

H26年度
スーパーコンピュータの高速化技法入門
自動ベクトル化による高速化技法

2014年 6月17日
大阪大学サイバーメディアセンター
日本電気株式会社

本資料は、東北大学サイバーサイエンスセンターとNECの共同により作成され、大阪大学サイバーメディアセンターの環境で実行確認を行い、修正を加えたものです。
無断転載等は、ご遠慮下さい。

目次

FORTRAN90 / SXの自動ベクトル化機能

- ベクトル化とは？
- 自動ベクトル化の条件
- 拡張ベクトル化機能
- 編集リストと変形リスト

性能チューニング

- ベクトル化による高速化
- 性能チューニングの手順
- 性能の分析
- チューニングの実施

ベクトル化における注意事項

目次

FORTRAN90 / SXの自動ベクトル化機能

- ベクトル化とは？
- 自動ベクトル化の条件
- 拡張ベクトル化機能
- 編集リストと変形リスト

性能チューニング

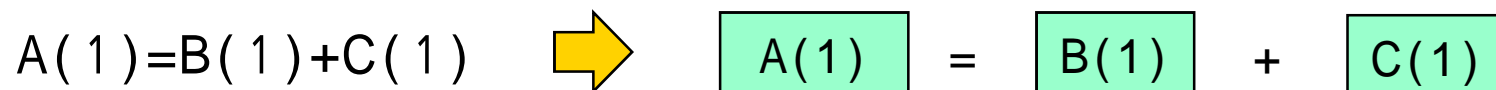
- ベクトル化による高速化
- 性能チューニングの手順
- 性能の分析
- チューニングの実施

ベクトル化における注意事項

自動ベクトル化

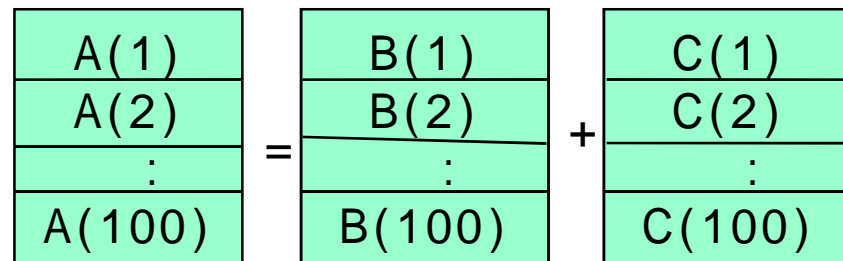
ループ中で繰り返される規則的に並んだ配列データ(ベクトルデータ)の演算に対してコンパイラがベクトル命令を適用すること

スカラ命令のイメージ

$$A(1) = B(1) + C(1)$$


ベクトル命令の適用のイメージ

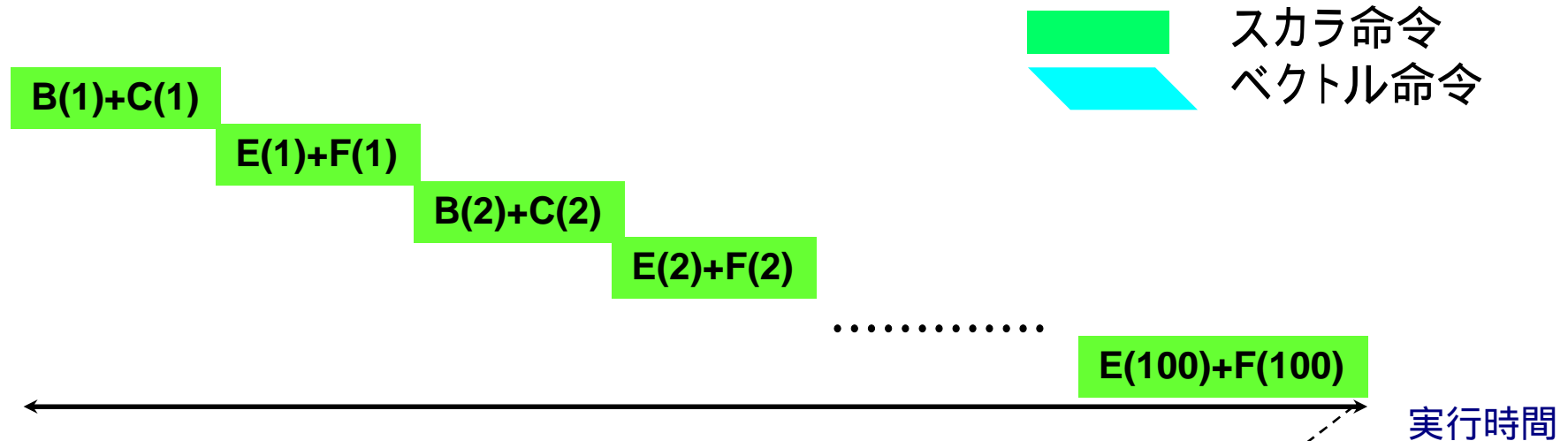
```
DO I=1,100  
  A(I)=B(I)+C(I)  
ENDDO
```



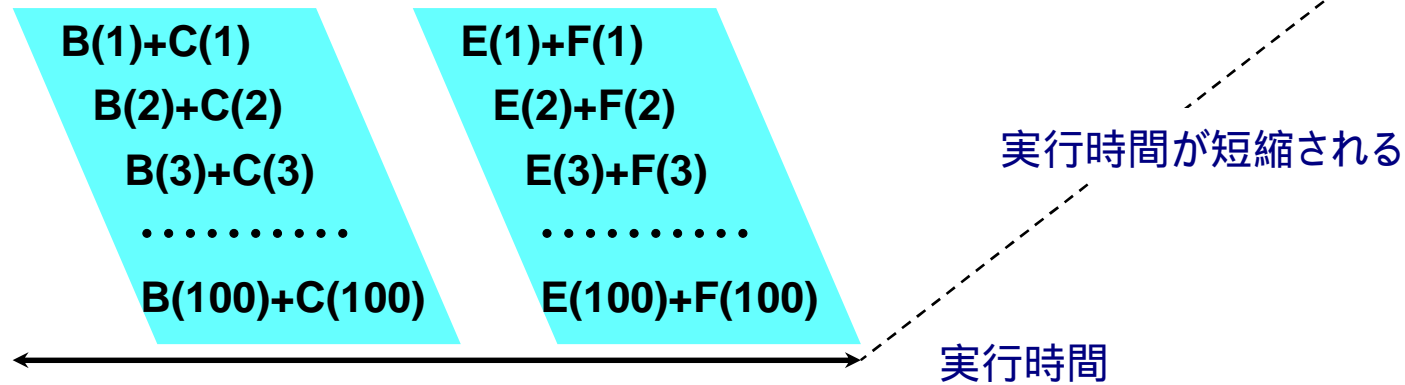
A(1)	=	B(1)	+	C(1)
A(2)		B(2)		C(2)
⋮		⋮		⋮
A(100)		B(100)		C(100)

スカラ命令とベクトル命令

スカラ命令 (スカラ加算) の実行イメージ



ベクトル命令 (ベクトル加算) の実行イメージ



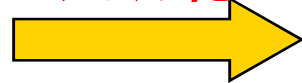
ベクトル化による実行順序の変更

ベクトル化を行うと、実行順序が変更される

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

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

ベクトル化



スカラでの実行順序

```
A(1) = B(1) + C(1)  
D(1) = E(1) + F(1)  
A(2) = B(2) + C(2)  
:  
:  
:  
A(100) = B(100) + C(100)  
D(100) = E(100) + F(100)
```

ベクトルでの実行順序

```
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)
```

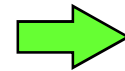
スカラ命令

ベクトル命令

配列構文のベクトル化

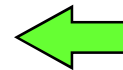
ソースプログラム

```
A(1:M,1:N)=B(1:M, 1:N)+C(1:M,1:N)  
B(1:M,1:N)=SIN(D(1:M,1:N))
```



コンパイラによる変形イメージ1

```
DO J = 1, N  
  DO I = 1, M  
    A(I,J)=B(I,J)+C(I,J)  
  ENDDO  
ENDDO  
DO J = 1, N  
  DO I = 1, M  
    B(I,J)=SIN(D(I,J))  
  ENDDO  
ENDDO
```



コンパイラによる変形イメージ2

```
DO J = 1, N  
  DO I = 1, M  
    A(I,J)=B(I,J)+C(I,J)  
    B(I,J)=SIN(D(I,J))  
  ENDDO  
ENDDO
```

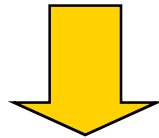
配列構文は、内部的にはDOループと同様に变形してから、ループ融合、一重化などの最適化を行い、最適な次元でベクトル化される。

IF文のベクトル化

条件分岐(IF文)もベクトル化される

```
DO I=1,100  
  IF (A(I) .LT. B(I)) THEN  
    A(I) = B(I) + C(I)  
  ENDIF  
ENDDO
```

ベクトル実行



```
mask(1) = A(1) .LT. B(1)  
mask(2) = A(2) .LT. B(2)  
      :      :      :  
mask(100) = A(100) .LT. B(100)
```

マスク生成

```
if (mask(1)) A(1) = B(1) + C(1)  
if (mask(2)) A(2) = B(2) + C(2)  
      :      :      :  
if (mask(100)) A(100) = B(100) + C(100)
```

マスク付ベクトル演算

maskが真の要素のみ
処理が行われる

目次

FORTRAN90 / SXの自動ベクトル化機能

- ベクトル化とは？
- 自動ベクトル化の条件
- 拡張ベクトル化機能
- 編集リストと変形リスト

性能チューニング

- ベクトル化による高速化
- 性能チューニングの手順
- 性能の分析
- チューニングの実施

ベクトル化における注意事項

自動ベクトル化の条件

コンパイラが自動ベクトル化を行う条件:

- (1) ベクトル化に適合するループ、文、型、演算であること。
 - 4倍精度実数型は、ベクトル演算命令が無いためベクトル化できない
 - 利用者手続(関数・サブルーチン)の呼び出しを含むループはベクトル化できない
 - 入出力文はベクトル化できない
- (2) 配列や変数の定義・引用関係に、ベクトル化を阻害する依存関係が無いこと
- (3) ベクトル化によって、性能の向上が期待できること。
 - ループ長が十分に長い
 - ループ分割など、ベクトル化で増加するコストよりも、ベクトル化による性能向上の効果のほうが大きい

ベクトル化の対象範囲

対象となるループ	配列式、DOループ、DO WHILEループ、IF文とGOTO文によるループ
対象となるループ中に許される文	代入文、CONTINUE文、GOTO文、CYCLE文、EXIT文、IF文、CASE構文 (CALL文、入出力文、ポインタ代入等は不可)
対象となるデータの型	4バイト、8バイトの整数型・論理型 単精度、倍精度の実数型・複素数型 (文字型、4倍精度、2バイトの整数型、 1バイトの論理型、構造型は不可)
対象となる演算	加減乗除算、べき算、論理演算、関係演算、 型変換、組込み関数 (利用者定義演算は不可)

ベクトル化を阻害する依存関係 (1)

ベクトル化を阻害する依存関係

以前の繰り返しで定義された配列要素や変数を後の繰り返しで引用するパターンのとき、ベクトル化を阻害する依存関係がある

例1

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

ベクトル化すると、更新された A の値が使用されなくなってしまうのでベクトル化できない

スカラーでの実行順序

$a(3) = A(2) * B(2) + C(2)$
 $a(4) = a(3) * B(3) + C(3)$
 $a(5) = a(4) * B(4) + C(4)$
 $a(6) = a(5) * B(5) + C(5)$
⋮

a : 更新されたAの値

ベクトルでの実行順序

$a(3) = A(2) * B(2) + C(2)$
 $a(4) = A(3) * B(3) + C(3)$
 $a(5) = A(4) * B(4) + C(4)$
 $a(6) = A(5) * B(5) + C(5)$
⋮

A : 元のAの値

例2

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

ベクトル化しても実行順序は変わらないのでベクトル化できる

スカラーでの実行順序

$a(1) = A(2) * B(2) + C(2)$
 $a(2) = A(3) * B(3) + C(3)$
 $a(3) = A(4) * B(4) + C(4)$
 $a(4) = A(5) * B(5) + C(5)$
⋮

a : 更新されたAの値

ベクトルでの実行順序

$a(1) = A(2) * B(2) + C(2)$
 $a(2) = A(3) * B(3) + C(3)$
 $a(3) = A(4) * B(4) + C(4)$
 $a(4) = A(5) * B(5) + C(5)$
⋮

A : 元のAの値

ベクトル化を阻害する依存関係 (2)

例3

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

変数の引用が、定義よりも先に
現れるループもベクトル化できない



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

例4

```
DO I=1, N  
  A(I) = A(I+K) + B(I)  
ENDDO
```

スカラでの実行順序

```
a(1) = S  
S = B(1) + C(1)  
a(2) = S  
S = B(2) + C(2)  
:
```

ベクトルでの実行順序

```
a(1) = S  
a(2) = S  
:  
a(N) = S  
S = B(1) + C(1)  
S = B(2) + C(2)  
:
```

プログラムを変形することで
ベクトル化できるようになる

コンパイル時にKの値が不明なため、
依存関係の有無が判定できないので、
ベクトル化できない

ベクトル化を阻害する依存関係 (3)

例5

```
S = 1.0
DO I = 1,N
  IF(A(I) .LT.0.0) THEN
    S = A(I)
  ENDIF
  B(I) = S + C(I)
ENDDO
```

変数の引用の前に定義があっても、
定義が実行されない可能性がある
とベクトル化できない

例6

```
DO I = 1,N
  IF(A(I) .LT.0.0) THEN
    S = A(I)
  ELSE
    S = D(I)
  ENDIF
  B(I) = S + C(I)
ENDDO
```

Sの参照の前に必ずSの定義がある
のでベクトル化できる

ベクトル化による性能の向上の判定

例7

```
DO I = 1,2  
  A(I)=B(I)+C(I)  
  D(I)=E(I)+ F(I)  
ENDDO
```

ループ長が2しかないため、ベクトル命令を適用しても高速化されないため、コンパイラは **ベクトル化しない**

例8

ベクトル化不可の依存

```
S = 1.0  
DO I=1,N  
  IF(A(I).LT.0.0) THEN  
    S = A(I)  
  ENDIF  
  WK = S + C(I)  
  B(I) = WK * 2.0  
ENDDO
```

~~自動変形~~

```
S = 1.0  
DO I=1,N  
  IF(A(I).LT.0.0) THEN  
    S=A(I)  
  ENDIF  
  WK(I) =S+C(I)  
ENDDO  
  
DO I=1,N  
  B(I) = WK(I) * 2.0  
ENDDO
```

ベクトル化されないループ

ベクトル化されるループ

コンパイラの拡張ベクトル化機能で部分的にベクトル化を行うことが可能だが、ベクトル化による高速化の効果より、作業配列を使うコストが大きいと判断して、**ベクトル化しない**

目次

FORTRAN90 / SXの自動ベクトル化機能

- ベクトル化とは？
- 自動ベクトル化の条件
- 拡張ベクトル化機能
- 編集リストと変形リスト

性能チューニング

- ベクトル化による高速化
- 性能チューニングの手順
- 性能の分析
- チューニングの実施

ベクトル化における注意事項

拡張ベクトル化機能(1)

拡張ベクトル化機能

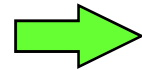
- そのままではベクトル化できない場合やより効率の良いベクトル化が可能な場合などに、コンパイラがプログラムを内部的に変形することで、ベクトル化の効果をさらに高める機能

- 文の入れ換え
- ループの一重化
- ループの入れ換え
- 部分ベクトル化
- 条件ベクトル化
- マクロ演算の認識
- 多重ループのベクトル化
- ループ融合
- インライン展開

拡張ベクトル化機能(2)

■ 文の入れ換え

```
ソースプログラム  
DO I=1,99  
  A(I)=2.0  
  B(I)=A(I+1)  
ENDDO
```



```
コンパイラによる変形のイメージ  
DO I=1,99  
  B(I)=A(I+1)  
  A(I)=2.0  
ENDDO
```

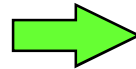
そのままベクトル化すると、B(1) ~ B(99)の値が全て2.0になってしまう。
このような場合に、ループ内の文の順序を入れ換えることにより、
ベクトル化できるように変形する

拡張ベクトル化機能(3)

多重ループの一重化

ソースプログラム

```
DIMENSION A(M,N), B(M,N), C(M,N)
DO J=1,N
  DO I=1,M
    A(I,J)=B(I,J)+C(I,J)
  ENDDO
ENDDO
```



コンパイラによる変形のイメージ

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

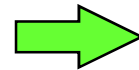
ループ長(ベクトル長)がより長くなるように、多重ループを一重化することにより、ベクトル化の効率をさらに高める

拡張ベクトル化機能(4)

多重ループの入れ替え

ソースプログラム

```
DO J=1,M  
  DO I=1,N  
    A(I+1,J)=A(I,J)+B(I,J)  
  ENDDO  
ENDDO
```



コンパイラによる変形のイメージ

```
DO I=1,N  
  DO J=1,M  
    A(I+1,J)=A(I,J)+B(I,J)  
  ENDDO  
ENDDO
```

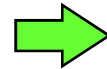
DO I=1,Nでベクトル化しようとする、配列Aに依存関係がありベクトル化できない。
ループを入れ換えると、DO J=1,Mのループに関しては依存関係がなくなりベクトル化できる。

拡張ベクトル化機能 (5)

部分ベクトル化

ソースプログラム

```
DO I=1,N  
  X = A(I) + B(I)  
  Y = C(I) + D(I)  
  WRITE (6,*) X, Y  
ENDDO
```



コンパイラによる変形のイメージ

```
DO I=1,N  
  WX(I) = A(I) + B(I)  
  WY(I) = C(I) + D(I)  
ENDDO  
DO I=1,N  
  WRITE (6,*) WX(I), WY(I)  
ENDDO
```

ループ構造や配列式に、ベクトル化できる部分とベクトル化できない部分が含まれている場合、ベクトル化可能な部分と不可能な部分に分割し、可能な部分だけをベクトル化する。

このとき必要であれば、作業配列（上の例ではWX, WY）を使用する。

拡張ベクトル化機能(6)

条件ベクトル化

依存関係チェックタイプ

```
DO I=N, N+100  
  A(I) = A(I+K) + B(I)  
ENDDO
```



```
IF (K.GE.0 .OR. K.LT.-100) THEN  
  ベクトル化したコード  
ELSE  
  ベクトル化しないコード  
END IF
```

ループ長タイプ

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



```
IF (N.GE.5) THEN  
  ベクトル化したコード  
ELSE  
  ベクトル化しないコード  
END IF
```

依存関係あるいはループ長が不明で、ベクトル化できるかどうかコンパイル時にわからない場合、実行時に依存関係あるいはループ長を調べ、ベクトル化コードを実行するか否かを決定する。

拡張ベクトル化機能(7)

マクロ演算の認識

総和型

```
DO I=1,N  
  S=S+A(I)  
ENDDO
```

漸化式型

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

最大値、最小値型

```
DO I=1,N  
  IF(XMAX.LT.X(I)) THEN  
    XMAX = X(I)  
  END IF  
ENDDO
```

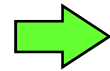
配列や変数の定義・引用関係に矛盾があり、本来はベクトル化できない場合でも、コンパイラが特別なパターンであることを認識し、特別なベクトル命令を用いることで、ベクトル化を行う。

拡張ベクトル化機能(8)

外側ループのベクトル化

ソースプログラム

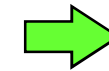
```
DO I=1,N  
  DO J=1,N  
    A(I,J) = 0.0  
  ENDDO  
  B(I)=1.0  
ENDDO
```



外側ループ
を分割

コンパイラによる変形 イメージ1

```
DO I=1,N  
  DO J=1,N  
    A(I,J)=0.0  
  ENDDO  
ENDDO  
DO I=1,N  
  B(I)=1.0  
ENDDO
```



ループを
一重化

コンパイラによる変形 イメージ2

```
DO I=1,N*N  
  A(I,1)=0.0  
ENDDO  
DO I=1,N  
  B(I)=1.0  
ENDDO
```

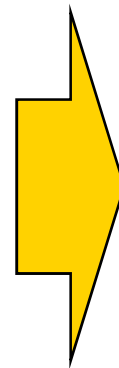
ベクトル化は、基本的には最内側ループをベクトル化するが、
外側のループを二つに分割することによって、外側のループに含まれる文も
ベクトル化を行う

拡張ベクトル化機能(9)

ループ融合

ソースプログラム

```
DO I=1,N  
  A(I) = B(I) + C(I)  
ENDDO  
DO I=1,N  
  D(I) = SIN(E(I))  
ENDDO
```



コンパイラによる変形のイメージ

```
DO I=1,N  
  A(I) = B(I) + C(I)  
  D(I) = SIN(E(I))  
ENDDO
```

コンパイラは同じ繰り返し回数を持つ複数のループ構造同士または、同じ形状(次元数と各次元のサイズ)を持つ複数の配列式同士を一つにまとめてベクトル化する。これをループ融合と呼ぶ。(-C hopt を指定したとき行われる)

拡張ベクトル化機能 (1 0)

■ 配列代入文のループ融合

ソースプログラム

```
A(1:M,1:N) = B(1:M,1:N) + C(1:M,1:N)
D(1:M,1:N) = E(1:M,1:N) * F(1:M,1:N) + S
```

コンパイラによる変形のイメージ

```
DO J=1,N
  DO I=1,M
    A(I,J) = B(I,J) + C(I,J)
    D(I,J) = E(I,J) * F(I,J) + S
  ENDDO
ENDDO
```

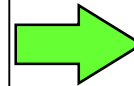
コンパイラは、同じ形状の配列式、ループ構造が連続していれば融合するが、間に形状の異なる配列式やループ構造、他の文があると融合できない。
高速化のためには、なるべく同じ形状の配列式、ループ構造を連続させるほうがよい。

拡張ベクトル化機能(11)

■ インライン展開によるベクトル化

ソースプログラム

```
DO I=1,N  
  CALL SUB(B(I),C(I))  
  A(I) = B(I)  
ENDDO  
  
:  
SUBROUTINE SUB(X,Y)  
X=SIN(Y)  
END
```



コンパイラによる変形のイメージ

```
DO I=1,N  
  B(I) = SIN(C(I))  
  A(I) = B(I)  
ENDDO  
  
:  
SUBROUTINE SUB(X,Y)  
X=SIN(Y)  
END
```

コンパイル時オプションに `-pi` を指定すると、インライン展開可能な利用者手続きを、呼び出し元にインライン展開する。

ループ中に手続きの呼び出しがあれば、展開後にベクトル化を行う。

目次

FORTRAN90 / SXの自動ベクトル化機能

- ベクトル化とは？
- 自動ベクトル化の条件
- 拡張ベクトル化機能
- 編集リストと変形リスト

性能チューニング

- ベクトル化による高速化
- 性能チューニングの手順
- 性能の分析
- チューニングの実施

ベクトル化における注意事項

編集リストと変形リスト

編集リスト

- ベクトル化 / 自動並列化に関する情報をソースプログラムの左側に表示
(コンパイラオプション -R5、-R2)
- どのループがベクトル化されたか？
- どの文が部分的にスカラで実行されるか？

変形リスト

- 拡張ベクトル化で行ったループの変形結果のソースを元のソースプログラムとマージして表示 (コンパイラオプション -R1、-R2)
- どのようにループが変形されたか？

編集リスト(1)

```
FILE NAME:t5.f
PROGRAM NAME: sub
FORMAT LIST
```

LINE	LOOP	FORTRAN STATEMENT
1:		subroutine sub(a,b,c,d,z,ix)
2:		real,dimension(100)::a,b,c,d,x,y,z
3:		integer ix(100)
4:	V----->	do l = 1,100
5:	l	call sub2(x,a,b,l)
6:		y(l) = c(l) + d(l)
7:	s	z(ix(l)) = z(ix(l)) + x(l) + y(l)
8:	V-----	enddo
9:		end

編集リスト(2)

◆ループ全体がベクトル化される場合

```
V-----> do I=1,100
|           a(I)=b(I)+c(I)
V-----  enddo
```

◆ベクトル化されない場合

```
+-----> do I=1,100
|         print *,a(I)
+-----  enddo
```

◆ループの一部がベクトル化される場合

```
V-----> do I =1,100
|         a(I)=b(I)+c(I)
|         S   print *,a(I)
V-----  enddo
```

ベクトル化不可の処理がある行には
“S” が表示される

編集リスト(3)

配列式をベクトル化した場合(1)

```
real a(90),b(90),c
V===== a(1:90)=b(1:90)+c
```

ループの先頭と最後の行が同じである場合、ループの構造は“=”で表示される

配列式をベクトル化した場合(2)

```
real a(90),b(90),c
integer d(90),e(90)
V-----> a(1:90)=b(1:90)+c
|          d(1:90)=int(a(1:90))
V----- e(1:90)=d(1:90)+1
```

ループ融合している場合は、その範囲について“V”で表示される

手続呼出しがインライン展開された場合

```
l call sub2(x,a,b,c,l)
```

インライン展開された手続がある行には“l”が表示される

編集リスト(4)

◆多重ループが一重化された場合

```
W-----> do J =1,100
|*-----> do I =1,100
||
||          a(I,J)=b(I,J)+c(I,J)
|*----- enddo
W----- enddo
```

一重化されたループの外側ループに“W”、内側ループに“*”が表示される

◆ループの入れ換えが行われた場合

```
X-----> do J =1,1000
|+-----> do I =1,10
||
||          a(I,J)=b(I,J)+c(I,J)
|+----- enddo
X----- enddo
```

入れ換えた結果ベクトル化されるループに“X”が表示され、ベクトル化されなくなるループには“+”が表示される

変形リスト

LINE	FORTRAN STATEMENT
1	subroutine sub(a,b)
2	real,dimension(100,100)::a,b
3	do j = 1,100
4	do i = 1,99
5	a(i+1,j)=a(i,j)+b(i,j)
6	enddo
7	enddo
.	do i = 1,99
.	!CDIR NODEP
.	do j = 1,100
.	a(i+1,j)=a(i,j)+b(i,j)
.	enddo
.	enddo
9	end

ループが入れ換えられて

ベクトル化不可の依存関係
がなくなった。

目次

FORTRAN90 / SXの自動ベクトル化機能

- ベクトル化とは？
- 自動ベクトル化の条件
- 拡張ベクトル化機能
- 編集リストと変形リスト

性能チューニング

- ベクトル化による高速化
- 性能チューニングの手順
- 性能の分析
- チューニングの実施

ベクトル化における注意事項

ベクトル化による高速化(3つのポイント)

■ ループ長

- ベクトル化対象のループ長は十分あるか？(100以上が望ましい)

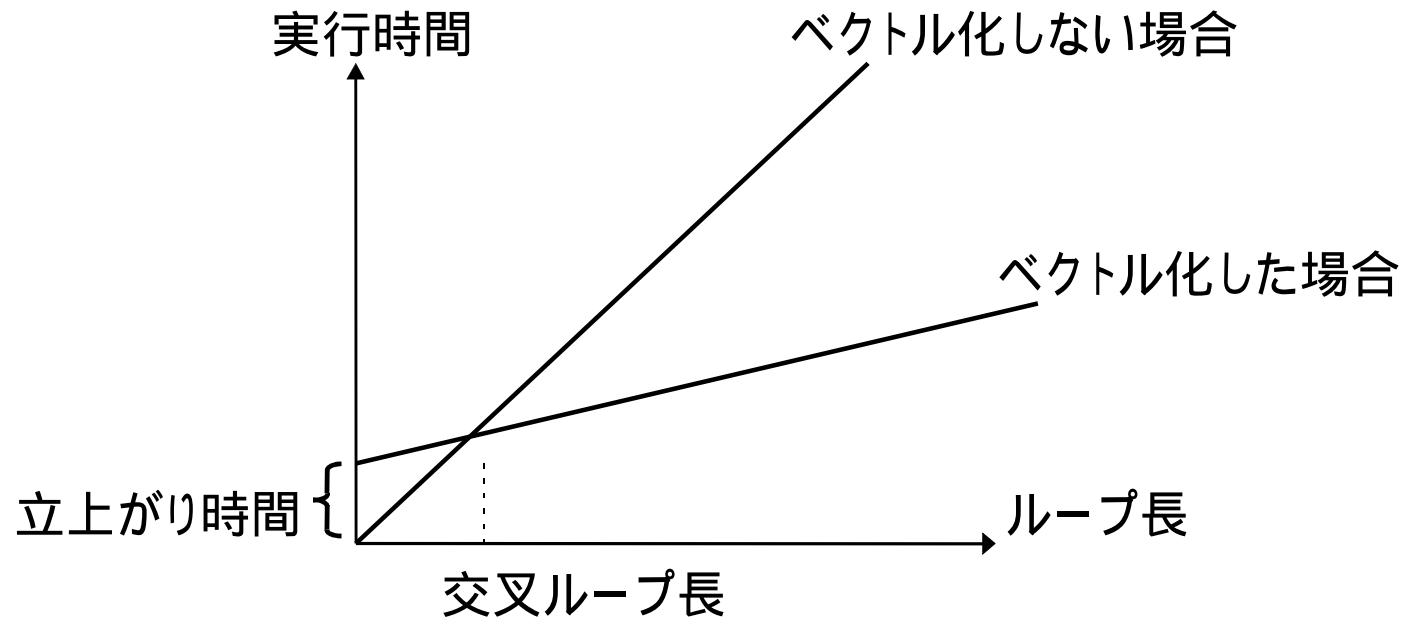
■ ベクトル化率

- ベクトル化率は、99%以上あるか？

■ メモリアクセス

- バンクコンフリクト時間の割合は小さいか？

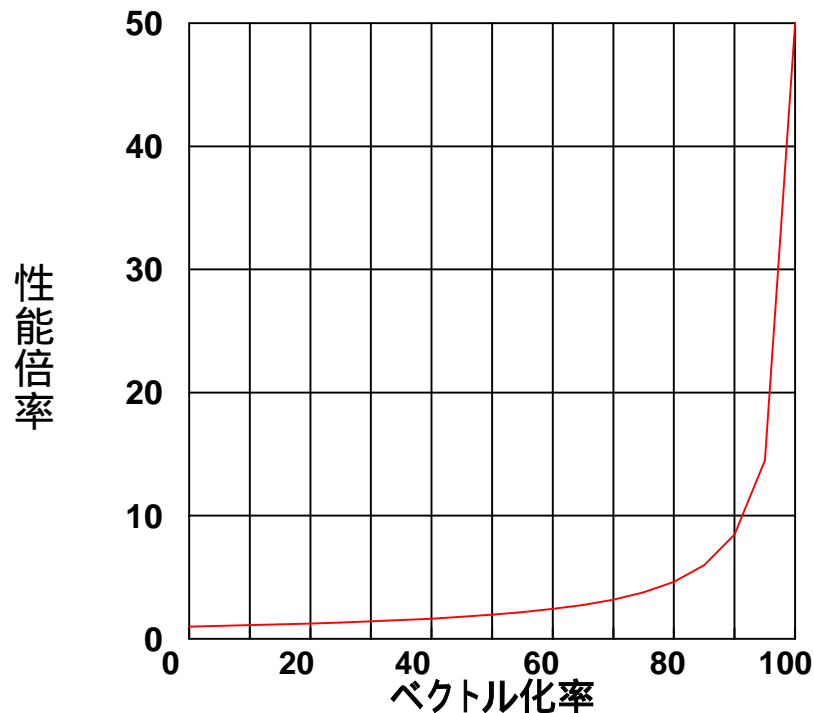
ループ長(ベクトル長)



ベクトル処理が開始されるまでに少し時間がかかる。(立ち上がり時間)
そのためループ長が非常に短い場合、ベクトル化しないほうが速い。
(交叉ループ長 5 程度)
ループ長をできるだけ長くした方が、ベクトル化による高速化の効果が大きいことがわかる。

ベクトル化率と性能(アムダールの法則)

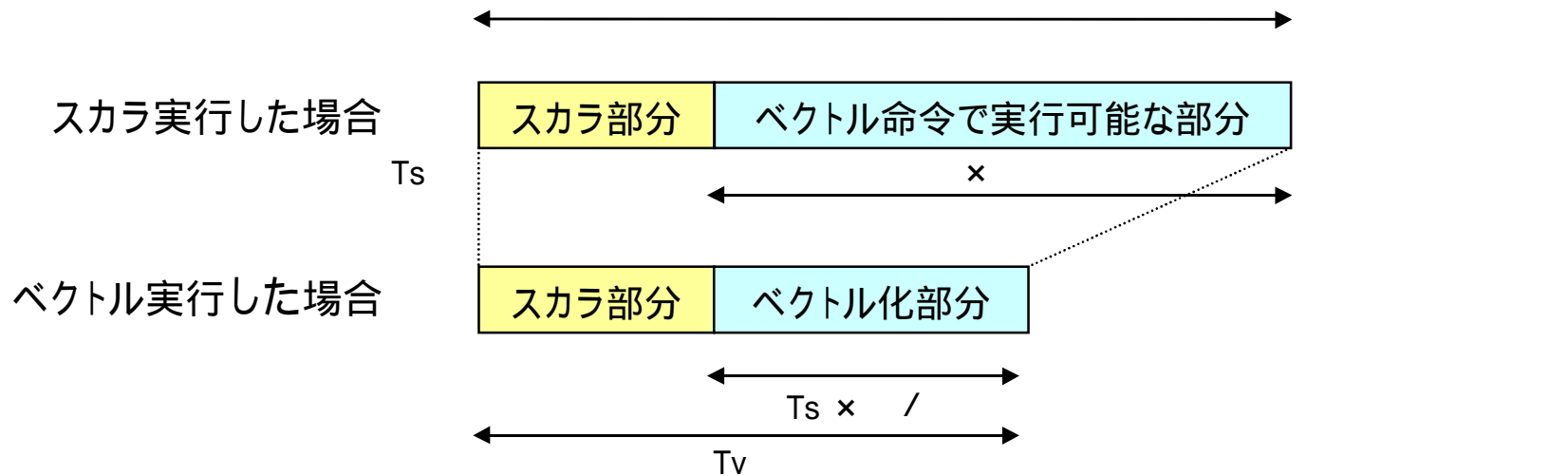
ベクトル化率が十分に高くなって初めて効果発揮



ベクトル化による性能向上比を50倍と仮定
ベクトル化率 50%、全体の性能は2倍
ベクトル化率 80%でも、全体の性能は4.6倍
にしかない。

ベクトル化で十分な高速化を行うためには、
ベクトル化率を可能な限り100%に近づける

ベクトル化率



α : ベクトル化率

$$\text{性能向上比} = T_s / T_v = 1 / (\alpha * (1 - \alpha) + 1)$$

β : ベクトル命令の性能向上比

T_s : スカラ実行したときの実行時間

$$\text{ベクトル演算率} = (T_s * \beta / \alpha) / T_v$$

T_v : ベクトル実行したときの実行時間

$$= \beta / (\alpha * (1 - \alpha) + 1)$$

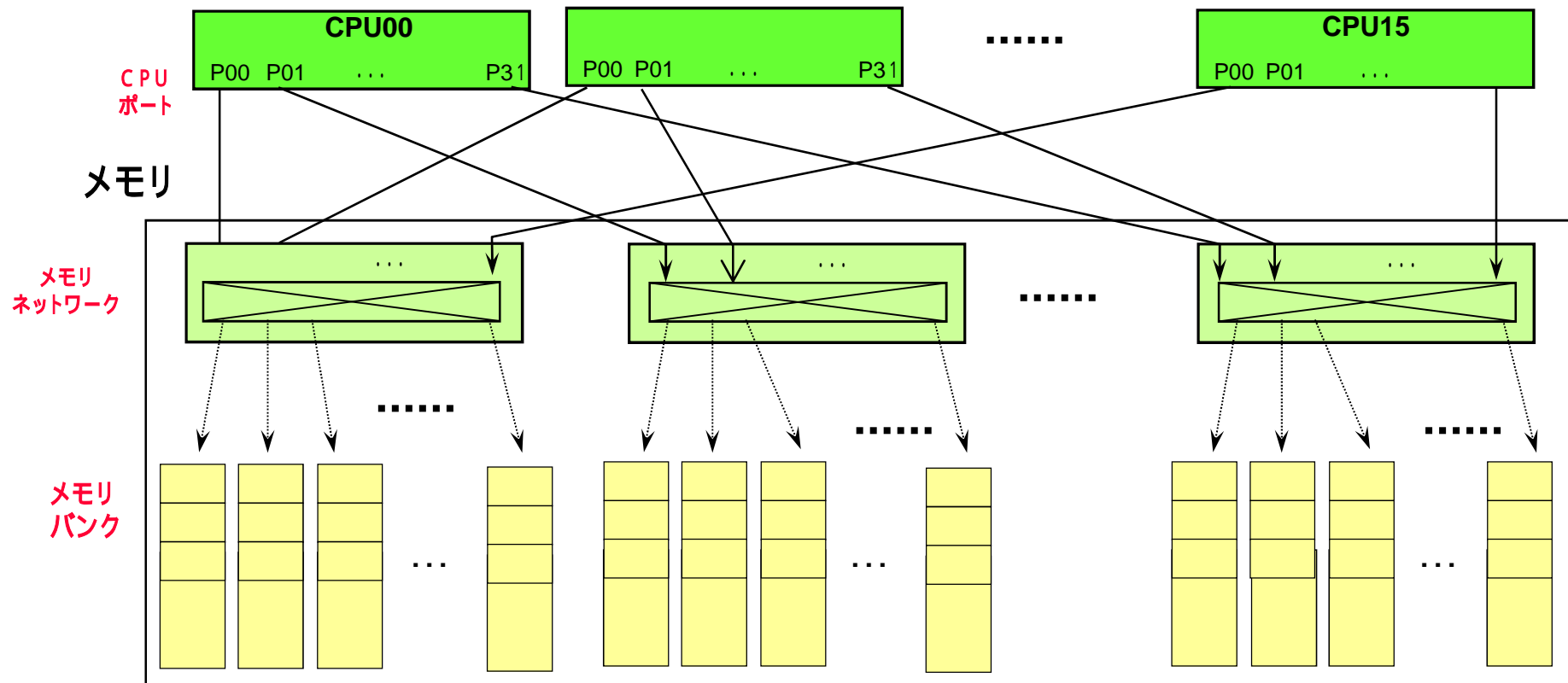
一般にベクトル化率を正確に求めることは困難であるため、ベクトル特性情報 (proginf) に表示される、[ベクトル演算率 \(後述\)](#) (ベクトル演算が実行された割合) で代用する。

メモリアクセスの効率化

SXのメモリは高速化のため多数のバンクから構成される。

- CPUとバンクは複数のパスにより接続
- ベクトルロード・ストアでは複数のバンク上のデータに同時にアクセス可能

SX-9のCPUとメモリの構成



バンク競合

複数のデータ転送が同一のバンク・同一のパスを使用するなどの要因により、データ転送性能が低下する現象

競合の発生する場所により

CPUポート競合
メモリネットワーク競合

の2種類に分けられる。

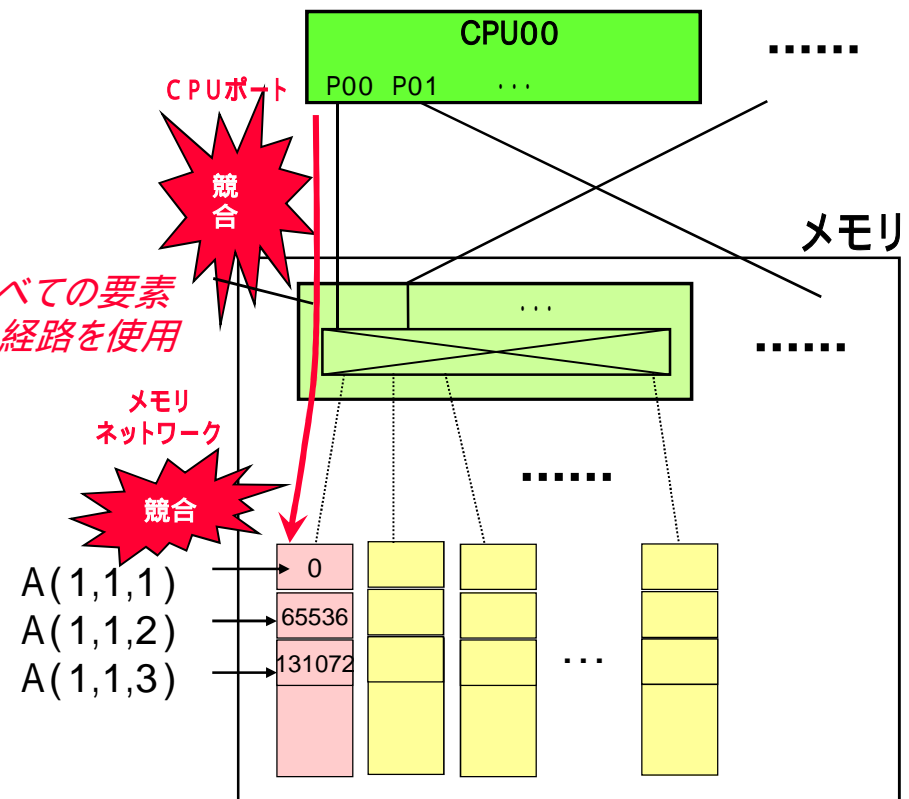
例

```
REAL*8 A(1024,64,100),B(100)
DO I=1,100
  A(1,1,I) = B(I)
ENDDO
```

アクセスされるAの要素

A(1,1,1)	Aの先頭
A(1,1,2)	Aの先頭 + 65536
A(1,1,3)	Aの先頭 + 131072
:	

これらのAの要素は同一のメモリバンク上に存在するためアクセス競合(コンフリクト)が発生し性能低下



目次

FORTTRAN90 / SXの自動ベクトル化機能

- ベクトル化とは？
- 自動ベクトル化の条件
- 拡張ベクトル化機能
- 編集リストと変形リスト

性能チューニング

- ベクトル化による高速化
- 性能チューニングの手順
- 性能の分析
- チューニングの実施

ベクトル化における注意事項

チューニングの手順

proginf情報からプログラム全体の性能を分析



プロファイラ情報、ftrace情報から、チューニングすべき
手続(サブルーチン・関数)を選択



ベクトル化診断メッセージから、チューニングすべき
ループ及び配列式を選択



チューニングの実施
・ベクトル化指示行の挿入
・ソースプログラムの変更

目次

FORTTRAN90 / SXの自動ベクトル化機能

- ベクトル化とは？
- 自動ベクトル化の条件
- 拡張ベクトル化機能
- 編集リストと変形リスト

性能チューニング

- ベクトル化による高速化
- 性能チューニングの手順
- 性能の分析
- チューニングの実施

ベクトル化における注意事項

PROGINF情報の利用

setenv F_PROGINF YES または DETAILで表示

- スーパーコンでは、既定値で設定されている (DETAILの表示例)

***** プログラム 情報 *****	
経過時間 (秒)	: 30.039067
ユーザ時間 (秒)	: 28.297595
システム時間 (秒)	: 0.017765
ベクトル命令実行時間 (秒)	: 28.263599
全命令実行数	: 11892009287.
ベクトル命令実行数	: 6099457558.
ベクトル命令実行要素数	: 1528191109757.
浮動小数点データ実行要素数	: 880815869041.
MOPS 値	: 54208.977057
MFLOPS 値	: 31126.881228
平均ベクトル長	: 250.545412
ベクトル演算率 (%)	: 99.622385
メモリ使用量 (MB)	: 512.031250
MIPS 値	: 420.248061
命令キャッシュミス (秒)	: 0.002944
オペランドキャッシュミス (秒)	: 0.011593
バンクコンフリクト時間	
CPU ポート競合 (秒)	: 0.455945
メモリネットワーク競合 (秒)	: 2.416176

平均ベクトル長、ベクトル演算率、
バンクコンフリクト時間に着目

ループ長は十分か？

ベクトル演算率は十分か？

メモリアクセス性能の
改善が必要か？

平均ベクトル長

- 一度のベクトル命令で演算を行った演算要素数の平均値
- ひとつのベクトル命令で、最大256要素同士の演算を行う

例:

```
DO I = 1, 300  
  A(I) = A(I) + B(I)  
ENDDO
```

256 要素の演算と44要素の演算の
2回で実行される。



平均ベクトル長 $(256+44)/2 = 150$

proginfの平均ベクトル長の表示が256を超えることは無い

プロファイラ情報の採取

```
> sxf90 -p test.f          -p を指定してコンパイル+リンク
> a.out
> prof a.out
```

%Time	Seconds	Cumsecs	#Calls	msec/call	Name	
35.0	56.80	56.80	1320	43.0303	sub7_	
28.2	45.73	102.53	20	2286.5	func15_	
12.8	20.80	123.34			ev_cdexp	← システムの手続
6.6	10.64	133.98	40	266.0	func23_	
5.5	8.86	142.84	15000	0.0005	sub12_	
:	:	:	:	:	:	

- : 全体に占める手続毎のCPU時間の割合 (%)
- : 手続毎のCPU時間 (秒)
- : 先頭からの合計のCPU時間 (秒)
- : 手続の呼出し回数 (-pを指定しない手続やシステムルーチンに対しては表示されない)
- : 一回の呼出し毎のCPU時間 (")
- : 手続の入口名 (ユーザ手続の場合、最後に_が付加される)

プロファイラ情報の利用

(1) profによる表示の上位の手続(関数・サブルーチン)に着目

(2) 呼出し毎の実行時間が大きい手続

➡ コストの高いループがベクトル化できているかを確認

⇒ ベクトル化されていれば、ループ長の拡大を検討
(多重ループの中で、ベクトル化されるループの変更を検討)

⇒ ベクトル化されていなければベクトル化の促進を検討

(3) 呼出し回数が大きく、呼出し毎の実行時間の小さい手続

➡ 手続のインライン展開を検討

簡易性能解析機能 : ftrace (1)

プログラム単位ごとに性能情報を採取

呼出し回数
呼出したプログラムを含まない実行時間とその割合
1回あたりの EXCLUSIVE TIME
proginf と同じ

```

*-----*
FLOW TRACE ANALYSIS LIST
*-----*

Execution Date : Fri Sep 19 17:58:12 2008
Total CPU Time : 0:00'28"266 (28.266 sec.)
    
```

PROG.NAME	FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CPU	CONFLICT PORT	NETWORK
sub5	1275	23.255(82.3)	18.239	61136.0	35800.2	99.62	250.2	23.248	0.0007	0.0036	0.034	0.911	
sub3	500	4.784(16.9)	9.567	20427.7	8298.5	99.69	256.0	4.781	0.0002	0.0010	0.061	1.840	
sub7	24	0.147(0.5)	6.124	72093.7	50785.4	99.93	250.3	0.146	0.0000	0.0005	0.001	0.006	
main	1	0.039(0.1)	39.140	47202.8	4145.6	99.52	256.0	0.029	0.0008	0.0040	0.000	0.001	
sub3	9	0.031(0.1)	3.432	49334.3	21004.3	99.79	256.0	0.031	0.0000	0.0000	0.002	0.013	
sub6	24	0.007(0.0)	0.301	67647.3	37418.1	99.57	256.0	0.007	0.0000	0.0000	0.000	0.002	
sub4	1	0.003(0.0)	2.501	43833.8	21628.6	99.80	250.3	0.002	0.0000	0.0000	0.000	0.000	
sub10	1	0.001(0.0)	0.892	150.6	1.8	0.96	7.6	0.000	0.0001	0.0002	0.000	0.000	
sub8	1	0.001(0.0)	0.563	150.5	0.8	39.48	244.1	0.000	0.0002	0.0002	0.000	0.000	
sub1	1	0.000(0.0)	0.245	18648.4	0.0	98.46	256.0	0.000	0.0000	0.0000	0.000	0.000	
sub9	1	0.000(0.0)	0.004	175.4	70.7	58.62	36.0	0.000	0.0000	0.0000	0.000	0.000	
total	1838	28.267(100.0)	15.379	54268.1	31160.9	99.62	250.5	28.245	0.0020	0.0094	0.098	2.773	

簡易性能解析機能 : ftrace (2)

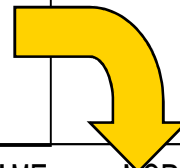
ユーザ指定リージョン

- プログラムの局所的な部分の性能を知りたい場合に使用する

```
PROGRAM MAIN
PRINT*, "TEST"
CALL INIT
CALL FTRACE_REGION_BEGIN("U_REGION")
CALL SUB
CALL SUB
CALL FTRACE_REGION_END("U_REGION")
END
```

以下の手続きの実行で挟まれる
範囲を測定可能

```
CHARACTER *(*) NAME
FTRACE_REGION_BEGIN(NAME)
FTRACE_REGION_END(NAME)
```



PROG.UNIT	FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	...
sub	2	1.539(99.9)	769.251	31597.3	20799.5	99.83	250.0	
init	1	0.001(0.1)	0.868	13982.2	0.0	98.84	256.0	
main	1	0.000(0.0)	0.160	272.9	0.2	76.53	250.9	

total	4	1.540(100.0)	384.882	31584.1	20785.6	99.83	250.0	
U_REGION	1	1.539(99.9)	1538.506	31597.2	20799.4	99.83	250.0	

簡易性能解析機能 : ftrace (3)

使用方法

```
% sxf90 -ftrace test.f90  
% a.out  
% ftrace
```

or

```
% sxf90 -ftrace test.f90  
% setenv F_FTRACE YES  
% a.out
```

実行後、カレントディレクトリにftrace.out(解析情報ファイル)が作成される

注意事項

-ftrace指定でコンパイルされた手続から -ftrace指定なしでコンパイルされた手続を呼び出す場合、呼び出し回数以外の測定値は呼び出し先の手続の性能情報を含んだ値となる

例: sub1.f

```
SUBROUTINE SUB1  
CALL SUB2(X,Y,Z)  
END
```

sub2.f

```
SUBROUTINE SUB2(X,Y,Z)  
X=SQRT(Y**2+Z**2)  
END
```

sub1.f のみを -ftraceでコンパイルした場合、ftraceリストにはsub2の情報は表示されず、sub1の性能情報にsub2の情報を含んだ値が表示される。

目次

FORTRAN90 / SXの自動ベクトル化機能

- ベクトル化とは？
- 自動ベクトル化の条件
- 拡張ベクトル化機能
- 編集リストと変形リスト

性能チューニング

- ベクトル化による高速化
- 性能チューニングの手順
- 性能の分析
- チューニングの実施

ベクトル化における注意事項

性能チューニング

■ チューニングの実施方法

- コンパイラオプション
 - 最適化レベルの指定
 - インライン展開
- 指示行
 - ベクトル化に関連する指示行
 - メモリアクセスに関連する指示行
- ソースコード修正

コンパイラオプション(1)

最適化レベルの指定

- -Chopt
 - 最大限の最適化処理およびベクトル化処理を行うことを指定する。最適化処理およびベクトル化処理に副作用を伴う場合があるので、注意が必要。
- -Cvopt(既定値)
 - 最大限の最適化処理を行い、規定レベルのベクトル化処理を行うことを指定する。
- -Cvsafe
 - 最適化処理およびベクトル化処理を行うが、副作用を伴う可能性のある機能は抑止することを指定する。
- -Csopt
 - 最大限の最適化処理を行い、ベクトル化処理を抑止することを指定する。
- -Cssafe
 - ベクトル化処理を抑止し、最適化処理も副作用を伴う機能は抑止することを指定する。
- -Cdebug
 - 原始プログラムのデバッグを行うことを指定する。

コンパイラオプション(2)

手続きのインライン展開による改善

```
DO I=1, N  
  A(I)=FUN(B(I),C(I))+D(I)  
ENDDO
```

```
FUNCTION FUN(X, Y)  
  FUN = SQRT(X) * Y  
END FUNCTION FUN
```

f90: vec(3): test.f, line 3: ベクトル化できないループである。
f90: opt(1025): test.f, line 4: 最適化を阻害する関数呼出しがある。
f90: vec(10): test.f, line 4: ループまたは配列式全体をベクトル化不可とする手続funが指定された



-pi 指定時のコンパイラの変形イメージ

コンパイル時オプション -pi を指定することにより、FUNがインライン展開され
上記ループはベクトル化される

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

f90: vec(1): test.f, line 3: ループ全体をベクトル化する。
f90: vec(24): test.f, line 3: ループの繰り返し数を最大5000と仮定する。
f90: opt(1222): test.f, line 4: 手続呼び出しをインライン展開した。

指示行

ベクトル化指示行

!CDIR オプション [, オプション]

!CDIRR オプション [, オプション]

!CDIR [BEGIN | CONT | END] オプション [, オプション]

- !CDIR は1 ~ 5桁目、!CDIRRは1 ~ 6桁目に書いた時だけ有効
(それ以外 はコメントとみなされる)
- !CDIRは直後にのみ、!CDIRRはそこから後ろ全てに、
BEGIN ~ END はその区間で有効となる

- | | |
|-----------------------|---------------------|
| ● NODEP | : 参照する配列要素に重なりがない |
| ● LOOPCHG / NOLOOPCHG | : ループ入れ換えを行う / 行わない |
| ● outerunroll | : アウターアンローリングを行う |
| ● ON_ADB | : ADBを利用する |
| ● VECTOR / NOVECTOR | : ベクトル化の対象とする / しない |

指示行の挿入によるベクトル化

NODEP指示行

ソースプログラム

```
DO I=1, N  
  A(IX(I)) = A(IX(I)) + B(I)  
ENDDO
```

A(IX(I))の依存関係不明でベクトル化されない。
-Wf, -pvctl listvec を指定することにより、ベクトル化を行うこともできるが...

-Wf, -pvctl fullmsg を指定することにより、以下のメッセージが出力される

```
f90: vec(1): test.f, line 5: ループ全体をベクトル化する。  
f90: vec(24): test.f, line 5: ループの繰り返し数を最大5000と仮定する。  
f90: opt(1036): test.f, line 6: 異なる繰り返しで定義された値を参照している可能性  
      がある。(nosync/nodepを指定すれば最適化を行う)  
f90: vec(26): test.f, line 6: List Vectorのマクロ演算としてベクトル化を行う。
```



!CDIR NODEP

```
DO I=1, N  
  A(IX(I))=A(IX(I))+B(I)  
ENDDO
```

配列IX(I)の値に、重複した要素のないことがわかっているならば指示行NODEPを挿入することで高速にベクトル化できる
(例: IX(I)が、9,3,2,4,1,5,7,10,8,...)

指示行の挿入によるループ長の拡大

ソースプログラム

```
DO I=1,N
  DO J=1,M
    A(I,J)=B(I,J)+C(I,J)
  ENDDO
ENDDO
```

DO I=1,Nを内側にした方が、メモリのアクセスが連続するため、コンパイラがループを入れ換える



指示行NOLOOPCHGを指定してループの入れ換えを抑止

```
DO I=1,N
  !CDIR NOLOOPCHG
  DO J=1,M
    A(I,J)=B(I,J)+ C(I,J)
  ENDDO
ENDDO
```

例えば M= 10000, N= 10 だとわかっているならば、ループを入れ換えないほうがベクトル演算の効率が高い

改善例の参考値 ユーザ時間 (秒) 0.087

ユーザ時間 (秒) 0.008
(約11倍の改善)

-Wf, -pvctl fullmsg を指定することにより、ループの入れ換えが行われたことをメッセージで表示

f90: opt(1589): test.f, line 7: 外側ループを内側ループと入れ換えた。

outerunroll指示行

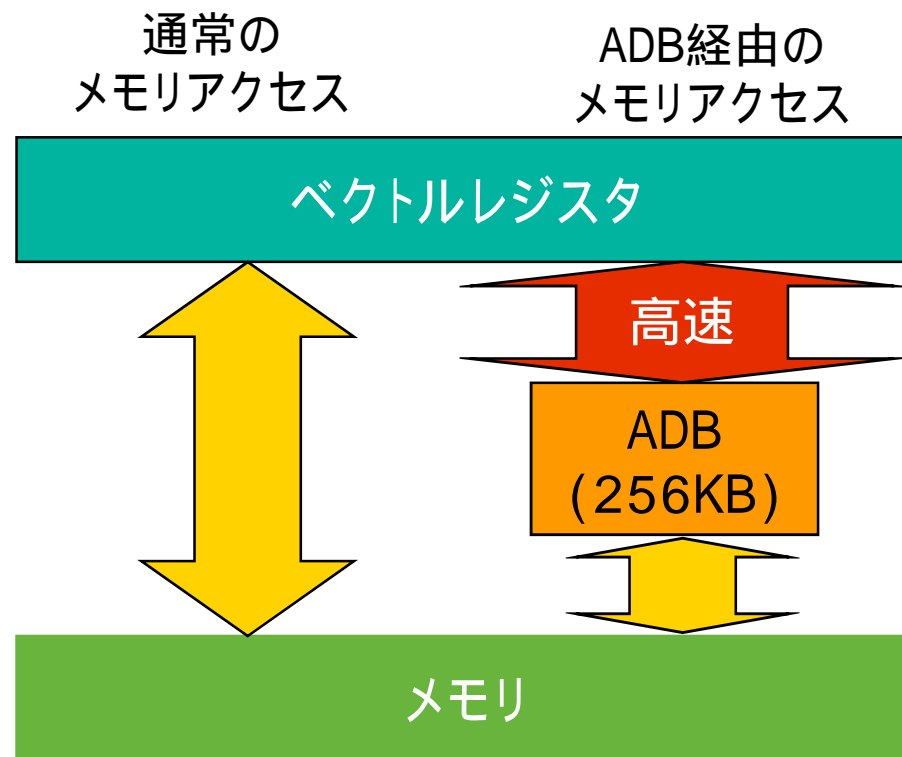
4 段outerunroll指示行の挿入例

- 配列aのメモリアクセスが1/4になるため、高速化が可能になる
- 段数は2のべき乗の値のみ有効

```
24      do j=1,n
25 !cdir outerunroll=4
26      do k=1,n
27          do i=1,n
28              a(i,j)=a(i,j)+b(i,k)*c(k,j)
29          enddo
30      enddo
.      do k = 1, 2048, 4
. !cdir  nodep
. !cdir  on_adb(a,b)
.      do i = 1, 2048
.          a(i,j) = a(i,j) + b(i,k)*c(k,j) + b(i,k+1)*c(k+1,j) + b(i,k+
.          1 2)*c(k+2,j) + b(i,k+3)*c(k+3,j)
.      enddo
.      enddo
31      enddo
```

ADB (Assignable Data Buffer)

- ベクトルデータバッファリング機能により、メモリアクセスを高速化
- コンパイラの既定値で利用可能



ADB指示行(1)

ON_ADB[(識別子[, 識別子]...)]

- 直後のループ中のすべての配列のベクトルロード、ストアにおいて配列をADBにバッファリングする
- 識別子が指定されたとき、識別子で指定された配列のベクトルロード、ストアにおいて配列をADBにバッファリングする。
- 直後のループに対してのみ有効(直後のループの内側ループに対しては無効)
- ADBの容量より指定した配列の大きさの方が大きい場合、後からバッファリングされたデータにより上書きされる。
- 指示行を利用した場合、コンパイラによる自動ADB機能は無効となる

ADB指示行(2)

```
REAL * 8 A(N),B(N),C(N)
```

```
...
```

```
!CDIR ON_ADB(A)
```

```
!CDIR ON_ADB(C)
```

```
DO I=1,N
```

```
... = A(I) + B(I)
```

```
C(I) = ...
```

```
ENDDO
```

```
...
```

```
!CDIR ON_ADB(A)
```

```
!CDIR ON_ADB(C)
```

```
DO I=1,N
```

```
...
```

```
... = A(I) + C(I)
```

```
...
```

```
ENDDO
```

配列AはメモリからADB経由でロード
(1回目のロード)
配列CをADB経由でストア

指示行がないとメモリからのロードになってしまう

配列AはADBからロード(2回目のロード)
配列CはADBからのロード

ADB指示行(3)

```
.      do j = 1, 2048, 4
.      !cdir  nodep
.      !cdir  on_adb(a,b)
.          do i = 1, 2048
.              a(i,j) = 0.0000000000000000e+000
.              a(i,j+1) = 0.0000000000000000e+000
.              a(i,j+2) = 0.0000000000000000e+000
.              a(i,j+3) = 0.0000000000000000e+000
.              b(i,j) = dfloat(2049 - max(i,j))
.              b(i,j+1) = dfloat(2049 - max(i,1 + j))
.              b(i,j+2) = dfloat(2049 - max(i,2 + j))
.              b(i,j+3) = dfloat(2049 - max(i,3 + j))
.              c(i,j) = float(2049 - max(i,j))
.              c(i,j+1) = float(2049 - max(i,1 + j))
.              c(i,j+
.              c(i,j+
.          enddo
.      enddo
```

ADBが利用された場合の編集リスト(Aの表示)

```
24: +----->          do j=1,n
25: |                !cdir nounroll
26: |+----->          do k=1,n
27: ||V- - ->          do i=1,n
28: |||                a(i,j)=a(i,j)+b(i,k)*c(k,j)
29: ||V- - -          enddo
30: |+-----          enddo
31: +-----          enddo
```

LINE LEVEL(NO.): DIAGNOSTIC MESSAGE
27 vec (29): ADB is used for array.: a
27 vec (29): ADB is used for array.: b

ベクトル化抑止による高速化

VECTOR / NOVECTOR指示行

- 直後の配列式 またはDOループを自動ベクトル化の対象とする (VECTOR) か、対象としない (NOVECTOR) ことを指定する

```
M=MIN(N,2)
!CDIR NOVECTOR
DO I=1,M
    A(I) = B(I) * C(I) + D(I) * E(I)
ENDDO
```

例えば「M は、1または2にしかかなり得ない」ことを利用者が知っている場合、NOVECTORを指定して、ベクトル化を抑止したほうが効率がよい

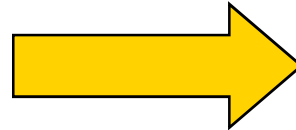
- コンパイル時にループ長が5以下の場合には、コンパイラはベクトル化を行わない

ソースプログラムの変更によるベクトル化率の向上(1)

ベクトル化できないループ

```
DO I=1, N
  IF(X(I).LT.S) THEN
    T = X(I)
  ELSE IF(X(I).GE.S)
  THEN
    T = -X(I)
  ENDIF
  Y(I) = T
ENDDO
```

Tが参照前に必ず定義されるように変形する



ベクトル化可能なループ

```
DO I=1, N
  IF(X(I).LT.S) THEN
    T = X(I)
  ELSE ! IF(X(I).GE.S) THEN
    T = -X(I)
  ENDIF
  Y(I) = T
ENDDO
```

Tは定義されないかもしれない

Tは必ず定義される

-Wf, -pvctl fullmsg を指定することにより、以下のメッセージが出力される

```
f90: vec(3): test.f, line 3: ベクトル化できないループである。
f90: vec(13): test.f, line 3: ループ分割によるオーバーヘッドが大きすぎる。
f90: opt(1019): test.f, line 5: スカラ変数が異なる繰り返して定義した値を参照している。
f90: vec(21): test.f line 5: ベクトル化不可の依存関係がある。
f90: vec(21): test.f line 7: ベクトル化不可の依存関係がある。
```

ソースプログラムの変更によるベクトル化率の向上(2)

ソースプログラム

```
DO I=1, N
  IF(A(I).GT.0.0) THEN
    S = S + B(I)
  ELSE
    S = S + C(I)
  END IF
ENDDO
```



```
DO I=1, N
  IF(A(I).GT.0.0) THEN
    T = B(I)
  ELSE
    T = C(I)
  END IF
  S = S + T
ENDDO
```

S は繰り返し間にまたがって定義・参照されるため、ベクトル化できない

ソースを書き換えることにより
総和型のマクロ演算に適合するので
ベクトル化される

ソースプログラムの変更によるベクトル化率の向上(3)

ソースプログラム

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

$X(I) = X(I-1) \dots$ にベクトル化を阻害する依存関係があるため、ベクトル化できない



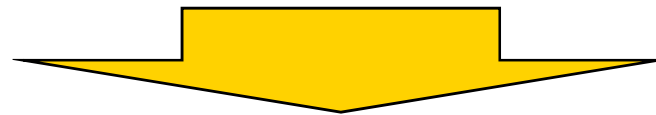
```
DO I=2, N  
  X(I)=X(I-1)*A(I)+Y(I)*A(I)+B(I)  
ENDDO
```

ソースを変形すると、漸化式型のマクロ演算
 $X(I) = \text{式} \pm X(I-1) * \text{式}$
に適合するのでベクトル化できる

メモリアクセス性能の改善(1)

配列の定義・参照は連続した要素のアクセスがもっとも速い。

- 2^n 飛び要素でアクセスすると性能低下
(CPUポート競合、メモリネットワーク競合時間増加)
- リストアクセス($X(I \times I)$ のような参照)は線形アクセス($X(2 * I + 1)$ のような参照)より遅い
(メモリネットワーク競合時間増加)



- ループを入れ換えて、配列要素のアクセスが連続するように変更
- 配列宣言の一次元目が奇数となるように変更(2^n の倍数を避ける)
- リストアクセスを使わない計算方法に変更

といった方法で性能を改善

メモリアクセス性能の改善(2)

ループを入れ換えて、配列要素が連続アクセスとなるように変更

```
DIMENSION
A(1024,1023),B(1024,1023)
CALL SUB(A, B, 1024, 1023)
END
SUBROUTINE SUB(A, B, M, N)
DIMENSION A(M,N), B(N,M)
DO J=1,1000
!CDIR NOLOOPCHG
DO I=1,1000   ベクトル化
  A(J, I) = B(J, I)
ENDDO
ENDDO
```



```
DIMENSION
A(1024,1023),B(1023,1024)
CALL SUB(A, B, 1024, 1023)
END
SUBROUTINE SUB(A, B, M, N)
DIMENSION A(M,N), B(N,M)
!CDIR NOLOOPCHG
DO J=1,1000   ベクトル化
  DO I=1,1000
    A(J, I) = B(J, I)
  ENDDO
ENDDO
```

Iでベクトル化すると、配列A,Bのアクセスが1024要素飛びになり、メモリアクセス性能が低下

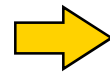
Jでベクトル化すると、配列A,Bは連続要素のアクセスとなる。

改善例の参考値	1024要素飛び	:
ユーザ時間	0.54msec	
	連続要素アクセス:	
	0.02msec	

メモリアクセス性能の改善(3)

どうしても飛びアクセスとなる場合、飛びが 2^n の倍数にならないように配列宣言を変更

```
REAL A(1024,1000)
DO I = 1, 1000
  A(J, I) = B(J, I)
ENDDO
```



```
REAL A(1025,1000)
DO I = 1, 1000
  A(J, I) = B(J, I)
ENDDO
```

配列A, Bのアクセスが1024要素飛びになり、性能低下

配列A, Bのアクセスが1025要素飛びになり、性能が改善される

改善例の参考値

ユーザ時間 : 59msec

ユーザ時間 : 4msec

目次

FORTTRAN90 / SXの自動ベクトル化機能

- ベクトル化とは？
- 自動ベクトル化の条件
- 拡張ベクトル化機能
- 編集リストと変形リスト

性能チューニング

- ベクトル化による高速化
- 性能チューニングの手順
- 性能の分析
- チューニングの実施

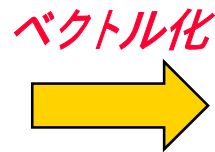
ベクトル化における注意事項

自動ベクトル化における注意事項

総和型演算は、ベクトル化によって演算順序が変わるため、桁落ちの誤差が発生するデータの場合には、演算誤差が生じることがある

ソースプログラム

```
S=0.0  
DO I=1,1024  
  S=S+A(I)  
ENDDO
```



```
VS(1:256) = 0.0  
DO I=1,1024,256  
  VS(1:256)=VS(1:256)+A(I:I+255)  
ENDDO  
S = S + vsum(VS(1:256))
```

vsum はベクトルレジスタ中のデータの合計を計算する命令ベクトル加算命令より遅い

ベクトル化した総和の演算順序

```
VS(1) = A(1)+A(257)+A(513)+A(769)  
VS(2) = A(2)+A(258)+A(514)+A(770)  
      :  
VS(256)=A(256)+A(512)+A(768)+A(1024)
```

備考:

総和型だけでなく、累積($S=S+A(I)$)や漸化式($A(I+1)=A(I)+B(I)$)でも同様の演算誤差が発生することがある