The University of Osaka
Institutional Knowledge Archive

Title	並列処理の注意点と作成手順 : SX-4のプログラム相談員より
Author(s)	
Citation	大阪大学大型計算機センターニュース. 1997, 104, p. 72-77
Version Type	VoR
URL	https://hdl.handle.net/11094/66213
rights	
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

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

The University of Osaka

並列処理の注意点と作成手順

— SX-4 のプログラム相談員より —

[ベクトル化機能と並列処理機能の最も違う点]

ベクトル化機能の場合は、使用する CPU 時間が減少し、さらに経過時間も短縮されます。並列処理機能の場合は、1CPU で行っていた仕事を複数 CPU を同時に使うことにより経過時間の短縮を図るもので使用する CPU 時間は減少しないことです。逆に、複数の CPU をコントロールするための処理(同期制御、排他制御等：並列オーバーヘッドと呼ばれる)により CPU 時間は増加します（ ただし、「第 4.2 並列化時における課金対象 cpu 時間」で示すように負担金は並列化を行なえば安くなるように優遇措置をとっております）。

自動並列化は、自動ベクトル化と同様に DO ループを対象とし、その部分を指定された CPU 数で分割、実行することにより並列化します。このときに並列オーバーヘッドが加わります。 この並列処理オーバーヘッドの時間をできるだけ短くし、確保した CPU を有効に利用するために以下の点に注意して、プログラムの作成をおこなってください。

○多重ループの場合はできるだけ外側の DO ループで並列化できるようにする。

この時内側ループがベクトル化されると高い処理性能を得られます。

並列処理オーバーヘッドの時間よりループの処理時間が十分大きくないと並列処理の効果を得られません。この対策として ANALYZER-P/SX や profiler により、DO ループの処理時間やループ長を確認して、大きなコストの DO ループから並列化する様にします。コンパイルオプションの他、指示行 *odir noconcur により特定の DO ループの並列化を抑止できます。

○対象 DO ループの処理が確保された 各 CPU に均一に分配されるようにする。

下記の例の様に外側 DO ループで並列化すると、内側の DO ループの処理が均一にならないためオプション f77: -Wf"by=4", f77sx: fopp by=4 (4: 分割時のループ長)を指定して、各 CPU の負荷を均一となるようにする。

【例】

```
DO I = 1,N
    DO J = I,M
      (ベクトル化)
    END DO
END DO
```

また、内側ループの処理が一定の場合は、外側ループのループ長が CPU 数で割り切れる様にループ長や CPU 数を調整する。

○最内側のループ長が十分大きい場合、またループ中の計算（演算量）が 大きい場合は、オプション f77: -Wo"-e i", f77sx: -fopp e=i を指定して並列化しても効果が出ます。

[並列化してはいけないプログラム]

サブルーチン内でのみ使用されるローカル変数あるいはローカル配列は、通常は静的に準備された領域(static 領域)に取られますが、並列処理時はスタック上にとられます。このため、以下のようなプログ

ラムは動かなくなりますので注意が必要です。なお、これらはいずれもFORTRAN規格では禁止されています。

○初期値としてゼロが入っていることを期待して、初期値設定をしないで引用しているプログラム。

○RETURN文実行後も値が保存されていることを期待しているプログラム。

本来、このようなプログラムはSAVE文で指定する必要がある。

【例】

```
subroutine make_random(x,n)
real x(n)
integer iseed
if iseed = 0 then iseed = 1        iseed の初期値としてゼロが入っていることを期待している
do i = 1,n
  iseed = iseed * 65537
  iseed = iand (iseed,2147483647)
  x(i) = float( iseed / 2147483647)
end do
return
end
```

このようなプログラムが含まれる可能性がある場合は、自動並列で実行する前に、まず、f77 -stack（ローカル変数あるいはローカル配列をスタック上にとる）を指定して実行し、スタック上にとっても問題がないことを確認の上で、自動並列化されることをおすすめします。

[並列化可能なプログラムか否かの確認]

並列処理時には共通ブロック(common)として宣言されていない変数や配列（以下ローカル変数とローカル配列と記述します）のメモリ割付け位置が、共通ブロックと同等の領域からCPU毎に用意されているスタック上に変更されます。これは 一つのサブルーチンが複数のCPUで同時に実行されても結果が正しくなる様にするためです。この結果これらの変数や配列の初期値は不定となりますので 、以下の2つの項目のいずれかに該当するプログラムは正しく修正する必要があります。

これらはいずれもFORTRAN規格では禁止されています。

○ローカル変数またはローカル配列の初期値としてゼロを期待し初期化せず参照しているプログラム。

[対策] ローカル変数やローカル配列の初期化処理を追加する。

○RETURN文実行後もローカル変数やローカル配列の値が保存されていることを期待しているプログラム。

[対策] SAVE文で指定する。

ほとんどの場合、以下の手順で該当しているか知ることが出来ます。

(a) コンパイル時に -P stack オプションを指定し、ローカル変数やローカル配列の割り付け位置をスタックに変更したLMを作成する。

(b) 作成したLMを実行し、結果が -P stack オプションを指定していないLMの結果と変わらな

いことを確認する。

【例】

```
% f77 -o static.out   sample.f
% f77 -o stack.out   -P   stack   sample.f
% static.out > static.result
% stack.out > stack.result
% diff static.result stack.result
```

もし、実行結果が一致せず該当するローカル変数やローカル配列があると分かった時、これらを検索し正しくプログラムを修正します。

このときプログラムのコンパイル時に出力される以下の警告メッセージやオプション -P stack -Wf'-L map' により得られる相互参照リストが参考になります。

** 106 : 変数あるいは配列が引用されているが，どこでも定義されていない : xx

xx : 対象となる変数あるいは配列

ただし初期化処理が IF 文下に記述されていたり、read 文によるデータ入力で入力データが不足している場合等、実行時の条件で処理が行われないことがある様な複雑なプログラムでは、コンパイラは該当する変数や配列を検出できませんので注意してください。

[実行時間の大きなループの検出及びその並列化]

並列処理では、並列処理オーバーヘッドと呼ばれる複数の CPU をコントロールするための処理(同期制御、排他制御等)が加わるため、ループの処理時間が十分大きくないと並列処理の効果を得られません。この対策として ANALYZER-P/SX やプロファイル機能により、DO ループの処理時間やループ長を確認して、大きな実行時間の DO ループから並列化する様にします。

このためには ANALYZER-P/SX の実行時間解析機能により DO ループの一回当たりの実行時間が 1 ミリ秒以上のものを自動並列化の対象とすると良いでしょう。

その他の DO ループは指示行 *odir noconcur により自動並列化を抑止します。また、サブルーチン中に一つも並列化の対象となる DO ループが無ければ、そのサブルーチンをオプション -P mulit でコンパイルし自動並列化をやめます。

サブルーチン毎にオプション設定するためには、事前に fsplit コマンドでプログラムをサブルーチン単位に分割しておくと便利です。

【例】ANALYZER-P/SX による実行時間解析の例

```
% fanp   -AO   do   foo.f
```

【例】プログラムの分割及び、プロファイル機能での情報採取

```
% fsplit   foo.f
% f77   -c   -P auto   -Wf"res=whole" MAIN.f
% f77   -c   -P auto   subp.f
% f77   -c   -P multi   subs.f
```

```
% f77   -P multi -p   MAIN.o   subp.o   subs.o
% setenv   F_RSVTASK   2
% a.out
% prof   a.out   >   prof_data_p
```

MAIN.f　：メインプログラム
subp.f　：並列化対象サブルーチン
subs.f　：非並列化対象サブルーチン

[仕事量の不均衡(各 CPU の負荷が不均一)の検出及びその対策]

自動並列化により並列化された DO ループは、サブルーチンに変形されます。

そのサブルーチン名は、 sub1$1 のように "(サブルーチン名)$n : n はサブルーチン内でシーケンシャルな番号、でネーミングされます。

プロファイラの結果から、この並列化された時のサブルーチン名により仕事量の不均衡を検出することができます。この仕事量の不均衡の対策として、DO ループの分割方法をコンパイルオプション f77 -Wo"-L 4" (ループを PARDO BY=4 で並列化します)により変更したり、CPU 数の変更により改善をおこないます。

以下に prof コマンドの出力結果を簡単に説明します。

```
% Res   T/M    Micro   CPU   #Calls     msec/call      Name
                        56.06                          gx_lpmpark      -- 1.
                 56.06                                 micro1
                        30.42                          subp$1_         -- 2.
                 21.66                                 micro1
                 8.76                                  micro2
                        17.68                          subp$2_         -- 3.
                 8.74                                  micro1
                 8.93                                  micro2
                        16.53                          gx_lpmret       -- 4.
                 1.31                                  micro1
                 15.22                                 micro2
```

各サブルーチン／関数の説明

1) gx_lpmpark

演算モードにより [ex_i|ex_l|fx_i|fx_l|gx_l]pmpark のいずれかが呼ばれます。

並列化されない部分の実行時に、確保された CPU が仕事を割り当てられるのを待っているサブルーチン。一定時間待ちその間次の並列処理サブルーチンがこなければ CPU を OS に一旦返します。なお待ち合わせの時間は、環境変数 F_PMTUNE で調整可能ですが(既定値 50 ミリ秒)、あまり小さくすると、OS との間で CPU のやり取り為の処理が増え 経過時間がのびてしまいます。

2) subp$1_

自動並列化によりDOループがサブルーチンに変形されたものです。

micro1, micro2は2つのCPUで並列処理が行われたことを意味します。もしCPU数が4ならば micro4まで表示されます。

この例ではmicro1, micro2のCPU時間が21.66 secと8.76 secで差が大きく仕事量の不均衡(各CPUの負荷が不均一)がおきていることを示しています。

3) subp$2_

これも自動並列化によりDOループがサブルーチンに変形されたものです。

この例では仕事量がほぼ均一になっています。

4) gx_lpmret

演算モードにより [ex_i|ex_l|fx_i|fx_l|gx_l]pmparkのいずれかがよばれます。

各CPUの同期待ちを行うサブルーチンです。特にこの関数は自動並列化されたDOループの出口部分で他のCPUの処理の終了待ちをおこないます。よってこの関数のCPU時間は、仕事量の不均衡の大きさを意味し、並列化されたDOループの仕事量の不均衡を各々修正していくことにより、この時間は逆に小さくすることができます。

[並列化率の向上]

最後に自動並列化されていないDOループを捜し、そのDOループの並列化を検討します。

指示行 *odir nosync (*vdir nodepでも可)や *odir cncall (並列化しても大丈夫なサブルーチン呼び出しを含む)等で並列化が可能となることがあります。これらの観点で自動並列化されていないDOループの見直しを行います。

[指示行による並列化促進方法]

自動並列化されていないコストの大きなDOループを捜し、そのDOループの並列化を検討します。

この時下記のオプションにより出力リストをとると便利です。

```
f77 -c -P auto -Wo'-l test.lst' test.f
```

または

```
f77sx -c -fopp par list=test.lst test.f
```

(ソース : test.f)

```
      subroutine sub(a,b,c,id,n,m)
      dimension a(n,m),b(n,m),c(n),id(n)
      real a,b,c
      integer id,i,j
c
      do j = 1,m
        do i = 1,n
            a(id(j)) = a(id(j)) + b(i,j) * c(i)
        end do
      end do
```

(出力リスト : test.lst)

```
fopp V5.3B43 9:58:23 4/09/97
 1. subroutine sub(a,b,c,id,n,m)
 2. dimension a(n),b(n,m),c(n),id(n)
 3. real a,b,c
 4. integer id,i,j
 5.c
 6.   D--- do j = 1,m
 7.   D   V-   do i = 1,n
 8.   D   V     a(id(j)) = a(id(j)) + b(i,j) * c(i)
 9.   D   V-   end do
10. D--- end do
```

```
Line Typ Msg ( T=Translation, D=Dependency, W=Warning, S=Syntax )
8 D Potential feedback of array elements – use directive if OK (A)
  conflict on line 8. The loop index is J
```

メッセージにあるようにリスト中でDで示されたループは、コンパイル時にデータ依存関係が不明 なため並列化されていません。

ここで問題となっている配列aの間接参照は、配列idの値に重なりがなく同一アドレスを参照することがないとすると並列化およびベクトル化が可能です。

この場合は指示行 *odir nosync (*vdir nodep でも有効)を指定するとにより、下記のように自動並列化されます。

```
   fopp V5.3B4310:02:10 4/09/97
1)  subroutine sub(a,b,c,id,n,m)
2)  dimension a(n),b(n,m),c(n),id(n)
3)  real a,b,c
4)  integer id,i,j
5)  c
6)  *odir nosync
7)  P---      do j = 1,m
8)  P    V-       do i = 1,n
9)  P    V          a(id(j)) = a(id(j)) + b(i,j) * c(i)
10) P   V-      end do
11) P---     end do
12) c
13) return
14) end
```

同様にDOループ中にcall sub2(...) の様なサブルーチン呼び出しがあると、データ依存関係がわからないため並列化されません。この時は以下の様なメッセージが出力されます。

```
Line Typ Msg ( T=Translation, D=Dependency, W=Warning, S=Syntax )
12 T Subr. calls are handled only when CNCALL directive is used
(SUB2)
```

呼び出されているサブルーチンを調査し並列に動作可能ならば、指示行 *odir cncall(sub2) を指定し並列化させます。この時、個々のサブルーチンを調査するのが大変であれば、インライン展開をオプションで指定して様子を見るのも方法の一つです。