H28年度
SX-ACE 高速化技法の基礎
（自動ベクトル化）

2016年　9月8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

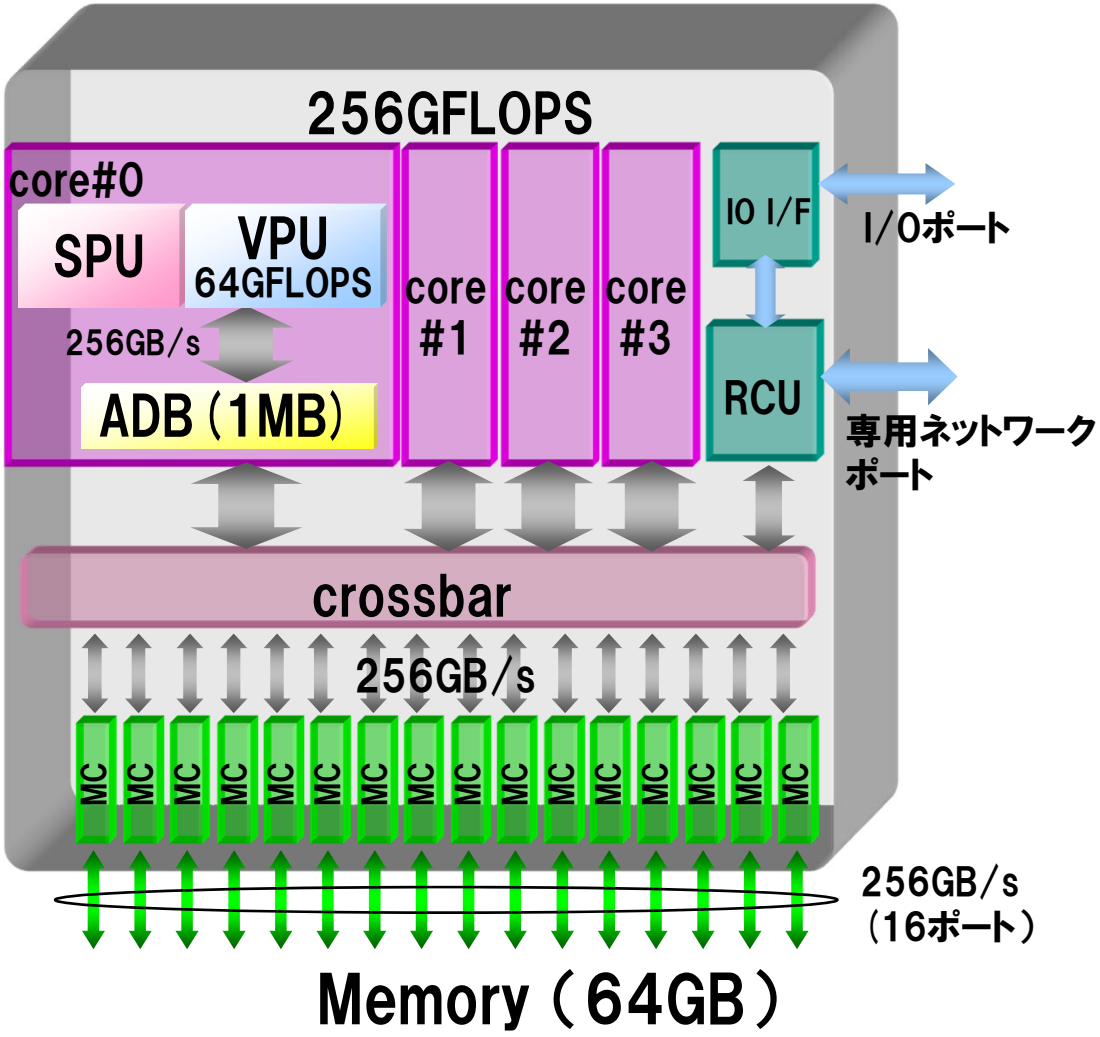
本資料は，東北大学サイバーサイエンスセンターとNECの共同により作成され，大阪大学サイバーメディアセンターの環境で実行確認を行い，修正を加えたものです．

はじめに

SX-ACEのご紹介

SX-ACEのご紹介(CPU構成)

- 演算性能　256GFLOPS（64GFLOPS/Core　×4Core）
- メモリバンド幅 256GB/s　（16GB/s /ポート　×16ポート）



Core	
演算性能	64GFlops
ADBサイズ	1MB
ADB帯域	256GB/s
CPU	
Core数	4
演算性能	256GFlops
メモリ帯域	256GB/s
Byte/Flop	1

SPU: Scalar Processing Unit
VPU: Vector Processing Unit
ADB: Assignable Data Buffer
RCU: Remote Access Control Unit
MC: Memory Controller

SX-ACEの強化ポイント（対SX-9）

Coreピーク性能比（0.6x）以上の実効性能向上を実現する、新ベクトルアーキテクチャ

ベクトル性能強化

短ベクトル性能強化
- ■SPUベクトルパイプライン新設
- ■演算器間のダイレクトバイパスチェイニング

リストベクトル性能強化
- ■メモリレイテンシ短縮
- ■命令追い越し実行機能強化

スカラ性能強化

- パイプライン構成見直し
- L1キャッシュ容量拡張（2倍化）
- 分岐予測機能強化
- 命令発行機能強化
- ハードウェアプリフェッチ機能

メモリ性能強化

- ADB容量拡張（1MB/Core）
- MSHRによる冗長なロードメモリアクセス数削減
- ストア圧縮による冗長なストアメモリアクセス数削減

目次

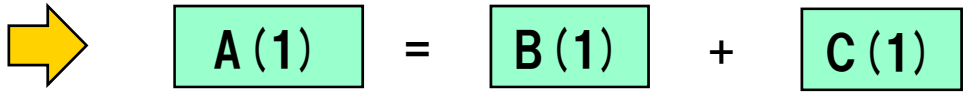
目次

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

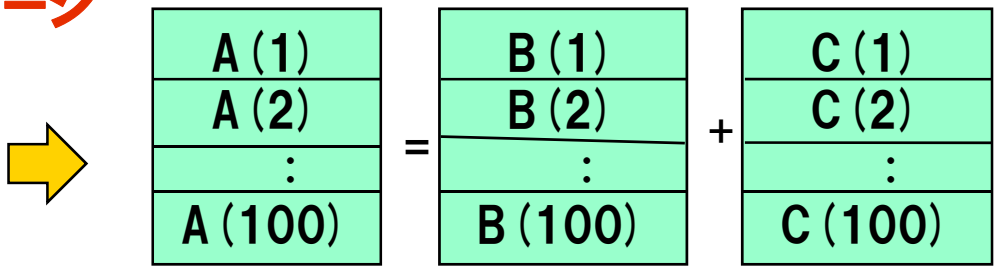
スカラ命令のイメージ

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



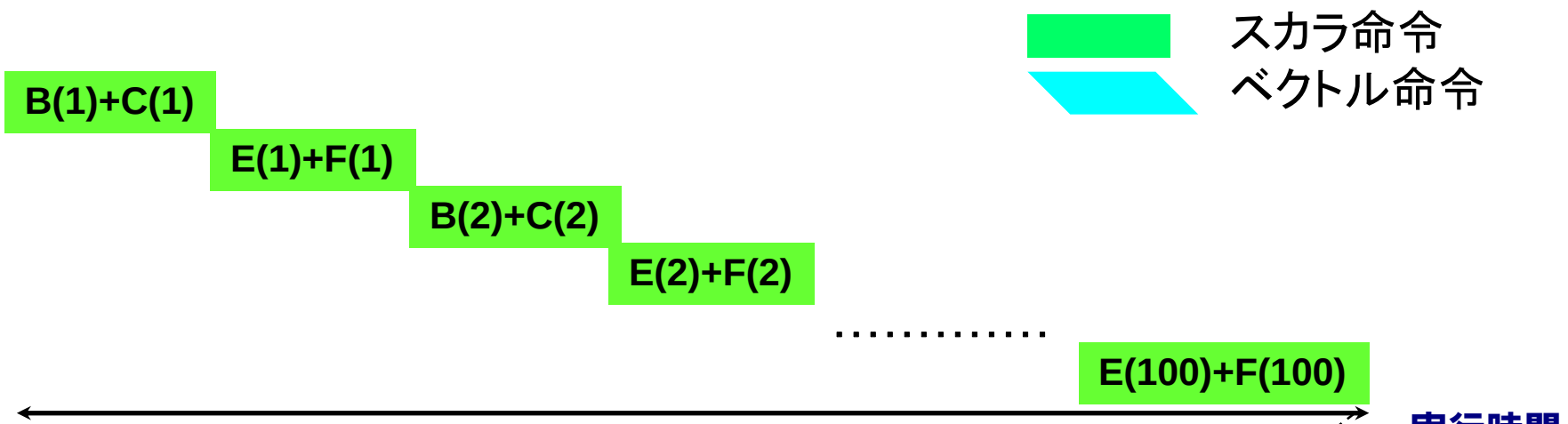
ベクトル命令の適用のイメージ

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

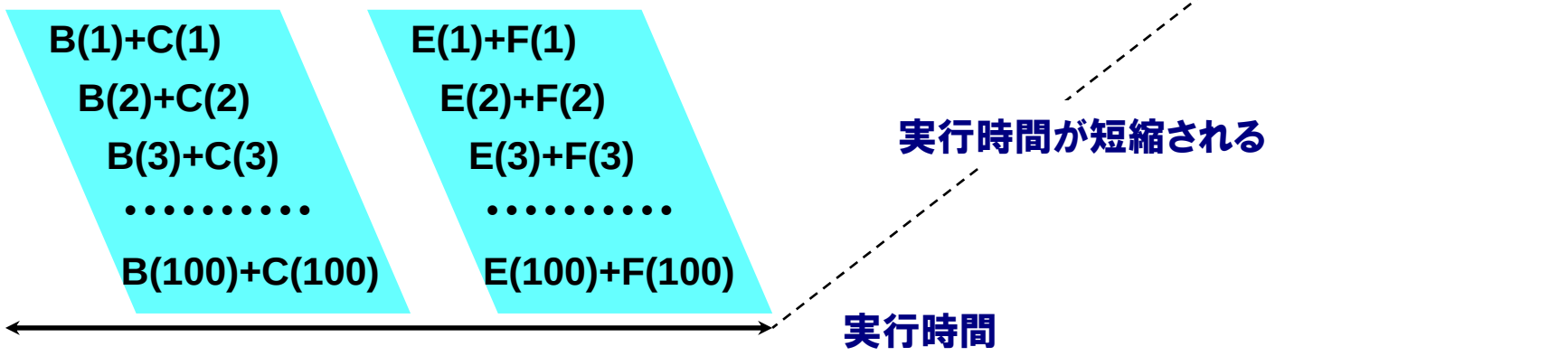


スカラ命令とベクトル命令

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
```

ベクトル化

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

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)
```

スカラ命令

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)
```

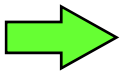
```
D(1)   =  E(1)   + F(1)
D(2)   =  E(2)   + F(2)
   :      :      :    :
D(100) =  E(100) + F(100)
```

ベクトル命令

配列式のベクトル化

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) )
```

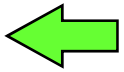


コンパイラによる変形イメージ1

```
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
```



配列式は、内部的にはDOループと同様に変形してから、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

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)
:

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:更新されたAの値　　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)
:

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:更新されたAの値　　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)
```

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

例4

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

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
高速化されないので、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

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

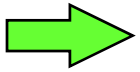
- 文の入れ換え
- ループの一重化
- ループの入れ換え
- 部分ベクトル化
- 条件ベクトル化
- マクロ演算の認識
- 多重ループのベクトル化
- ループ融合
- インライン展開

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
```

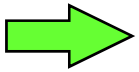
そのままベクトル化すると、B(1)～B(99)の値が全て2.0になってしまう。
このような場合に、ループ内の文の順序を入れ換えることにより、
ベクトル化できるように変形する

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
```

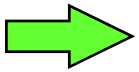
ループ長（ベクトル長）がより長くなるように、多重ループを一重化することにより、ベクトル化の効率をさらに高める

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
```

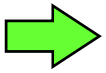
DO I=1,Nでベクトル化しようとすると、配列Aに依存関係がありベクトル化できない。
ループを入れ換えると、DO J=1,Mのループに関しては依存関係がなくなりベクトル化できる。

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
```

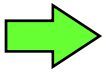
配列や変数の定義・引用関係に矛盾があり、本来はベクトル化できない場合でも、コンパイラが特別なパターンであることを認識し、特別なベクトル命令を用いることで、ベクトル化を行う。

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

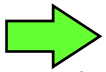
```
DO  I=1, N
   DO  J=1, N
      A(I, J)  =  0.0
   ENDDO
   B(I)=1.0
ENDDO
```



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
```

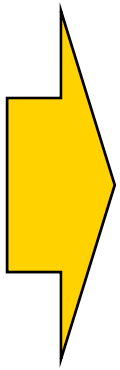
ベクトル化は、基本的には最内側ループをベクトル化するが、
外側のループを二つに分割することによって、外側のループに含まれる文も
ベクトル化を行う

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(１０)

配列式のループ融合

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
```

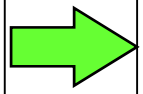
コンパイラは、同じ形状の配列式、ループ構造が連続していれば融合するが、
間に形状の異なる配列式やループ構造、他の文があると融合できない。
高速化のためには、なるべく同じ形状の配列式、ループ構造を連続させるほうがよい。

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

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

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

インライン展開された手続がある行には “I” 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

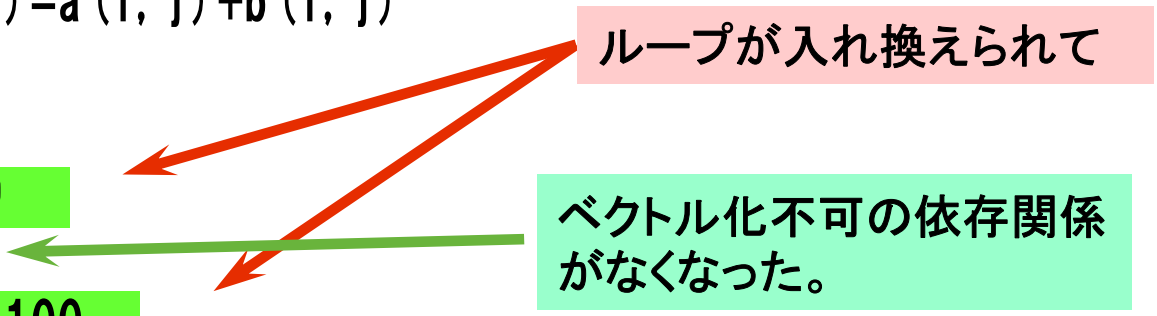
◆ループの入れ換えが行われた場合

```
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
```



目次

ベクトル化による高速化（３つのポイント）

ループ長

- ベクトル化対象の平均ループ長は、可能な限り２５６に近付けているか？

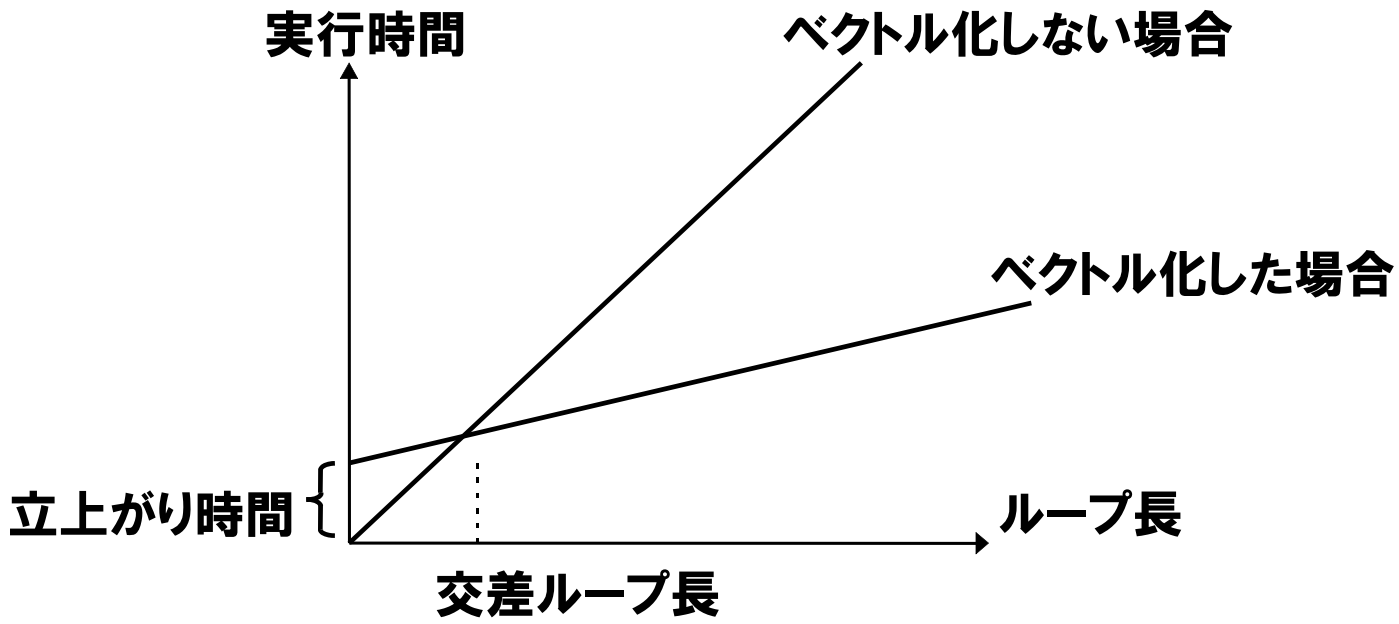
ベクトル化率

- 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？

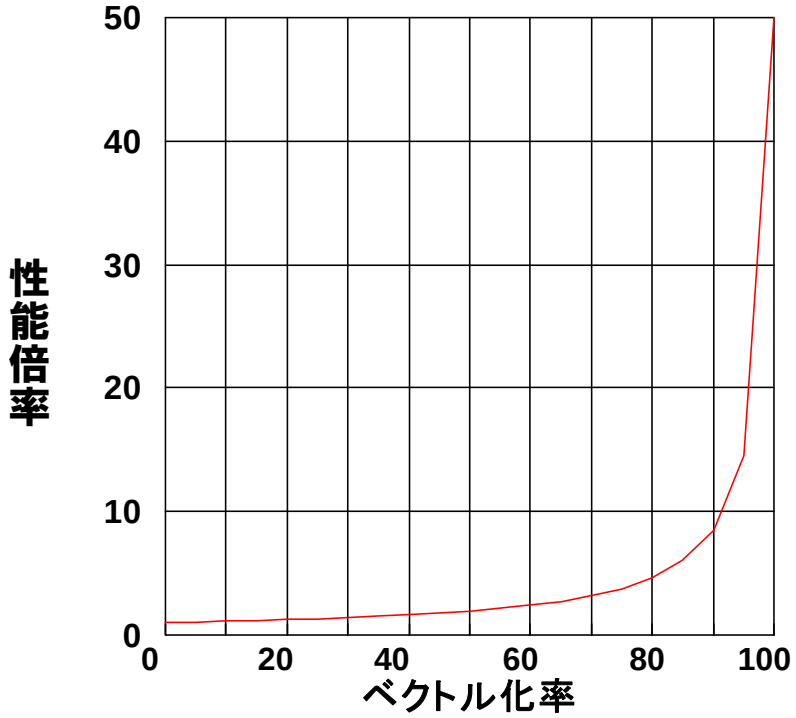
ループ長(ベクトル長)



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。
(交差ループ長 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)

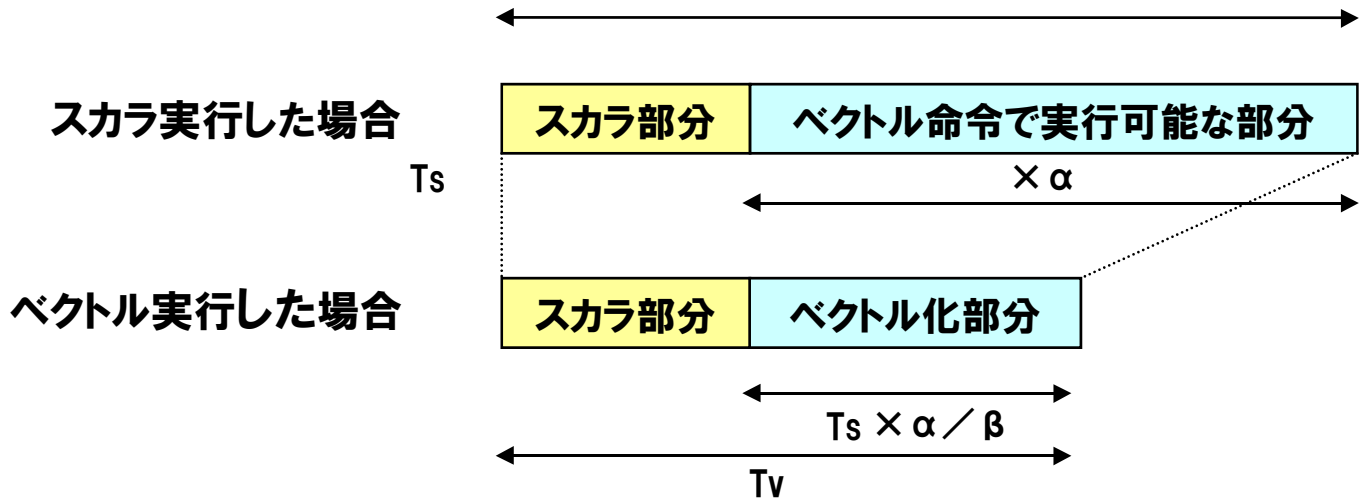
ベクトル化率が十分に高くなって初めて効果発揮



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
にしかならない。

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

β：ベクトル命令の性能向上比

Ts　:スカラ実行したときの実行時間

Tv :ベクトル実行したときの実行時間

性能向上比＝Ts／Tv＝β／　　（α＊(1－β)＋β）

ベクトル演算率＝　　(Ts＊α／β)／Tv

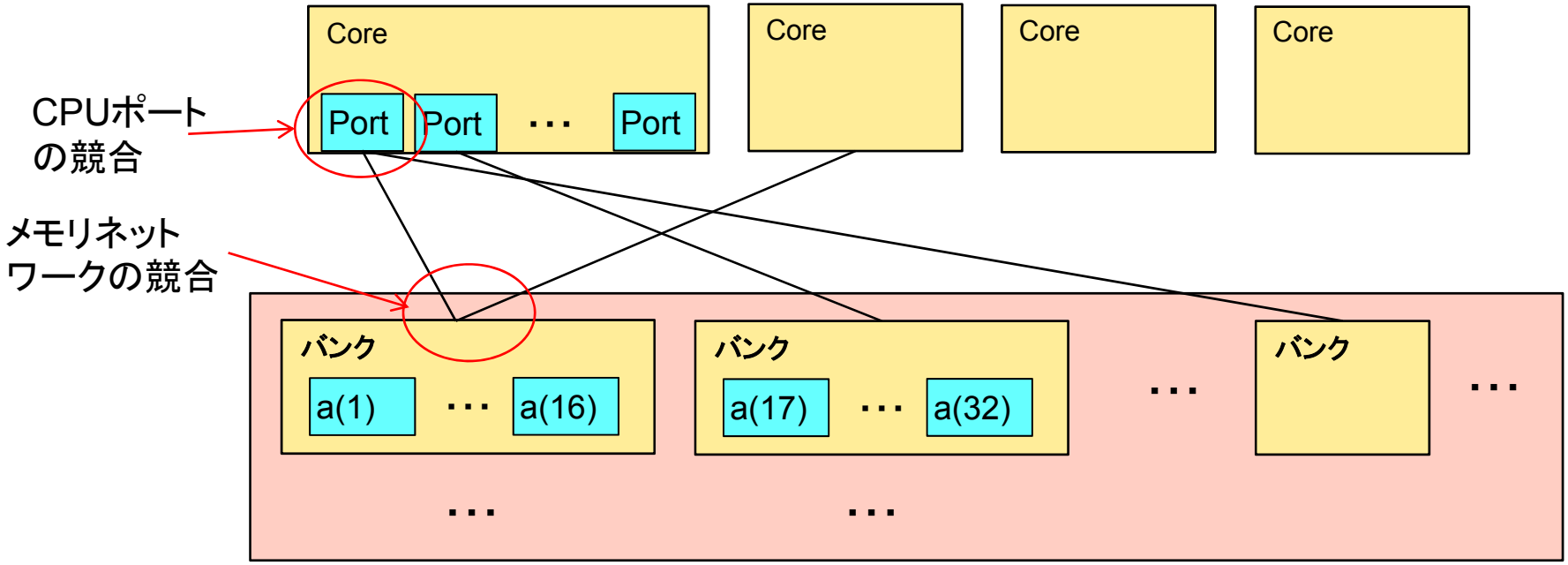
＝　α／（α＊(1－β)＋β）

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、
プログラム実行解析情報(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-ACEのメモリは高速化のため128個のバンクから構成される。

- 別々のバンクに対して並列にロード・ストアが可能
- Coreあたり16個のCPUポートを持ち，倍精度浮動小数点型の場合1ポートあたり16個の配列要素を処理



a(i):倍精度浮動小数点型の配列とすると，ひとつのバンクに16要素ずつ格納

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

(a) CPUポート競合

Core内の同一ポートにロード・ストアの要求が集中したときに発生する

（例）

```
real a(256,10000),b(256,10000)
          :
do i = 1,10000
   a(1, i) = b(1, i)
endo do
```

配列a,bの1次元目の要素数が256である．
配列の1次元目の添字式がループ内で不変であり，2次元目の添字式がループの繰り返しにしたがって1ずつ増加しているので，要素間距離256の等間隔ベクトルとなり，CPUポート競合が発生する．

(b) メモリネットワーク競合

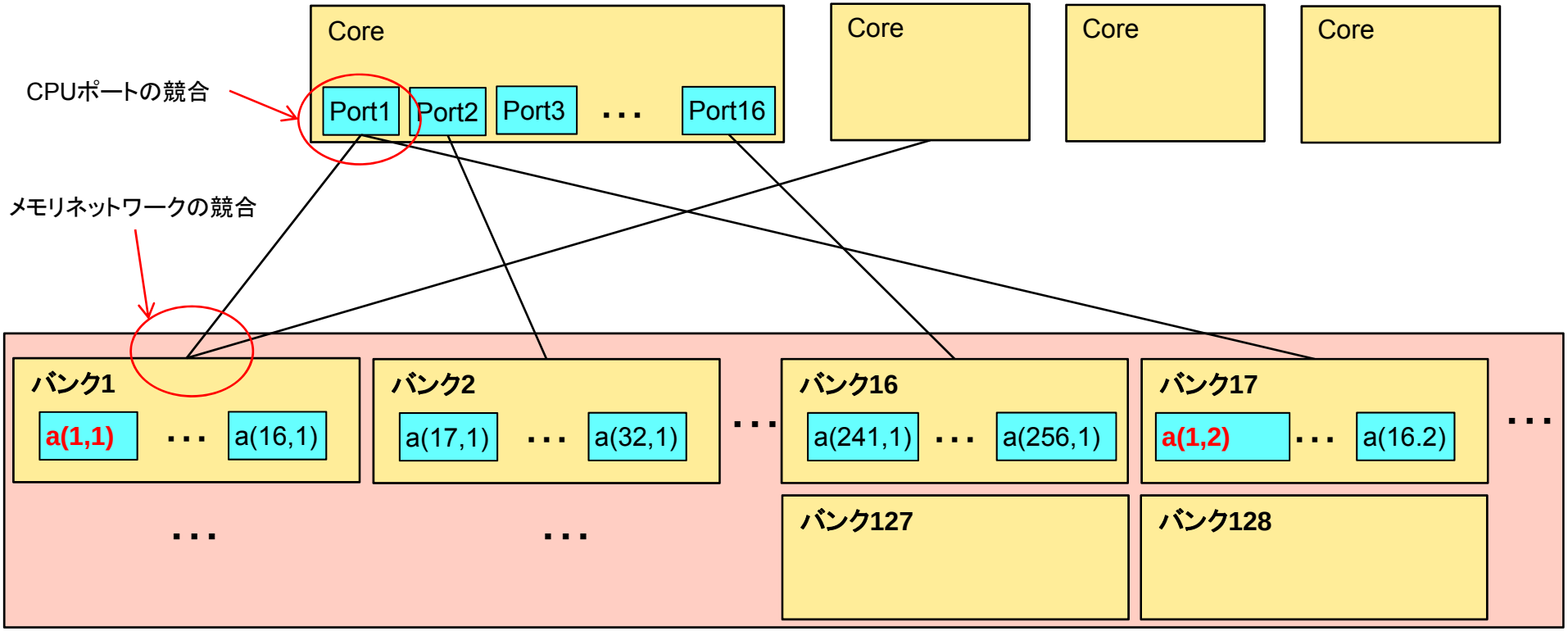
以下のようなケースで発生する

- CPUポート競合が発生しているケース
- 自動並列，OpenMP並列化されたプログラムの複数のタスクから同時に同一バンクにアクセスしたケース．同一の配列要素，スカラ変数を複数のタスクで参照しているときなどに発生する
- 別のプロセスが，たまたま同一のバンクに繰返しアクセスしていたケース

メモリアクセスの効率化（佐藤の補足）

SX-ACEのメモリは高速化のため128個のバンクから構成される。

- 別々のバンクに対して並列にロード・ストアが可能
- Coreあたり16個のCPUポートを持ち，倍精度浮動小数点型の場合1ポートあたり16個の配列要素を処理



a(i):倍精度浮動小数点型の配列とすると，ひとつのバンクに16要素ずつ格納

目次

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

PROGINF情報の利用

setenv F_PROGINF YES または DETAILで表示

- SX-ACEでは、既定値で設定されている(DETAILの表示例)

```
 ******  Program Information  ******
Real Time (sec)                :          0.429530
User Time (sec)                :          0.428730
Sys  Time (sec)                :          0.000722
Vector Time (sec)              :          0.428555
Inst. Count                    :        240708300.
V. Inst. Count                 :        117679817.
V. Element Count               :      30126031456.
V. Load Element Count          :      10741743662.
FLOP Count                     :      17179869321.
MOPS                           :      70555.034495
MFLOPS                         :      40071.535281
A. V. Length                   :        255.999986
V. Op. Ratio (%)               :         99.593282
Memory Size (MB)               :        256.031250
MIPS                           :        561.444965
I-Cache (sec)                  :          0.000040
O-Cache (sec)                  :          0.000061
Bank Conflict Time
  CPU Port Conf. (sec)         :          0.000016
  Memory Network Conf. (sec)   :          0.109435
ADB Hit Element Ratio (%)      :         19.954172
```

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
   ①      ②        ③        ④          ⑤           ⑥
%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_
    :       :       :        :            :           :
```

①：全体に占める手続毎のCPU時間の割合（％）
②：手続毎のCPU時間（秒）
③：先頭からの合計のCPU時間（秒）
④：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）
⑤：一回の呼出し毎のCPU時間（〃）
⑥：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）

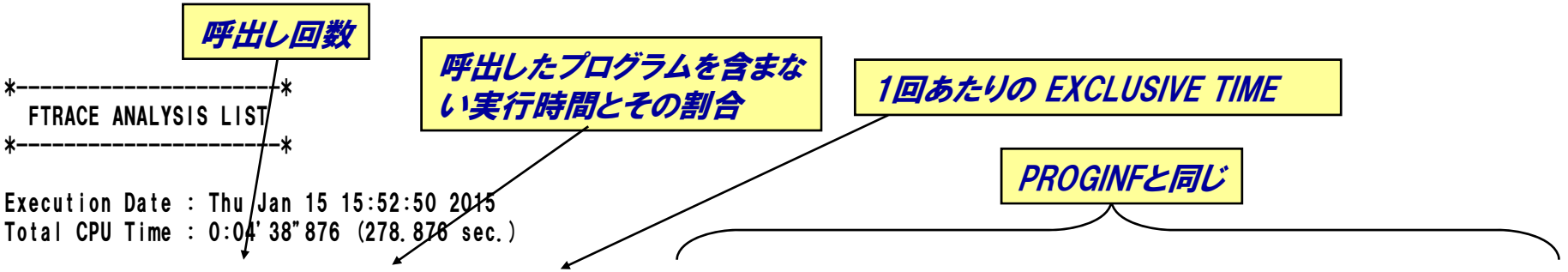
⇨ ベクトル化されていなければベクトル化の促進を検討

（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




```
*----------------------*
  FTRACE ANALYSIS LIST
*----------------------*

Execution Date : Thu Jan 15 15:52:50 2015
Total CPU Time : 0:04'38"876 (278.876 sec.)
```

PROC.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 CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%
sub5	15792	271.375 (97.3)	17.184	843.1	2.3	18.77	136.0	4.106	7.603	27.078	0.000	0.000	0.00
sub3	1616	2.779 (1.0)	1.719	842.7	2.3	18.77	136.0	0.042	0.078	0.278	0.000	0.000	0.00
sub9	1178190	2.184 (0.8)	0.002	11087.6	0.0	97.32	100.0	1.737	0.000	0.000	0.972	0.001	99.85
sub10	1178190	2.004 (0.7)	0.002	29645.8	11756.1	99.14	250.0	1.282	0.000	0.000	0.919	0.000	99.86
main_	1	0.380 (0.1)	380.484	281.8	0.0	0.00	0.0	0.000	0.001	0.004	0.000	0.000	0.00
sub2	1235	0.151 (0.1)	0.122	16496.3	0.0	99.06	250.0	0.151	0.000	0.000	0.059	0.062	4.33
sub1	1235	0.001 (0.0)	0.001	3950.2	0.0	96.43	250.0	0.000	0.000	0.000	0.000	0.000	0.00
sub7	1616	0.001 (0.0)	0.000	129.6	0.0	40.00	10.0	0.000	0.000	0.000	0.000	0.000	88.71
sub4	16	0.000 (0.0)	0.018	799.9	2.2	18.74	136.0	0.000	0.000	0.000	0.000	0.000	0.00
sub8	16	0.000 (0.0)	0.013	1954.5	0.0	79.84	10.0	0.000	0.000	0.000	0.000	0.000	0.64
sub6	16	0.000 (0.0)	0.000	10685.6	4211.1	98.52	250.0	0.000	0.000	0.000	0.000	0.000	4.80
total	2377923	278.876 (100.0)	0.117	1138.1	86.8	40.44	160.4	7.319	7.683	27.360	1.950	0.064	95.09

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        AVER.TIME     MOPS   MFLOPS V.OP   AVER.   …
                        TIME[sec](  %  )    [msec]                    RATIO 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
ジョブの実行
% sxftrace
```

or

```
% sxf90 -ftrace test.f90
環境変数の指定
( setenv F_FTRACE YES)
ジョブの実行
```

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

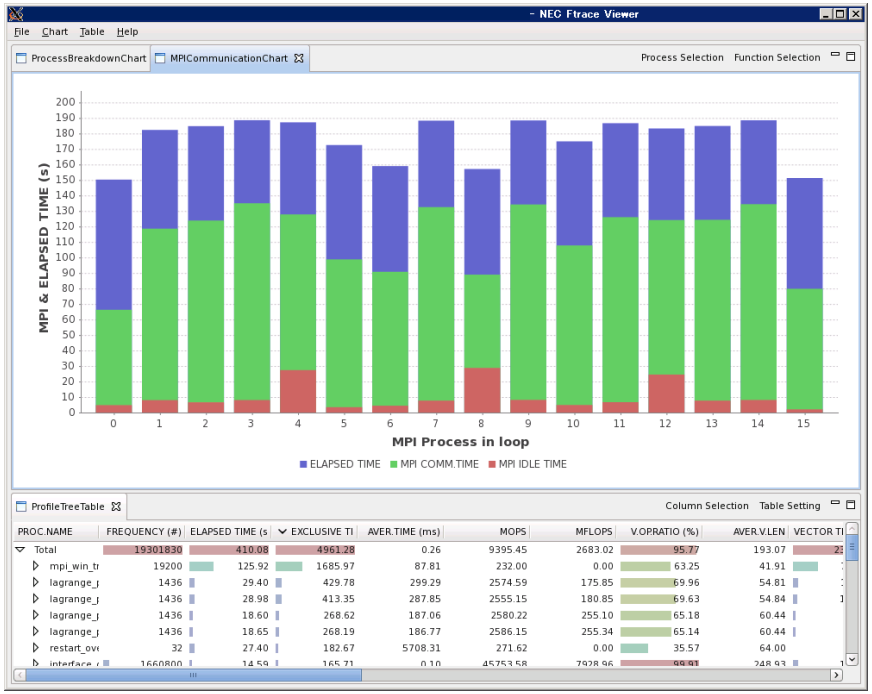
```
SUBROUTINE SUB2(X,Y,Z)
X=SQRT(Y**2+Z**2)
END
```

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。

NEC Ftrace Viewer

簡易性能解析機能(Ftrace)情報をグラフィカルに表示するためのツール

- 関数・ルーチン単位の性能情報を絞り込み、多彩なグラフ形式で表示できます.
- 自動並列化機能・OpenMP、MPIを利用したプログラムのスレッド・プロセス毎の性能情報を容易に把握できます.



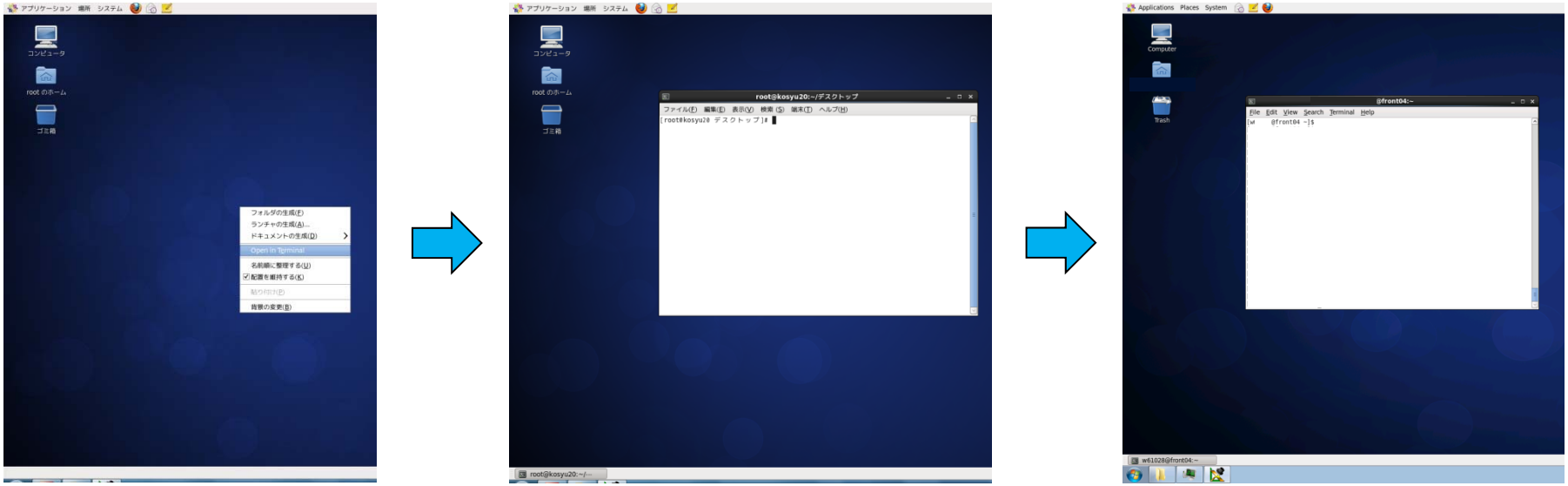
NEC Ftrace Viewer

NEC Ftrace Viewerの使用方法について

1．環境（Xサーバ）の準備（Exceedの場合）

フロントエンドマシンへのログイン

- Exceedの起動
- 端末画面の起動
  - マウス右クリックで表示されるメニューから “Open in Terminal” を選択
- フロントエンドマシンへログイン



1. 環境 (Xサーバ) の準備 (Xmingの場合)

フロントエンドマシンへのログイン

- Xmingの起動
  - 「すべてのプログラム」→「Xming」→「Xming」でXmingを起動.
  - Windows環境では，起動するとタスクバーにXmingのアイコンが表示される.
- TeraTermの設定
  - 「設定」→「SSH転送」→「リモートの (X) アプリケーション…」のチェックを入れてOKを押下.
  - xeyesコマンドなどで，画面転送ができているか確認して下さい.
- フロントエンドマシンへログイン

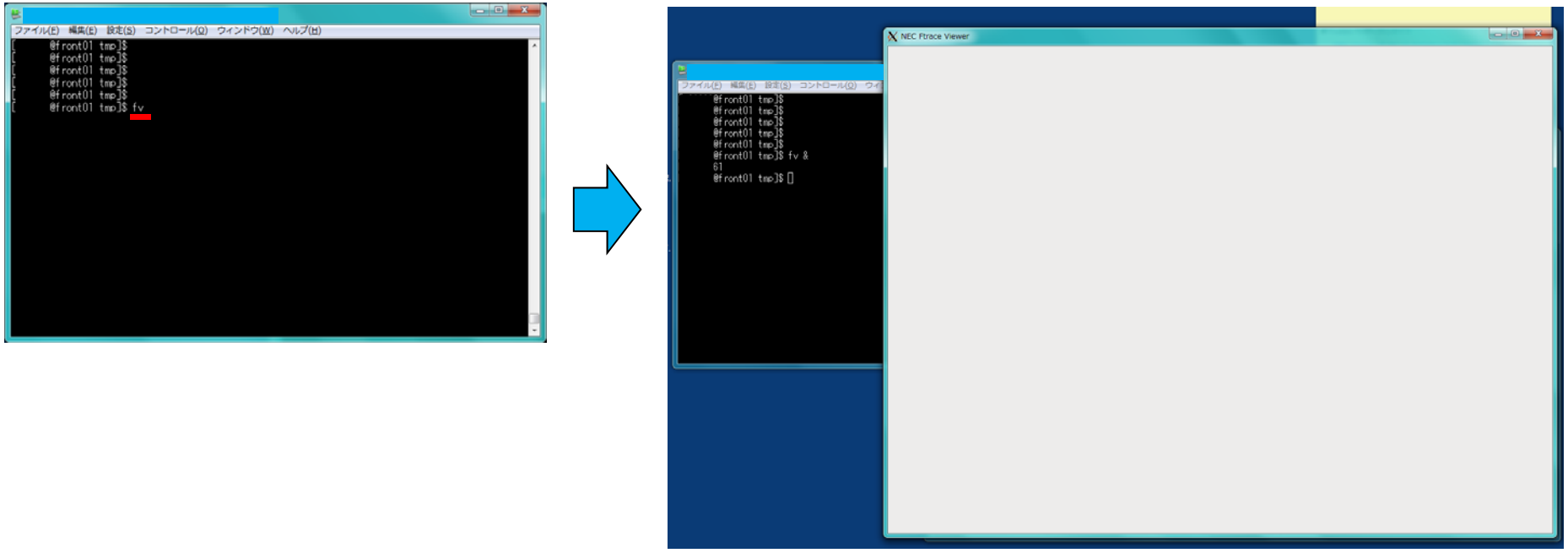
※次ページからの説明は<u>Windows環境でXmingを使用した場合</u>を例にしています.

2. NEC Ftrace Viewer の起動

GUI 画面の表示

- “fv”コマンドの実行

  Xmingウィンドウが立ち上がり，NEC Ftrace Viewer画面が表示される．



3. ファイルの読み込み

初期画面の上部メニュー「File」から表示する ftrace.out を選択

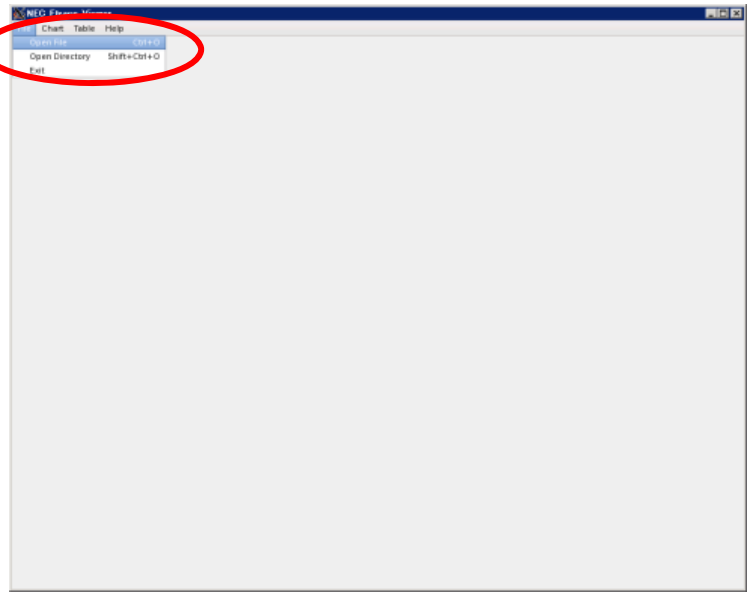
- Open File
  - 指定した ftrace.out もしくは ftrace.out.n.nn を1つ読み込みます.
- Open Directory
  - 指定したディレクトリ直下の ftrace.out もしくは ftrace.out.n.nn を全て読み込みます.

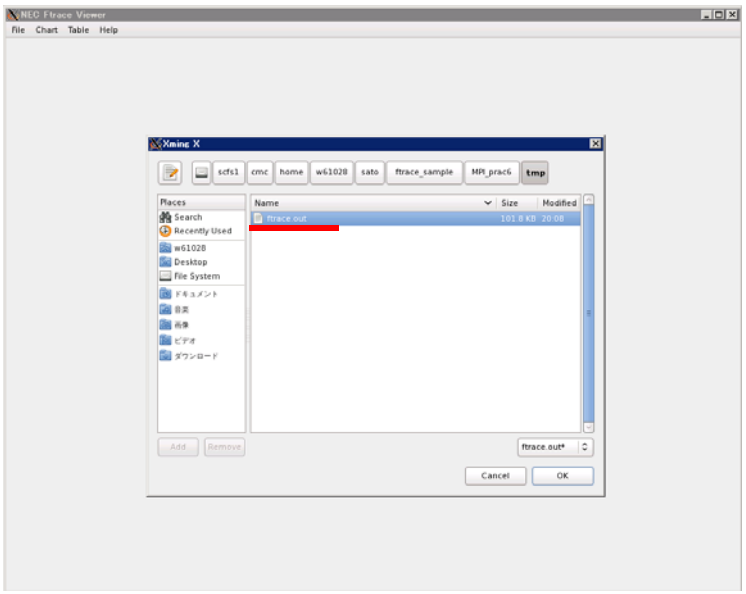
  ※ ftrace.out と ftrace.out.n.nn が同じディレクトリにある場合、読み込みに失敗します.

3. ファイルの読み込み

シリアル/SMP実行の場合(1/2)

- ftrace.out ファイルの読み込み
  - 「File」→「Open File」から読み込みたい ftrace.out を選択して「OK」を押下

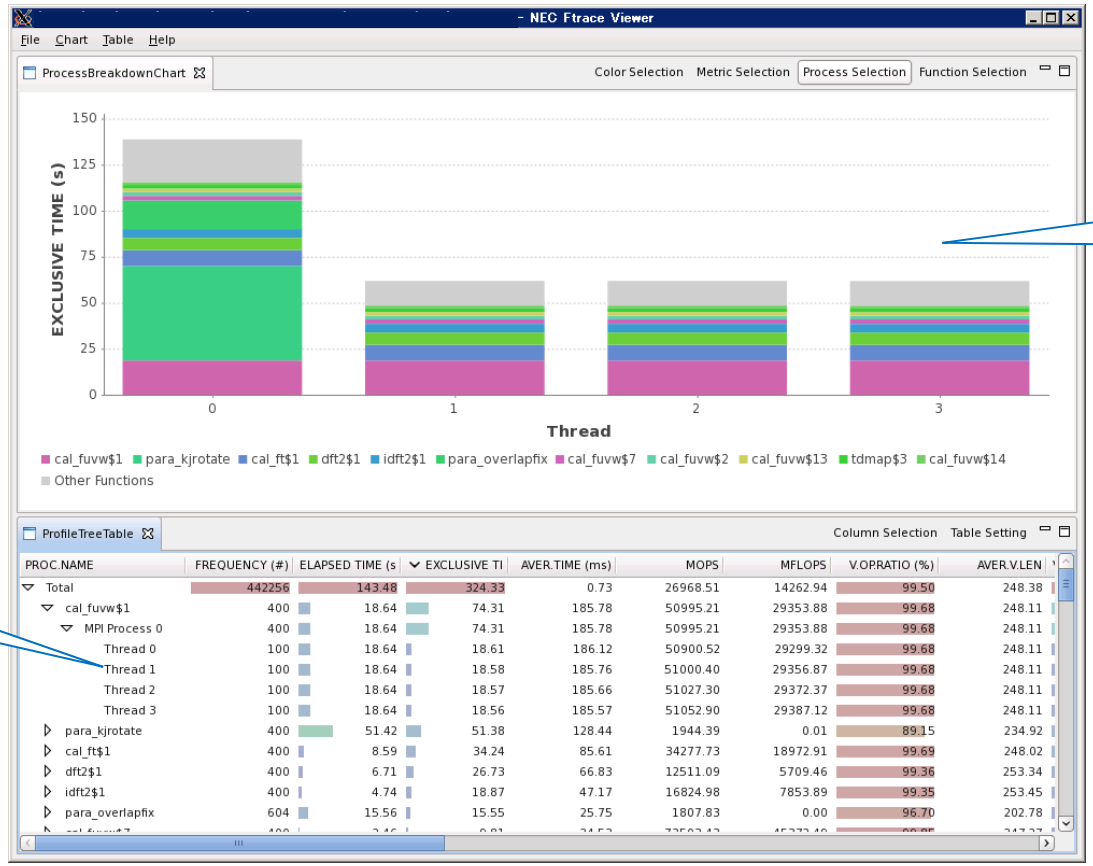




3. ファイルの読み込み

シリアル/SMP実行の場合(2/2)

- GUI画面の例(4SMP実行の結果)
  - “Process Breakdown Chart”モード



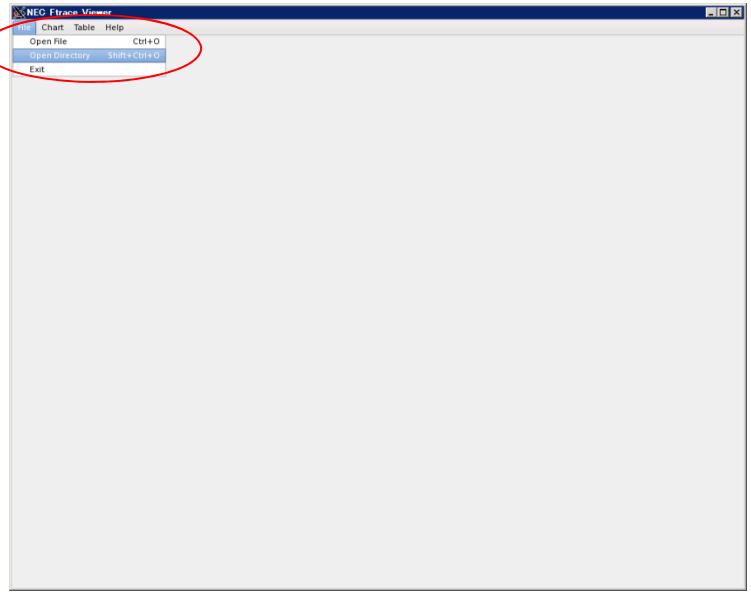
右クリックでグラフを
画像として保存できます

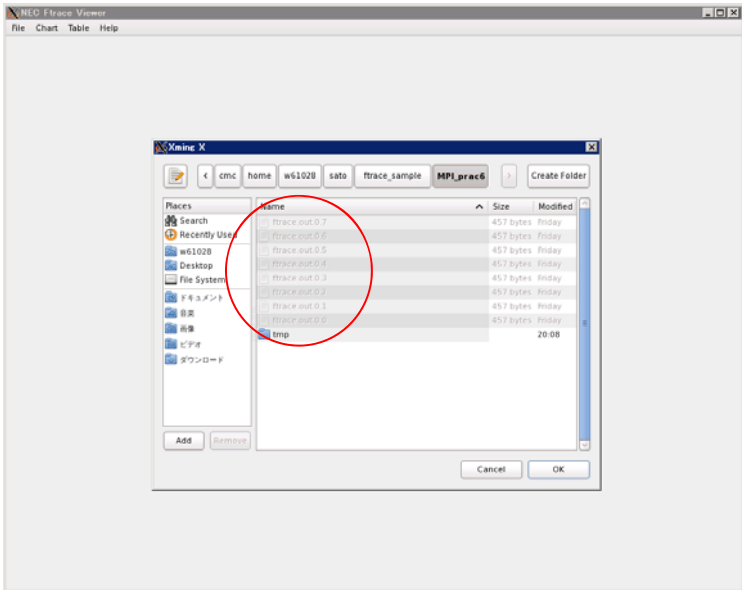
SMP並列プロセスごとの
性能情報が表示されます

3. ファイルの読み込み

MPI実行の場合(1/2)

- ftrace.out.n.nn ファイルの読み込み
  - 「File」→「Open File」から読み込みたい ftrace.out.n.nn があるフォルダを選択して「OK」を押下
    - ✓ 今回は，MPIプロセス分の ftrace.out ファイルを読み込む場合を例にしています．
    - ✓ 1プロセス分だけを表示させる場合は，「シリアル/SMP実行の場合」のようにプロセスに対応した ftrace.out.n.nn を指定して下さい．

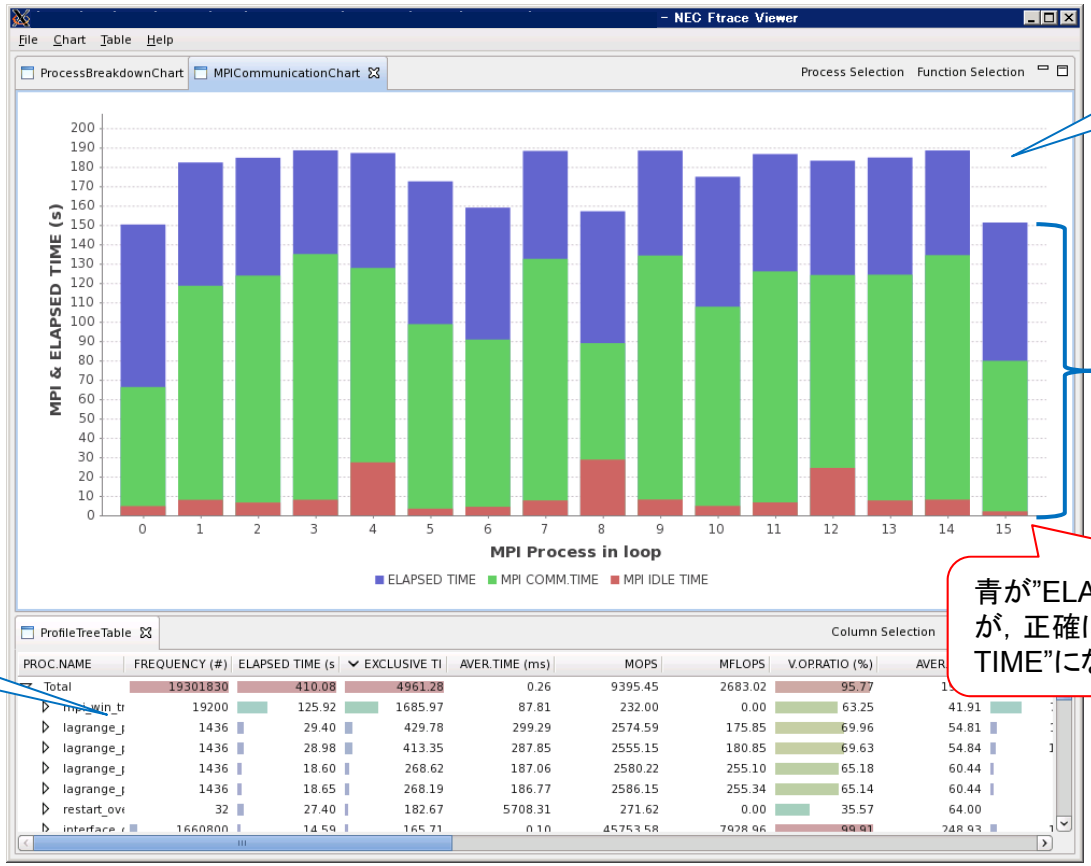




3. ファイルの読み込み

MPI実行の場合(2/2)

- GUI画面の例(16MPI実行の結果)
  - “MPI Communication Chart”モード



右クリックでグラフを
画像として保存できます

ELAPSED TIME

MPI並列プロセスごとの
性能情報が表示されます

青が”ELAPSED TIME”になっていますが，正確には青＋緑+赤が”ELAPSED TIME”になります．

1．演習問題：オリジナルコードのコンパイルと実行

目的

- 現状のプログラムの性能を把握する．

手順

- コンパイル（リストの確認）
- 実行（結果，性能の確認）

ディレクトリ

- practice_1

2. 演習問題：性能解析（Ftraceの利用）

目的

- 性能解析ツールFtraceを使い，性能情報を採取する．

手順

- ソースコードの修正（Ftrace_Regionの挿入）
- コンパイルオプションの追加（-ftrace）
- 実行（結果，性能の確認）

ディレクトリ

- practice_2

目次

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
```

-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: 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
```

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

-pi 指定時のコンパイラの変形イメージ

```
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/NOON_ADB	:ADBを利用する/しない
●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(SX-ACE向け)

SX-ACEでは長ベクトル・ストライドアクセスのループと，短ベクトル・連続アクセスのループであれば，後者の方が望ましい．

Loop1
- 長ベクトル
  →ループ長25600
- ストライドアクセス
  →65要素飛びアクセス

```
27: +------>      do i = 1,64
28: |           !cdir noloopchg
29: |V----->       do j = 1,25600
30: ||                d1(i,j) = d1(i,j)+a1(i,j)+b1(i,j)*c1(i,j)
31: |V-----        enddo
32: +------       enddo
```

Loop2
- 短ベクトル
  →ループ長64
- 連続アクセス

```
36: X------>      do i = 1, 64
37: |+----->       do j = 1, 25600
38: ||                d1(i,j) = d1(i,j)+a1(i,j)+b1(i,j)*c1(i,j)
39: |+-----        enddo
40: X------       enddo
```

ストライドアクセスの場合，
SX-ACEだとバンク競合時間が増大

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
20000	9.836(22.6)	0.492	26747.4	9994.5	99.64	256.0	9.823	0.000	0.000	4.429	7.445		loop1(SX-9)
20000	83.800(73.0)	4.190	3138.6	1173.1	99.67	256.0	83.797	0.000	0.000	3.695	80.709	0.00	loop1(SX-ACE)
20000	10.611(24.3)	0.531	25140.1	9264.6	98.27	64.0	10.596	0.000	0.001	0.277	8.056		loop2(SX-9)
20000	9.172(8.0)	0.459	29083.9	10718.0	98.27	64.0	9.169	0.000	0.000	0.000	3.577	0.00	loop2(SX-ACE)

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 outerunoll=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
```

3. 演習問題：outerunroll指示行

目的

- アウターアンローリング指示行の使い方を理解する.
- ４段アウターアンロールを行う.

手順

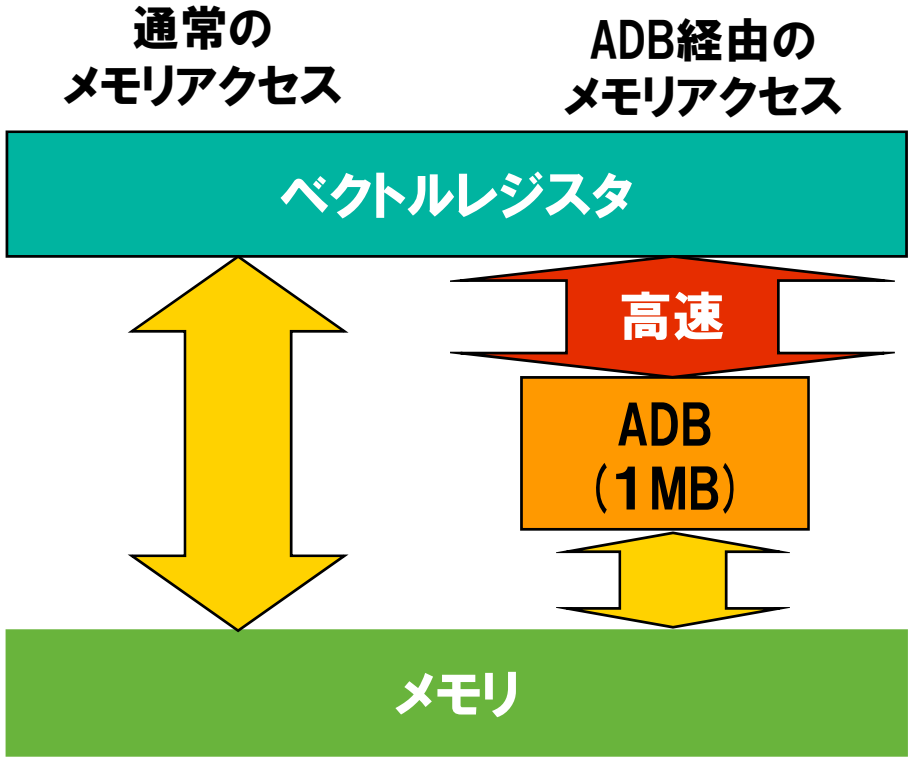
- ソースコードの修正
- コンパイル（リストの確認）
- 実行（結果，性能の確認）

ディレクトリ

- practice_3

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の利用(1)

- コンパイラはADBを利用することで高速化が見込める配列に対して自動的にON_ADB指示行を指定するほか，ベクトル版数学関数や行列積ライブラリなどの組込み関数においてADBを使用した高速化を行っている．

```
  7: +------>          do j=1,m
  8: |V----->            do i=1,n
  9: ||      A             d(i,j)=a(i,j)*b(j) + a(i,j-1)*c(j)
 10: |V-----              enddo
 11: +------            enddo

  8  vec   (   1): Vectorized loop.
  8  vec   (  29): ADB is used for array.: a
```

配列aの2次元目の添字式がjとj-1であるので，2次元目のループに繰返しで同じ配列aの要素が参照される（再利用性あり）ので，コンパイラはADBに配列aを乗せることを指定する．

- ADBの容量(Coreあたり1MByte)より指定する配列の大きさの方が大きい場合は，後からバッファリングするデータにより上書きされる．
- ON_ADB指示行やNOON_ADB指示行を用いて，再利用性の高いデータを選択してADBを効率よく利用する．

ADBの利用(2)

NOON_ADB指示行で指定された配列はADBを利用しない.

```
32: +------>           DO K=2,NZ-2
33: |+----->           DO J=2,NY-2
34: ||           !CDIR NOON_ADB(B1,B2,B3,B4)
35: ||V---->           DO I=2,NX-2
36: |||      A         D= B1(I,J,K)*(A(I+1,J,  K)  +A(I-1,J,  K))
37: |||                &  +B2(I,J,K)*(A(I,  J+1,K)  +A(I,  J-1,K))
38: |||                &  +B3(I,J,K)*(A(I,  J,  K+1)+A(I,  J,  K-1))
39: |||                &  +B4(I,J,K)*(A(I,  J+1,K+1)-A(I,  J+1,K-1)
40: |||                &              -A(I,  J-1,K+1)+A(I,  J-1,K-1))
41: |||      A          C(I,J,K)=A(I,J,K)*D
42: ||V----             END DO
43: |+-----             END DO
44: +------             END DO

35  vec   (   1): Vectorized loop.
35  vec   (  29): ADB is used for array.: a
```

コンパイラは自動で，配列Aだけでなく B1,B2,B3,B4もADBの利用をしようとしている.
ADBの容量，再利用性を考慮し，配列Aだけ利用するため，NOON_ADB指示行で，B1,B2,B3,B4を対象外にする.

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
1000	11.602(97.5)	11.602	42818.4	19250.5	99.55	255.2	11.602	0.000	0.000	0.000	5.235	20.05	指示行なし
1000	8.948(96.8)	8.948	55519.5	24960.8	99.55	255.2	8.947	0.000	0.000	0.000	3.138	53.27	指示行あり

上記例では，ヒット率が約2.6倍向上し，性能は約1.3倍向上.

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は定義されない
かもしれない

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で定義した値を参照 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)....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) = 式 ± X(I-1) * 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)

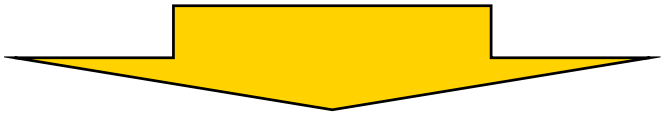
配列の定義・参照は連続した要素のアクセスがもっとも速い。

- 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(IX(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)
DO I=1,1000
  DO J=1,1000        ベクトル化
    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。

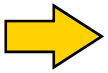
改善例の参考値
ユーザ時間

1024要素飛び　：0.146msec
連続要素アクセス：0.001msec

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： 4.108msec

ユーザ時間： 0.255msec

4. 演習問題：自動インライン展開

目的

- 自動インライン展開のオプションの使い方を理解する.

手順

- インライン展開前の性能の確認
  - コンパイル（リストの確認）
  - 実行（結果，性能の確認）
- インライン展開後の性能の確認
  - コンパイルスクリプトへオプション追加，再コンパイル（リストの確認）
  - 再実行（結果，性能の確認）

ディレクトリ

- practice_4

5. 演習問題：行列積ライブラリの利用

目的

- 行列積ライブラリの性能を確認する.

手順

- コンパイルスクリプトの修正
- コンパイル（リストの確認）
- 実行（結果，性能の確認）

ディレクトリ

- practice_5

目次

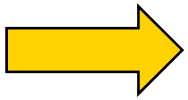
自動ベクトル化における注意事項

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
```

ベクトル化



```
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