HPF/VPP 入門 (1)

浅岡香枝 *　平野彰雄 *

1　はじめに

HPF(High Performance Fortran) は、 Fortran を拡張したデータ並列言語であり、分散メモリ型並列計算機用に開発された並列化言語です。本センターでサービスしている分散メモリ型ベクトル並列計算機 VPP800/63 でも HPF コンパイラをサポートしており、利用することができます。

本稿では、 VPP で HPF を利用するにあたって、知っておけば役に立つというような情報やプログラミングガイド的なことをまとめて、何回かにわけて解説していこうと思います。今回は、 HPF の概要や、 HPF 指示文、コンパイルおよび実行方法について紹介します。

2　HPF プログラムの基本

2.1　基本処理

分散メモリ型並列計算機で並列プログラミングを行うには、**データマッピング** (配列データをどうプロセッサに割り付けるか)、**計算マッピング** (計算処理をどうプロセッサに分割するか)、**通信処理** (プロセッサ間のデータ転送や同期処理) を考える必要がありますが、 HPF は、このうちのデータマッピングさえユーザが指定すればあとの処理はすべてコンパイラが行ってくれるという自動並列化を目指して設計されています。

コンパイラは、ユーザが指示したデータマッピングに基づいて、 Owner Computes Rule により計算のマッピングを決めループを並列化します。 Owner Computes Rule とは、演算の左辺に現れるデータを保持するプロセッサがその演算を行うというものです。また、 HPF では、すべての変数はグローバルに扱われます。他プロセッサ上にあるデータへのアクセスが生じた場合にはプロセッサ間のデータ転送が自動で生成されます。

2.2　HPF のプログラミング

HPF のプログラミングは、従来の Fortran プログラムに HPF 指示文を挿入し、並列化を行います。

HPF 指示文には、データマッピング、計算のマッピング、データ転送を指示するものがあります。

2.3　HPF の仕様

HPF の仕様は HPF Forum[3] によって検討され、最新は HPF2.0 (基本仕様 + 公認拡張仕様)[1] となっています。また、 HPF/JA[1] という拡張仕様が JAHPF(Japan Association of High Performance Fortran)[4] によってまとめられています。 HPF/JA 拡張仕様は、 HPF2.0 基本仕様と公認拡張仕様の大部分をカバーしており、きめ細かな通信の指定など、独自の機能が追加されています。仕様の包含範囲を図 1に示します。

VPP の HPF コンパイラ (以降、 HPF/VPP と呼ぶ) は、 HPF/JA 拡張仕様に基づいています。

3　HPF 指示文

3.1　指示文の形式

HPF 指示文は Fortran の注釈行の形であり、行の先頭から `!hpf$` で始まります。例えば、次のような記述です。

```
!hpf$ distribute a(block,block) onto p
```

* あさおかかえ、ひらのあきお (京都大学 学術情報メディアセンター)

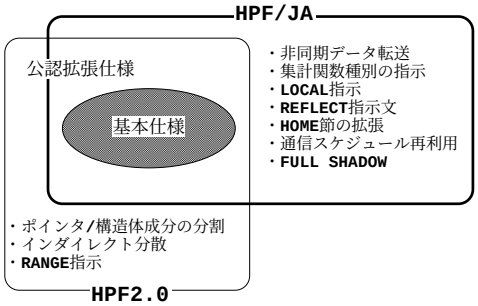


図 1. 仕様の包含範囲

固定形式の場合には、!hpf$ の代りに chpf$ で記述することもできます。

指示文が複数行にまたがる場合は、継続行も!hpf$ で始まる必要があります。次に、継続行の例を示します。

(1) 自由形式

```
!hpf$ distribute a(block,block) &
!hpf$                    onto p
```

(2) 固定形式

```
!hpf$ distribute a(block,block)
!hpf$*                   onto p
```

また、HPF/JA 拡張仕様の独自の指示文は、!hpfj で始まります。

3.2 データマッピング

3.2.1 プロセッサ構成の宣言

配列データの分散先となる抽象プロセッサについて宣言します。抽象プロセッサを配列の形状に並べたものをプロセッサ構成と呼び、 processors 指示文で宣言します。

配列と同じ形式で宣言し、例えば、大きさ 4 の 1 次元のプロセッサ構成 p1 を宣言するには、次のように記述します。

```
!hpf$ processors p1(4)
```

同様に、 2×2 の 2 次元のプロセッサ構成 p2 を宣言するには、次のように記述します。

```
!hpf$ processors p2(2,2)
```

3.2.2 配列データの分散と分散形式

配列データの分散は、 distribute 指示文で指定します。 distribute 指示文に、分散配列名、分散形式、 processors 指示文で宣言したプロセッサ構成を指定して、例えば次のように記述します。

```
!hpf$ distribute a(block) onto p1
```

この例は、一次元配列 a を block という分散形式で、プロセッサ構成 p1 に対して分散することを指示します。

分散形式には、次のものがあります。

(1) block(幅)

均等に割った連続区間を各プロセッサに分散します。括弧で区間の幅を指定することもできます。

(2) cyclic

循環して各プロセッサに分散します。仕様では、幅の指定はできますが、 HPF/VPP では制限機能となっています。

(3) gen_block(*array*)

不均等な大きさの連続区間を各プロセッサに分散します。 *array* は各プロセッサへ分散する区間幅を指定した一次元配列です。

(4) *(アスタリスク)

指定された次元において分散しないことを表します。多次元配列の分散時に使われます。

次に、大きさ 12 の一次元配列を 4 つのプロセッサに対して分散する例を示し、分散の様子を図 2に示します。

```
      real(8) :: a(12),b(12),c(12)
!hpf$ processors p1(4)

!!!!!! (1)  block 分散 !!!!!!!
!hpf$ distribute a(block) onto p1

!!!!!! (2)  cyclic 分散 !!!!!!
!hpf$ distribute b(cyclic) onto p1

!!!!!! (3)  gen_block 分散 !!!!!!
!hpf$ distribute &
!hpf$   c(gen_block((/5,4,2,1/))) &
!hpf$   onto p1
```

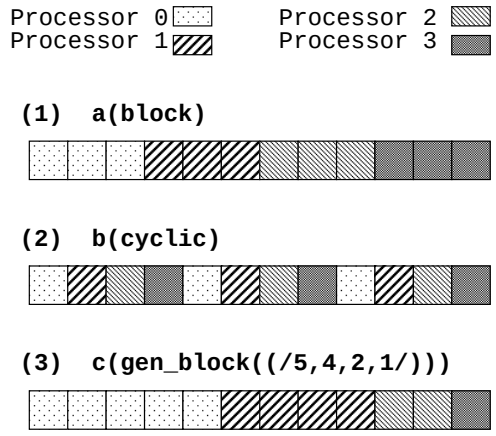


図 2. 分散の様子 (1 次元配列)

また、多次元配列の分散の例を次に示します。例 (1) は、 2 次元配列を 2 つの次元で block 分散した例で、例 (2) は、 2 次元配列を 2 次元目だけで block 分散した例です。これらの分散の様子を図 3に示します。

例 (1)

```
      real(8) :: a(8,8)
!hpf$ processors p2(2,2)
!hpf$ distribute a(block,block) onto p2
```

例 (2)

```
      real(8) :: a(8,8)
!hpf$ processors p1(4)
!hpf$ distribute a(*,block) onto p1
```

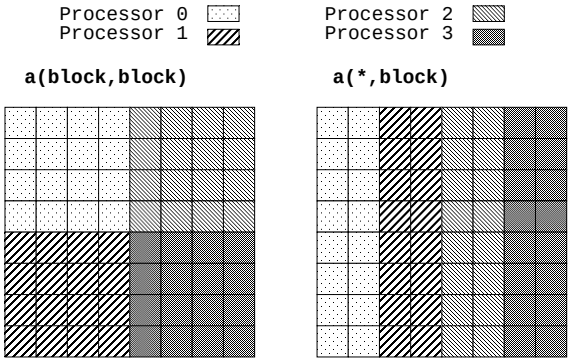


図 3. 分散の様子 (2 次元配列)

3.2.3 テンプレート配列

配列の分散は、実配列に対して指定するだけでなく、テンプレート配列に対しても指定することができます。

テンプレート配列とは、配列としての形状はもちますが型や値をもたず、メモリ領域をとらない仮想的な実体で、 template 指示文を使って実配列と同じ形式で宣言します。

次の例は、大きさ 100 の 1 次元のテンプレート配列 t を宣言し、これを block 分散する例です。

```
!hpf$ template t(100)
!hpf$ processors p1(4)
!hpf$ distribute t onto p1
```

3.2.4 整列

例えば、次のようなプログラムがあります。

```
      real(8) :: b(11),x(12)
      do i=1,11
         b(i)=x(i+1)+1.0d0
      end do
```

この例でプロセッサ間の通信を必要とせずに演算を行うためには、 b(i) と x(i+1) が同じプロセッサ上にマッピングされていなければなりません。

このようなマッピングを指定するために、 align 指示文があります。 align 指示文は、 distribute 指示文で分散が指定された配列に対して、別の配列の分散を整列させます。 align 指示文を用いて、 b(i) と x(i+1) を同じプロセッサ上にマッピングするには、次のように記述します。

```
!hpf$ align b(i) with x(i+1)
```

これは、配列 x に対して配列 b を整列させており、変数 i を用いて整列の関係を指定しています。変数 i をダミー変数といいます。この例のように、整列関係はダミー変数を用いた 1 次式で表します。

またこのとき、変数 i は、配列 b の宣言範囲すべての添字について有効でなければなりません。したがって、次のようなプログラムは書けません。

```
      real(8) :: x(12),y(12)
!hpf$ processors p1(4)
!hpf$ distribute y(block) onto p1
!hpf$ align x(i) with y(i+1)
```

このような場合にはテンプレート配列が有効です。 align 指示文の指定に必要な範囲の上下限をもつテンプレート配列を宣言し、これを整列先の配列とすることで、無駄なメモリ領域をとらずにマッピングが指定できます。先ほどの例を、テンプレート配列を利用すると次のようになります。

```
      real(8) :: x(12),y(12)
!hpf$ template :: t(13)
!hpf$ processors p(4)
!hpf$ distribute t(block) onto p
!hpf$ align x(i) with t(i)
!hpf$ align y(i) with t(i+1)
```

次に align 指示文を使った他の整列の例を紹介します。それぞれの整列の様子を図 4に示します。

次の例は、最も単純な例です。

```
!hpf$ align a(i) with x(i)
```

この例のように、ダミー変数が単純引用されている場合は、これを ":"で置き換え、次のように記述することもできます。

```
!hpf$ align a(:) with x(:)
```

1 要素飛ばしで整列させる場合は、次のように記述できます。

```
!hpf$ align c(i) with x(2*i-1)
```

また、次元の異なる配列間の整列は、 "*"を使って指定します。

```
!hpf$ align d(*,i) with x(i)
```

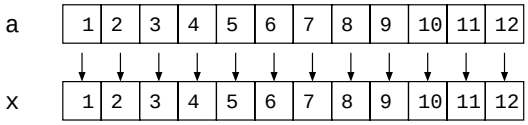
この例では、配列 d の 2 次元目を配列 x に整列させることを意味します。

3.3 手続きの並列化

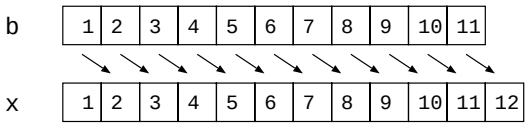
3.3.1 並列化の明示

並列化したいループの直前に、 independent 指示文を指定します。ただし、回帰的演算のような並列化不可能な do ループに対して independent 指示文を指定しても無視されます。

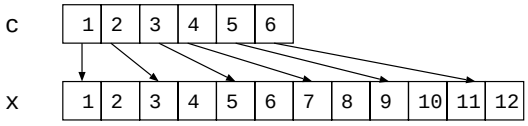
align a(i) with x(i)



align b(i) with x(i+1)



align c(i) with x(2*i-1)



align d(*,i) with x(i)

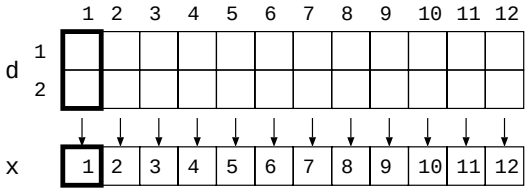


図 4. 整列の例

3.3.2 new 節

例えば、次のようなループがあります。

```
do j=1,n
 do i=1,n
  s=a(i,j)**2+b(i,j)**2
  c(i,j)=s
 end do
end do
```

HPF/VPP では、この do ループは、コンパイラによって自動的に並列化はされませんが、independent 指示文を外側 do ループに指定すれば、並列化されます。しかし、変数 s および並列化 do ループ内にある do ループの制御変数 i について、各プロセッサ持つ s、 i の値を一致させるための通信が発生し、性能劣化の原因となります。

したがって、これらの変数を new 節に指定します。 new 節に指定された変数は、並列化 do ループの各繰り返しに閉じたプライベートな変数として扱

われるので、不必要な通信を削減することができます。ただし、 new 節に指定された変数の値を並列化ループの終了後に参照することはできません。

また、並列化 do ループの制御変数 j は、ループの終了後にとる n+1 の値を保証するための通信が発生します。もし、 do ループ終了後に j の値を参照しないのであれば、 j を new 節に指定することで、この通信を削減することができます。

先ほどのプログラム例に、 independent 指示文と new 節を指定したものを次に示します。

```
!hpf$ independent,new(s,i,j)
      do j=1,n
       do i=1,n
        s=a(i,j)**2+b(i,j)**2
        c(i,j)=s
       end do
      end do
```

HPF/VPP では、コンパイラが自動で並列化した do ループについても new 節を自動的に誘導することはできませんので、全ての並列化ループについて、 independent 指示文および new 節を指定する必要があります。

independent 指示文が指定されていなくても並列化されたループは、後述するコンパイル時に出力されるソースリスト、または、次のコンパイルメッセージで見つけることができます。

```
joo7101i-i  "a.hpf", line 21: This DO
loop has been parallelized.
(Home Variable:c(:,j))
```

また、 new 節に指定する変数もソースリストやコンパイルメッセージから知ることができます。independent 指示文が指定されている do ループの中に、 new 節や後述の reduction 節にも指定されていない変数が代入側に現れる場合、次のようなコンパイルメッセージが出力されます。

```
joo7103i-i  "a.hpf", line 21: DO variable
of the inner loop does not appear in the
NEW clause. (name:i)
joo7102i-i  "a.hpf", line 21: A scalar
variable appears on the left side of
an assignment statement within DO loop,
but does not appear in the NEW clause
nor the REDUCTION clause. (name:s)
```

これらのメッセージに注意して、適切な指定を行って下さい。

3.3.3 集計演算と reduction 節

総和演算を行うための do ループは、繰り返し間に依存関係が発生するので基本的には並列化できません。しかし、図 5に示すように、各プロセッサ毎に部分和 (s1,s2) を求め、最後に部分和を足すことで (s1+s2)、並列に計算ができます。

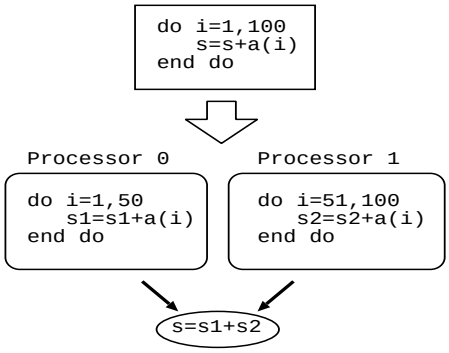


図 5. 総和演算の並列計算

このような集計演算を行うために、 reduction 節が用意されています。 reduction 節で指定できる集計演算の種別には、表 1に示すものがあります。

表 1. reduction の集計種別

種類	集計種別
加算	+
乗算	*
論理演算	.and. , .or. , .eqv. ,.neqv.
最大, 最小	max,min
最大値と位置	fastmax,lastmax
最小値と位置	fastmin,lastmin
ビット演算	iand,ior,ieor

reduction 節は independent 指示文の節として記述し、指定形式は次のとおりです。

reduction(集計種別: 集計変数 / 位置変数 /)

位置変数は、集計種別に fasttmax、 lastmax、fastmin、 lastmin を使う場合に使用します。

総和と最大値検索の例を次に示します。

```
!hpf$ independent,new(i),&
!hpf$  reduction(+:vsum),&
!hpf$  reduction(max:vmax)
      do i=1,n
        vsum=vsum+a(i)
        if( a(i) > vmax) then
           vmax = a(i)
        end if
      end do
```

この例では、vsum に総和演算の結果が求まり、vmax に最大値検索の結果が求まります。

また、最大値 (最小値) と見つけたインデックスの値を求めるのに、複数のプロセッサで最大値 (最小値) が求まった場合、 fastmax、 fastmin は最小のインデックスの値を返し、 lastmax、 lastmin は最大のインデックスの値を返します。

次に、 lastmax を使用した例を示します。

```
!hpf$ independent,new(i),&
!hpf$ reduction(lastmax:vmax/iloc/)
      do i=1,n
        if( a(i) >= vmax) then
           vmax = a(i)
           iloc=i
        end if
      end do
```

この例では、vmax に最大値が求まり、最大のインデックスの値がiloc に求まります。

3.3.4 分割の指示

基本的に、 do ループは Owner Computes Rule によって各プロセッサに分割されますが、 on 指示文を使ってユーザが指定することもできます。

次の例は、 Owner Computes Rule では a(i) を保持するプロセッサが i 番目の繰り返しを実行しますが、 b(i) を保持するプロセッサが実行するように on 指示文で指定しています。

```
      real(8) :: a(12),b(12),c(12)
!hpf$ processors p1(4)
!hpf$ distribute a(block) onto p1
!hpf$ distribute b(cyclic) onto p1
!hpf$ align c(i) with b(i)
        :
!hpf$ independent,new(i)
     do i=1,12
!hpf$ on home (b(i)) begin    ! ここから
        a(i)=b(i)+c(i)
!hpf$ end on                  ! ここまで
     end do
```

分割の指定は、並列化ループの中の演算を on 指示文と end on 指示文で囲み、 on 指示文の home 節に分散配列を基準としたループ分割の指定をします。

この例では、配列 b、配列 c のデータマッピングが同じなので、このような計算マッピングを指定することで、演算中に発生する通信を 2 回 (b(i) と c(i) のアクセス) から 1 回 (a(i) アクセス) に削減することができます。

3.4 隣接プロセッサ上データのアクセス

例えば、次のようなプログラムがあります。

```
      real(8) :: a(6,6),b(6,6)
!hpf$ processors p1(2)
!hpf$ distribute a(*,block) onto p1
!hpf$ align b(i,j) with a(i,j)
!hpf$ independent,new(i,j)
      do j=2,5
!hpf$ on home(a(:,j)) begin
       do i=1,6
        a(i,j)=b(i,j-1)+b(i,j)+b(i,j+1)
       end do
!hpf$ end on
      end do
```

この例では、図 6に示すように、配列 a、配列 b ともに 2 つのプロセッサ 0 に 3 列ずつ分散されいます。そして、 j=3、 j=4、の繰り返しが実行される際に、隣接プロセッサ上のデータに対するアクセスが発生します。

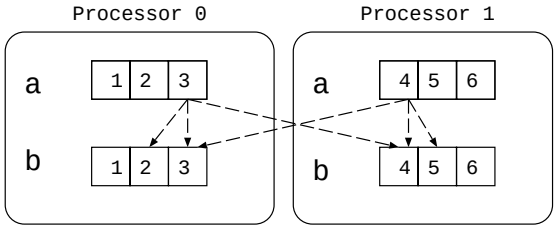


図 6. 隣接プロセッサ上データへのアクセス

そこで、図 7に示すように、隣接プロセッサ上のデータを保持するための領域 (shadow 領域といいます) を宣言し、この領域に必要となるデータをあらかじめコピーし、演算中にこれを参照すれば、プロセッサ間の通信を行わずに演算ができます。

shadow 領域を使用するために、先ほどのプログラムに必要な指示文を挿入したものを次に示します。

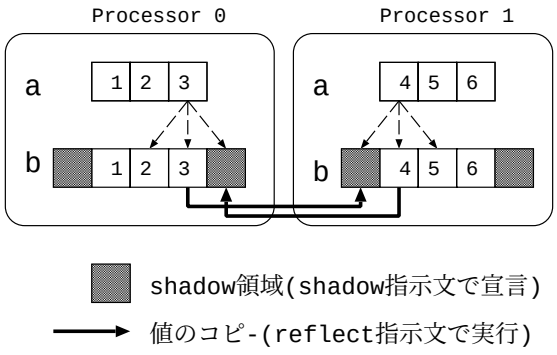


図 7. shadow 領域へのアクセス

```
      real(8) :: a(6,6),b(6,6)
!hpf$ processors p1(2)
!hpf$ distribute a(*,block) onto p1
!hpf$ align b(i,j) with a(i,j)
!hpf$ shadow b(0,1:1) !shadow 領域宣言
!hpf$ reflect b       ! 値のコピー
!hpf$ independent,new(i,j)
      do j=2,5
!hpf$ on home(a(:,j)) begin
!hpfj local begin     ! 通信不要 (ここから)
       do i=1,6
        a(i,j)=b(i,j-1)+b(i,j)+b(i,i+1)
       end do
!hpfj end local       ! 通信不要 (ここまで)
!hpf$ end on
      end do
```

shadow 領域は、 shadow 指示文で宣言され、分散後のローカルな部分配列の下端と上端に確保されます。 shadow 指示文には、配列名と配列の各次元における shadow 領域の幅を指定します。幅の指定は、上端に確保する領域幅と下端に確保する領域幅を ":"でつなげて記述し、領域を確保しない場合は 0 と記述します。上記の例では、配列 b に対して、2 次元目のみ上端と下端に 1 ずつの shadow 領域を宣言しています。

そして、 shadow 領域に元の配列から値をコピーするには、 reflect 指示文を使います。上記の例ように reflect 指示文には、値をコピーする配列名を指定します。

また、 local 指示文は、データへのアクセスがプロセッサ内に閉じており、通信が不要であることを明示するものです。 shadow 領域をアクセスする場合には必須となります。

3.5 配列データの一括転送

例えば、次のようなプログラムがあります。

```
      real(8) :: u(n1,n2),t(n2,n1)
!hpf$ processors p(4)
!hpf$ distribute u(*,block) onto p
!hpf$ distribute t(*,block) onto p
        :
!hpf$ independent,new(i,j)
      do j=1,n2
         do i=1,n1
            t(j,i) = u(i,j)
         end do
!hpf$ end on
      end do
```

この例は転置配列をつくるためのデータ転送を do ループで行っています。並列化はされますが、t(j,i) と u(i,j) は同じプロセッサ上にないため 1 要素単位の転送となり、非常に効率が悪いです。

そこで、 1 要素単位でなく、一括してデータ転送するために、次のように非同期転送構文に置き換えます。非同期転送構文は、本来、指定したデータ転送とその他の実行文を非同期に行うための機能ですが、データ転送を一括してして行うため、配列間のデータ転送を効率良く行うことができるのです。

```
      real(8) :: u(n1,n2),t(n2,n1)
!hpf$ processors p(4)
!hpf$ distribute u(*,block) onto p
!hpf$ distribute t(*,block) onto p
!hpfj asyncid id1
        :
!hpfj asynchronous(id1), nobuffer
      forall(i=1:n2,j=1:n1) t(j,i)=u(i,j)
!hpfj end asynchronous
!hpfj asyncwait(id1)
```

asyncid 指示文は転送識別子を宣言します。転送識別子とは、非同期転送の開始と終了を対応づけるための ID です。 asynchronous 指示文と end asynchronous 指示文で転送のパターンを囲みます。転送のパターンは forall 文、もしくは、配列代入文を記述します。上記の例では、 forall 文で先ほどの do ループと同じ内容のデータ転送を記述しています。また、 asynchronous 指示文には転送識別子を指定します。そして、 asyncwait 指示文で、指定した転送識別子に対応する転送の終了を待ちます。

この置き換えにより、実行性能を大幅に向上させることができます。

4　コンパイルと実行

では最後に、 HPF プログラムのコンパイル方法および実行方法について簡単に解説します。

コンパイルは、 Fortran のコンパイルコマンド ”frt”に ”-Wh”オプションを指定します。このとき、ソースファイルの拡張子が、 ”.f”であれば固定形式であるとみなしてコンパイルし、 ”.f90”、 ”.f95”、 ”.hpf”であれば自由形式であるとみなしてコンパイルされます。

【使用例】

```
% frt -Wh samp.hpf
```

また、オプションに ”-Wh,-Ldt,-Ol3 -Wv,-m3 -Pso”を指定すると、コンパイラは並列化ループの抽出を最大限に行い、また、分散情報および図 8に示す並列化情報を含むソースリストを出力します。

【使用例】

```
% frt -Wh,-Ldt,-Ol3 -Wv,-m3 -Pso samp.hpf
```

```
v p 1           do i=1,n
v p               x(i)=real(i,8)
v p             end do

        !hpf$ independent,new(i), &
        !hpf$          reduction(+:dot)
v p           do i=1,n
  p     !hpf$ on home(x(i)),local(x,y)
v p             dot = dot + x(i) * y(i)
v p           end do
```

図 8. ソースリスト

図 8のソースリストには、プログラムソースの左にベクトル化・並列化の情報が表示されています。一番左はベクトル化の域で、英字 v はベクトル化されたことを示しています。続く英字 p はループが並列化されていることを示します。そして数字は、通信ライブラリの呼出し数で、プロセッサ間通信の目安になります。

プログラムの実行は、デバッグプログラムなど、少数の PE を使い短時間で終了するものであれば、会話型で実行し、そうでなければ NQS バッチ型で実行するのがよいでしょう。

【会話型で実行】 PE 数 4 で./a.out を実行する

```
% jobexec -vp 4 ./a.out
```

【NQS で実行】

```
% qsub sample.sh
```

```
# ----サンプルスクリプト sample.sh-------
# sample script sample.sh
# @$-q g          # キュー g に依頼
# @$-lT 20:00:00 # CPU 時間の指定 (20 時間)
# @$-lM 7gb       # メモリサイズの指定 (7GB)
# @$-lP 4         # PE 数の指定
# @$-eo           # 標準出力と標準エラー
#                 # 出力をまとめる
#
set -x
cd $QSUB_WORKDIR
./a.out           # ./a.out の実行
# ---------------------
```

使用できる PE 数、メモリ量、 CPU 時間などの制限や、詳細な使い方などはスーパーコンピュータのホームページ [5] をご覧ください。

5　おわりに

今回は、 HPF の概要、 HPF 指示文、および、簡単なコンパイル方法と実行方法を解説しました。次回は、手続き種別と手続きの呼び出しについて紹介する予定です。

なお、 HPF のオンラインマニュアル [2] はホームページ [5] においてありますので、興味を持たれた方はご覧ください。

参考文献

[1] High Performance Fortran Forum: High Performance Fortran 2.0 公式マニュアル、シュプリンガー・フェアラーク東京 (1999)

[2] UXP/V HPF 使用手引書 V20L20 L01041 TX01646 用、 J2U5-0450-01、富士通株式会社

[3] http://www.crpc.rice.edu/HPFF/

[4] http://www.hpfpc.org/jahpf/

[5] http://www.kudpc.kyoto-u.ac.jp/Supercomputer/