SX-ACEでのプログラミング（ベクトル化編）

［大規模科学計算システム］

SX-ACE でのプログラミング（ベクトル化編）

工藤 淑裕　横谷 雄司
日本電気株式会社

概 要

SX-ACE システムのハードウエア性能を引き出すために重要となるプログラムのベクトル化、並列化に関わるコンパイラの機能、およびプログラミングの際にご留意頂きたい点について、「ベクトル化編」、「並列化編」の 2 回に分けてご紹介します。

SX-ACE システムでは主なプログラミング言語として、Fortran 95、Fortran 2003、C、C++言語が利用できます。本稿では、SX-ACE システムの Fortran 95 言語コンパイラである FORTRAN90/SX（以降、単にコンパイラと略す）を用いて、コンパイラが持つ自動ベクトル化機能の特長、ベクトル化プログラミングでの性能向上についてご紹介します。

1. SX-ACE の CPU

SX-ACE システムは、4 個の CPU コアを内蔵した CPU を持つノードを複数結合した「スケーラブル・パラレル・スーパーコンピュータ」です。各 CPU コアは一つの命令で最大 256 個のデータを処理できるベクトル演算器を 6 個(加算器 2 個、乗算器 2 個、除算器 1 個、論理演算器 1 個)ずつ含みます。

SX-ACE のベクトル命令には、一般的なスカラプロセッサで導入されている複数のデータを一度に処理できる SIMD 命令に対して、次の特長があります。

- 最大 256 個までの大量のデータを一つのベクトル命令で演算できる
- 一つのベクトル命令で処理するデータの個数を自在に変更できる

SX-ACE の CPU コアのハードウエア性能を十分に引き出すためには、個々の CPU コアの中でベクトル命令を使って効率的に計算するためのベクトル化が大変重要となります。

2. 自動ベクトル化機能

SX-ACE システムでのベクトル化技法は、SX-9 システムの場合と基本的には同じですが、今回はスーパーコンピュータを初めて使われる方にもご理解頂けるようベクトル化の基本概念からご紹介します。

2.1. ベクトル化の基本概念

通常の演算命令は、一度に一組のデータを演算できます。このような演算命令をベクトル命令と対比させるためにスカラ命令と呼びます。これに対してベクトル命令は、複数の組のデータに対する演算を一つの命令で行うことができます。

例-1 のプログラムは、配列 B と配列 C を加算し配列 A に代入、配列 E と配列 F を加算し配列 D に代入する DO ループです。

例-1

```
DO I = 1, 100
   A(I) = B(I) + C(I)
   D(I) = E(I) + F(I)
ENDDO
```

例-1 の DO ループ中の加算をスカラ命令、ベクトル命令で実行したときの実行イメージは図-1 のとおりです。一つのスカラ命令は、一度に一組のデータに対する演算処理を行います。これに対して、SX-ACE のベクトル命令では一つの命令で一度に複数(この場合は 100 個、最大 256 個)のデータに対する演算処理を行うことができます。

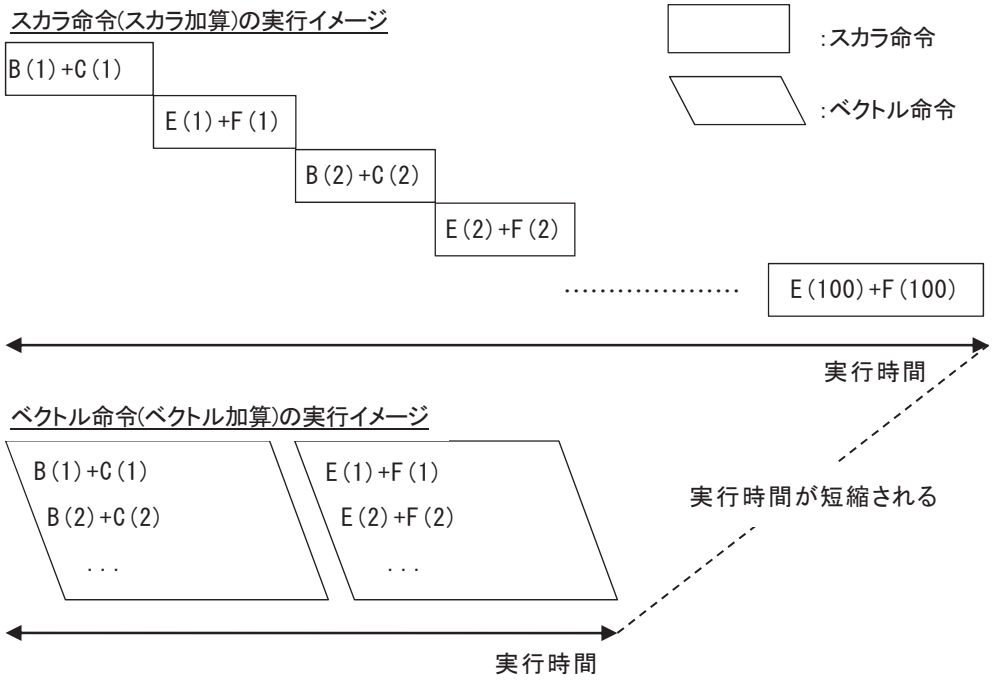


図-1　加算の実行イメージ

ループ中で計算される行列の要素など、規則的に並んだ配列データに対してベクトル命令を適用することをベクトル化(の適用)と呼び、ベクトル化することによって高速な演算が可能となります。

コンパイラの自動ベクトル化機能は、ソースプログラムを解析し、ベクトル命令で実行できる部分を自動的に検出しベクトル化を適用します。

2.2.　ベクトル命令の適用例

例-2 は、DO ループが自動ベクトル化されたときのベクトル命令の適用例です。二つの配列のメモリロード、ベクトル加算、メモリストアの四つのベクトル命令で実行されます。

例-2

```
DO I = 1, 100
   C(I) = A(I) + B(I)
ENDDO
```

↓

```
VR1    ←  配列A          (配列Aからベクトルレジスタに100個のデータをロード)
VR2    ←  配列B          (配列Bからベクトルレジスタに100個のデータをロード)
VR3    ←  VR1 + VR2      (100個のデータをベクトル加算)
配列C ←  VR3            (配列Cに100個の演算結果をストア)
                                          VRn : ベクトルレジスタ
```

2.3. ベクトル化の対象範囲

自動ベクトル化機能は、表-1 で示すループ、および、ループに含まれる文、データ、演算を対象としてベクトル化を適用します。

表-1 自動ベクトル化の対象

対象	Fortran 言語要素
ループ	配列式、DO ループ、DO WHILE ループ、FORALL ループ
文	代入文、CONTINUE 文、GOTO 文、CYCLE 文、EXIT 文、IF 文、SELECT 構文(CALL 文、入出力文等は不可)
データ型	INTEGER(KIND=4)、INTEGER(KIND=8)、REAL(KIND=4)、REAL(KIND=8)、COMPLEX(KIND=4)、COMPLEX(KIND=8)
演算	加減乗除算、べき算、論理演算、関係演算、型変換、組込み関数(利用者定義演算，ポインタ代入等は不可)

2.4. データの依存関係

ループにベクトル化を適用するには、「2.3. ベクトル化の対象範囲」で示した対象範囲に加えて、ベクトル化の適用による文、演算の実行順序が変更されても、データの定義・参照順番（データの依存関係）が変わらないことが条件となります。

例-3 は、二つの配列を定義する DO ループです。図-2 は、例-3 の DO ループのベクトル化適用前、すなわち、スカラでの文の実行順序とベクトル化後の文の実行順序のイメージを示しています。

例-3

```
DO I = 1, 100
   A(I) = B(I) + C(I)
   D(I) = E(I) + F(I)
ENDDO
```

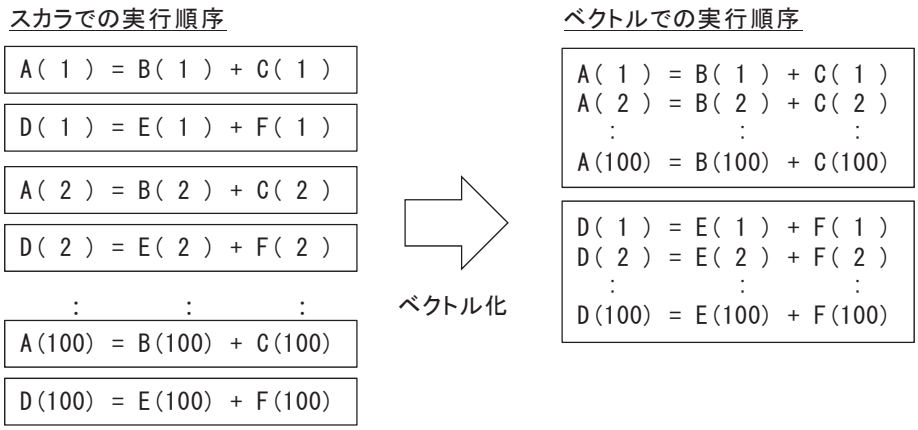


図-2　文の実行順序

スカラでの実行では、A(1)=B(1)+C(1)、D(1)=E(1)+F(1)、A(2)=B(2)+C(2)、...と実行されますが、ベクトル化した場合は一つの命令で複数のデータを処理するため、A(1)=B(1)+C(1)、A(2)=B(2)+C(2)、...、A(100)=B(100)+C(100)、D(1)=E(1)+F(1)、D(2)=E(2)+F(2)、...、D(100)=E(100)+F(100)と配列 B、配列 C の加算、配列Aへの代入後、配列E、配列Fの加算、配列Dへの代入が実行されます。ベクトル化した場合、このように文、演算の実行順序が変わります。

このような文の実行順序の変更により、データの依存関係が変わってしまうことがあります。例-4 はベクトル化後にデータの依存関係が変わってしまう例です。以前の繰返しで定義された配列要素や変数を後の繰返しで参照するパターンのとき、図-3 のようにデータの依存関係が変わり正しい結果が得られなくなるため、コンパイラはベクトル化を適用しません。

例-4

```
DO I = 2, N
   A(I+1) = A(I) * B(I) + C(I)
ENDDO
```

図-3 は、例-4 の DO ループの文の実行順序示すイメージです。ベクトル化すると、「2.2. ベクトル命令の適用例」で示した例-2 のように、配列 A のデータをまとめてメモリからロードするため、更新された A の値が使用されず正しい演算結果が得られません。

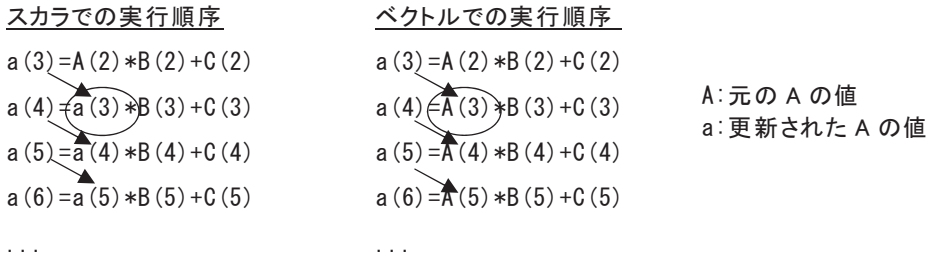


図-3　ベクトル化を阻害するデータの依存関係

例-5 はベクトル化を適用してもデータの依存関係が変わらない例です。

例-5

```
DO I = 2, N
   A(I-1) = A(I) * B(I) + C(I)
ENDDO
```

図-4 は、例-5 の DO ループの文の実行順序を示すイメージです。ベクトル化してもデータの依存関係は変わらないのでコンパイラはベクトル化を適用します。

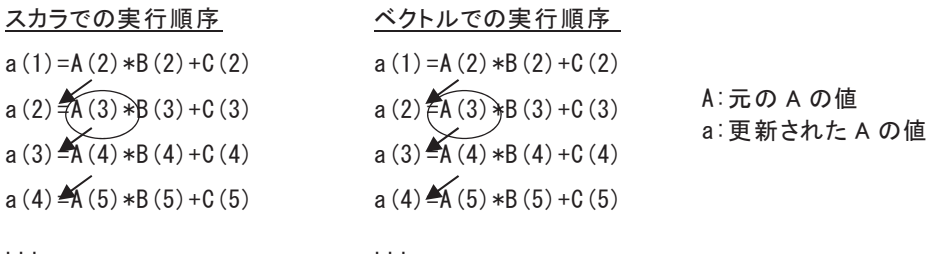


図-4 ベクトル化可能なデータの依存関係

コンパイラの自動ベクトル化機能は、ソースプログラムを解析して、ベクトル命令で実行できる部分を自動的に検出するとともに、必要ならベクトル化に適合するようにプログラムを変形して、ベクトル化を適用できる範囲を広げます。

プログラムのループを記述する際、文をループの繰返しごとに並べたとき、図-3 のような右下向きの矢印の依存関係ができないようにすると、ループにベクトル化を適用でき、ループ内の処理を高速化できる可能性が高まります。

3. 拡張ベクトル化機能

コンパイラの自動ベクトル化機能は、ベクトル化を適用できる範囲を拡げるため、データの依存関係などの理由でベクトル化できないループを変形してベクトル化したり、プログラムを変形することによってベクトル化の効果をさらに高めたりします。これを拡張ベクトル化機能と呼びます。ここでは、コンパイラの持つ拡張ベクトル化機能のうち主なものをご紹介します。

3.1. 文の入れ換え

「2.4 データの依存関係」の例のようなループは、ループ中の文を入れ換えるとベクトル化できることがあります。コンパイラは、実行結果が正しく保てるのであれば、例-6 に示すように自動的に文を入れ換えてベクトル化を適用します。

例-6

（文の入れ換え前）

```
DO I = 1, 99
   A(I) = 2.0       ! 定義
   B(I) = A(I+1)    ! 参照
ENDDO
```

（文の入れ換え後のイメージ）

```
DO I = 1, 99
   B(I) = A(I+1)       ! 参照
   A(I) = 2.0          ! 定義
ENDDO
```

3.2. ループの一重化

ループの一重化は、多重ループの外側のループと内側のループを一つのループにまとめて、ループの繰返し数を大きくする最適化です。

SX-ACE のベクトル命令は一つの命令で最大 256 個のデータを処理できます。よって、ベクトル命令で処理するときはできるだけ 256 個ずつ演算した方が命令の実行回数を減らすことができ効率が良くなります。さらに外側ループの繰返し制御のための時間も省くことができます。

自動ベクトル化では最内側ループをベクトル化の対象とします。例-7 では最内側ループの繰返し数は最大でも 100 回と 256 より短いので、このままでは一度に 100 個ずつ処理するようベクトル化されてしまいます。このループに一重化を適用すると、繰返し数を 10,000(=100×100)にでき、256 個ずつ処理できるようになります。

例-7

（ループの一重化前）

```
INTEGER::M, N
PARAMETER(M=100, N=100)
REAL(KIND=8)::A(M,N), B(M,N), C(M,N)
DO I = 1, M
   DO J = 1, N
      A(J,I) = B(J,I) + C(J,I)
   ENDDO
ENDDO
```

（ループの一重化後の変形イメージ）

```
DO IJ = 1, M*N
   A(IJ,1) = B(IJ,1) + C(IJ,1)
ENDDO
```

3.3. ループの入れ換え

多重ループのとき、ループを入れ換えることによりベクトル化できないデータの依存関係の問題が解消されてベクトル化できるようになる場合や、内側のループよりも外側のループの繰返し数が大きく、入れ換えた方が効率がよいと判断された場合には、例-8 のようにコンパイラが自動的にループを入れ換えてベクトル化を適用します。

例-8

（ループの入れ換え前）

```
DO J=1, 1000
   DO I = 1, 999
      A(I+1,J) = A(I,J) + B(I,J)
   ENDDO
ENDDO
```

（ループの入れ換え後のイメージ）

```
DO I = 1, 999
   DO J = 1, 1000
      A(I+1,J) = A(I,J) + B(I,J)
   ENDDO
ENDDO
```

なお、SX-ACE では、ハードウエア特性を考慮し、ループの繰返し数を長くするよりも配列要素の連続アクセスを優先して最適化することがあります。すなわち、外側ループの繰返し数が内側より大きく SX-9 ではループ入れ換えされていたループが、内側ループが連続アクセスできるときは SX-ACE ではループ入れ換えの対象とならない場合があります。

3.4. 条件ベクトル化

条件ベクトル化とは、一つのループに対してあらかじめ二つ以上の命令コードを用意しておき、実行時に最も効率よくに実行できる命令コードを選択して実行するベクトル化です。

例-9 は、データの依存関係がベクトル化に適合しているかどうかコンパイル時に不明であったとき、ベクトル化した命令コードとベクトル化しない命令コードの両方を用意しておき、プログラムを実行するときそのどちらかを選択して実行する条件ベクトル化の例です。

例-9

（条件ベクトル化前）

```
DO I = N, N+10
   A(I) = A(I+K) + B(I)
ENDDO
```

（条件ベクトル化後のイメージ）

```
IF (K.GE.0.OR.K.LT.-10) THEN
!CDIR NODEP                ! ベクトルでの実行
   DO I = N, N+10
      A(I) = A(I+K) + B(I)
   ENDDO
ELSE
   DO I = N, N+10          ! スカラでの実行
      A(I) = A(I+K) + B(I)
   ENDDO
ENDIF
```

例-9 の条件ベクトル化後のイメージの IF 文が実行時にコードを選択するための判定文で、THEN ブロックはベクトルで実行するコード、ELSE ブロックはスカラで実行するコードです。IF 文の条件式に現れる「K.GE.0」が真のときは例-5 と同じようにデータの依存関係が変わりません。また、「K.LT.-10」のときには、「DO I=N,N+10」であることから常に I≠I+K が成り立ちデータの依存関係がありません。よって、条件式が真のときベクトルでの実行(THEN ブロック)となります。逆に、IF 文の条件式が偽であるとき、「2.4 データの依存関係」の例-4、図-3 で示したようなベクトル化後にデータの依存関係が変わってしまう(右下向きの矢印ができてしまう)のでスカラでの実行(ELSE ブロック)となります。

例-9 はデータの依存関係に着目した条件ベクトル化(依存関係による条件ベクトル化)です。他の条件ベクトル化として、ループの繰返し数による条件ベクトル化も行います。

4. プログラミングの際の留意点

Fortran 言語でプログラムを作成する場合、実行性能の面で注意が必要な言語機能があります。このセクションでは、使い方によって性能に大きな影響が出る可能性のある、配列代入文と配列ポインタおよび形状引継ぎ配列を使う場合の注意点について説明します。

4.1. 配列代入文

Fortran 言語では、配列データの演算を一括して行う配列式や配列代入文が利用できます。配列式や配列代入文は基本的にベクトル化され高速に実行できますが、複数の配列代入文を書く場合、同じ形状の配列代入文をなるべく連続して書くようにしてください。

同じ形状の配列代入文が連続するとき、コンパイラはそれらの配列代入文を1個のループにまとめてベクトル化するため、より効率のよい実行命令列が得られます。

例-10

```
SUBROUTINE SUB(A,B,C,D,N,X,Y,M)
REAL::A(N), B(N), C(N), D(N)
REAL::X(M), Y(M)

A = B**2 + C**2          ①
X = SIN(Y)               ②
D = SQRT(A)              ③

END SUBROUTINE SUB
```

例-10 では、配列 A、B、C の大きさが N、配列 X、Y の大きさが M となっています。すなわち、大きさ N の配列代入文①と③の間に、大きさが M の配列代入文②が割り込む形になっています。この場合、例-11 のように 3 個のループで書いた場合と同じような実行命令列が生成されます。

例-11

```
DO I = 1, N
   A(I) = B(I)**2 + C(I)**2
ENDDO
DO I = 1, M
   X(I) = SIN(Y(I))
ENDDO
DO I = 1, N
   D(I) = SQRT(A(I))
ENDDO
```

これを、例-12 のように同じ大きさの配列代入文①と③を並べて書くようにすれば、コンパイラはこの二つをまとめて一つのループとして実行命令列を作成します。

例-12

```
A = B**2 + C**2          ①
D = SQRT(A)              ③
X = SIN(Y)               ②
```

上記コードに対するコンパイラの最適化イメージ

```
DO I = 1, M
   A(I) = B(I)**2 + C(I)**2
   D(I) = SQRT(A(I))
ENDDO
DO I = 1, N
   X(I) = SIN(Y(I))
ENDDO
```

複数の配列代入文が一つのループにまとめられたかどうかは、最適化診断メッセージ、または、編集リストで確認できます。診断メッセージで確認するときは-Wf,-pvctl,fullmsg、編集リストで確認するときは-R2、または、-Wf,-L,fmtlist のコンパイラオプションを指定してください。例-12 で二つの配列代入文が一つのループにまとめられた場合のメッセージとリストは例-13 のようになります。

例-13

最適化診断メッセージ

```
f90: opt(11): ex.f90, line 5: 配列代入を融合した。:line 5 - 6
```

編集リスト

```
      LINE    LOOP      FORTRAN STATEMENT

      1:              SUBROUTINE SUB(A,B,C,D,N, X,Y,M)
      2:              REAL::A(N), B(N), C(N), D(N)
      3:              REAL::X(M), Y(M)
      4:
      5: V------>     A = B**2 + C**2
      6: V------      D = SQRT(A)
      7: V======      X = SIN(Y)
      8:
      9:              END SUBROUTINE SUB
```

編集リスト上で、7 行目のように、「V======」と表示されている場合は、その配列代入文が単独でループになっていることを示しています。この表示が複数並んでいるときは、同じ形状の配列代入文がないか、もしあったらそれらを連続するように並べ換えできないか確認してください。

4.2. 配列ポインタ、形状引継ぎ配列

通常、配列の各要素はメモリ上連続して配置されています。しかし、配列ポインタと形状引継ぎ配列の場合、隣接する要素がメモリ上不連続になることがあります。

例-14

```
INTERFACE
   SUBROUTINE SUB(X)
   REAL::X(:)              ! 形状引継ぎ配列
   END SUBROUTINE SUB
END INTERFACE

REAL,POINTER::P(:)         ! 配列ポインタ
REAL,TARGET::A(100)
REAL::B(5,5)

P=>A(1:100:2)              ! PはAの奇数要素と結合
CALL SUB(B(1,:))           ! SUBの形状引継ぎ配列XはBの1次元目の添字が1の要素と結合
...
```

例-14 において、ポインタ代入後の配列ポインタＰと、サブルーチン SUB が呼び出されたときの形状引継ぎ配列 X の要素のメモリ上の配置は図-5 のようになります。

P(1)		P(2)		P(3)	...	P(49)		P(50)	

X(1)				
X(2)				
X(3)				
X(4)				
X(5)				

図-5 メモリ上の配置

通常は、このようにメモリ上不連続となっていることをあまり意識する必要はありませんが、これらの配列を手続の実引数に指定したときに性能上問題となることがあります。

例-15

```
INTERFACE
   SUBROUTINE SUB2(Y)
   REAL::Y(50)
   END SUBROUTINE SUB2
END INTERFACE

REAL,POINTER::P(:)         ! 配列ポインタ
REAL,TARGET::A(100)
P=>A(1:100:2)              ! PはAの奇数要素と結合
CALL SUB2(P)
```

例-15 では、配列ポインタＰをサブルーチン SUB2 の実引数としています。P に対応する SUB2 の仮配列は、Y(50)と宣言されています。Fortran 言語では、配列ポインタと形状引継ぎ配列以外の配列の要素はメモ

リ上連続領域でなければならないと定められています。このため、コンパイラは SUB2 を呼び出す前に P の要素が連続領域となっているかどうかをチェックし、もし不連続なら P の全要素の値を一旦作業用の領域にコピーし、その作業領域を SUB2 に渡し、SUB2 から戻った時点で作業領域から P に値を書き戻す処理を挿入します。

コンパイラが生成する処理のイメージは、以下のようになります。

例-16

```
f_copyin(P,wk)      ! この実行時ルーチンはPが連続領域でなければ作業領域に
                    ! コピーしwkにその作業領域のアドレスを設定する。
                    ! Pが連続ならwkはPの結合先のアドレスが設定される。
CALL SUB2(wk)       ! 連続領域であることが保証されているwkをSUB2に渡す。
f_copyback(wk,P)    ! この実行時ルーチンはwkとPが同じメモリ領域なら何も
                    ! しない。そうでなければwkからPに値をコピーする。
```

P が連続領域でない場合、2 回コピー処理が行われ、その分余計な時間がかかることになります。P が連続領域である場合でも、コピーほどではないにせよ、連続領域かどうかのチェックのためのオーバヘッドがかかります。連続領域へのコピー処理が生成されたかどうかは、コンパイラオプション-Wf,-msg,o を指定したとき出力される以下の注意メッセージで確認できます。

例-17

注意メッセージ

```
f90: observ(1): wk.f90, line 11: 実引数の配列を連続領域にコピーするコードを生成した。
```

このようなコピー処理が生成されるのを避けるには、ポインタまたは形状引継ぎ配列を実引数として結合される可能性のある仮引数を形状引継ぎ配列として宣言します。

以下に例を示します。

例-18

```
INTERFACE
   SUBROUTINE SUB3(Z)
   REAL::Z(:)           !Zは形状引継ぎ配列
   END SUBROUTINE SUB3
END INTERFACE

REAL,POINTER::P(:)      ! 配列ポインタ
REAL,TARGET::A(100)

P=>A(1:100:2)           !PはAの奇数要素と結合
CALL SUB3(P)            !Pと対応するZが形状引継ぎ配列であるためコピー処理は生成されない。
```

配列ポインタに関しては、もう一つ注意点があります。

Fortran 言語では、同一の配列が複数の異なる配列ポインタと結合することを許しています。配列ポインタが実際にどの配列と結合しているかは、コンパイル時には確定できませんので、言語仕様に忠実に解釈すると、型と次元数が同一であるすべての配列ポインタおよびそれと結合可能な配列の間には、依存がある可能性を考慮してベクトル化することになります。

たとえば、以下の例-19 において、配列ポインタ P1 と P2 が同一の配列と結合していた場合、依存関係があるものとみなされるためベクトル化できません。

例-19

```
REAL,POINTER::P1(:), P2(:)
REAL::X(100)
DO I = 1, 100
   P1(I+10) = P2(I) + X(I)   ！P1とP2が同一配列と結合しているとベクトル化できない依存関係がある
ENDDO
```

しかし、実際のプログラムでは、同一の配列を別々の配列ポインタに結合して一つのループ中でそれらを混在させて使用するようなことはほとんどありません。そのため、コンパイラの既定値の動作では、異なる配列ポインタは別の配列に結合しているものと仮定して、例-19 のようなループにも最適化・ベクトル化を適用します。

もし、上記のような仮定が成り立たないループをベクトル化すると正しい実行結果が得られません。そのようなプログラムをコンパイルする場合は、コンパイラオプション-Cvsafe を指定してください。なお、このオプションを指定すると、最適化・ベクトル化に様々な制約が課せられるため既定値と比べてベクトル化できなくなるループが増加してしまいます。したがって、一つの手続内で同一配列を複数の異なる配列ポインタに結合して使うことは避けてください。

5. ベクトルデータ・バッファリング機能

5.1. ADB

SX システムでは、ADB(Assignable Data Buffer)と呼ばれるハードウエア機構を利用したベクトルデータ・バッファリング機能を利用することで、ベクトル命令によるメモリアクセス性能を改善できます。

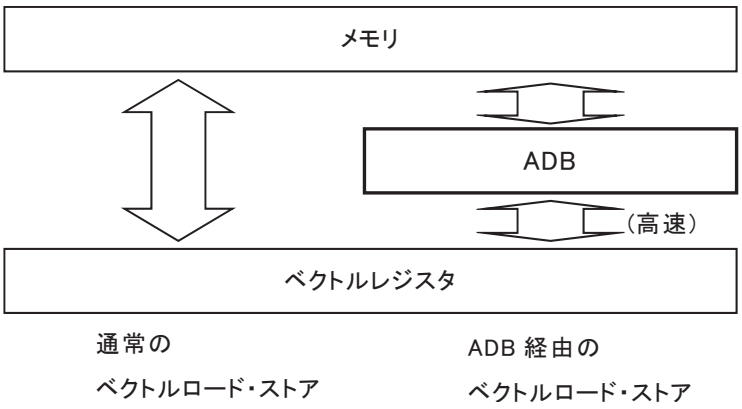


図-6 ADB (Assignable Data Buffer)

ADB を利用するとき、配列データの最初のベクトルロード、または、ベクトルストア時に、データが ADB にバッファリングされます。次にそのデータを利用するときには、ADB にバッファリングされたデータがロードされます。ADB からはメモリからロードするより高速にデータをロードできるため、2 回目以降のベクトルロードを高速化できます。

コンパイラは、配列データの再利用性、すなわち、ある配列データが 2 回以上アクセスされるかどうかに着目し性能向上が期待できる配列を自動的に選択し ADB にバッファリングされるようにします。

5.2. SX-ACE の ADB

ADB は SX-9 で初めて導入されましたが、その容量が SX-ACE では CPU コアあたり 1 メガバイトに大きく増加されました。また、ハードウエアのバッファリング機構の仕組みも改善され、バッファリングされたデータをより高速に置き換えられるようになりました。これに伴い、コンパイラでの ADB の利用ポリシーが変更されました。

表-2 SX-ACE と SX-9 の ADB

	SX-ACE	SX-9
ADB 容量/コア	1 メガバイト	256 キロバイト
利用ポリシー	再利用されることが明確なデータに加えて、リストベクトル、作業ベクトル、等間隔ベクトルなどで再利用性が不明な場合でも積極的に ADB にバッファリングし、できるだけ多くのベクトルデータをバッファリング	再利用性のあるデータをコンパイラが自動選択し、バッファリング
コンパイラオプション	-pvctl,on_adb=*カテゴリ指定*のコンパイラオプションが追加	-pvctl,on_adb
コンパイラ指示行	右の指示行に加えて、データをプリフェッチするための指示行が追加	!CDIR ON_ADB

5.3. コンパイラオプションによるカテゴリ指定

ADB の容量には限りがあるため、すべての配列データを ADB にバッファリングすることはできません。配列の大きさが ADB の容量(1メガバイト)以下でありアクセス頻度が高い配列データを選択的にバッファリングする方が性能向上を期待できます。

コンパイラは、こういった配列データをコンパイラが自動的に選択する機能の他に、コンパイラ指示行 !CDIR ON_ADB でユーザが配列を指定する機能、特別な属性を持つ配列データをコンパイラオプションでまとめて指定する機能(コンパイラオプションによるカテゴリ指定機能)を持ちます。コンパイラオプションによるカテゴリ指定機能は SX-ACE で新たに強化されました。

コンパイラオプションによるカテゴリ指定では、連続、リストベクトルなどのアクセスパターンごと、共通ブロック、モジュール変数、引数などのデータの種類ごと、特定の用途で利用されるものごとなど、バッファリング

する配列データをその種類によって一括指定できます。

表-3 コンパイラオプションによるカテゴリ指定

	カテゴリ指定	バッファリングされるベクトルデータ
-pvctl,on_adb=	arg	仮引数、実引数
	common	共通ブロック要素
	contiguous	連続ベクトル
	indirect	リストベクトル
	module	モジュール変数
	reuse	コンパイラが自動選択した再利用されるベクトルデータ
	stride	等間隔ベクトル
	work	部分ベクトル化で利用される作業ベクトル

カテゴリを複数指定するときは、「:」(コロン)で区切って指定します。カテゴリ指定の前に「no」を付けると、そのカテゴリに属するデータを ADB にバッファリングしなくなります。カテゴリ指定を省略したとき、contiguous 以外のすべてのカテゴリが指定されたものとみなされます。例-20 にコンパイラオプションの指定例を示します。

例-20

```
% cat a.f90
SUBROUTINE SUB(C)
COMMON /COM/A,B
REAL(KIND=8)::A(1000), B(1000), C(1000)
B = A * 2
A = B + C
END SUBROUTINE
% sxf90 -Wf,-pvctl,on_adb=arg:nocommon a.f90
```

5.4. ADB ヒット要素率

ADB にバッファリングされたデータの再利用性を確認するための性能値として「ADB ヒット要素率」がプログラム実行解析情報、簡易性能解析機能/Ftrace Viewer の解析リスト/グラフに出力されます。出力形式については、「6. 性能解析」を参照してください。

あるベクトルデータをベクトルロードする際、その配列データがあらかじめ ADB にバッファリングされていたとき ADB ヒットとみなされます。ADB ヒット要素率は、ベクトルロードした全要素のうち ADB ヒットした要素

の比率を表します。

この性能値の利用にあたっては次のことにご留意ください。

- ある配列要素が二度ベクトルロードされるとき、1 回目は必ずミスヒット、2 回目は 1 回目でバッファリングされていればヒットとなります。ADB ヒット要素率の値は大きいほどよいですが、アクセスパターン、頻度によっては値が小さくても性能上問題ありません。
- 手続中に再利用されるベクトルデータがない(少ない)、すなわち、二度以上アクセスされるデータがないとき、ADB ヒット要素率の値は 0%に近づきます。

6. 性能解析

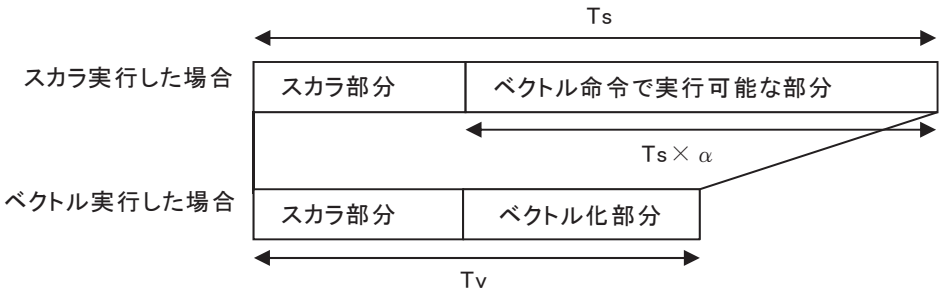
SX-ACE においては、プログラムに含まれるループをできるだけベクトル命令を使って、データを最大ベクトル長である 256 個ずつ実行できているかどうかが性能分析のポイントになります。

本セクションでは、プログラムの性能を確認するための性能値の取得方法、ベクトル化が十分できていない手続を絞り込む方法などを説明します。

6.1. ベクトル化率

プログラムをスカラ命令だけで実行させた場合の実行時間に占めるベクトル命令で実行可能な部分の時間の割合をベクトル化率と呼びます。一般にベクトル化率を正確に求めることは困難であるため、ベクトル化率の近似値としてベクトル演算率を用います。

ベクトル演算率は、プログラムで処理された全演算数に占めるベクトル命令で処理された演算数の割合を求めたものです。SX-ACE では、このベクトル演算率を大きくすることを目標にチューニングしてください。ベクトル演算率は、プログラム中のループをベクトル化することにより実行性能が向上しますが、ベクトル演算率が 50%程度では、スカラ実行時の高々2 倍の性能にしかならないことは、図-7 からもわかると思います。一般にはベクトル演算率が 98%以上を目指すことになります。ベクトル演算率は、以降で示すプログラム実行解析情報、簡易性能解析機能/Ftrace Viewer で参照できます。



Ts：スカラ実行したときの実行時間　　α：ベクトル化率

Tv：ベクトル実行したときの実行時間

図-7 ベクトル化率

6.2. プログラム実行解析情報

プログラム実行解析情報は、プログラムの実行時に参照される環境変数 F_PROGINF に YES、または、DETAIL が設定されているとき、プログラムの実行終了時に標準エラー出力ファイルに出力されます。この情報から、プログラムがハードウエアの性能を十分に引き出しているか否かを判断できます。環境変数 LANG に値 japan、または、ja_JP.UTF-8 が設定されているときは日本語、そうでないときは英語で出力されます。

例-21 にプログラム実行解析情報の出力例を示します。

例-21

```
    ******  Program Information  ******
Real Time (sec)                :          875.787967    経過時間
User Time (sec)                :         3498.747107    ユーザ時間
Sys  Time (sec)                :            2.455352    システム時間
Vector Time (sec)              :         3329.276575    ベクトル命令実行時間
Inst. Count                    :       597017752356.    全命令実行数
V. Inst. Count                 :       464204772550.    ベクトル命令実行数
V. Element Count               :    100858351225030.    ベクトル命令実行要素数
V. Load Element Count          :     11548190101324.    ベクトルロード命令実行要素数
FLOP Count                     :     43776313912608.    浮動小数点データ実行要素数
MOPS                           :        28864.951114    MOPS値
MFLOPS                         :        12511.997173    MFLOPS値
A. V. Length                   :          217.271250    平均ベクトル長
V. Op. Ratio (%)               :           99.868490    ベクトル演算率
Memory Size (MB)               :         5376.000000    メモリ使用量
MIPS                           :          170.637584    MIPS値
I-Cache (sec)                  :            0.605043    命令キャッシュミス時間
O-Cache (sec)                  :           23.921375    オペランドキャッシュミス時間
Bank Conflict Time
  CPU Port Conf. (sec)         :           35.571349    CPUポート競合時間
  Memory Network Conf. (sec)   :         1597.203733    メモリネットワーク競合時間
ADB Hit Element Ratio (%)      :           25.849985    ADBヒット要素率
```

プログラム実行解析情報のベクトル演算率が 98%未満のとき、プログラムを調べ、ベクトル化できていないループがないかなどを調査します。

6.3. 簡易性能解析機能／Ftrace Viewer

プログラムの規模が大きいとき、プログラム内のコードすべてについてループのベクトル化状況を調べるのは作業効率がよくありません。ベクトル性能を引き出すには、実行時間が長く、ベクトル演算率が低い手続に絞って調べることが効果的です。この絞り込みには簡易性能解析機能、またはFtrace Viewerが有効です。

Ftrace Viewerでは、手続ごとの実行回数、実行時間、FLOPS値、ベクトル演算率、ADBヒット要素率などの様々な性能値に加えて、それらをグラフ(Function Metric Chart)表示し、視覚的にプログラムの性能を分析できます。

図-8は、実行時間とベクトル演算率を重ねて表示したグラフです。これを参照し、実行時間が長く、ベクトル演算率の低いものを探し、その中のループのベクトル化状況を調べます。

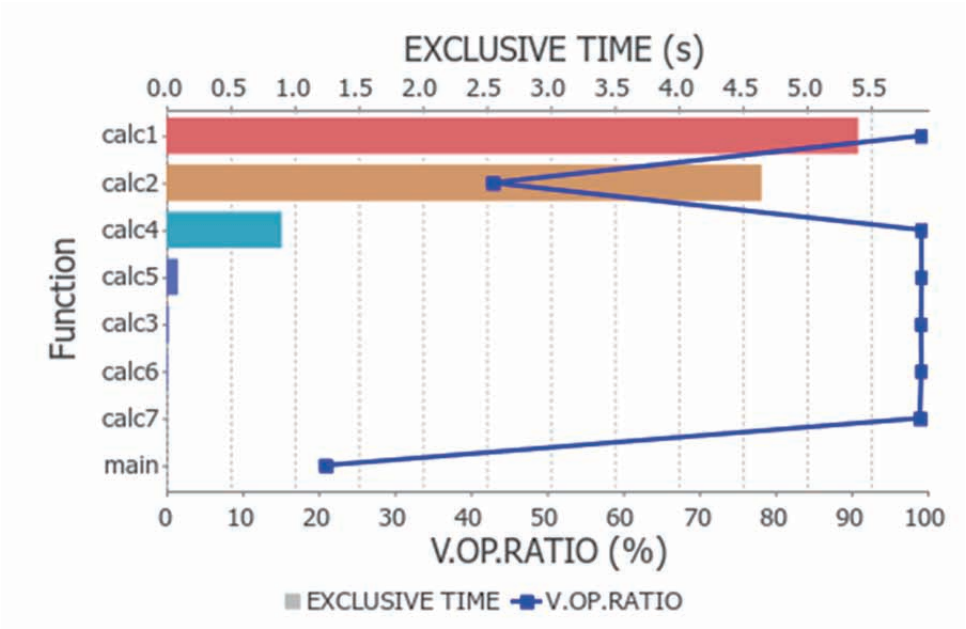


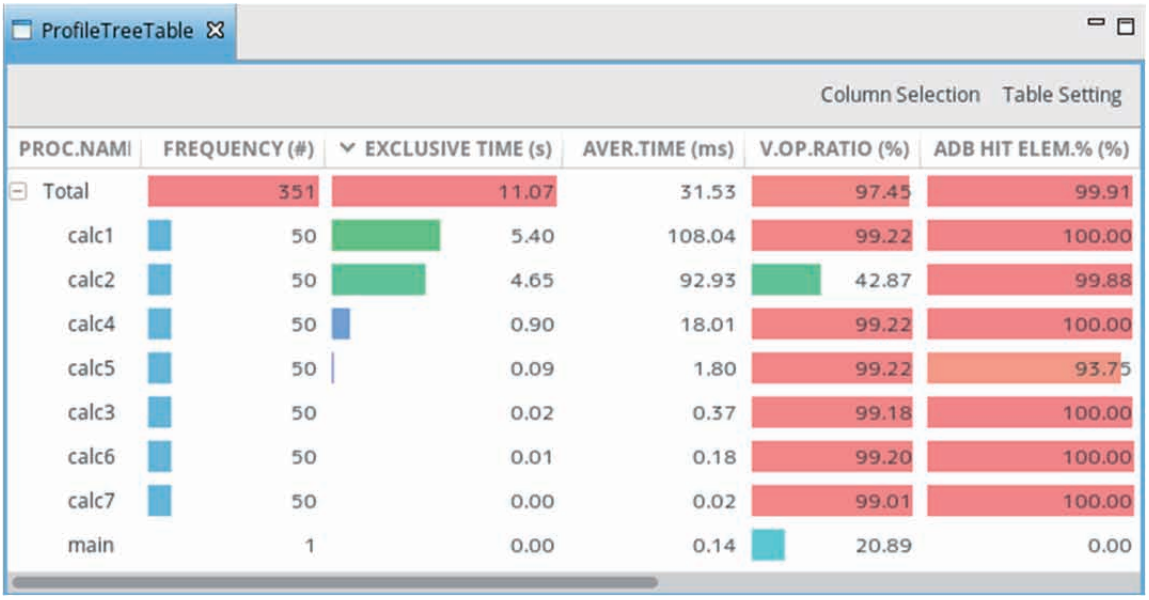
図-8　Function Metric Chart

図-8 では、縦軸が手続名、上横軸が実行時間(EXCLUSIVE TIME、単位は秒)、下横軸がベクトル演算率(V.OP.RATIO、単位は%)です。実行時間は棒グラフ、ベクトル演算率は折れ線グラフで表示されています。

このグラフから、実行時間が長くベクトル演算率が低い手続は「calc2」と分かります。「calc1」は実行時間は長いですがベクトル演算率も高く、ベクトル演算率を高めるという観点からのチューニングの余地は少なそうです。他のベクトル演算率の低い手続に main がありますが、これは実行時間が短いのでチューニングしてもプログラム全体の実行時間の短縮は望めません。

図-8 の例では、手続「calc2」に絞ってソースコードを調べることが必要だと判ります。

個々のサブルーチン、手続の詳細な性能値を調べたいとき、グラフと一緒に表示されるテーブル（Profile Tree Table）を参照します。



PROC.NAME	FREQUENCY (#)	EXCLUSIVE TIME (s)	AVER.TIME (ms)	V.OP.RATIO (%)	ADB HIT ELEM.% (%)
Total	351	11.07	31.53	97.45	99.91
calc1	50	5.40	108.04	99.22	100.00
calc2	50	4.65	92.93	42.87	99.88
calc4	50	0.90	18.01	99.22	100.00
calc5	50	0.09	1.80	99.22	93.75
calc3	50	0.02	0.37	99.18	100.00
calc6	50	0.01	0.18	99.20	100.00
calc7	50	0.00	0.02	99.01	100.00
main	1	0.00	0.14	20.89	0.00

図-9　Profile Tree Table

図-9 から、手続「calc2」の実行回数(FREQUENCY)は 50 回、一回当たりの実行時間(AVER.TIME)は 92.93 ミリ秒で総実行時間は 4.65 秒だったことがわかります。さらに、ベクトル演算率(V.OP.RATIO)は 42.87%、ADB ヒット要素率は 99.88%だったこともわかります。

Ftrace Viewer で性能値を参照するには、コンパイラオプション-ftrace を指定してコンパイルされたプログラムを実行して、解析情報ファイル(ファイル名 ftrace.out.*.*)を出力することが必要です。

7.　おわりに

以上、FORTRAN90/SX コンパイラの自動ベクトル化機能を中心にご紹介させて頂きました。SX-ACE のコンパイラは、他にも本稿でご紹介できなかった種々のベクトル化機能を持っています。詳細につきましては、マニュアル「FORTRAN90/SX プログラミングの手引」をご参照ください。

皆様が SX-ACE をご利用になる上で、本稿が多少なりともお役に立てれば幸いです。

参考文献

[1] FORTRAN90/SX プログラミングの手引　日本電気株式会社 G1AF07

[2] Ftrace Viewer 利用の手引　日本電気株式会社 G1AF33