The University of Osaka
Institutional Knowledge Archive

Title	ベクトル計算機用小規模行列の加算・乗算高速化支援プリプロセッサ : 短い DO ループを早く計算させたい方の為に
Author(s)	西野, 友年
Citation	大阪大学大型計算機センターニュース. 1994, 91, p. 3-12
Version Type	VoR
URL	https://hdl.handle.net/11094/66040
rights	
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

The University of Osaka

研究開発

ベクトル計算機用小規模行列の加算・乗算高速化支援プリプロセッサ

—短いＤＯループを速く計算させたい方の為に—

東北大学理学部基礎物理学講座・西野 友年

はじめに

SXを使っている方で、ベクトル化率が低い という悩みをお持ちの方はいらっしゃいませんか?このプログラムは、そのような方々のうち「短いDoループの計算が多いので高速化はあきらめている」という方に、ちょっとした高速化のお手伝いをする為に存在します。それから、「ACOSではデバッグの為に使っている部分を、SXで走らせる時にはマスクして実行させない様にしたい!」と思っている方はいらっしゃいませんか? そういう場合にもこのプログラムを使う事ができます。逆に、数値計算プログラムのある部分を、SXを使用する時だけに実行させたい場合にも使えます。

詳しい事は、以下で順を追って説明する事にして、まず簡単に使用環境を示しておきましょう。このプログラムは、ACOS上にあるSX用のフォートランのソースプログラム (以後入力ファイルと呼ぶ) を、コメント行中に埋め込まれたメッセージに従って書き換えて出力する (以後出力ファイルと呼ぶ) プリプロセッサもどきです。プログラム本体は FORTRAN77で書かれていますが、ファイルの入出力やユーザーインターフェースに関連する部分で ACOS 固有の組み込みサブルーチンを使用していますので、ACOS上で使用します。

なお、このプログラムは大阪大学計算機センターの研究開発課題として認められたもので、開発に当たっては同センターの補助 (開発にかかる計算費免除)を受けました。

本プログラム開発の背景

ここで解説するプログラムの一番の目的は、ループ長が10から100くらいの短いDoループを含んだSX用のプログラムをなるべく速く計算させるという事です。既知のこととは思いますがDoループ長とは、一番内側のDoループ繰り返し回数のことです。例えば行列の代入

```
      DO 20 J = 0, 999
      DO 10 I = 0,  95
        C(I,J)= B(I,J)
   10 CONTINUE
   20 CONTINUE
```

だと変数Ｉは0から95まで回っていますから、ループ長は96です。一般的に言って、ループ長が長い方がスーパーコンピューターの特性を活かすことが出来て、高速演算に都合が良いのです。しかし、計算の目的によっては、いつも長いDoループばかり出て来るとは限りません。

著者は、ここで解説するプログラムの開発を始めた頃、量子モンテカルロという物理シミュレーションを行なっていました。この量子モンテカルロ・シミュレーションでは、小さなサイズの行列計算が頻繁に出現し、場合によってループ長が10から100という短い長さのDoループを何億回も繰り返さないといけない事があります。しかし、スーパーコンピューターは、こういった短いDoループを効率良く処理するのが苦手で、ベクトル化率が70%台・実行速度はSXのカタログ性能の1/100という事も再々ありました。そこで、少しでも計算費を節約するために、次のような工夫を始めました。簡単の為に行列の加算を例に取って説明しましょう。

120行 120列の行列同士の加算 C = A + B を計算する必要があるとしましょう。一般に、行列のサイズと、行列を収納する配列変数のサイズは異なっているのが普通ですから、ここでも行列 A,B,C に対する配列は

```
      REAL*8 A(0:120,0:120), B(0:120,0:120), C(0:120,0:120)
```

と宣言しておくことにします。つまり、A(i,j)と書いた時に添え字 i,j は 0 から 119まで走り、 120番目はどうでも良い任意の要素だとします。教科書通りにプログラムすると

```
      DO 20 J = 0, 120 -1
      DO 10 I = 0, 120 -1
        C(I,J)= A(I,J) + B(I,J)
   10 CONTINUE
   20 CONTINUE
```

となります。(119をわざと120-1と書いてあるのは、こう書いた方がデバッグの際に都合が良いからです。) ここで、添え字 I についてのループが、一番内側に来ていることに注意して下さい。もしまかり間違って

```
      DO 20 I = 0, 120 -1
      DO 10 J = 0, 120 -1
        C(I,J)= A(I,J) + B(I,J)
   10 CONTINUE
   20 CONTINUE
```

などというプログラミングをすると、実行速度は地に落ちます。(その理由は、SXのマニュアル・Fortran77利用の手引き を参照して下さい。)

最初に挙げた方の加算プログラムをSX-3で走らせて、実行速度を見てみましょう。測定はアナライザSXというSXの高速化支援ソフトを使って行ないました。この加算プログラムの実行時間は非常に短いので、測定には多少の誤差があります。以下で出て来る数字 (MFLOPS値) は参考値だと思って下さい。実行結果: 上記の加算プログラムは450MFLOPSで実行されました。ここでは加算と代入という演算しか行なわれていませんから、SX-3の最高速度6.4GFLOPSは絶対に出ないのですが、それを差し引いても遅いですね。

では、もっと高速に演算をするにはどうすれば良いのでしょうか? 一つの方法は、行列 A, B, C を別のサブルーチンに一次元配列として渡して、A(120,120) といった不要な要素も含めて加算してしまうことです。

```
      SUBROUTINE ADDABC( A, B, C )
      REAL*8 A(0:*), B(0:*), C(0:*)
      DO 10 I = 0, 121*120 -1
          C(I)= A(I) + B(I)
   10 CONTINUE
      RETURN
      END
```

このようにすれば、実行速度は一気に820MFLOPSまで上がります。わざわざサブルーチンコールをしなくても、次のように配列変数の添え字の範囲と、Doループ変数 i,j の移動範囲が一致していれば、

```
      REAL*8 A(0:120,0:120), B(0:120,0:120), C(0:120,0:120)
      DO 20 J = 0, 120
      DO 10 I = 0, 120
        C(I,J)= A(I,J) + B(I,J)
   10 CONTINUE
   20 CONTINUE
```

SXのコンパイラは自動的にA,B,Cを一次元配列として扱ってくれます。この場合でも同様に800MFLOPS程度の速度が出ます。

これで一件落着かというと、世の中そう甘くはありません。行列の加算を先に示した様に一次元化出来るのは、あくまでも行列A,B,Cの大きさが宣言時に揃っている場合のみで、例えば

```
      REAL*8 A(0:120,0:240), B(0:240,0:240), C(0:400,0:400)
```

という風に宣言されていたならば、振り出しに戻って高速化の手立てを考えなければなりません。また、たとえ一次元化出来たとしても、必要な行列のサイズが、宣言した配列のサイズよりもずっと小さい場合には、かえって低速になってしまいます。そういう訳で、もっと融通の効く方法を紹介しましょう。次の様にDoループの中での独立な演算を増やしてやればよいのです。

```
      DO 20 J = 0, 120 -1, 4
      DO 10 I = 0, 120 -1
      C(I,J+0)= A(I,J+0) + B(I,J+0)
      C(I,J+1)= A(I,J+1) + B(I,J+1)
      C(I,J+2)= A(I,J+2) + B(I,J+2)
      C(I,J+3)= A(I,J+3) + B(I,J+3)
   10 CONTINUE
   20 CONTINUE
```

このようにすれば、実行速度は590MFLOPSになります。もっと頑張って

```
      DO 20 J = 0, 120 -1, 8
      DO 10 I = 0, 120 -1
      C(I,J+0)= A(I,J+0) + B(I,J+0)
      C(I,J+1)= A(I,J+1) + B(I,J+1)
      C(I,J+2)= A(I,J+2) + B(I,J+2)
      C(I,J+3)= A(I,J+3) + B(I,J+3)
      C(I,J+4)= A(I,J+4) + B(I,J+4)
      C(I,J+5)= A(I,J+5) + B(I,J+5)
      C(I,J+6)= A(I,J+6) + B(I,J+6)
      C(I,J+7)= A(I,J+7) + B(I,J+7)
   10 CONTINUE
   20 CONTINUE
```

という風に展開すれば、さらに実行速度は上がって650MFLOPSを達成できます。ここで調子にのって、もっともっと展開して

```
      DO 20 J = 0, 120 -1, 24
      DO 10 I = 0, 120 -1
      C(I,J+ 0)= A(I,J+0) + B(I,J+0)
      C(I,J+ 1)= A(I,J+ 1) + B(I,J+ 1)
      C(I,J+ 2)= A(I,J+ 2) + B(I,J+ 2)
```

- 中略 -

```
      C(I,J+ 3)= A(I,J+ 3) + B(I,J+ 3)
      C(I,J+20)= A(I,J+20) + B(I,J+20)
      C(I,J+21)= A(I,J+21) + B(I,J+21)
      C(I,J+22)= A(I,J+22) + B(I,J+22)
      C(I,J+23)= A(I,J+23) + B(I,J+23)
   10 CONTINUE
   20 CONTINUE
```

とやっても、実行速度はもはや増加しません。それどころか、僅かですが遅くなって来ます。良く、「Doループは中身を増やせば増やすほど速くなる」という迷信を持って、ただ長いプログラムを書く方がいらっしゃいますが、過ぎたるはなお及ばざるが如し、展開も程々にしておいた方が無難です。

こうやって、Doループを展開すれば実行速度が速くなる事がわかっていても、ソースリスト中にDoループ展開すべき場所が何十とあったら、どうでしょうか? ぞっとしますね。おまけに、最適の展開数 n (上の場合では n=8) を見つける為に、ソースリストを何回も書き直す必要に迫らせるかもしれません。ああ、めんどくさいですね。 では、当時 Dc 院生だった遊び人(著者)は、どうやってDoループを展開したかというと、予めソースリスト中に

```
      DO 20 J = 0, 120 -1, 24
      DO 10 I = 0, 120 -1
CC    EXPAND IK = 0, 23
        C(I,J+IK)= A(I,J+IK) + B(I,J+IK)
CC    EXPEND
   10 CONTINUE
   20 CONTINUE
```

という風にコメント行を忍ばせておいて、ちょっとしたプログラムでこれを

```
      DO 20 J = 0, 120 -1, 24
      DO 10 I = 0, 120 -1
      C(I,J+ 0)= A(I,J+ 0) + B(I,J+ 0)
      C(I,J+ 1)= A(I,J+ 1) + B(I,J+ 1)
      C(I,J+ 2)= A(I,J+ 2) + B(I,J+ 2)
```

- 中略 -

```
      C(I,J+21)= A(I,J+21) + B(I,J+21)
      C(I,J+22)= A(I,J+22) + B(I,J+22)
      C(I,J+23)= A(I,J+23) + B(I,J+23)
   10 CONTINUE
   20 CONTINUE
```

と展開させたのです。このDoループ展開支援ソフト (と言えば聞こえがいいものの、実体はただの手抜きプログラム) が、これから述べる開発プログラムのプロトタイプです。

本プログラムで出来る事

読者には、もう大体予想がついていると思いますが、ここで紹介するプログラムは、ACOS向けの編集行付きのフォートランのソースリスト (入力ファイル) を読み込んで、コメント行中に埋め込まれた指示に従って編集行なしの (ストリップされた) SX向けの出力ファイルを作成する、一種のプリプロセッサです。では、ちょっと形式的になりますが、このプログラムの「文法・機能」を紹介しましょう。まずは簡単な方から...

○機能1: コメント行の実行文化

ACOSでは実行したくないけど、SXでは実行したいという操作があったとします。例えば、デバッグ時に、ACOSで動かすと死ぬほど時間がかかるので、パスしたいルーチンがあるーといった場合です。このような時には、入力ファイルの該当する行を「行番号+'CSX ' 実行文」というふうにコメント行にしておいて下さい。例えば入力ファイルに

```
1230CSX   CALL SUB( A, B, C... )
```

という部分があれば、出力ファイルには

```
      CALL SUB( A, B, C... )
```

というものが出力されます。つまり「'CSX'で始まるコメント行が検出された場合、文字列'CSX'は3文字のスペース'　　'に変換される」わけです。(これくらいの事なら、エディターでも出来ますね。)

○機能2: 実行文のマスク

ACOSではデバッグの為に実行しているけど、SXでは実行させたくないという部分があったとしましょう。そういう場合には、次のようにコメントを書き入れます。

```
1230C     DELETE
1240       CALL DEBUG( A, B, C... )
1250C      DELEND
```

こうしておけば、出力ファイルには、この3行 (正確には'DELETE'と'DELEND'に挟まれた行) は出力されません。ここで注意して頂きたいのですが、'DELETE'とか'DELEND'とかいう文字列は、コメント行中の最初の文字列でなくてはなりません。ですから、

```
1230C      Debug Routine --- DELETE
1240       CALL DEBUG( A, B, C... )
1250C      DELEND
```

などと書かれていれば、1230行目は無視されて、1250行目でエラーになります。こういう場合は、順序を逆にして下さい。

```
1230C      DELETED --- ebug Routine
1240       CALL DEBUG( A, B, C... )
1250C      DELEND
```

○機能3: コメント行の水増し

これはSXとは直接関係の無い機能なのですが、行頭に C が一文字だけのコメント行を任意の数だけ生成します。

```
1230CC     COMMENT = 10
```

というコメントがあれば

```
c
c
c
c
c
c
c
c
c
c
```

が出力されます。この機能は、サブルーチンの頭に長大な説明を書き込む時などに、余白を取る時に使用します。(もっとも、高機能端末が目の前にあれば不要な機能ですが。) この場合も'COMMENT'はコメント行の最初に書いて下さい。

○機能4: 展開

これが最も大切な機能で、コメント行中に 'EXPA ND' 文字列1 = 定数1, 定数2 という部分があれば、その次の行から 'EXPEND' が現われるまでの文中にある文字列1 を、定数1 から定数2 までの数字に書き換えて展開します。例えば、次の例では

```
0150CC     EXPAND IL = 10, 15
0160C             IL IN COMMENT LINE IS ALSO CHANGED.
0170       A(IL)=B(IL+2)
0180CC     EXPEND
```

'IL'という文字列が 160行と 170行に存在しますから、

```
C             10 IN COMMENT LINE IS ALSO CHANGED.
      A(10)=B(10+2)
      A(10)=B(10+2)
C             11 IN COMMENT LINE IS ALSO CHANGED.
      A(11)=B(11+2)
```

```
      A(11)=B(11+2)
C            12 IN COMMENT LINE IS ALSO CHANGED.
      A(12)=B(12+2)
      A(12)=B(12+2)
C            13 IN COMMENT LINE IS ALSO CHANGED.
      A(13)=B(13+2)
      A(13)=B(13+2)
C            14 IN COMMENT LINE IS ALSO CHANGED.
      A(14)=B(14+2)
      A(14)=B(14+2)
C            15 IN COMMENT LINE IS ALSO CHANGED.
      A(15)=B(15+2)
      A(15)=B(15+2)
```

と展開されます。展開される部分の中に、文字列1 が存在すると、強制的に(何でもかんでも)変換してしまうので、

```
1230C      EXPAND TP = 1, 3
1240CSX    CALL SUBTP( A, B, C... )
1250C      EXPEND
```

は

```
      CALL SUB1( A, B, C... )
      CALL SUB2( A, B, C... )
      CALL SUB3( A, B, C... )
```

と変換されます。

ここで示したように、展開中でも'CSX'で始まる部分は実行文に書き換えられます。この性質を使うと、ACOSでもSXでもコンパイルエラーなしで動くソースリストを作れます。例えば、

```
1200CSX    DO 20 J = 0, 120 -1, 24
1210CSX    DO 10 I = 0, 120 -1
1220CC     EXPAND IK = 0, 23
1230CSX      C(I,J+IK)= A(I,J+IK) + B(I,J+IK)
1240CC     EXPEND
1250    10 CONTINUE
1260    20 CONTINUE
```

というコメント行は本プログラムで展開後、SX用の実行文になります。

○機能5: 変数の定義

展開機能を使う時に、展開の上限と下限を変数で指定することも出来ます。どうするかというと、予め

```
0100C      EXPDEF ABC = 10
0110C      EXPDEF DEF = 15
```

という風に、変数ABC, DEFを定義しておいて、展開部分でこれらを使用します。

```
0150CC     EXPAND IL = ABC, DEF
0160C            IL IN COMMENT LINE IS ALSO CHANGED.
0170       A(IL)=B(IL+2)
0180CC     EXPEND
```

展開部分で、このように変数を使用する時には必ず先だって'EX PDEF'を使って値を定義しておいて下さい。そうしないとエラーになります。

○諸制限

定数・変数ともに3桁までしか受け付けません。というのも、Doループ内に1000個もの代入文があったら、コンパイラ (f77sx) が受け付けないからです。また、変数は100個までしか使用できません。　また、変換の結果、出力ファイル中の一行の長さが80文字以上になる場合、動作の保証はできません。

○代表的な変換例

以上の機能がまとめて入っている入力ファイルの例をあげておきます。

```
0010C      REMARK   LINE
0020CAB    REMARK   LINE
0030CSX    ACTIVATED LINE
0040C
0050CC     EXPDEF ABC = 10
0060CC     EXPDEF EFG = 20
0070C
0080C      DELETE
0090       THIS LINE WILL BE DELETED
0100CSX    THIS LINE WILL BE DELETED
0110C      DELEND
0120CC--->
0130CC     COMMENT = 10
0140CC--->
0150CC     EXPAND IL = ABC, EFG
0160C             IL IN COMMENT LINE IS ALSO CHANGED.
0170       A(IL)=B(IL+2)
0175CSX    A(IL)=B(IL+2)
0180CC     EXPEND
0190C
0200CC     EXPAND I = 1,3
0240            A(I)=I
0250CC     EXPEND
```

これは、次のように変換されます。

```
C      REMARK   LINE
CAB    REMARK   LINE
       ACTIVATED LINE
C
C
CC--->
C
C
C
C
C
C
```

```
C
C
C
C
CC--->
C            10 IN COMMENT LINE IS ALSO CHANGED.
      A(10)=B(10+2)
      A(10)=B(10+2)
C            11 IN COMMENT LINE IS ALSO CHANGED.
      A(11)=B(11+2)
      A(11)=B(11+2)
C            12 IN COMMENT LINE IS ALSO CHANGED.
      A(12)=B(12+2)
      A(12)=B(12+2)
C            13 IN COMMENT LINE IS ALSO CHANGED.
      A(13)=B(13+2)
      A(13)=B(13+2)
C            14 IN COMMENT LINE IS ALSO CHANGED.
      A(14)=B(14+2)
      A(14)=B(14+2)
C            15 IN COMMENT LINE IS ALSO CHANGED.
      A(15)=B(15+2)
      A(15)=B(15+2)
C            16 IN COMMENT LINE IS ALSO CHANGED.
      A(16)=B(16+2)
      A(16)=B(16+2)
C            17 IN COMMENT LINE IS ALSO CHANGED.
      A(17)=B(17+2)
      A(17)=B(17+2)
C            18 IN COMMENT LINE IS ALSO CHANGED.
      A(18)=B(18+2)
      A(18)=B(18+2)
C            19 IN COMMENT LINE IS ALSO CHANGED.
      A(19)=B(19+2)
      A(19)=B(19+2)
C            20 IN COMMENT LINE IS ALSO CHANGED.
      A(20)=B(20+2)
      A(20)=B(20+2)
C
           A(1)=1
           A(2)=2
           A(3)=3
```

プログラムの使用法

ACOS上の編集行付きのプログラム FILE1 を FILE2 に変換する場合は、本プログラムをTSS上でRUNさせて、入力促進メッセージに従って FILE1 [CR] FILE2 [CR] と入力して下さい。操作の対象となるコメント行が検出されると、画面上にメッセージが現われます。最後にGood Luckと表示されれば、一応うまく動作したと思って下さい。

使用上の注意

このプログラムは、入力ファイルを 98 番に、出力ファイルを 99 番に割り付けます。ですから、 98, 99番を他のプログラム・ユティリティーで使用している場合は、装置番号の競合に気をつけて下さい。また、このプログラムは、一応バグ取りはしてあるものの、予期せぬバグが潜んでいる可能性は否定出来ません。従って、このプログラムを使用される時には、予め入力ファイルのバックアップを取っておいて下さい。また、出力ファイルに、既に存在するものを指定した場合にも警告は出ないので、気を付けて下さい。なお、このプログラムを使用したがために引き起こされた損害に対しての補償は行なわれません。

終わりに

このプログラムを開発した目的は、その昔阪大にあったSX-2Nというスーパーコンピューターを何とか速く走らせる事でした。昨年の初めに導入されたSX-3/14Rは、SX-2Nと比べると劇的に実行速度が上がっているのですが、それでも小規模行列はまだ苦手なようです。ただ一つの例外、行列同士の乗算を除いては。ですから小規模行列がわんさか出てくるプログラムをお持ちの方は、以上に述べた様な方法で高速化を計ってみて下さい。

最後に一公平を期す為に一他にも色々なループ展開の方法が存在する事を申し添えておきます。それは例えば、SXのフォートランFortran 77/SXのベクトル化指命令の「*VDIR EXPAND ... 」であったり、SXの上でawkという便利なプログラミング言語を使っての作業であったりします。詳しくはSX-3のマニュアルを見て調べてみて下さい。きっとご利益があるものと確信しています。