数 値 解 析 39－5
（１９９１．１２．６）

連立一次方程式の並列計算

福井 義成
(株)東芝 総合情報システム部

川端 裕一
東芝ＣＡＥシステムズ(株)

縦ブロックガウスは行列を複数の列ベクトルからなるブロックに分割し、ブロック単位で消去を進める方法である。本来、実記憶の容量よりもはるかに大きい大規模密行列を解くためのものであるが、密行列の連立一次方程式を並列計算で解く方法として用いることができる。ここではまず、この縦ブロックガウスを並列計算に適用する方法について述べ、つぎにＣＲＡＹ Ｙ-ＭＰの並列処理についての説明を行う。ＣＲＡＹ Ｙ-ＭＰ上でテスト問題を解いた結果、３ＣＰＵで2.7倍から3倍に近い高速化を得ることができた。

Parallel computation of linear equations

Yoshinari Fukui
Toshiba Corp. Total Information & Systems Division

Yuichi Kawabata
Toshiba CAE Systems Inc.

In Column Blocked Gaussian elimination method, a matrix is divided into blocks and each block is eliminated as a unit. This method is originally used for a matrix that is considerably larger than the capacity of real memory, but it may also be used to solve dense linear equations with parallel computation.

Method of applying the Column Blocked Gaussian elimination in parallel computation is discussed and the parallel computation in CRAY Y-MP is explained.

When this method is applied to the test problem using the CRAY Y-MP, speed up of 2.7 to nearly 3 times is observed with 3 cpus.

１．はじめに

縦ブロックガウス(文献［３］)とはもともと実記憶よりもはるかに大きい大規模密行列を解くための計算方法である。図1.1に示すように行列全体を複数の列ベクトルからなるブロックに分割し、各ブロック単位で消去計算を進めて行く。

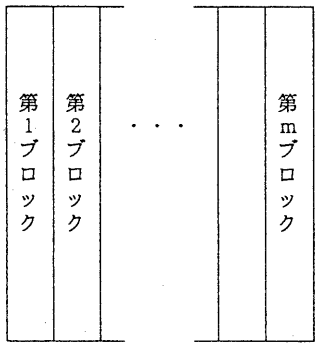

図1.1

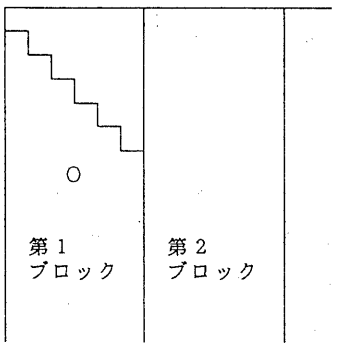

図1.2

具体的に手順を示すと次のようになる。

手順１　：　第１ブロック内で消去計算を行う　(図1.2)
手順２　：　第１ブロックによって第２ブロックを消去する
手順３　：　第２ブロック内で消去計算を行う　(図1.3)
手順４　：　第１ブロックによって第３ブロックを消去する
・・・・・・

以下、順次計算を進め、第(m-1)ブロックによる第mブロックの消去を行った後(図1.4)、最後に第mブロック内での消去計算を行って行列全体の三角化を完了する。

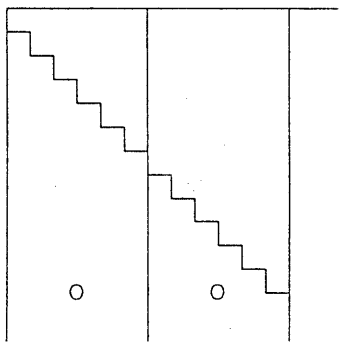

図1.3

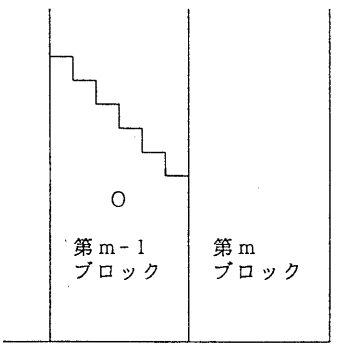

図1.4

本論文では最後に実プログラムによる性能評価を行っている。そのときに用いているプログラムは、kji型ガウスの消去法に、段数kに対する２重のアンローリングをほどこしたもの(文献［１］)をもと

にして、並列化のための改良を行ったものである。このアンローリングは図1.5のようなループ構成で表わされ、1回のストアに対して2回の積和演算が行われるため、全体ではストア回数が1/2、ロード回数が約3/4になる。

```
do k=1,n,2
  do j=k+2,n
    do i=k+2,n
      a(i,j)=a(i,j)+w1(i)t+w2(i)t1
```

図1.5

2．並列縦ブロックガウス

前節に示した手順1，2，3，．．．は必ずしもこの順番で行う必要はなく、第iブロックによって第jブロックを消去する作業を(i,j)と書けば、次の2つの条件さえ満足していればよいことがわかる。

・(i,j)を実行するには(i,i)と(i-1,j)が終了していなければならない
・(i,i)を実行するには(1,i) ... (i-1,i)が終了していなければならない

これを満たす作業が複数あればそれらは並列に処理できることになる。この方針に沿うよう、作業の順序を管理するために待行列を導入し、実行可能になった作業を順次この待行列に登録して行く。さらに作業が実行可能か(待行列に登録してよいか)どうかの判断はサイズがブロック数mの作業用配列を2つ用意する(ここでは仮にCHK1とCHK2とする)ことで行うことができる(図2.1)。

BKNO	1	2	3	4	5	6		m	
CHK1	1	1	1	0	0	0	...	0	<-- フラグ
CHK2	0	1	2	0	0	0		0	<-- カウント

図2.1

計算中、図2.1に示す配列CHK1(i)には(i,i)が終わった時点でフラグが立てられ、配列CHK2(j)は(k,j),k=1,2,...,j-1が1つ終わるごとに1づつカウントされる。CHK1(k)=1で(k-1,j)が終了したら前述2方針のうち前者が満たされたとして(k,j)を待行列へ登録し、CHK2に1が加えられる。後者については前者の定義から自動的に、(i-1,i)が終了した時点で(1,i) ... (i-1,i)は終了しているとして待行列に(i,i)が登録される。

3CPUの計算機を使い、行列を8ブロックに分割して実行したとすると図2.2のようなイメージで処理が進められる。

勿論、実際には各作業にかかる時間は異なり、一般にはこの図に示すような順序の通りに処理は進まない。CPU1の行に記した (1,1) --> (1,2) --> (2,2) --> (2,3) --> (3,3) ... と続く一連の流れはこの順で行われなければならない処理で、ここはこれ以上短縮することができない。このため実

プログラムでは待行列に(i,i)を登録する際には待行列の先頭に登録し、優先的に作業が実行されるようにした。

```
CPU1   (1,1) --> (1,2) --> (2,2) --> (2,3) --> (3,3) ...
CPU2             (1,3) --> (1,5) --> (1,7) --> (2,4) ...
CPU3             (1,4) --> (1,6) --> (1,8) --> (2,5) ...
```

図2.2

3．マルチタスキング

前記の手法をＣＲＡＹ Ｙ-ＭＰに適用する。ＣＲＡＹ Ｙ-ＭＰにおける並列処理はマルチタスキングと呼ばれ、現在、次の３つの方法がある。

・マクロタスキング　：サブルーチンレベルでタスクが定義され、タスクの発生、タスク間の同期などは全てライブラリを呼ぶことによって行われる。

・マイクロタスキング：DOループなど、プログラムの行レベルでマルチタスキングを行う。タスクの管理はシステムにまかせ、プロセッサの数と並列処理する部分をコンパイラディレクティブで指定する。

・オートタスキング　：マイクロタスキングと同じレベルのマルチタスキングで、コンパイラおよびプリプロセッサのレベルで自動的に並列処理を行ってくれる。

本論の手法で用いるのはこのうちのマクロタスキングである。マクロタスキングの主なライブラリを表3.1に示す。ここに示したライブラリを用いて記述するとアルゴリズムの概観は図3.1-図3.3のようになる。以下の説明は３つのＣＰＵを使うものと仮定して行う。

表3.1　マルチタスキングのライブラリ

タスク制御ルーチン	CALL TSKSTART(TASKID,Subname,[Arg]) 　　タスクを発生させる CALL TSKWAIT (TASKID) 　　他のタスクを待つ
イベント制御ルーチン	CALL EVPOST (EVENT) 　　イベントを発生させる CALL EVWAIT (EVENT) 　　イベントを待つ CALL EVCLEAR(EVENT) 　　イベントを抹消する
ロック制御ルーチン	CALL LOCKON (LOCK) 　　メモリへのアクセスをロックする CALL LOCKOFF(LOCK) 　　メモリへのアクセスロックを解除する

まず、親タスクがTSKSTARTをコールすることによって２つの子タスクを発生する(図3.1)。計３つのタスクはそれぞれ独立に異なった部分の消去計算を行うが、作業(i,j)においてi=jか否かでその処

理は異なるため、PGAUSSではその状態を判断して、実際に消去計算をしているサブルーチンPGAUSSAあるいはPGAUSSBをコールする(図3.3)。

各タスクは次に行うべき作業を待行列からとってきて、1つの作業が終わったらそれによって実行可能になった作業を待行列に登録する。この部分に限って各タスクが同じ配列を同時にアクセスする可能性があるので、イベント制御やロック制御をしなければならない(図3.2)。新しい作業がなくなるとタスクは待ちの状態になり、最後の作業(m,m)の終了があるタスクで確認されると行列全体に対するガウスの消去計算が終了したとして、各タスクとも待ち状態から抜けて処理を終了する。

```
初めの作業を待行列に入れる
CALL TSKSTART(ITSK1, PGAUSS, ...
CALL TSKSTART(ITSK2, PGAUSS, ...
CALL PGAUSS( ...

CALL TSKWAIT(ITSK1)
CALL TSKWAIT(ITSK2)
```

図3.1

```
CALL LOCKON (LOCK)
CALL EVWAIT (EVNT)
CALL EVCLEAR(EVNT)
CALL LOCKOFF(LOCK)

待行列へのアクセスを
ともなう一連の処理

CALL EVPOST (EVNT)
```

図3.2

```
      SUBROUTINE PGAUSS
100   CONTINUE
      待行列の先頭から次の作業をとってくる
      IF (待行列が空で全タスクが作業を行っていない) RETURN

      IF (ブロック内の消去作業) THEN
         CALL PGAUSSA
      ELSE (他のブロックを消去する作業)
         CALL PGAUSSB
      END IF

      新しく実行可能になった作業を待行列に加える

      IF (最終ブロックの作業を終了した) RETURN

      GO TO 100
```

図3.3

待行列が空で他のＣＰＵによる登録を待つ場合、ＣＰＵでループして待っているためＣＰＵを多く消費する可能性がある。通常の待ちはイベントを使用して行うことができるが、計算終了の判定をすべてのタスクに伝えるために、このような処理を行っている。イベントによる待ちに改良を加え、ループを使うことを避ける方法もあり得るが、現段階では詳細な検討は行っていない。

４．実行結果

以上に述べた並列縦ブロックガウスを行列サイズがそれぞれ1000,2000,3000元の問題について計算した。ここでは３つのＣＰＵを使ってFrank行列を解いた結果を示す。計算機がすいている状態でプログラムを実行した結果が表4.1の(a)から(c)である。表中のnbは１ブロック内の列数で、cpu timeは３ＣＰＵ合計のＣＰＵ時間を示している。いずれも計算時間(use time)の３倍にかなり近い数字となっており、並列計算の効果がよくでていることが分かる。３ＣＰＵ程度の並列度では負荷バランスはかなり容易にとれるものと思われる。

次に、マルチプログラミング環境の異なる状況で複数回の測定をした結果、最もよくないと思われる結果を表4.2のそれぞれ(a)-(c)に示した。条件の悪い場合、倍率が１を下回るものもある。これは非常に混んでいる状況で複数のＣＰＵが確保できず、マルチタスクのためのオーバーヘッドによる悪影響だけが表面化したものと考えられる。したがって、マルチプログラミング状況での並列処理は条件に注意して行わなければならない。

nbが200,400の場合、ＣＰＵ時間が増加しているが、これは作業の粒度が大きくなることによって、各ＣＰＵの作業待ちの状態が多くなり、その部分でＣＰＵを消費しているものと考えられる。

表4.1

n = 1000

nb	cpu time (sec)	use time (sec)	倍率
4	3.985	1.411	2.824
6	2.936	1.064	2.759
8	2.724	0.989	2.754
10	3.001	1.083	2.771
20	2.540	0.927	2.740
50	2.572	0.940	2.736
100	2.761	1.001	2.758
200	3.901	1.388	2.811
400	6.815	2.358	2.890

(a)

n = 2000

nb	cpu time (sec)	use time (sec)	倍率
4	22.371	7.572	2.954
6	20.097	6.810	2.951
8	19.330	6.540	2.956
10	19.016	6.435	2.955
20	18.605	6.289	2.958
50	18.639	6.304	2.957
100	18.985	6.418	2.958
200	20.359	6.883	2.958
400	33.575	11.562	2.904

(b)

n = 3000

nb	cpu time (sec)	use time (sec)	倍率
4	69.656	23.854	2.920
6	67.470	23.008	2.932
8	62.678	21.072	2.974
10	62.260	20.962	2.970
20	63.406	21.310	2.975
50	61.296	20.594	2.976
100	63.989	21.633	2.958
200	64.040	21.528	2.975
400	72.642	24.399	2.977

(c)

表4.2

n = 1000

nb	cpu time (sec)	use time (sec)	倍率
4	5.114	2.892	1.768
6	2.929	1.026	2.855
8	2.727	0.976	2.794
10	3.989	1.387	2.876
20	2.859	1.013	2.822
50	2.811	1.005	2.797
100	2.772	0.989	2.803
200	4.836	1.794	2.696
400	6.416	3.985	1.610

(a)

n = 2000

nb	cpu time (sec)	use time (sec)	倍率
4	24.254	23.460	1.034
6	21.539	19.915	1.082
8	21.791	25.689	0.848
10	21.700	24.213	0.896
20	21.225	22.233	0.955
50	26.164	27.175	0.963
100	19.365	18.779	1.031
200	23.276	30.604	0.761
400	27.544	31.562	0.873

(b)

n = 3000

nb	cpu time (sec)	use time (sec)	倍率
4	76.905	82.244	0.935
6	67.993	69.373	0.980
8	64.812	60.005	1.080
10	63.245	54.355	1.164
20	63.461	56.659	1.120
50	61.986	51.347	1.207
100	62.904	135.422	0.465
200	64.079	38.741	1.654
400	70.423	42.776	1.646

(c)

注)

1. nb : 1ブロック内の列数
2. 倍率 : (cpu time)/(use time)
3. 並列用に改良していない2段同時ガウスのプログラムを実行したときのＣＰＵ時間はそれぞれ
   n=1000 : 2.583sec, n=2000 : 18.949sec, n=3000 : 61.823sec

５．おわりに

表4.1の結果は環境が整えば本論で述べた並列縦ブロックガウスが、密行列を係数とする連立一次方程式の高速計算に有効であることを示している。今回のような例では、ベクトル化のために十分な大きさの粒度と、ＣＰＵの負荷バランスとの両方を満足させていたために、かなり良い結果を得ることができた。今回のようなマルチタスキングの方法がＣＰＵの負荷に対して良い結果を与えることは、以前にも回路シミュレーションの例で示されている(文献［５］)。

本論の方法はＣＰＵ数に対して十分大きな数の処理単位を持つ問題に対して、非常に有効であると考える。ベクトル型スーパーコンピュータの並列処理では比較的少ないＣＰＵでの並列処理となるため、並列処理が必要となるような大規模問題では効果が大きい。一方、超並列計算機で各ＣＰＵがスカラーである場合、粒度を小さくしても計算効率はベクトル計算機に比べてそれほど落ちないと考えられるため、この場合も有効な手段となることが期待される。

謝辞

本論文の作成にあたりお世話になった、図書館情報大学の長谷川 秀彦先生に深く感謝します。

参考文献

［１］長谷川 秀彦，川端 裕一:「連立一次方程式の高速解法について」:情報処理学会 数値解析研究会資料35-3，1990.

［２］福井 義成:「汎用マルチプロセッサによる並列計算」:情報処理Vol.28，No11，1987.

［３］村田 健郎，名取 亮，唐木 幸比古:「大型数値シミュレーション」:岩波書店，1990.

［４］長谷川 秀彦:「密行列を係数とする連立一次方程式の解法（II）」:図書館情報大学研究報告 Vol.7，No.2，1988.

［５］福井，大吉，加藤，渡辺:「マルチプロセッサシステムにおける回路解析コードの並列処理」:情報処理学会 数値解析研究会資料17-5，1986.

［６］Fukui Y., Yoshida H., Higono S.:「Supercomputing of Circuit Simulation」:ACM Publication Order No.415892，1989.