FORTRAN90/SX の自動並列化について

日本電気株式会社 第一コンピュータソフトウェア事業部　橋本 ユキ子

概要

FORTRAN90/SX は、 SX-7 のもつ並列処理機能を利用して、その性能を十分に引き出すための高度な自動並列化機能を備えている。本文では、並列処理の概念、自動並列化機能、および並列化促進のための技法について紹介する。

1. はじめに

SX-7 は、既に実績のあるベクトル処理に並列処理機能を融合した「スケーラブル・パラレル・スーパーコンピュータ」です。このハードウェアのもつ高い能力をいかんなく発揮させるためには、コンパイラのベクトル化/並列化機能が重要な役割を果たします。

FORTRAN90/SX は、ハードウェアに密着した高度な最適化、ベクトル化、並列化機能を有した Fortran90 コンパイラです。言語仕様としては、Fortran95（JIS X3001-1:1998）をサポートしています。SX-7 は、32 台のプロセッサの範囲内では、利用しやすい共有メモリ方式を採用しており、FORTRAN90/SX の並列処理は、この共有メモリ方式を使った並列化機能を提供しています。

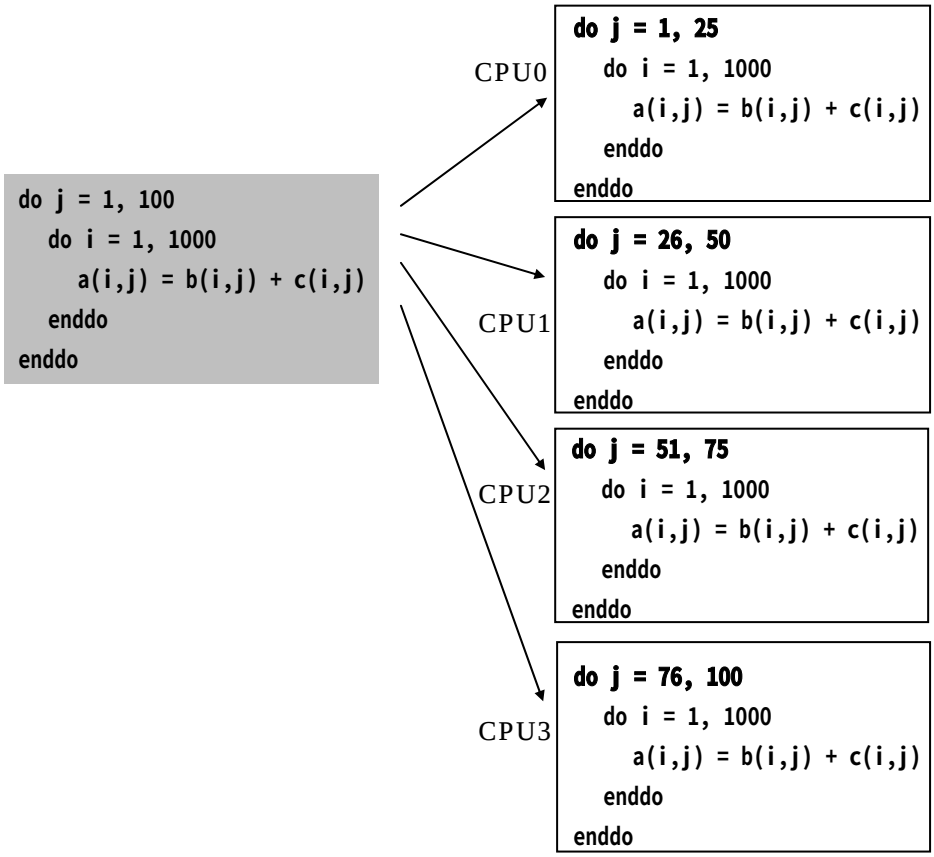
一般に、並列化を行う時には、並列実行しても結果が変わらないことを保証するために、データの依存関係の解析を行い、細心の注意を払ってプログラムの変形や指示行の挿入をしなければなりませんが、自動並列処理機能を利用すると、それらの作業を自動的にコンパイラが行います。また、ベクトル化と同様に、効果的に並列化を行うためのオプションや指示行が用意されており、十分にプログラムのチューニングを行うことが可能です。本稿では、並列処理の概念と、自動並列化機能および関連する指示行についてご紹介します。

2. 並列処理とは

並列処理とは、1 つの仕事をいくつかの小さな仕事に分割し、それを複数のタスク（CPU）で並列に実行することです。FORTRAN90/SX コンパイラが備えている自

動並列処理機能とは、「コンパイラがプログラムを解析して、並列に実行可能なループや文の集まりを抽出し、ループの繰り返しや文の集まりを複数のタスクに自動的に割り当てて実行時間を短縮する機能」です。

コンパイラが、do ループを 4 つのタスクに分割して実行するイメージは、次の図ようになります。この例では、外側ループの 100 回の繰り返しを 4 つに分割して、各 CPU 上で各々を並列に実行します。

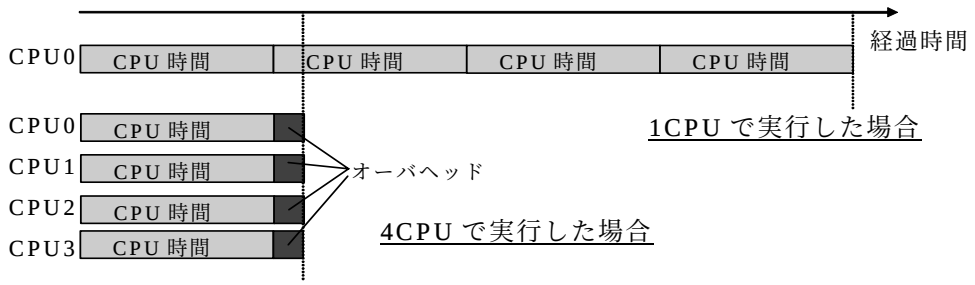


この例の配列 a は分割されて、各タスク毎に値（a(i,1)～a(i,25)、 a(i,26)～a(i,50)、a(i,51)～a(i,75)、 a(i,76)～a(i,100)）が計算され、定義されるので、a は各タスクから共通に参照できるグローバルな領域に割り当てられます。このような各タスクから共通に参照できるデータをタスク間共有データと呼びます。これに対して、並列実行される各タスクから、非同期に定義/参照を行うと、結果が不正になってしまうようなデータ（上例では i）は、各タスク毎にローカルな領域（スタック）に割り当てられ、タスク固有なデータと呼びます。自動並列化機能は、このようなデータの割当ても適切に行います。

この自動並列化機能は、オプション“-P auto”を指定するだけで利用可能です。

```
sxf90  -P auto  program.f
```

ここで、“実行時間の短縮”には注意が必要です。並列処理は、1 つの仕事を分割して、並列に実行を行うわけですから、CPU 時間が削減されるわけではなく、経過時間が短縮されることになります。また、仕事を各タスクで並列実行させるための処理（オーバーヘッド）も必要となり、CPU 時間は、かえって増加することになります。たとえば、CPU 時間と経過時間の関係は、以下のようになります。



2.1　並列処理とベクトル処理

ここで、“ベクトル化による実行時間の短縮”と“並列化による実行時間の短縮”との相違を明確にしたいと思います。ベクトル処理とは、規則的に並んだ複数個の配列データを一度に演算する高速なベクトル命令を使って処理を行うことであり、この場合、CPU 時間が短縮され、同時に経過時間も短縮されます。これに対して、並列処理では、先に述べた通り、合計の CPU 時間は並列化のオーバヘッドにより、単一 CPU で実行した時よりも増加することになりますが、経過時間を短縮することによって高速化を図ります。したがって、ベクトル化の場合は、単一 CPU での実行ですが、上手にベクトル化できれば、スカラで実行した時よりも、一般的に 10 倍以上の性能向上が期待できます。並列化の場合に期待できる性能向上の効果は、最大で使用可能な CPU の個数倍となります。

これらのことより、基本的には、ベクトル化と並列化を組み合わせて利用し、多重ループの内側ループについてはベクトル化を行い、外側ループを並列化することが、高速化を図る最善の方法となります。

また、並列処理した場合には、オーバヘッド時間が加わりますので、並列に実行される仕事量（粒度と呼びます）が十分に大きくなければ、並列化の効果は期待できません。当然ですが、並列処理のオーバヘッド時間よりも並列実行される部分の実行時

間の方が小さければ、並列化したことにより、実行時間（経過時間）がかえって多くなってしまうことになります。

ベクトル化できるプログラムは、ベクトル化すればほとんど全ての場合に性能向上が図れますが、並列化できるプログラムは、並列化したからといって必ずしも性能が向上するとは限らないことになります。すなわち、どんなプログラムでも自動並列化すれば性能が向上するということではないことに注意して下さい。効果的な並列化の方法について、この後、説明をしていきたいと思います。

3. 自動並列化

自動並列化機能を使用すれば、その名の通り、並列用の指示行を直接使って並列プログラミングをする場合に比べ、格段に容易にプログラムを並列化することができます。自動並列化機能は、プログラムを解析し、並列化した場合の効果も調べて、並列化を行います。たとえば、並列化すれば十分性能が向上するだけの粒度をもっているか、do ループの繰り返しを並列実行しても結果不正になるような文を含んではいないかなどを調査し、可能な場合は、並列化できるようにループやデータの定義を書き直して並列化を行います。

自動並列化は、内部的に次の図のようなイメージで行われます。

コンパイラが並列可能なループを検索し、そのループを新たにサブルーチンとして切り出し、マイクロタスク機能を使って並列化します。並列実行されるループをサブルーチンとして切り出すことによって、並列化効率を最大限に引き出すことが可能となります。

(a) ソースプログラム

```
program main
   .......
   do i = 1, n
      ～
   enddo
   .......
end program
```

コンパイラによる変形

(b) 変形イメージ

```
      program main
!cdir reserve
        .........
        call sub$1
        .........
!cdir release
      end program
      subroutine sub$1
        初期化
!cdir pardo for
        do i = 1, n
           ～
        enddo
      return
      end subroutine sub$1
```

上記例の!cdir で始まる行はコンパイラによって挿入された指示行で、reserve はタスクの確保、release はタスクの解放、pardo はループを並列実行することを指示します。また切り出されたサブルーチン名には、元のサブルーチン名に$1、$2、...とサフィックスが付きます。コンパイル時に、オプション-R1 を指定することによって、並列化された様子を示すリスト（変形リスト）を参照することができます。

3.1 自動並列化の条件

自動並列化の条件は、以下の通りです。

対象となる部分	DO ループ IF 文と GOTO 文によるループ 配列式
対象となるデータの型	文字型以外の型（4 倍精度も可能）
対象となるループ中に許される文	代入文、IF 文、GOTO 文、CONTINUE 文、CALL 文、SELECT 構文
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

- ループ中のデータに依存関係がある場合に、ベクトル化や並列化ができなくなることがありますが、その条件には違いがあります。次の例では、ベクトル化は可能ですが、並列化はできません。配列 a の引用（演算）される要素（a(i)）と定義される要素（a(i-1)）がループの繰り返し間でオーバラップしているため、ループの繰り返しを並列実行すると、結果不正になる可能性があります。

```
do i = 2, n
  a(i-1) = a(i) * b(i) + c(i)
enddo
```

  並列化の場合は、ループの繰り返し間でデータの依存関係があると、並列化できなくなるため、ベクトル化よりも条件が厳しくなります。
- call がループ中にある場合、ベクトル化はできませんが、呼び出すサブルーチンが並列実行可能であれば、そのループを並列化することができます。
- ベクトル化では、4 倍精度用のベクトル命令がないため、4 倍精度のベクトル化は不可能でしたが、並列化においては、ハード的な制約はないため、並列化が可能です。

3.2　自動並列化方法

並列実行される場合は、先にも述べた通り、粒度が十分に大きくなければその効果が期待できません。また、SX-7 はベクトルマシンであるため、ベクトル化が性能向上には欠かせない要因となります。これらのことを考慮し、自動並列化機能は、基本的には、多重ループの内側ループをベクトル化し、外側ループを並列化します。内側ループがベクトル化できない場合は、スカラコードのまま外側ループが並列化されます。

```
subroutine sub(a,h)
real a(600,100,100)
integer h(100)
do i = 1,100          並列化
  do j = 1,100
    do k = 1,599
      a(k,j,i) = (a(k+1,j,i) + a(k,j,i)) * 0.5
    enddo             ベクトル化
  enddo
  h(i) = h(i) + 1
enddo
end
```

更に、コンパイラは、並列化の効果を高めるため、可能であればループ変形などの最適化を行い、並列化を促進します。これらの並列化の工夫について、いくつか例を紹介します。

- **一重ループの場合**

  基本的には、ベクトル化を行いますが、ループのコストが大きい、つまり粒度が大きい場合は、ループを分割して、ベクトル化＋並列化を行います。ベクトル化できない場合は、並列化だけが行われます。

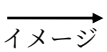
```
do i= 1,100
  a(i) = b(i) + c(i)          ベクトル化
enddo


do i = 1,10000
  a(i) = b(i) + c(i)          ベクトル化＋並列化
enddo
```

◆ **ループ融合や一重化が可能なループの場合**

ループ融合や一重化などのループの最適化を行った後に、並列化を行います。

```
subroutine sub(a,b,c)
real a(10000,4),b(10000,4),c(10000,4)
do j= 1,4
  do i=1,10000
    a(i,j) = sqrt(b(i,j))
  enddo
enddo
do j= 1,4
  do i=1,10000
    b(i,j) = c(i,j) - a(i,j)
  enddo
enddo
return
```




```
real a(10000,4),b(10000,4),c(10000,4)
do j=1,10000*4
    a(j,1) = sqrt(b(j,1))
    b(j,1) = c(j,1) - a(j,1)
enddo
```

並列化

◆ **条件並列化**

ループ長（粒度）あるいは依存関係が不明で、並列化の効果がコンパイル時に判断できない場合、実行時に粒度や依存関係を調べて並列コードを実行するかどうかを選択できるように条件並列化を行います。

次の例では、nx*ny>n によって、粒度が並列化するのに十分かどうかを調べています。これは、ループの繰り返し数が十分に大きいかどうかを実行時にチェックしているわけですが、このとき、コンパイラは単純に繰り返し数だけではなく、do ループ中の各演算（加算や乗算など）に対して重み付けを行い、do ループの演算コストから並列化の効果が十分に期待できる値（n）を計算して、条件並列化を行います。

また、ループ中のデータに依存関係ある場合は並列化すると結果不正になりますが、

この例では、配列 y を定義している y(ic+i)と y(id+i)の ic と id の関係を実行時に調べること（id-ic=0 .or. abs(id-ic)>=nx）によって、このループの実行中に、配列 y の同じ領域にデータを書き込まない場合のみ並列化されるようにしています。

```
do i = 1,nx
  aa = a
  bb = b
  do j = 1,ny
    aa = aa + x(i+1) * g(j-1)
    bb = bb + x(i-1) * g(j-1)
  enddo
  y(ic+i) = -aa
  y(id+i) = -bb
enddo
```

イメージ

```
if (nx*ny > n .and.
   (id-ic = 0 .or. abs(id-ic) >= nx) then

    並列コード

else

    非並列コード

endif
```

◆ **手続き呼び出しを含むループの場合**

手続き呼び出しを含むループは、手続き間解析によって、依存関係を判定し、並列化を行います。次の 2 つのサブルーチン中のループは、展開イメージからわかるように、データに依存関係が存在しないため、並列化が可能です。

```
real x(100,100),y(100,100),z(100,100)
do i = 1,100
  call abc(x(1,i),y(1,i),100)
  call def(y(1,i),z(1,i),100)
enddo
end
subroutine abc(a,b,n)
real a(n), b(n)
do i = 1,n
  b(i) = a(i) + 1.0 / a(i)
enddo
end
subroutine def(a,b,n)
real a(n), b(n)
do i = 1,n
  b(i) = b(i) + sqrt(a(i))
enddo
end
```

展開イメージ

```
!サブルーチン abc を呼び出したイメージ
do i = 1,100
  y(i) = x(i) + 1.0 / x(i)
enddo
!サブルーチン def を呼び出したイメージ
do i = 1,100
  z(i) = z(i) + sqrt(y(i))
enddo
```

並列化イメージ

```
real x(100,100),y(100,100),z(100,100)
do i = 1,100
  call abc(x(1,i),y(1,i),100)
  call def(y(1,i),z(1,i),100)
enddo
end
```

並列化

◆ **配列構文の場合**

配列構文の場合は、内部的には、do ループのイメージに展開され、その後で、ベクトル化、並列化が行われます。

```
real a(999, 1000), b(999, 1000), c(999, 1000)

      a = b * c                    do j = 1, 1000      並列化
      b = sin(c)        イメージ      do i = 1, 999
                                       a(i, j) = b(i, j) * c(i, j)
                                       b(i, j) = sin(c(i, j))
                                     enddo          ベクトル化
                                   enddo
```

4. 並列化の阻害要因と並列化指示行

4.1 並列化の阻害要因

先にも述べた通り、ループの繰り返し間にデータの依存関係がある場合は、並列化はできません。並列化を妨げる要因をいくつか紹介します。

◆ 添え字に重なりがある場合

```
do i = 1,n
  a(i) = b(i+1)
  b(i) = c(i)
enddo
```

この例も 3 節で述べた例と同様に、ループの繰り返し間にデータの依存関係があるために並列化ができない例です。ただし、ベクトル化は可能です。依存関係によるベクトル化可/並列化不可の理由をもう少し具体的に説明します。

ベクトル化の場合は、ループの繰り返しの実行順序は保証されますが、並列化の場合は、ループの実行順序は保証されません。上記例においてループの繰り返しと配列 b の定義/参照関係に着目すると以下のようになります。この例では、ベクトル化の場合は、参照と定義の順番は保証され、 たとえば、b(3)の値は必ず参照してから定義されることになります。

ループの繰り返し	参照	定義
1	b(2)	b(1)
2	b(3)	b(2)
3	b(4)	b(3)
...	....	....

すなわち、ループの繰り返しにまたがっての定義/参照関係は、プログラム通りの正しい関係が保持されることになります。次に並列化の場合ですが、簡単のために、ループの繰り返し 2 回毎に並列化する場合を例に考えてみます。

ループの繰り返し	参照	定義	
1	b(2)	b(1)	
2	b(3)	b(2)	タスク 1 で実行
3	b(4)	b(3)	タスク 2 で実行
4	b(5)	b(4)	
...	....	....	....

この場合は、タスク 1、タスク 2、... が並列に実行されることになるため、b(3)の値がタスク 1 で参照されるタイミングとタスク 2 で定義されるタイミングの順序は保証できません。すなわち、先にタスク 2 で定義された値（c(3)の値）をタスク 1 で参照して、a(2)に代入してしまう可能性があるわけです。

◆ 定義と引用が閉じていない場合

```
do i = 1,n
  c(i) = t
  t = b(i)
enddo
```

ループ中で、変数 t を引用してから、定義しているため、ループの繰り返しを並列実行すると、結果不正となります。文の意味は変わってしまいますが、次のように変数 t を定義してから、引用していれば、並列化が可能です。

```
do i = 1,n
  t = c(i)
  …
  b(i) = t
enddo
```

◆ if 文下に do 変数以外のインデックス変数がある場合

```
do j = 1,n
  do i = 1,n
    if (a(i,j) >= del) then
     ii = ii + 1
     ic(ii,j) = ii
    endif
  enddo
  do i = 1,ii
    b(i,j) = ic(i,j) + sin(c(ii,j))
  enddo
enddo
```

ii の更新（if 文 then 節の実行）が a(i,j)の値に左右され、ii は外側ループの繰り返しのおいても加算されていきますが、外側ループで並列化された場合、各タスクで並列実行されるループ毎に ii が加算されてしまい、ii の値が正しく計算されなくなり、b の結果が不正となります。

◆ ループからの飛び出しがある場合

```
do j = 1,n
  do i = 1,n
    a(i,j) = sqrt(b(i,j))
    if (a(i,j) >= del) go to 100
    if (c(i,j) >= 0) then
     b(i,j) = c(i,j) - a(i,j)
    else
     b(i,j) = c(i,j) + a(i,j)
    endif
  enddo
enddo
100  continue
```

並列実行されると、ループの繰り返しの実行順序が保証されないため、ループから飛び出すタイミングがプログラム通りにならない場合があり、配列 b の値が不正になる可能性があります。

4.2 指示行の利用とソースプログラムの書き換えによる並列化促進

並列化の阻害要因は、上記以外にもありますが、比較的発生し易い状況は、依存関

係によって並列化が妨げられる場合や、各タスクで同じデータに書き込みを行ってしまうことによって結果不正を引き起こしてしまうような場合です。これらは、プログラミング上の工夫によって、回避することが可能な場合も多々あります。また、ベクトル化の場合と同じように、プログラマには、依存関係がないことがわかっており、それをコンパイラに教えてやることによって並列化が可能になる場合もあります。このような状況に対応できるように、FORTRAN90/SX コンパイラには、指示行が用意されています。本節では、指示行の紹介と利用法、およびプログラミング上の工夫の例を紹介します。

(1) 並列化指示行

並列化指示行は、カラム 1 から

!cdir オプション

の形式で指定します。並列化用の主な指示行のオプションとその利用法は、次の通りです。

- **concur/noconcur**

  直後のループを自動並列化の対象とする/しないを指定します。

  たとえば、並列化するとかえって性能が劣化するループをプログラムが含んでいる場合に、そのループの先頭に noconcur を指定します。

- **inner/noinner**

  最内側ループあるいは一重ループを自動並列化の対象とする/しないを指定します。

  最内側ループは、既定値では自動並列化の対象とはならないため、最内側ループを並列化できると効果がある場合に、inner を指定します。また、一重ループの場合も、並列化効果がコンパイル時に不明な場合は、並列化の対象とはなりませんが、inner を指定することによって並列化をすることが可能となります。

```
!cdir inner
    do i = 1,n
      a(i) = sqrt(b(i)**2 + c(i)**2)
    enddo
```

→ イメージ

```
if (n > 294) then
  ベクトル+並列コード
else
  ベクトルコード
endif
```

この例では、コンパイラは、ループ中のコストを計算して、条件並列化を

行っています。

◆ **nosync**

ループ中の配列要素に重なりがないことを指定します。

次の例で、k1 と k2 の値がコンパイル時に不明な場合は、k1=k2 であった場合に結果不正になる可能性があるため、このループを自動並列化することはできません。k1 と k2 の値が同じでないことがわかっている場合は、nosync を指定することによって並列化が可能となります。

```
!cdir nosync
do j = 1,ny
  do i = 1,nx
    a(i,k1,j+1) = a(i,k2,j) + b(i)
  enddo
enddo
```

並列化

◆ **select(concur)**

多重ループにおいて、指定されたループを優先して並列化します。

多重ループにおいて、並列化した場合に最も効率のよいループを指定することができます。たとえば、最外側ループの繰り返し回数が 1 や 2 である場合、最外側ループで並列化しても、その効果は期待できません。このような場合は、次の例のように、 select(concur)を指定することによって、より効率よく並列化することが可能となります。

```
    do k = 1,nz
!cdir select(concur)
      do j = 1, ny
       do i = 1,nx
         c(i) = b(i,j,k) / dble(nx)
         a(i,j,k) = a(i,j,k) + c(i) / 2.0
       enddo
      enddo
    enddo
```

イメージ

```
do k = 1,nz
 do j = 1, ny
  do i = 1,nx
    c(i) = b(i,j,k) / dble(nx)
    a(i,j,k) = a(i,j,k) + c(i) /  2.0
  enddo
 enddo
enddo
```

並列化

◆ **cncall**

並列化されてもよい手続きであることを指定します。

あらかじめ、ループ中で呼び出しているサブルーチンが並列実行されても問題がないことがわかっている場合、cncall を指定することによっ

て並列化が可能となります。自動並列化機能では、引数として渡す変数が、ループ中で明示的に定義されていない（ループ中で値が代入されない）場合は、タスク間共有データ（タスク間での共通の領域を参照するデータ）となるため、渡されたデータがサブルーチン中で更新されないということが、並列実行しても結果不正にならない重要な条件となります。

```
!cdir cncall
do i = 1,n                    並列化
  call sub(a(i), x)
enddo
```

たとえば、この例の場合、x はタスク間共有データであるため、サブルーチン sub 中で、x が更新されると、結果不正を引き起こすことになります。したがって、x が sub 中で変更されない場合のみ、cncall 指示行を指定して並列化することが可能です。

◆ **強制並列化指示行**

上記指示行等を指定してもコンパイラにそのループが並列化可能であることが認識できず並列化が行なわれない場合でも、利用者には、そのループが並列化可能であることがわかっている場合があります。このような場合にコンパイラに強制的に並列化を行なわせることを指定する指示行が、強制並列化指示行です。

ただし、この指示行を指定した場合、コンパイラはデータの依存関係などのチェックは行なわずに並列化をするので、並列化したときの動作の妥当性については利用者が保証しなければなりません。

```
!cdir parallel do private(wk)
do j = 1, 10
  do i = 1, 100
    wk(i) = a(i) + b(i)
  enddo
  call sub(x(j), wk)
enddo                         並列化
```

parallel do 指示行は直後のループを並列実行することを指定します。各ループの繰り返しで作業用に使用している変数（wk）のようにタスク

固有のデータは、private オプションで指定します。ただし、do 変数はコンパイラが自動的にタスク固有データと解釈するので、明示的に private で指定する必要はありません。private に指定されなかったデータはタスク間共有データとなります。

```
!cdir parallel do
   do i = 1, 100
       call sub(a(i), b(i), x)
!cdir atomic
       s = s + a(i) * b(i)
   enddo
```

並列化

atomic 指示行は、parallel do 指示行で並列化されたループ中の総和や内積など排他的に処理しなければならない代入文に指定します。

(2) プログラミング上の工夫による並列化促進

◆ ループの入れ換えによる並列化促進

次のループは外側ループで並列化すると、a(i,j)と a(i,j-1)の間に依存関係があり、実行結果が不正となります。この場合は、ループを入れ換えることによって並列化が可能となります。

```
do j = 2,m                          do i = 1,n
  do i = 1,n                  ⟶      do j = 2,m
    a(i,j) = a(i,j-1) * b(i,j)          a(i,j) = a(i,j-1) * b(i,j)
  enddo                               enddo
enddo                               enddo
```

◆ 仮引数の配列サイズ変更による並列化促進

引数として渡ってきたデータは、タスク間で共有となるため、次の例では、配列 c がタスク間共有変数となります。このため、最外側ループ（do k=1,nz）で並列化を行うと、配列 c の領域を各タスクで書き換えることになり、結果不正となってしまいます。

そこで、配列 c の次元の宣言を変更し、最外側ループで異なる領域を使用することにすれば、並列化が可能となります。もちろん、この sub を呼び出しているサブルーチン中の対応する配列に対しても修正が必要です。

```
subroutine sub(a,b,c,nx,ny,nz)
real*8 a(100,100,100),b(0:100,100,100)
real*8 c(0:100)
do k = 1,nz
  do j = 1, ny
   do i = 1,nx
     c(i) = b(i,j,k) / dble(nx)
   enddo
   do i = 1,nx
     a(i,j,k) = a(i,j,k) + (c(i-1)+c(i))
                                  / 2.0
   enddo
  enddo
enddo
end
```

```
subroutine sub(a,b,c,nx,ny,nz)
real*8 a(100,100,100),b(0:100,100,100)
real*8 c(0:100,100)
do k = 1,nz
  do j = 1, ny
   do i = 1,nx
     c(i,k) = b(i,j,k) / dble(nx)
   enddo
   do i = 1,nx
     a(i,j,k) = a(i,j,k) + (c(i-1,k)+c(i,k))
                                  / 2.0
   enddo
  enddo
enddo
end
```

◆ **作業領域の受け渡しをしないことによる並列化促進**

上例において、サブルーチン sub を呼び出している側が、配列 c を参照していない、すなわち配列 c を単なる作業領域として確保している場合、配列 c を引数として渡さずに、サブルーチン sub 側で配列宣言することによって、並列化が可能となります。

```
subroutine sub(a,b,c,nx,ny,nz)          ← 引数 c の削除
real*8 a(100,100,100),b(0:100,100,100)
real*8 c(0:100)
do k = 1,nz
  do j = 1, ny
   do i = 1,nx
     c(i) = b(i,j,k) / dble(nx)
   enddo
   do i = 1,nx
     a(i,j,k) = a(i,j,k) + (c(i-1)+c(i))/2.0
   enddo
  enddo
enddo
end
```

5. あとがき

以上、Fortran90 で利用可能な自動並列化機能について、簡単に説明してまいりました。　並列処理においては、その処理方式を理解した上で、最適に並列処理が行われるようにプログラムのチューニングを行うことが、性能向上には大変重要です。こらの詳細な説明は、号を改めて行いたいと思います。

最後に、並列実行されたプログラムの性能情報をみることができる proginf の結果を紹介して、本稿の説明を終わりたいと思います。

```
******  Program Information  ******
 Real Time (sec)       :       2241.800319    経過時間
 User Time (sec)       :       8903.817840    ユーザCPU時間
 Sys  Time (sec)       :          0.848539    システムCPU時間
 Vector Time (sec)     :       8713.069899    ベクトル命令実行時間
 Inst. Count           :     716897080005.    全命令実行数
 V. Inst. Count        :     331530260017.    ベクトル命令実行数
 V. Element Count      :   81369094603294.    ベクトル命令処理要素数
 FLOP Count            :   35064202883826.    浮動小数点データ処理要素数
 MOPS                  :       9181.955751    1秒あたりの実行演算数
 MFLOPS                :       3938.108743    1秒あたりの浮動小数点データ処理要素数
 MOPS   (concurrent)   :      36469.877067    1秒あたりの実行演算数(実効時間)                ※
 MFLOPS (concurrent)   :      15641.802862    1秒あたりの浮動小数点データ処理要素数(実効時間) ※
 VLEN                  :        245.434895    平均ベクトル長
 V. Op. Ratio (%)      :         99.528629    ベクトル演算率
 Memory Size (MB)      :       5440.000000    メモリサイズ
 Max Concurrent Proc.  :                4.    実行プロセッサ数                               ※
  Conc. Time(>= 1)(sec):       2241.698300    少なくとも1台のCPUが同時に(並列に)動いた時間   ※
  Conc. Time(>= 2)(sec):       2220.965807    少なくとも2台のCPUが同時に(並列に)動いた時間   ※
  Conc. Time(>= 3)(sec):       2220.934612    少なくとも3台のCPUが同時に(並列に)動いた時間   ※
  Conc. Time(>= 4)(sec):       2220.223084    少なくとも4台のCPUが同時に(並列に)動いた時間   ※
 Event Busy Count      :                0.    event待ち回数(マクロタスク用)                  ※
 Event Wait (sec)      :          0.000000    event待ち時間(マクロタスク用)                  ※
 Lock Busy Count       :                0.    lock待ち回数(マクロタスク用)                   ※
 Lock Wait (sec)       :          0.000000    lock待ち時間(マクロタスク用)                   ※
 Barrier Busy Count    :                0.    barrier待ち回数(マクロタスク用)                ※
 Barrier Wait (sec)    :          0.000000    barrier待ち時間(マクロタスク用)                ※
 MIPS                  :         80.515695    1秒当たりの命令実行数
 MIPS (concurrent)     :        319.800876    1秒当たりの命令実行数(実効時間)                ※
 I-Cache (sec)         :          0.239609    命令キャッシュミス時間
 O-Cache (sec)         :          5.363951    データキャッシュミス時間
 Bank (sec)            :          9.214590    バンクコンフリクト時間

 Start Time (date)  :  2002/08/21 21:21:08    開始時刻(日付)
 End   Time (date)  :  2002/08/21 21:58:29    終了時刻(日付)
```

この情報は、実行時オプション setenv　F_PROGINF　YSE あるいは DETAIL を指定することによって採取できます（東北大学では規定値として DETAIL が設定されています）。　は、並列処理使用時に表示される情報です。上記、Conc. Time を調べることによって、このプログラムが 4 つの CPU で実行されたことがわかります。

並列化は、ベクトル化と異なり、すべてを自動にまかせて性能を向上させること

は難しい機能です。本稿が、自動並列化利用の手助けになれば幸いです。

参考文献

［1］FORTAN90/SX プログラミングの手引き　　日本電気　G1AF07
［2］FORTAN90/SX 並列処理機能利用の手引き　日本電気　G1AF08