関数呼び出しの所要時間も抑えたい。もう一つは、分割を細かくした際に生じる計算のオーバーヘッドを最小限にすることである。これはキャッシュなどの問題からある程度避けられないが、境界部分の扱いをできるだけ特別なものにしないようにすることでコンパイラの最適化をしやすくする効果をねらっている。なお、コンパイラは gcc でオプションとして`-O2 -funroll-loops -msupersparc` を用いた。

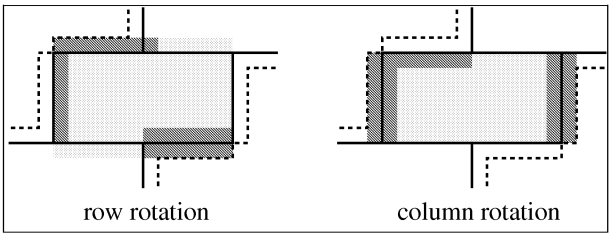

図 7. 計算の割り当て

QR の 1 ステップは 3 行 3 列の計算であるため、時にはプロセッサを跨いだ計算が必要となる。プロセッサを跨ぐ計算をどちらに割り振るかによっていくつかのデータ分割が考えられるが、今回は図 7 のように割り当てた。図は中央のブロックが行変換・列変換を行なう際に、所有しているプロセッサが計算を行ない更新する部分に網を掛けて示している。網掛けの濃い部分は計算後にデータを送る部分である。破線はこのプロセッサが余計に持たなければならない「縁」の範囲をあらわしている。この縁を含めてデータ領域を取るとすると、プロセッサが確保する領域は図 3 で示されていた長方形のブロックを単純に並べただけのものではなくなるので、通常の 2 次元配列では効率的に実現できない。今回の実装では C 言語のポインタを用いて行列を `double **matrix;` のように 1 次元配列へのポインタの配列とすることにより、複雑な領域を統一的に扱うこととした。これによって領域の縁の部分を特別扱いする必要がなくなり、主要な 2 重ループが非常にシンプルになった。

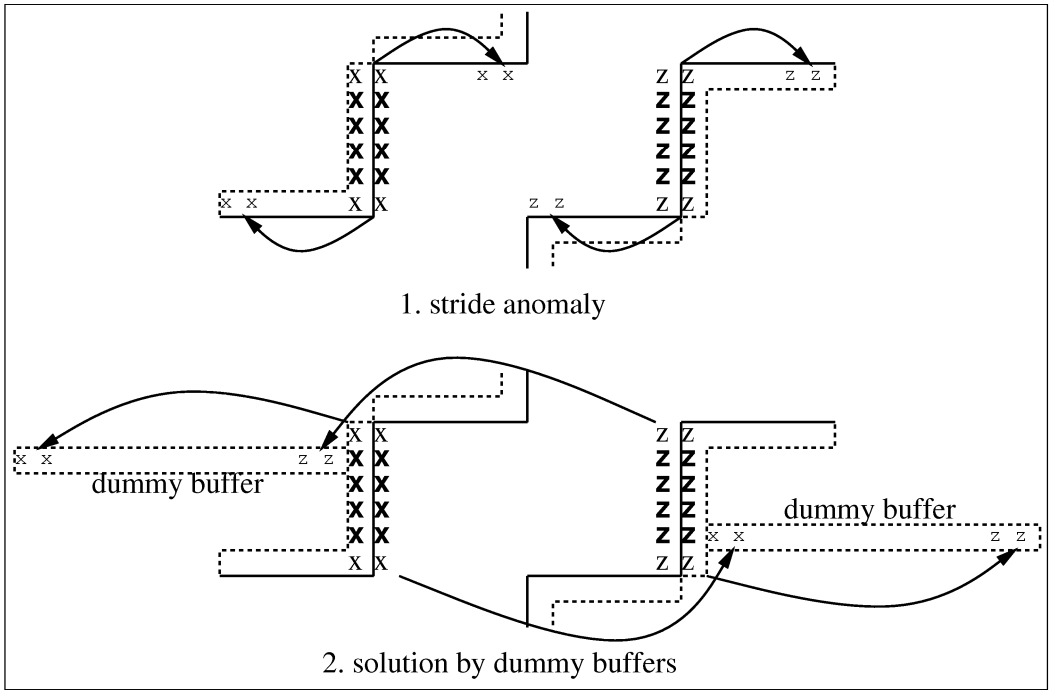

図 8. ストライド転送のためのダミーバッファ

通信のオーバーヘッドを最小限に抑えるためには、AP1000+ では put を用いるのが有効である。列行ともに縁の領域があるが、長方形の配列の場合には一定間隔でならんでいる複数の一定幅のメモリ領域を一度に転送する put_stride を用いれば列行どちらを通信する場合もバッファリングは不要となる。しかし、今回はデータ領域が不規則な形状をしているため領域の最初と最後の行は幅が異なり、一定間隔のストライド転送を用いて送信することはできない。図 8 の上図はこの状況を示しており、ストライド転送では小さい文字で示された位置のデータが転送されてしまう。この問題は図 8 の下図に示すダミーバッファを追加することにより解決できる。ダミーバッファの長さはストライドと同じに取って最初と最後の行のデータがちょうどダミーバッファと対応するようにする。転送の前後に本体の配列とダミーバッファとの間でデータをコピーすることにより、ストライド転送で列のデータを送ることができるようになる。

§4 性能評価

この節では AP1000+ 上に実装された並列 HQR の性能評価を行なう。基本的には並列化効率を指標とするが、今回の評価では以下の 2 つの条件を変えて行なった。まず、問題として全固有値を求めるもののほか、最初に固有値が求まるまでの計算のみ（以下、一固有値）のデータも取った。一固有値の場合、§2.2 で述べた最適化はほとんど働かないため、提案したデータ分割が理想的な形で働く。これによって提案したデータ分割で想定通り受信待ちなしにパイプラインが進むことを確認することができるほか、今回行なわなかったデータの再分配を今後行なった場合の性能を予測する上で有用な情報を得ることができる。もう一つの実験条件は一台での性能の測定条件である。行列が大きくなるとキャッシュなどの影響でブロック化を行なった方が性能が高い。並列化効率としては一台の場合に最適なブロック化を行なったものを参照すべきと考えるが、他の論文のなかにはブロック化していない場合もあるので、本論文では両方を併記することにする。なお、行列は [0, 1) の一様乱数を用いて Hessenberg 行列を生成している。

表 1. 計算時間（十固有値）time: ms

proc	size	seri	para	block	eff
2	80	4.261	3.918	(20 1)	0.5438
2	160	15.40	10.83	(20 2)	0.7108
2	240	34.51	21.44	(30 2)	0.8055
2	320	61.15	35.18	(40 2)	0.8692
2	400	96.42	52.68	(50 2)	0.9152
4	160	15.40	7.325	(20 1)	0.5257
4	320	61.15	20.93	(40 1)	0.7303
4	480	138.9	41.46	(30 2)	0.8374
4	640	250.2	67.44	(80 1)	0.9276
4	800	389.0	102.3	(100 1)	0.9501
8	320	61.15	14.35	(20 1)	0.5329
8	640	250.2	40.98	(40 1)	0.7632
8	960	558.1	80.91	(60 1)	0.8621
8	1280	987.5	132.3	(80 1)	0.9328
8	1600	1525	201.5	(100 1)	0.9464
16	640	250.2	28.57	(20 1)	0.5475
16	1280	987.5	81.27	(40 1)	0.7595
32	1280	987.5	57.28	(20 1)	0.5387

表 1 は一固有値の計算時間を示している。実際は何回かの反復のうち、一反復にかかった時間の中央値を ms 単位で示してある。列は左からプロセッサ数、問題サイズ、一台での所要時間、並列での所要時間、ブロックサイズ、並列化効率である。ブロックサイズが $(b\ k)$ となっているのは b が基礎のブロックサイズ、k は §2.1 で示した細分化のパラメタで、細分化された帯はブロック分割で割り当ててある。一台での時間は最適なブロック化を行なったものである。表のように、n/p が 120 程度以上では効率が 80%, 160 以上では 90% を越え、高い並列化効率が実証されている。

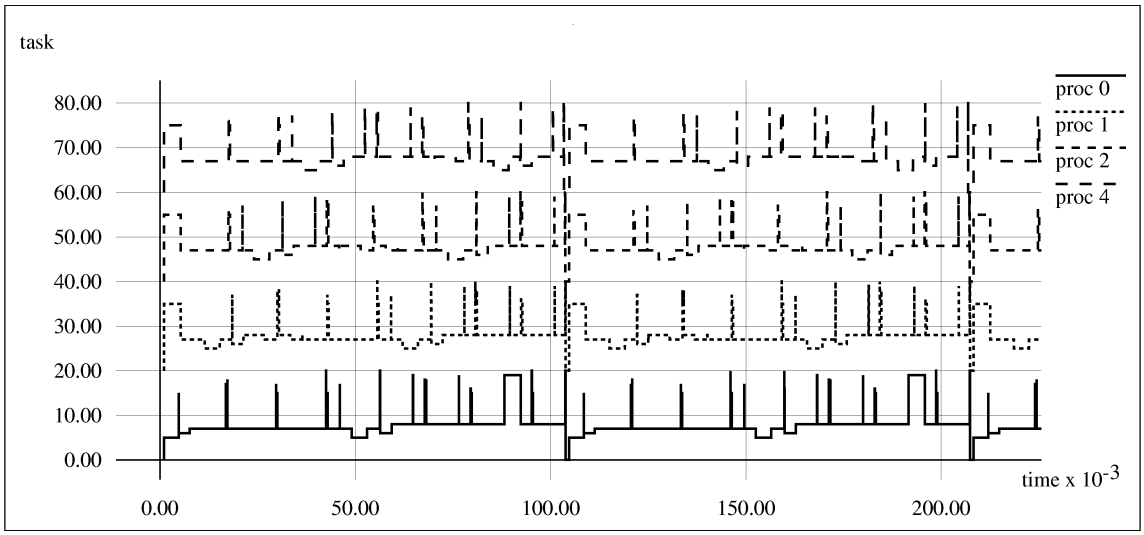

図 9. トレースの結果

図 9 は 800 元の行列を 4 台で解いた時のおよそ 2 反復分の実行状況のトレースである。横軸は時刻、縦軸は仕事の種類を示している。上に向かって針が立っているように見えるのは受信の部分で、大抵の場合データの到着が受信より早く、すぐに制御が計算に戻るためこのように見える。受信待ちの場合はグラフが台状になるが、これは反復の最初と最後のステップにすこしずつみられる。このように受信待ちはほとんどなく、パイプラインがきれいに流れていることがわかる。

表 2. 計算時間（全固有値）time:s

proc	size	seri1	seri2	para	block	eff1	eff2
2	120	0.864	0.864	0.794	(31 1)	0.544	0.544
2	200	3.539	3.546	2.622	(43 1)	0.675	0.676
2	280	9.225	10.596	6.101	(31 2)	0.756	0.868
2	360	20.152	24.970	11.926	(38 2)	0.845	1.047
4	240	6.172	6.640	2.961	(27 1)	0.521	0.561
4	400	27.144	37.773	10.330	(45 1)	0.657	0.914
4	560	68.279	115.730	24.009	(61 1)	0.711	1.205
4	720	143.717	270.928	47.403	(77 1)	0.758	1.429

表 2 は全固有値を求めるのにかかった時間を秒で示している。列のうち seri1 と

eff1 は一台での時間として最適なブロック化を行なったもの、seri2 と eff2 は一台ではブロック化を行なわなかったものである。後者の評価では並列化効率が 1 をはるかに越えてしまうことからも、提案した方式の高い並列性がわかる。これらの結果は一固有値の結果をはるかに下回っているが、これはデータ分散がアルゴリズムの最適化によって壊されているのが原因である。このためデータの再分配を行なうことによりさらに性能の改善が可能であると考えられる。

上の表ではプロセッサ数が 2 から 4 と少ないが、さらにプロセッサ数が多くなってもほぼ同様の並列化効率が得られることを示したい。しかし全固有値問題は時間がかかるため 1 台での最適なブロック分割を求めることが非常に困難であるので、上記のような並列化効率を求めることが事実上できない。そこで、ここでは並列化効率が n/p でほぼ決定されることを確認することを目的とし、ブロックを細分化しない場合について、単純に計算時間をプロットすることにした。図 10 はこのようにして計測した全固有値問題に対する計算時間を示している。行列のサイズが 1920, 960, 480 の場合について、プロセッサ数を変化させた場合の n/p と実際の所要時間の関係を折れ線で、($n = 1$ の代わりに) $n/p = 120$ の場合の計算時間を基準とするリニアスピードアップを直線で示している。このグラフからわかるように、プロセッサ数が変化した場合の計算時間の変化はほぼ n/p で決定されており、大規模並列環境でも提案した手法が有効に働いていることがわかる。なお、行列が大きくプロセッサ数が多い方が相対的な並列化効率がすこし高くなっているが、これはおそらく $O(1/p)$ となる最初と最後のステップのためのオーバーヘッド (§2.1 の直前で述べた) の影響と思われる。

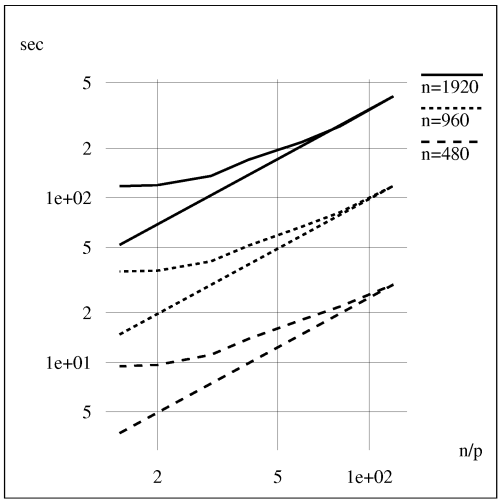

図 10. 計算時間と n/p との関係

図 11 はいくつかの論文に報告されている実装と並列化効率を比較したものである。Wu (n-block) は Wu and Chu [5] によるもので、彼らのデータ分割では一台の場合にはブロック化されない。このため我々の結果の ours (n-block) が比較として適当と思われる。彼らはプロセッサとしては iPSC/2 を用いている。Henry (5iter) は Henry と Geijn [6] のもので、一台の性能として LAPACK の出す最高性能として 10MFlops を用い、最初の 5 反復の性能を出している。我々の結果では ours (1 val) としたものが対応するものと考えられる。プロセッサは Paragon XP/S 140 である。このほかの論文には Boley らのもの [2] があるが、これは固有ベクトルまで求めているため比較の対象にはならない。また [3, 1] などは性能が低過ぎて並列化されているとは言い難く、比較にならない。

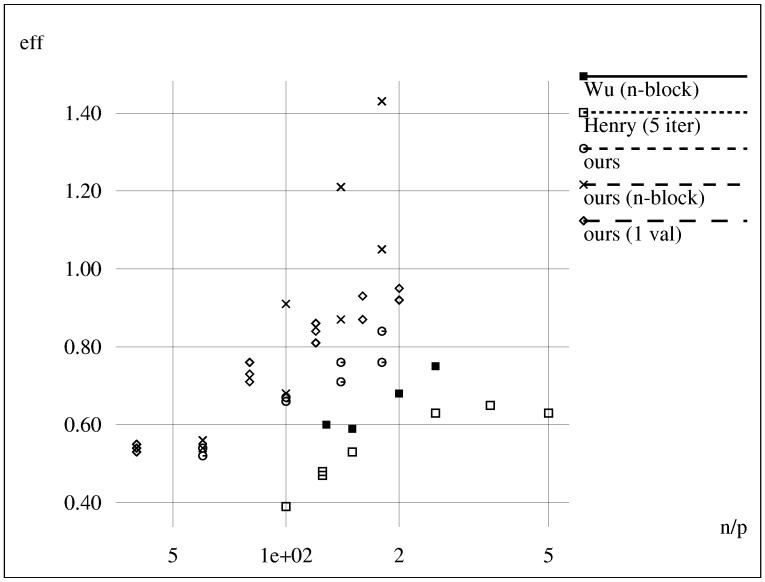

図 11. 並列化効率の比較

これらのデータは並列化効率という同一のスケールで与えられているものの、使用している計算機、逐次の場合のインプリメント、実際のプロセッサ数や行列サイズ、行列の性質などがそれぞれ異なるため単純に比較はできない。しかし我々の結果はこれまでのデータをはるかに上回る高い並列化効率を出しており、我々の手法によって QR 法の持っている並列性が極めて効率良く引き出されていることがわかる。また §1 で述べたように Hessenberg QR では並列化効率が低いと 1 プロセッサあたりに必要なメモリ量が増大してしまうという問題があるため、Hessenberg QR では並列化効率というものの重要性が他の問題と根本的に異なっている。我々の実装では並列化効率 1/2 を与える n/p が Henry ら [6] のおよそ 1/4 になっており、メモリ必要量の問題も格段に軽減されている。

§4.1 残された課題

以上のように今回提案したデータ分割とスケジューリングはこれまでに知られていた方法よりもはるかに優れていることが実証された。しかし未解決の問題は沢山ある。第一に、ブロックを細分化したものが

予想以上に性能が低かったことが問題である。少ないプロセッサ数で荒い分割を行なうとパイプラインの立上りのオーバーヘッドが問題になるため細分化した方が良いと予想されたが、実際には細分化しない方が良いことが多かった。細分化した時に性能があまり良くない原因は今のところ明らかではないが、行変換が列変換よりもなぜか速いこと（これは [6] でも報告されている）が原因の一部なのかも知れない。

第二に、データの再分配を行なって常に理想的なデータ分割を維持することが課題である。再分配にはかなりのオーバーヘッドが予想されるので再分配が有効かどうかは実験をしてみなければわからない。また、再分配の場合には行列サイズ n が小さくなった時にプロセッサ数を減らすことも検討する必要があると思われる。これはかなり詳しく検討し、実験も慎重に行なって評価を行なう必要があると思われる。第三に、ブロック内をタイリングして性能を向上させることが必要である。QR 法では行列・行列積を用いないためタイリングによる高速化は比較的少ないと予想されるが、ブロックサイズによる絶対性能の違いをなくして性能の解析をしやすくする効果があるかもしれない。

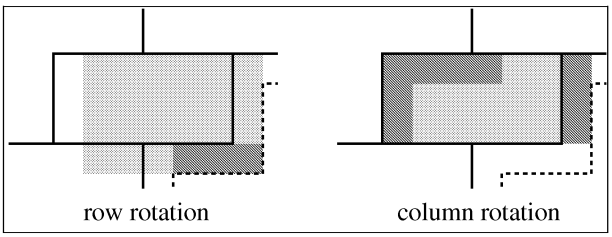

図 12. 改良した計算割り当て

第四に、図 12 に示す改良した計算割当の実装が挙げられる。ここでは行変換が二要素右にシフトしており、列変換の範囲も一要素下に移動している。これにより通信量と通信回数を現在のものに比べて削減することができる。この割当を用いてもダミーバッファにより put_stride を使用することができる。また、図 13 は提案したデータ分割と実質的に同じ分割で、スケジューリングも全く同じものが使える。これは斜めに切っているため最も重要な二重ループが複雑になってしまうが、縁の領域は半減し、上記の改良した計算割り当てに比べても通信量と通信回数は 3/4 になるという利点がある。内部をタイリングすることで欠点を解決できるのであれば有効である可能性がある。これらの課題の解決と共にさらに高い並列化効率を目指して今後も研究を行ないたい。

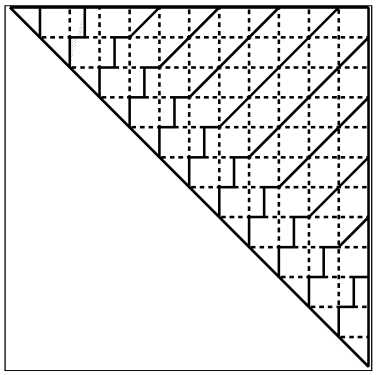
図 13. 別のデータ分割

§5 まとめ

本論文では Hessenberg 行列に対するダブルシフト QR 法に対してこれまでよりもはるかに性能の良いデータ分割とスケジューリングを提案した。このデータ分割は次ステップの枢軸部分の先行計算を含めてロードバランスを取ることができ、必要な通信では 1/4 ステップ以上の余裕を持たせて通信遅延を隠蔽させることが可能である。我々はこれを AP1000+ に実装することにより、全く待ちのないなめらかなパイプラインが実現されることを実証した。本手法によりこれまでに他の研究で報告されているものよりもはるかに高い並列化効率が実現され、並列 QR 法の問題であるプロセッサあたりのメモリ量の問題も格段に軽減されたことになる。さらに高い並列化効率と性能を目指して今後も研究を進めたい。

謝辞

並列実装に関して貴重な助言をして下さった建部修見氏 (電総研) に感謝致します。

参考文献

[1] Z. Bai and J. Dammel, “On a block implementation of Hessenberg multishift QR iteration,” *International Journal of High Speed Computing*, Vol. 1, No. 1 (1989) 97–112.

[2] D. Boley, R. Maier and J. Kim, “A parallel QR algorithm for the nonsymmetric eigenvalue problem,” *Computer Physics Communications*, 53 (1989) 61–70.

[3] G.A. Geist and G.J. Davis, “Finding eigenvalues and eigenvectors of unsymmetric matrices using a distributed-memory multiprocessor,” *Parallel Computing*, 13 (1990) 199–209.

[4] R. A. van de Geijn, “Storage schemes for Parallel Eigenvalue Algorithms,” *Numerical Linear Algebra, Digital Signal Processing and Parallel Algorihtms*, NATO ASI Series Vol. F70, Springer-Verlag 1991, 639–647.

[5] L. Wu and E. Chu, “New distributed-memory parallel algorithms for solving nonsymmetric eigenvalue problems,” Proc. 7th SIAM Conference on Parallel Processing for Scientific Computing, 1995, 540–545.

[6] G. Henry and R. van de Geijn, “Paralllelizing the QR algorithm for the unsymmetric algebraic eigenvalue problem: myths and reality,” *SIAM J. Sci. Comput*, Vol. 17, No. 4, pp. 870–883, July 1996.