計算機アーキテクチャ 123－3
（1997. 3. 7）

# 分散メモリ型並列計算機による固有値計算のための ブロック化 Householder 法 の性能評価

片桐孝洋† 金田康正††

この論文では，一般行列を Hessenberg 形に変換するためのブロック化 Householder 法の並列アルゴリズムを説明している．またこのアルゴリズムを分散メモリ型並列計算機 AP1000+ に実装して，性能を評価した．その結果，ブロック化アルゴリズムの性能を引き出すためにはループアンローリングが効果的であること，データ分割方式としてサイクリック分割方式がブロックサイクリック分割方式に対して有効となることがわかった．

# Performance Evaluation of Blocked Householder Algorithm for the Eigenvalue Problem on Distributed Memory Parallel Machine

TAKAHIRO KATAGIRI† and YASUMASA KANADA††

In this paper, we describe parallel implementations for the reduction of general matrix to Hessenberg form. To implement our algorithm, the blocked Householder algorithm is used. In addition, we implemented our algorithms to message-passing multiprocessor AP1000+, and evaluated its performance. Results from our experiments, we found that the loop unrolling is effective to increase the performance of blocked algorithm, and the cyclic distribution is better than the block-cyclic distribution.

## 1. はじめに

Householder 法は数ある行列計算手法のうちの最も重要な手法の一つである．例えば，行列の固有値や固有ベクトルを計算する際には，まずその行列の固有値を変えない変換 (相似変換) を Householder 法を用いて行い，それから次に固有値や固有ベクトル計算を行うのが一般的である．

このような重要な計算である Householder 法を並列計算機を用いて並列化し，高速に実装する試みは近年，並列計算機の普及により多数なされるようになった．それらすべてを列挙することは出来ないが，現在の主流となっているメッセージ交換を前提とした分散メモリ型並列計算機の実装手法に関して，その先駆けとなったのは Stewart[1] であろう．

一方で Hoseholder 法において，主メモリからのデータのロード回数を減らし，出来るだけキャッシュ，レジスタなどの高速アクセス可能な記憶を利用するアルゴリズムが知られている．このアルゴリズムは Bischof と Loan[2] らによって開発され，ブロック化 Householder 法と呼ばれる．

ブロック化 Householder 法を用いた相似変換の並列化に関する，初期の重要な並列アルゴリズムは Dongarra と Geijn[3] との文献に書かれている．また，同種の研究開発報告としては 米国 Oak Ridge 国立研究所の Choi[4] らによる技術報告書にも，並列ブロック化アルゴリズムの性能が報告されている．だが Householder 法におけるブロック化アルゴリズム性能を評価した文献は数少ない．

そこで本論文では，逐次ブロック化アルゴリズムの実現および高速化技法の一つであるループアンローリグの性能評価まで立ち帰り，その性能を評価する．さらにその並列アルゴリズムの実装法を出来るだけ詳細に検討し，富士通の高並列計算機 AP1000+に実装して評価を行う．

## 2. 並列実行環境と表記法

ここで並列計算機は $p$ 個の均質的な PE(Processing Element) で構成され，それらは一次元的に $P_0$, $P_1$, $\cdots$, $P_{p-1}$ とラベル付けされているものとする．さらに各 PE はメッセージの放送 (同一メッセージを一対多の通信で放送すること) や，各 PE が局所的に所有

† 東京大学大学院理学系研究科情報科学専攻
Department of Information Science, Graduate School of Science, University of Tokyo
†† 東京大学大型計算機センター
Computer Centre, University of Tokyo

するデータの要素に対して総和演算を行うような計算(一対一もしくは全対全のなどの通信により実現)が行えるように，ネットワーク網により結合されているものとする．

定式化のために，表1に示す表記を用いることとする．

表1　数学表記法とそれらの説明

| 表記 | 説明 |
|---|---|
| $\alpha, \mu$ | スカラ $\in \Re$ |
| $x, y, u$ | ベクトル $\in \Re^n$ |
| $d, e, f$ | ベクトル $\in \Re^m$ |
| $X, Z, U$ | 行列 $\in \Re^{n\times m}$ |
| $A$ | 行列 $\in \Re^{n\times n}$ |
| $[A]_{i:j,k:l}$ | $A$ の行 $i,\cdots,j$，列 $k,\cdots,l$ 部分行列 |
| $[A]_{*,k:l}$ | $A$ の行全て，列 $k,\cdots,l$ 部分行列 |
| $A^{(k)}$ | 反復 $k$ における行列 $A$ |

## 3. Householder 変換の逐次アルゴリズム

### 3.1 未ブロック化アルゴリズム

ここで行列 $A = A^{(1)}$ から $H^{(1)}A^{(1)}H^{(1)} = (I-\alpha_1 u_1 u_1^T)A^{(1)}(I-\alpha_1 u_1 u_1^T) = A^{(2)}$ の 変換を続け，Hessenberg 形 $A^{(n-2)}$への変換を行うことを考える．いま，$k$反復時に必要な変換行列を $H^{(k)} = (I-\alpha_k u_k u_k^T)$ とおいた．このとき，この変換処理は図1のようになることが知られている[1),3)]．

図1に示したように，反復 $k$における消去過程では消去対象行列 $[A]_{k:n,k:n}$における 列ベクトル $[A]_{*,k}$ が必要になる．このベクトルのことを枢軸ベクトル生成列とよび，それを用いることによって計算されるベクトル $u_k$を枢軸ベクトル (pivot vector) と呼ぶことにする．

ここで $\alpha_k$ は $[A]_{*,k}$ のノルムをもちいて計算されるスカラ値である．このベクトル $u_k$とスカラ$\alpha_k$との組への変換を $H^{(k)}([A]_{*,k}) = (u_k, \alpha_k)$ と記述する．

### 3.2 ブロック化アルゴリズム

ここでは，図1のアルゴリズムに示したHouseholder変換 $m$ 回を累積することを考える．Dongarra と Geijn[3)]とによるとこのアルゴリズムは以下のように書ける．

$$H^{(k+m-1)}\cdots H^{(k)}A^{(k)}H^{(k)}\cdots H^{(k+m-1)} = A^{(k)} - UZ^T - XU^T \quad (1)$$

ただし $k=(q-1)m+1$ であり，かつ $q=1,\cdots,n/m$ とする．ここで議論を単純にするため，$n$ は $m$ で割り切れるとする．このとき各ステップの計算は $j=1,\cdots,m$ とすると

$$H^{(k+j-1)}\left([A^{(k+j-1)}]_{*,k+j-1}\right) = ([U]_{*,j}, \alpha_{k+j-1}) \quad (2)$$

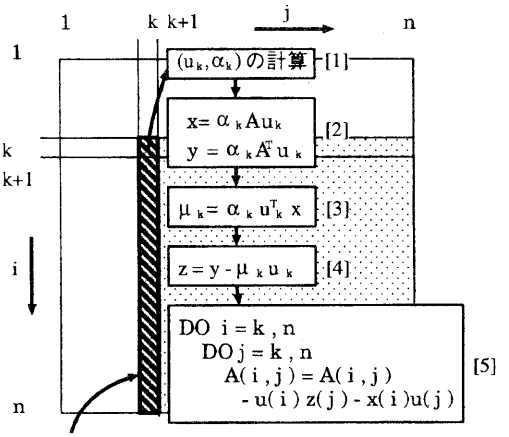


図1　Hessenberg reduction の逐次アルゴリズム

$$\begin{aligned}[X]_{*,j} &= \alpha_{k+j-1}A^{(k+j-1)}[U]_{*,j}\\ &= \alpha_{k+j-1}(A^{(k)} - [U]_{*,1:j}[Z]_{*,1:j}^T\\ &\quad -[X]_{*,1:j}[U]_{*,1:j}^T)[U]_{*,j}\\ &= \alpha_{k+j-1}A^{(k)}[U]_{*,j}\\ &\quad -[U]_{*,1:j}f - [X]_{*,1:j}e \end{aligned} \quad (3)$$

$$\begin{aligned} y &= \alpha_{k+j-1}A^{(k+j-1)T}[U]_{*,j}\\ &= \alpha_{k+j-1}(A^{(k)} - [U]_{*,1:j}[Z]_{*,1:j}^T\\ &\quad -[X]_{*,1:j}[U]_{*,1:j}^T)^T[U]_{*,j}\\ &= \alpha_{k+j-1}A^{(k)T}[U]_{*,j}\\ &\quad -[Z]_{*,1:j}e - [U]_{*,1:j}d \end{aligned} \quad (4)$$

$$\mu_{k+j-1} = \alpha_{k+j-1}[U]_{*,j}^T[X]_{*,j} \quad (5)$$

$$[Z]_{*,j} = y - \mu_{k+j-1}[U]_{*,j} \quad (6)$$

となる．ここで

$$d = \alpha_{k+j-1}[X]_{*,1:j}^T[U]_{*,j} \quad (7)$$

$$e = \alpha_{k+j-1}[U]_{*,1:j}^T[U]_{*,j} \quad (8)$$

$$f = \alpha_{k+j-1}[Z]_{*,1:j}^T[U]_{*,j} \quad (9)$$

とおいた．ただし，$j=1$ のときは 図1と同じ計算を行い，その計算結果のベクトル $u_k$，$x$，$y$をそれぞれ行列 $[U]_{*,1}, [X]_{*,1}, [Z]_{*,1}$ に蓄積する．また，$m=1$ の場合はこのブロック化アルゴリズムは未ブロック化アルゴリズムと同一になる．よって，未ブロック化アルゴリズムの拡張とみなすことができる．

## 4. 並列アルゴリズムの説明

### 4.1 列ブロックサイクリック分割方式の定義

ここで，入力行列 $A$ をどのように各 PE に分散させるかを定義しておく．いま入力行列の大きさを $n$，使用する PE の番号を 0, 1, $\cdots$, $p-1$，入力行列の列のインデックスを $i$, $(0 \le i < n)$，ブロック幅を $m$ とする．このとき，列 $i$ を所有している PE 番号

を $OwnerPEnum$，各PE内の局所インデックスを $LocalI$，$i$ 行が示すブロック内番号を $LocalBlkNum$ とする．このとき，列ブロックサイクリック分割方式☆とは $i$ を次に示す組，すなわち $\langle OwnerPEnum, LocalI, LocalBlkNum\rangle$ に写像する分割方式である．

$$i \mapsto \langle r \bmod p,\ \lfloor r/p \rfloor \times m + s,\ s\rangle \qquad (10)$$

ただし，$r = \lfloor i/m \rfloor$，$s = i \bmod m$ とおいた．

さらに，ブロック幅 $m = 1$ の特殊な場合を，列サイクリック分割方式☆☆といい，つぎの写像関数で表される．

$$i \mapsto \langle i \bmod p,\ \lfloor i/p \rfloor,\ 0\rangle \qquad (11)$$

### 4.2 列ブロックサイクリック分割方式に伴う並列アルゴリズム

列ブロックサイクリック分割方式に基づき各PEが所有している行列データ $A$ に対してのみ演算をすることを考える．このときこのデータ分割方式に基ずく単純な計算方式では，行列 $U, X$ およびベクトル $d, e, f$ は全PEが分割せずに全て所有し，行列 $Z$ は列ブロックサイクリックに，ベクトル $y$ はブロックサイクリックに分割するのが妥当である．よって，この並列アルゴリズムは図2の様になる．

ここで図2中の枠，たとえば ⟨12⟩ は各PEが所有している異なるベクトルに対して，その要素の総和を求めて，全PEがその値を所有する処理を意味している．この処理は通信と計算とを必要とする．

この並列アルゴリズムとほぼ同様のアルゴリズムとしては，Dongarra と Geijn[3] とのアルゴリズムが挙げられる．

### 4.3 問 題 点

4.2節の列ブロックサイクリック分割方式に基づくブロック化アルゴリズムの並列化に対して以下の問題点が指摘できる．

1) ブロック幅 $m$ を大きくすると並列処理での負荷バランスが悪くなる．
2) ブロック化アルゴリズムを並列化すると未ブロック化アルゴリズムの並列化版に対して通信回数と通信量とが増す．
3) ブロック化アルゴリズムは未ブロック化アルゴリズムに対して計算量が多くなる．

1) の問題の解決法としては，データ分割方式は列サイクリック分割方式を採用しつつ，計算はブロック化アルゴリズムを行うという戦略がある．本来，逐次処理での処理単位であるブロックごとにデータ分割する必然性はない．このアルゴリズムではブロック内で変換を累積する際にも各反復で通信を必要とするから，ブロック単位でデータを分割する利益は生じないことに留意する．また 2)，3) の問題は，計算性能と通信性能とのトレードオフであり，最適なブロック幅の決定が困難といえる．このことから，現状では経験的に最適なブロックサイズが決定されている．

⟨1⟩ **for** (iter=1; iter ≤ n-2; iter+=m) {
⟨2⟩ $[wA]_{iter:n,1:m} = [A^{(iter)}]_{iter:n,iter:iter+m-1}$;
⟨3⟩ **for** (blk=1; blk ≤ m; blk++) {
⟨4⟩ **if** ($[A^{(iter+blk-1)}]_{*,\,iter+blk-1}$ を所有)
⟨5⟩ $[wA]_{iter+blk-1:n,blk}$ の放送;
⟨6⟩ **else**
⟨7⟩ $[wA]_{iter+blk-1:n,blk}$ の受信;
⟨8⟩ $H^{(iter+blk-1)}([wA]_{*,blk}) = ([U]_{*,blk}, \alpha_{iter+blk-1})$;
⟨9⟩ **if** (blk == 1) {
⟨10⟩ /* 未ブロック化アルゴリズム */
⟨11⟩ $[X]_{*,1} = \alpha_{iter+blk-1} A^{(iter)} [U]_{*,1}$;
⟨12⟩ ベクトルリダクション $[X]_{*,1}$;
⟨13⟩ $y = \alpha_{iter+blk-1} A^{(iter)T} [U]_{*,1}$;
⟨14⟩ $\mu_{iter+blk-1} = \alpha_{iter+blk-1} [U]^T_{*,1} [X]_{*,1}$;
⟨15⟩ $[Z]_{*,1} = y - \mu_{iter+blk-1} [U]_{*,1}$;
⟨16⟩ } **else** {
⟨17⟩ /* ブロック化アルゴリズム */
⟨18⟩ $d = \alpha_{iter+blk-1} A^{(iter)} [X]^T_{*,1:blk} [U]_{*,blk}$;
⟨19⟩ $e = \alpha_{iter+blk-1} A^{(iter)} [U]^T_{*,1:blk} [U]_{*,blk}$;
⟨20⟩ $f = \alpha_{iter+blk-1} A^{(iter)} [Z]^T_{*,1:blk} [U]_{*,blk}$;
⟨21⟩ ベクトルリダクション $f$;
⟨22⟩ $[X]_{*,blk} = \alpha_{iter+blk-1} A^{(iter)} [U]_{*,blk} - [U]_{*,1:blk} f - [X]_{*,1:blk} e$;
⟨23⟩ ベクトルリダクション $[X]_{*,blk}$;
⟨24⟩ $y = \alpha_{iter+blk-1} A^{(iter)T} [U]_{*,blk} - [Z]_{*,1:blk} e - [U]_{*,1:blk} d$;
⟨25⟩ $\mu_{iter+blk-1} = \alpha_{iter+blk-1} [U]^T_{*,blk} [X]_{*,blk}$;
⟨26⟩ $[Z]_{*,blk} = y - \mu_{iter+blk-1} [U]_{*,blk}$;
⟨27⟩ }
⟨28⟩ $[wA]_{iter+blk-1:n,blk:m} -= [U]_{iter+blk-1:n,blk} [Z]^T_{iter+blk-1:iter+m-1,blk} + [X]_{iter+blk-1:n,blk} [U]^T_{iter+blk-1:iter+m-1,blk}$;
⟨29⟩ }
⟨30⟩ $[A^{(iter+m-1)}]_{iter:n,iter:iter+m-1} =$
⟨31⟩ $[wA]_{iter:n,1:m}$;
⟨32⟩ $[A^{(iter+m-1)}]_{iter:n,iter+m:n} -= [U]_{iter:n,1:m} [Z]^T_{iter+m:n,1:m} + [X]_{iter:n,1:m} [U]^T_{iter+m:n,1:m}$;
⟨33⟩ }

図2 列ブロックサイクリック分割方式に伴う並列アルゴリズム

## 5. ループアンローリング

計算処理を高速に行うため，ループアンローリングは最初に試みるべき重要な技法であることが知られている．よってここでは，Householder 変換の計算処理のループアンローリングを考える．

☆ 文献 3) では panel-wrapped storage と呼んでいる．
☆☆ 文献 3) では column-wrapped storage と呼んでいる．

Householder 変換では列方向のアクセスが頻発する．しかしC言語を用いて実装する理由から，行方向に連続アクセスさせる実現法が好ましいため，全ての行列を転置した形で記憶しておくとする．このことから最内ループが行方向連続アクセスとなるように留意して，式(1)～式(9)の計算処理を実装することを考える．このとき式(3)，式(4)，式(7)～式(9)の計算処理は次に示す型の二重ループのいずれかとなる☆．

```
(a) for (i=iter; i<n; i++) {
      y[i] = 0.0;
      for (j=iter; j<n; j++)
        y[i] += A[i][j] * u[j]; }

(b) for (i=iter; i<n; i++)
      x[i] = 0.0;
    for (i=iter; i<n; i++)
      for (j=iter; j<n; j++)
        x[j] += A[i][j] * u[i];
```

図3 Householder 変換の計算処理に用いられる二重ループの二つの型

このとき，(a)，(b)についての二段ループアンローリングとは$i$ループについてストライド2になるように式を展開することである．例えば図3(a)の場合は

```
m = (n-iter) / 2;
i = iter;
for (l=0; l<m; l++) {
  y[i  ] = 0.0;
  y[i+1] = 0.0;
  for (j=iter; j<n; j++) {
    y[i  ] += A[i  ][j] * u[j];
    y[i+1] += A[i+1][j] * u[j];  }
  i += 2;  }

/* 余りの処理 */
for (i=iter+m*2; i<n; i++) {
  y[i] = 0.0;
  for (j=iter; j<n; j++)
    y[i] += A[i][j] * u[j];  }
```

のようになる☆☆．多段のアンローリングも同様に実現できる．

一方，式(1)で示される行列更新の計算処理は行列$A$の行方向に連続アクセスすることを考慮すると，次の三重ループになる．このとき図4のループをアンローリングする方法はi) $i$ループについて，ii) $k$ループについて，iii) $i,k$の両ループについて，の三通りが考えられる．

```
for (i=iter+m-1; i<n; i++)
  for (k=0; k<m; k++)
    for (j=iter; j<n; j++)
      A[i][j] -= U[k][j] * Z[k][i]
               + X[k][j] * U[k][i];
```

図4 ブロック化 Householder 変換の更新処理における三重ループ

## 6. 性能評価

ここでは256PEの富士通AP1000+を用いて並列アルゴリズムの性能の評価を行なった結果を示す．AP1000+の各PEの理論ピーク性能は50MFlops，PE間は二次元トーラス網で結合されており，その最大転送性能は25Mbyte/sである☆☆☆．

### 6.1 逐次アルゴリズムの性能

まず最初に，逐次アルゴリズムの性能を評価する．この評価には，ブロック化アルゴリズムの評価とループアンローリングの評価とが含まれている．ここでは，問題サイズ$n = 1000$に固定し，ブロックサイズ$m$を1～40まで変化させる場合と，ループアンローリングをしない～6段まで行う場合とに対して，双方1づつ変化させて効果を調べた．その結果，行列の更新処理ではiii) $i,k$の両ループについてのアンローリングが最も効果的であったので，その結果のみを示す．

図5はブロック化 Householder 変換の更新処理である三重ループのうちの$i,k$の両ループについてアンローリングを行った性能を表している．図5から，ループアンローリングをしない場合，ブロック化アルゴリズムを用いる効果がなく，むしろ実行時間が遅くなることがわかった．しかしながらアンローリングを行うと，ブロック化による効果が得られるいることがわかる．また，ブロックサイズに依存して実行時間が変化する理由は，a) $k$ループはブロック幅$m$に依存するループである，b) $k$ループのアンローリングを行うと飛び飛びのメモリアクセスが生じる，ためであると考えられる．さらに紙面の都合上からすべての結果は載せられないが，2段から6段まで1づつアンローリングを増やして実行時間を調べた結果，$i,k$の両ループについての4段のアンローリングまでは効果があるが，5段以降では実行時間が低下することがわかった．

図6はブロック化 Householder 変換の更新処理である三重ループのうちの$i,k$の両ループについてアン

☆ このまま実装するとコンパイラによってはループに独立な変数を認識せず，メモリからのロードを行うコードを生成して実行効率が悪くなる場合が考えられる．よって性能を評価するプログラムにはループに依存しない変数，例えば図3(a)の$y[i]$などはスカラ変数をもちいて実装している．

☆☆ ただし余りの処理は二段アンローリング用に特化し，$i$ループを外すことができるが，ここでは一般化の説明ためにそのようには記述していない．

☆☆☆ 東京大学情報科学専攻が所有している AP1000+ のうち256台すべてを使用した．また，セルプログラム用のコンパイラとして gcc version2.6.3，オプションとして -O3 -msupersparc -fcaller-saves -fomit-frame-pointer -funroll-loops を指定した．測定日は1996年12月19日から97年2月5日である．

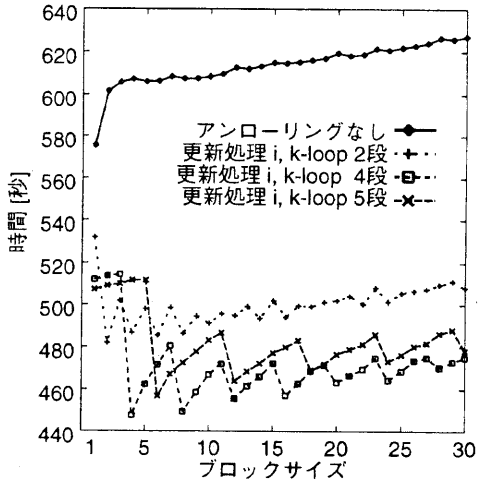


**図 5** 逐次ブロック化 Householder 変換における更新処理の $i, k$ の両ループのアンローリングの効果 ($n = 1000$)

ローリングを 4 段づつ行い，かつ図 3の様な二重ループの $i$ ループに関するアンローリングを 2 段～6 段まで行った性能を表している．図 6から，この条件での

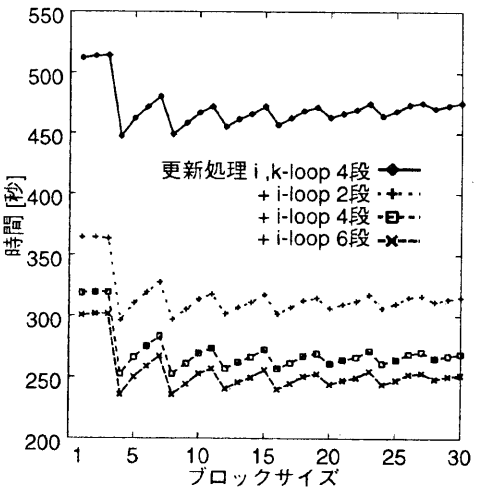


**図 6** 逐次ブロック化 Householder 変換における更新処理および二重ループ処理のループアンローリングの効果 ($n = 1000$)

実装では二重ループにおいて $i$ ループのアンローリングは 6 段づつ行うのが最も良いと考えられる．この場合，ブロック化アルゴリズムの効果は約 1.2 倍であった．図 5，6の実験の結果として，この条件ではブロック化アルゴリズムとループアンローリングの組合せによって，何も行わない実装方式に対して約 2.4 倍の速度向上が得られることがわかった．

### 6.2 並列アルゴリズムの性能

並列実装する際には，逐次と異なる理由でループアンローリングの段数の決定が問題となる．なぜならば，$k$ ループについてアンローリングの段数を増やすとブロック幅も大きくしないとその効果がない．しかしブロック幅を増やすと通信時間も増えることはすでに述べた．よってこの理由から逐次の場合と状況が異なり，最適な段数が決定しにくい．ここでは，いろいろ実験した結果から 4 段のアンローリングを実装している．

図 7，8に問題サイズ $n = 1000$ に固定し，PE 台数を 4，16，32，64，128，256 と変化させて実行した時間を載せる．図 7，8から，データ分割を列サイク

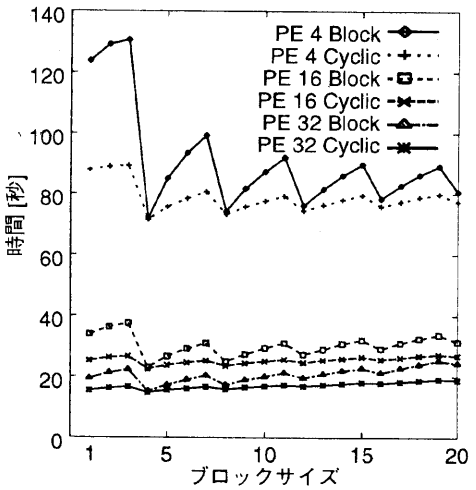


**図 7** ブロック化 Householder 法の実行時間 I ($n = 1000$, $p = 4, 16, 32$)

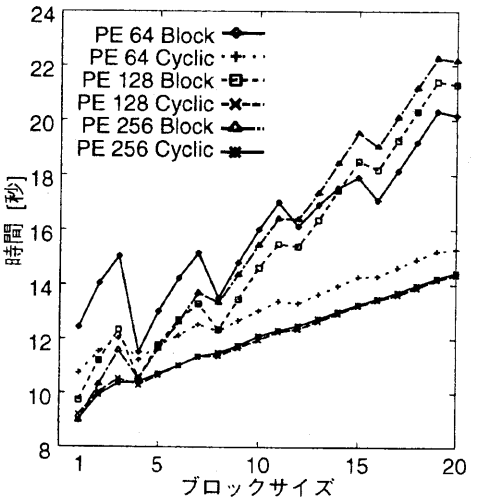


**図 8** ブロック化 Householder 法の実行時間 II ($n = 1000$, $p = 64, 128, 256$)

リックに行う方が列ブロックサイクリックに行う場合と比較して高速となっている．この理由は二通り考えられる．すなわち，i) アンローリング実装の難しさ，ii) 負荷バランスの悪さ，である．i) の理由は，本実装ではブロック幅 $m$ でアンローリングしていることが挙げられる．例えば4段アンローリングの場合，ブロック幅 $m$ が4以上ないとその効果がない．もちろん所有する列の全ての長さを調べてアンローリングすると，列サイクリック分割方式の結果とほぼ一致するはずである．その証拠として，各PEに所有するデータが十分大きな場合には，$m = 4, 8, 16, 20$ の時にほぼ実行時間が一致している．しかしながら，そのようなアンローリングを列ブロックサイクリック分割方式に実装するのは困難である．また ii) の理由は，列ブロックサイクリック分割方式の本質的な欠点である．図8のように各PEが所有するデータが約32Kbyte～約131Kbyteと十分小さな場合に，性能低下が約1.6倍～約2.4倍と甚だしいのはその理由によるところが大きいと考える．

さらに，PEに所有するデータが十分大きい場合のブロック化アルゴリズムの効果を調べる．そのためにPE台数 $p = 4$ に固定し，問題サイズ $n$ を変化させて性能を調べた．ここでブロック化アルゴリズムにおける $m = 1$ の場合は，ブロック化アルゴリズムに拡張するために余分なワークエリアが存在し，元来の未ブロック化アルゴリズムと異なったものになっている点に注意する．その理由から余分なワークエリアを取り除いて，最適化したプログラム☆との実行性能も比較する必要がある．その結果を図9に示す．ここで，Mflops値の計算のための演算回数には $10/3n^3$ を用いている．図9から問題サイズが大きくなるに従い，最適化された未ブロック化アルゴリズムでさえも，その性能が低下していく．しかしブロック化アルゴリズムでは，よりいっそうの性能の向上が期待できることがわかる．ここで問題サイズが $n = 512, 1024$ の場合に，約1.6倍の極端な性能低下が生じるが，これはバンクコンフリクトが多発しているためであり，配列確保時の要素数を1増やすことで解決ができる．

## 7. おわりに

分散メモリ型並列計算機AP1000+での実験の結果から，ブロック化アルゴリズムの性能を十分に引き出すためには，ループアンローリングの実装が効果的であることがわかった．またブロック化Householder法の並列化のためのデータ分割方式は，ブロックサイクリック分割方式よりもむしろサイクリック分割方式の方が実装の簡易さと性能との面からみても有効である．

今後の課題として，ネットワークアーキテクチャ及び計算方式の異なる分散メモリ型並列計算機に実装して評価することが挙げられる．一方，未ブロック化アルゴリズムにおいては片桐・金田により，行方向，列方向ともにサイクリック分割する並列アルゴリズムが列サイクリック分割方式に対して超並列処理を行う場合に有効となることが示されている[5]が，この並列アルゴリズムのブロック化は今後の課題である．

☆ ブロック化アルゴリズムにおける $m = 1$ の場合と同様の4段のループアンローリングを実装している．

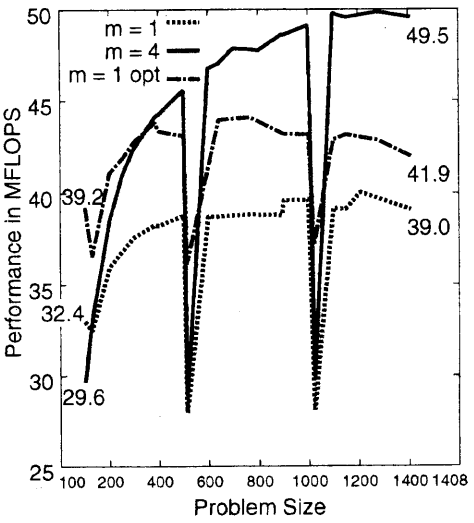


図9 サイクリック分割方式によるブロック化 Householder 法の性能 ($p = 4$)

## 参考文献


1) Stewart, G.: A Parallel Implementation of the QR-Algorithm, *Parallel Computing*, Vol. 5, pp. 187–196 (1987).

2) Bischof, C. and van Loan, C.: The WY Representation for Products of Householder Matrices, *SIAM J. Sci. Stat. Comput.*, Vol. 8, No. 1, pp. s2–s13 (1987).

3) Dongarra, J. J. and van de Geijn, R. A.: Reduction to Condensed Form for the Eigenvalue Problem on Distributed Memory Architectures, *Parallel Computing*, Vol. 18, pp. 973–982 (1992).

4) Choi, J., Dongarra, J. J. and Walker, D. W.: The Design of a Parallel Dense Linear Algebra Software Library: Reduction to Hessenberg, Tridiagonal, and Bidiagonal Form, Technical Report ORNL/TM-12472, Oak Ridge National Laboratory (1995).

5) 片桐孝洋, 金田康正: 分散メモリ型並列計算機による Householder 法の性能評価, 情処研報, 96-HPC-62, pp. 111–116 (1996).